Universidad de Murcia
Facultad de Informática

Trabajo Fin de Máster

Paralelización de algoritmos *QFT* mediante *OpenMP* y *CUDA*

Isaac Martínez Forte
48474854-P
isaac.martinez@um.es

Director:
Joaquín Cervera López
jcervera@um.es

Máster Nuevas Tecnologías de la Informática

Murcia - julio 2014

LaTeX

Resumen

En este trabajo se aborda la optimización y paralelización de distintos algoritmos relacionados con la técnica de control robusto *QFT*. Los algoritmos estudiados corresponden a las primeras fases de diseño *QFT*, concretamente un algoritmo de cálculo de plantillas, un algoritmo de cálculo del contorno de las plantillas y un algoritmo de cálculo de fronteras. Con este proyecto se plantea iniciar una vía de trabajo poco estudiada hasta ahora, la paralelización de algoritmos relacionados con *QFT*, para explotar al máximo las posibilidades de aceleración de cálculo que proporciona el paralelismo, con el objetivo de proporcionar la mayor interactividad posible al usuario, siendo esta interactividad un factor clave para una buena explotación de las posibilidades del diseño con *QFT*.

Las actuaciones concretas que se han realizado han sido la paralelización de dichos algoritmos en la *CPU*, utilizando las librerías *OpenMP*, y la implementación paralela de los mismos algoritmos en la *GPU*, utilizando el lenguaje de programación para tarjetas gráficas *CUDA*. Estas acciones se han planteado para su uso en el ámbito de un ordenador personal, con el fin de aprovechar los procesadores multinúcleo que incorporan hoy en día los ordenadores personales, así como la utilización de la potencia de cálculo en punto flotante y masivamente paralelo que ofrece la tarjeta gráfica.

Este trabajo se enmarca en un proyecto a largo plazo iniciado en ([Martínez [2013]]), que persigue la creación de una herramienta software libre, que permita aunar las diferentes investigaciones y desarrollos realizados en *QFT* a lo largo de los años. Al ser software libre dará la posibilidad a los investigadores y usuarios de disponer de una herramienta, que puede ser adaptada a sus necesidades y ampliada con los nuevos descubrimientos que deseen compartir con la comunidad.

Las actuaciones se han centrado en primer lugar, en un estudio teórico de los algoritmos implementados, determinando para cada uno de ellos su complejidad en los distintos casos planteables. A continuación, se ha revisado el código secuencial para mejorarlo y reducir así el número de operaciones que se ejecutan. En tercer lugar se ha realizado una implementación paralela de los algoritmos, tanto en *CPU* usando *OpenMP* como en *GPU* usando *CUDA*. Para terminar, se ha realizado un estudio experimental sobre el rendimiento de las distintas implementaciones desarrolladas de los algoritmos. Se han comparado los tiempos de ejecución de los algoritmos secuenciales con la versión paralela en *CPU* y con la versión implementada en *GPU*, para distintos ejemplos ilustrativos.

Abstract

This Master Thesis we deal with the optimization and parallelization of different algorithms related to QFT, a robust control design technique. The studied algorithms correspond to the first stages of the *QFT* design, in particular a templates computation algorithm, an algorithm for computation the contour template, and a boundaries calculation algorithm. This project proposes the beginning of a working path little studied so far: the parallelization of algorithms related to *QFT* for the purpose of exploiting the full potential of accelerating the calculations that parallelism provides. The aim is to provide the greatest possible interactivity to the user, being this interactivity a key factor for a good exploitation of the design possibilities of *QFT*.

The specific actions carried out are: the parallelization of these algorithms in the *CPU*, using the *OpenMP* libraries, and the parallel implementation of these algorithms in the *GPU*, using *CUDA*, a programming language for graphics cards. These actions have been proposed for the use in the context of a personal computer in order to take advantage of multicore processors incorporated nowadays in personal computers, as well as the use of computing power and massively parallel floating point offered in graphics cards.

This work is part of a long-term project started in ([Martínez [2013]]), which pursues the creation of a free software tool that allows the combination of different research and development conducted in *QFT* over time. As it is free software, it will provide researches and users with the opportunity to have a tool that can be adapted to their needs and expanded with further discoveries.

The actions have focused firstly, on a theoretical study of the implemented algorithms, determining the complexity in the various possible cases. In addition, the sequential code has been revised in order to improve it and reduce the number of operations executed. Thirdly there has been a parallel implementation of the algorithms both in *CPU*, using *OpeMP*, and in *GPU*, using *CUDA*. Finally, an experimental has been carried out study on the performance of different implementations of the algorithms developed. There have been compared the execution times of the sequential algorithms with the parallel version in the *CPU*, and the implemented version in *GPU* for different illustrative examples.

Agradecimientos

En primer lugar quiero agradecer al profesor *Joaquín Cervera López* por el tiempo que le ha dedicado a este proyecto. Han sido muchos meses de duro trabajo en los cuales siempre ha estado disponible para resolver mis dudas incluso en periodo de vacaciones. También agradecerle los 4 años que hemos trabajado juntos donde he podido descubrir mi vocación por la informática industrial y por el control. Y por supuesto agradecerle enormemente que aceptara ser mi tutor de tesis doctoral para los próximos años.

En segundo lugar agradecer al profesor *Javier Cuenca Muñoz* por atenderme en su despacho siempre de buena gana para resolver mis dudas sobre *CUDA*. Así como, agradecer a otros profesores que me han ayudado en momentos puntuales cuando lo he necesitado.

Una vez acabados los agradecimientos "académicos", uno empieza a pensar en todas aquellas personas que le han ayudado directa e indirectamente en la conclusión de este proyecto, personas que me han animado y soportado durante este tiempo. Mis compañeros de piso *Pedro*, *Cati*, *José* y *Mari* los cuales venían a rescatarme con sus risas cuando necesitaba huir de las líneas de código del proyecto.

Por supuesto agradecer a mi familia, a mi hermana *Irene* y a mis padres, *Fini* y *José Rafael*, que junto con mis amigos siempre me dieron ánimo y mostraron interés por mi proyecto, aunque la mayoría de ellos se arrepintieran de preguntarme "¿De qué va tu proyecto?", siempre intentaron aparentar que "algo" habían entendido.

Agradecer a las chicas de la Junta Directiva del *CEUM* el año que hemos pasado trabajando juntos, ellas me han demostrado que a la Universidad no se viene solo a estudiar, si no que poniendo cada uno de nosotros un granito de arena podemos construir un futuro mejor.

Y para terminar un agradecimiento muy especial para *Elena* y *Sofía*, las dos personas que más han tenido que aguantar mis nervios a lo largo de este curso, pero siempre han estado ahí para darme todo su apoyo y ánimo, una parte de este TFM os corresponde.

Gracias a todos

Isaac Martínez Forte

Notaciones

Notaciones usadas en QFT

- k_{hf}: ganancia de alta frecuencia un lazo $L(s)$,

$$k_{hf} = \lim_{s \to \infty} s^{n_{pe}} L(s) \tag{1}$$

  donde n_{pe} es el exceso de polos sobre ceros, es decir, el número de polos menos el número de ceros.

- *Diagrama de Nichols* o *plano de Nichols*: forma de representación del plano complejo, basada en la representación polar de los números complejos. En el eje de abscisas se representa la fase en grados y en el de ordenadas la magnitud en decibelios. Algunos autores lo llaman también plano complejo logarítmico.

- *Círculo-M*: en el plano de Nichols, lugar de los puntos p que cumplen

$$\left| \frac{p}{1+p} \right| = \lambda$$

  para un cierto λ. Se utilizan típicamente para definir fronteras de estabilidad robusta.

- Ω: conjunto de *frecuencias de diseño*, frecuencias a las que, dado un problema de control a resolver con QFT, se calcularán las fronteras dadas por las especificaciones y se comprobará la no violación de dichas fronteras.

- *Función de transferencia*: Modelo matemático que a través de un cociente de polinomios en la variable compleja s relaciona la respuesta de un sistema con una señal de entrada o excitación. En la teoría de control, a menudo se usan las funciones de transferencia para caracterizar las relaciones de entrada y salida de componentes o de sistemas que se describen mediante ecuaciones diferenciales lineales e invariantes en el tiempo.

- *Diagrama de Bode*: Es una representación gráfica que sirve para caracterizar la respuesta en frecuencia de un sistema. Normalmente consta de dos gráficas separadas, una que corresponde con la magnitud de dicha función en decibelios y otra que corresponde con la fase, los ejes de ambas frecuencias son logarítmicos.

Otras notaciones usadas

- *GUI*: Interfaz gráfica de usuario, acrónimo en inglés.

- *IDE*: Entorno de desarrollo integrado, acrónimo en inglés.

- $T_m(n)$*: Tiempo de ejecución de un algoritmo en el mejor caso.*

- $T_M(n)$*: Tiempo de ejecución de un algoritmo en el peor caso.*

Índice

Índice de figuras

1 Introducción

El proyecto planteado en este trabajo es continuación directa del realizado el curso pasado como proyecto final de carrera. En aquel trabajo se realizó la implementación de una serie de algoritmos enmarcados en la técnica de control robusto *QFT* (1.2). Se realizó la implementación de 3 algoritmos los cuales pueden llegar a tener una carga computacional alta, dependiendo de la entrada aportada por el usuario. El proyecto del curso pasado consistió en una implementación funcional de los algoritmos sin estudiar en profundidad el rendimiento de los mismos, cuestión que se ha abordado en este trabajo fin de máster. Por tanto, el objetivo de este proyecto es estudiar los algoritmos ya implementados desde el punto del vista del rendimiento y aplicar las soluciones más adecuadas para conseguir cumplir dicho objetivo.

Trabajar con la técnica de control *QFT* requiere de la interactividad del usuario con el software que esté utilizando, por ello un tiempo de ejecución alto dificulta que el usuario pueda trabajar de forma adecuada con *QFT*, por tanto es primordial disponer de algoritmos rápidos que permitan al usuario navegar de forma interactiva por las distintas fases de *QFT* tanto hacia adelante como hacia atrás.***

1.1 Motivación

La motivación por la cual se inició este proyecto el curso pasado era muy clara, el estado actual de la disciplina de *QFT* (*Quantitative feedback theory*) es que está en constante investigación, se están desarrollando nuevos y mejores algoritmos para las fases principales del proceso de diseño *QFT*, que serán descritas en la sección (1.2). De hecho para la fase *ajuste del lazo* no se puede decir que su informatización haya sido completada ni para los tipos de problemas más sencillos. Con lo cual es una disciplina en constante evolución y cambio, esta evolución se ha dado de manera heterogénea, con algoritmos individuales para cuestiones muy concretas desarrollados por diferentes investigadores. *QFT* es tan amplio que cada nueva investigación realiza avances en un parte específica del proceso completo, con la desventaja que conlleva para la unificación de todos estos resultados.

El grupo de investigación de Informática Industrial de la Universidad de Murcia, tenía la necesidad de unificar los esfuerzos investigadores realizados durante años, en un software orientado a investigación y futuro desarrollo. Pero que fuera construido de manera que su uso fuera asequible para personas no expertas, de modo que conjugara las dos vertientes (investigación y uso sencillo).

La idea de aglutinar los distintos proyectos e investigaciones llevadas a cabo en el pasado junto con construir el sistema pensando en su ampliación futura, son dos de los objetivos principales que iniciaron el proyecto, el tercero de los objetivos se centra en el rendimiento del sistema.

Durante el curso pasado, y como proyecto fin de carrera se realizó la implementación de las tres primeras fases de *QFT*, los algoritmos principales de esas fases fueron implementados con el foco de atención en la funcionalidad y sin apenas, al ser una primera fase

de desarrollo, en el rendimiento. Por tanto, este trabajo fin de máster fija su objetivo en analizar los algoritmos ya implementados y mejorar el rendimiento que ofrecen, así como, ampliarlos donde fuera necesario para terminar de completar las tres primeras fases de *QFT*.

Para mejorar el rendimiento de los algoritmos se han seguidos tres caminos principales de desarrollo, por un lado, se hecho una revisión completa del código de los algoritmos para estudiar detenidamente que posibilidades de mejora ofrecía el código implementado el curso pasado. El segundo camino estudiado ha sido la paralelización del código en la CPU utilizando para ello las librerías *OpenMP* con el objetivo de aprovechar la posibilidades de paralelización que ofrecen los procesadores actuales. Y por último, se ha trabajado en la paralelización masiva de los algoritmos utilizando la tarjeta gráfica, utilizando para ello el lenguaje de programación *CUDA* para *GPUs* de *NVIDIA*.

También se estimó importante que el software continuara siendo libre, para que en el futuro pudiera ser ampliamente utilizado y desarrollado más allá del contexto del grupo de investigación Informática Industrial y de los alumnos que colaboren con él. Es por ello que el software está bajo la licencia GPLv3 (ver Anejo 1: GNU GENERAL PUBLIC LICENSE), para que cualquier interesado pueda modificar el software con la obligación principal de liberar las modificaciones que se realicen, de esta manera se asegura que el futuro del proyecto pasa por ser software libre y sobre todo que se basa en la colaboración de los distintos interesados en usar y mejorar el software.

1.2 Introducción a *QFT*

El objetivo de un sistema de control es conseguir que un sistema dinámico, denominado planta, tenga un comportamiento deseado, dado por unas especificaciones de comportamiento. Esto se consigue diseñando uno o más sistemas dinámicos adicionales, denominados controladores, que se conectan a la planta para modificar su comportamiento. Los sistemas dinámicos pueden controlarse de diferentes formas, siendo las dos principales *lazo abierto* (figura 1) y *lazo cerrado* (figura 2).

En este contexto tenemos que la señal de entrada $u(t)$ y la señal de salida $y(t)$ se entienden como una función en el tiempo con codominio en los reales, $u : t \to \mathbb{R}$, y representa la evolución en el tiempo de una magnitud física continua. Si el sistema es *LTI*[1], pueden ser expresadas en el dominio de la frecuencia (transformadas de Fourier o Laplace de las señales temporales), obteniendo señales tipo $U(j\omega)$ o $U(s)$, respectivamente.

En los sistemas de *lazo abierto* no hay relación de realimentación entre la señal $y(t)$ que se obtiene a la salida, y la señal $u(t)$ que se tiene en la entrada, es decir, no hay información de la salida del sistema en la entrada. Con lo cual, la planta y las perturbaciones que le puedan afectar deben ser conocida a la perfección y no existir incertidumbre sobre ella. En este proyecto se ha trabajado con sistemas *LTI* que se pueden representar mediante su versión en el dominio de Laplace y representar el modelo del sistema mediante una

[1]Lineal e invariante en el tiempo

función de transferencia. La función de transferencia de *lazo abierto*, si aplicamos las reglas básicas de interconexión de sistemas *LTI* en Laplace es la siguiente:

$$L(s) = C(s)P(s) \tag{2}$$

Donde $C(s)$ es el controlador, $P(s)$ es el sistema a controlar y $L(s)$ es el sistema controlador.

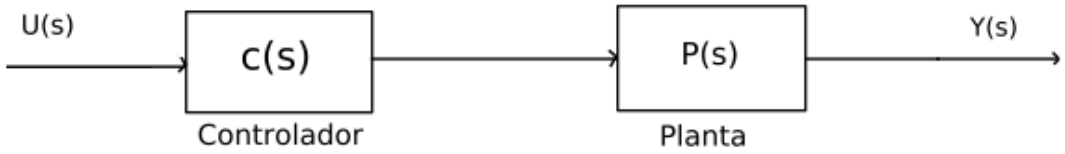

Figura 1: Sistema de control de *lazo abierto*.

En cambio, en un sistema de *lazo cerrado*, la señal $y(t)$ que se obtiene a la salida de un sistema y la señal de entrada $u(t)$ se relacionan entre sí en forma de realimentacion, es decir, se tiene información de la salida que se puede utilizar para modificar la señal de entrada y por tanto la señal de salida $y(t)$ influye en la entrada $u(t)$. La función de transferencia de *lazo cerrado*, si aplicamos las reglas básicas de interconexión de sistemas *LTI* en Laplace es la siguiente:

$$T(s) = \frac{L(s)}{1 + F(s)L(s)} \tag{3}$$

Donde $F(s)$ es la realimentación en el sistema y $T(s)$ es el sistema controlador.

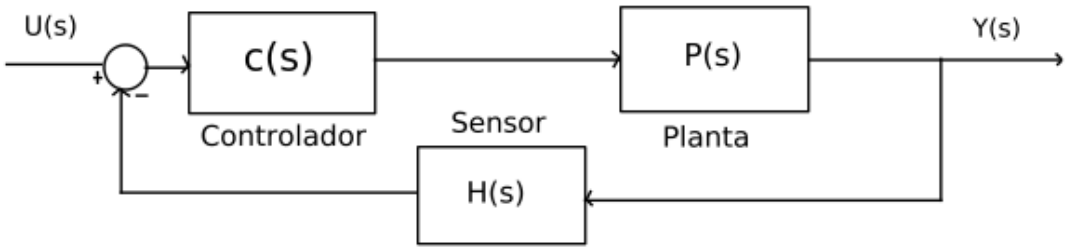

Figura 2: Sistema de control de *lazo cerrado*.

Se habla de control robusto cuando el controlador es diseñado de modo que sea capaz de cumplir las especificaciones a pesar de la incertidumbre y las perturbaciones sobre la planta. *QFT* es una técnica de control robusto en el dominio de la frecuencia que se focaliza en cuantificar la relación entre incertidumbre y esfuerzo de control necesario para

cumplir especificaciones a pesar de esa incertidumbre sobre el sistema de control.

En el año 1945 Bode publicó el libro *Network Analysis and Feedback Amplifier Design* [Bode [1945]]. Donde introdujo los fundamentos de la realimentación, del análisis de sistemas y del diseño de compensadores en el dominio de la frecuencia.

En 1959 Isaac M. Horowitz publicó un artículo titulado *Fundamental theory of automatic linear feedback control systems* [Horowitz [1959]] donde introdujo el concepto de lo cuantitativo a las ideas principales de Bode, introduciendo con ello la idea de que es necesario cuantificar el diseño del controlador respecto a las especificaciones requeridas y la incertidumbre de la planta. Este es el inicio del control cuantitativo en el dominio de la frecuencia. En 1963, Horowitz ampliaría estas ideas en su libro *Synthesis of Feedback Systems* [Horowitz [1963]], aportando novedosas ideas como el control basado en dos grados de libertad, entre otras. Ya en 1972 Horowitz y Sidi publicaron en libro *Synthesis of feedback systems with large plant ignorance for prescribed time-domain tolerances* [Horowitz y Sidi [1972]] donde se acuña el término *QFT*

QFT es un método de ingeniería de control en el cual se usa la realimentación cuantitativa para paliar los efectos de la incertidumbre en la planta. En general, la realimentación no es estrictamente necesaria a menos que exista incertidumbre en el problema de control a resolver. En las técnicas de control robusto se incluye la incertidumbre como forma de asegurar que el sistema de control funcione correctamente para cualquier dinámica del proceso real, siempre que pertenezca al conjunto utilizado para modelar la planta y las perturbaciones, incluyendo la incertidumbre. De este modo, por el que se incorpora a priori un modelo de la incertidumbre al proceso de diseño, se garantiza que el diseño realizado satisface las especificaciones para cualquier planta del conjunto considerado. Obviamente, en la resolución de este problema existen restricciones, especialmente en términos del esfuerzo de control.

En *QFT*, la incertidumbre de la planta se representa mediante plantillas de incertidumbre dependientes de la frecuencia (*templates* en la literatura sobre el tema). Tanto la incertidumbre del modelo de la planta $P(j\omega)$ como sus especificaciones frecuenciales y temporales son trasladadas a un conjunto de curvas $B(j\omega_i)$ (para cada frecuencia de diseño ω_i) en el diagrama de Nichols, llamadas *fronteras*, *bounds* o *boundaries*, que respresentan las restricciones de diseño. El paso principal, denominado *ajuste del lazo* o *loop shaping*, consiste en buscar una función de transferencia nominal de lazo abierto $L_0(j\omega) = C(j\omega)P_0(j\omega)$ para dichas restricciones de modo que se obtenga un controlador lo más simple posible y que satisfaga el conjunto de especificaciones deseadas para cualquier posible planta dentro de la incertidumbre al tiempo que se minimiza una cierta función objetivo, que tradicionalmente se considera minimizar el ancho de banda del controlador y por tanto la amplificación de ruido del sensor. El lazo (y por tanto controlador) que minimiza la función objetivo se denomina lazo (respectivamente controlador) óptimo.

Las principales fases del método son las siguientes:

- Modelado de la planta con su incertidumbre.

- Elección del conjunto de frecuencias de diseño (ω).
- Generación de plantillas (*templates*) y elección de planta nominal $P_0(j\omega)$.
- Definición de especificaciones.
- Generación de fronteras $B(j\omega)$ (*boundaries*).
- Síntesis del controlador $C(j\omega)$.
- Síntesis del pre-filtro $F(j\omega)$ (opcional).
- Validación del diseño.

A continuación se detalla una breve explicación de todas las fases del método, para ejemplificar la explicación se ha usado el ejemplo número uno del *Toolbox de QFT de Matlab* [Borghesani et al. [2003]].

1.2.1 Modelado de la planta con su incertidumbre.

Esta fase es exclusiva de introducción de datos por parte del usuario, en ella el usuario deberá introducir el modelo de la planta que desea controlar, en este caso del ejemplo es el siguiente:

$$P(s) = \frac{k}{(s+a)(s+b)} \tag{4}$$

$k \in [1, 10]$

$a \in [1, 5]$

$b \in [20, 30]$

QFT permite la descripción del modelo dinámico de la planta a controlar desde muy diversas perspectivas, mediante datos de respuesta frecuencial, a partir de datos experimentales, mediante funciones de transferencia lineales, variantes (LTV) o invariantes (LTI) en el tiempo, mediante ecuaciones no lineales y mediante ecuaciones en espacio de estados, todas ellas con incertidumbre no paramétrica y/o paramétrica, la cuál a su vez puede ser, entre otras intervalar, afín y probabilística. En este trabajo se utilizarán funciones de trasnferencia LTI.

1.2.2 Elección de las frecuencias de diseño.

En esta fase se trata simplemente de elegir las frecuencias de diseño que el usuario estime oportunas, suelen ser pocas las frecuencias elegidas. Las formas más habituales de introducir las frecuencias de diseño son las siguientes.

- Espaciado lineal entre un inicio y un fin (rango) con un número de puntos determinado.

- Espaciado logarítmico entre un inicio y un fin (rango) con un número de puntos determinado.
- Frecuencias introducidas manualmente.

En el ejemplo utilizado en esta sección las frecuencias de diseño, introducidas de forma manual, son las siguientes:

$$\omega = 0.1,\ 5,\ 10,\ 100 \text{ rad/s}$$

1.2.3 Cálculo de plantillas

Los modelos de la planta son trasladados al dominio de la frecuencia, representándose en el diagrama de Nichols, para cada frecuencia de diseño un conjunto de puntos (números complejos con magnitud en decibelios y fase en grados). Las plantillas representan la información disponible de la planta. En *QFT*, las plantillas pueden responder a diferentes niveles de estructura en la incertidumbre. El nivel más alto es la incertidumbre paramétrica, siendo usualmente una buena práctica utilizar parámetros que tengan significado físico. El nivel más bajo es la incertidumbre no estructurada, en torno a un modelo nominal. La variaciones respecto del modelo nominal no tienen información de fase, solo de magnitud (típicamente se expresan a partir de una norma). Representaciones más estructuradas se pueden desestructurar, obteniendo resultados más conservadores. Sin embargo, el paso inverso no es posible, debido a la pérdida de información en una representación no estructurada.

En el caso de incertidumbre paramétrica, el problema general de cálculo de las plantillas no está completamente resuelto, salvo por el método de fuerza bruta, de hecho este es un tema de investigación activo en la actualidad. Sin embargo, existen en la literatura ([Bailey y Hui [1989]], [Chen y Ballance [1998]] y [Nordin [1993]]) una serie de técnicas que permiten calcular, de forma eficiente, las plantillas para diferentes niveles de estructura en los parámetros, y que son adecuadas para la gran mayoría de los problemas prácticos.

Respecto al ejemplo usado, en la figura (3) se puede observar el resultados para cada frecuencia de diseño, estos han sido calculados implementando un algoritmo sencillo, propiciado por el conocimiento previo que tenían los desarrolladores del *Toolbox de QFT de Matlab* [Borghesani et al. [2003]] sobre la planta, utilizando varios bucles anidados que calculan directamente el contorno de la nube de puntos.

1.2.4 Definición de especificaciones

En *QFT*, las especificaciones de lazo cerrado se dan en el dominio de la frecuencia, en términos de cotas admisibles sobre la respuesta en frecuencia de las funciones de transferencia de lazo cerrado.

Las restricciones que típicamente se han venido considerando en *QFT* son las siguientes:

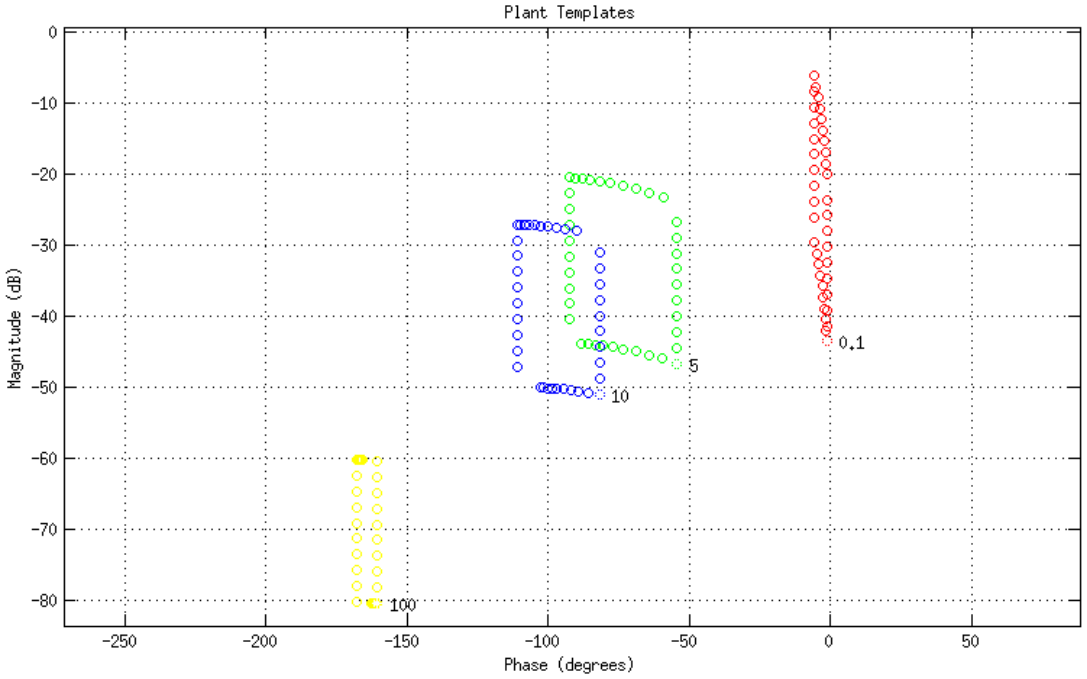

Figura 3: Plantillas de la planta (4).

(1) Seguimiento:

$$\alpha(\omega) \leq \left| \frac{L(j\omega)}{1+L(j\omega)} F(j\omega) \right| \leq \beta(\omega) \ \forall \omega > 0 \tag{5}$$

(2) Estabilidad:

$$\left| \frac{L(j\omega)}{1+L(j\omega)} \right| \leq \lambda \ \forall \omega > 0 \tag{6}$$

(3) Eliminación de ruido del sensor:

$$\left| \frac{L(j\omega)}{1+L(j\omega)} \right| \leq \delta_n(\omega) \ \forall \omega > 0 \tag{7}$$

(4) Eliminación de perturbaciones en la salida de la planta:

$$\left| \frac{1}{1+L(j\omega)} \right| \leq \delta_{po}(\omega) \ \forall \omega > 0 \tag{8}$$

(5) Eliminación de perturbaciones en la entrada de la planta:

$$\left| \frac{P(j\omega)}{1+L(j\omega)} \right| \leq \delta_{pi}(\omega) \ \forall \omega > 0 \tag{9}$$

(6) Esfuerzo de control:

$$\left| \frac{C(j\omega)}{1+L(j\omega)} \right| \leq \delta_{ce}(\omega) \ \forall \omega > 0 \tag{10}$$

Obsérvese que la especificación de estabilidad robusta se da en términos de una limitación sobre la magnitud de lazo cerrado, lo que conlleva una especificación simultánea, y por tanto ligada, sobre el margen de fase y el margen de ganancia. Esta limitación conjunta suele recibir el nombre, en el plano de Nichols, de círculo-M.

1.2.5 Cálculo de fronteras

Las especificaciones se combinan con la incertidumbre del sistema, dada en forma de plantillas (sección (1.2.3)), para obtener restricciones, o fronteras en terminología *QFT*.

Este es uno de los pasos clave en *QFT* en el que se traducen de las especificaciones en el dominio de la frecuencia a regiones en el plano de Nichols donde deberá mantenerse la función $L_0(j\omega)$. Estas regiones, que pueden ser conexas o no, están definidas por sus fronteras. Un cálculo preciso de las fronteras es importante, dado que determinarán el coste de la realimentación en términos de ganancia y ancho de banda del controlador.

El cálculo de fronteras requiere del uso de herramientas informáticas y no es trivial en el caso general, en que pueden ser multivaluadas. Además las fronteras pueden ser abiertas o cerradas. En general, una frontera delimita una región (conexa) del plano de Nichols, y puede expresarse como una función $M = F(\phi)$, donde F es la realimentacion, $F = |1+L|$. Se entiende por frontera multivaluada aquella en la cual la función F es multivaluada. Las fronteras cerradas, típicamente las de estabilidad, son al menos bivaluadas. Las fronteras de seguimiento, que típicamente son abiertas, pueden ser multivaluada en muchos casos, dichas fronteras representadas en el *plano de Nichols* para una fase pueden tener más de una magnitud. En la figura (4) se puede observar una frontera multivaludada con tres valores para una misma fase.

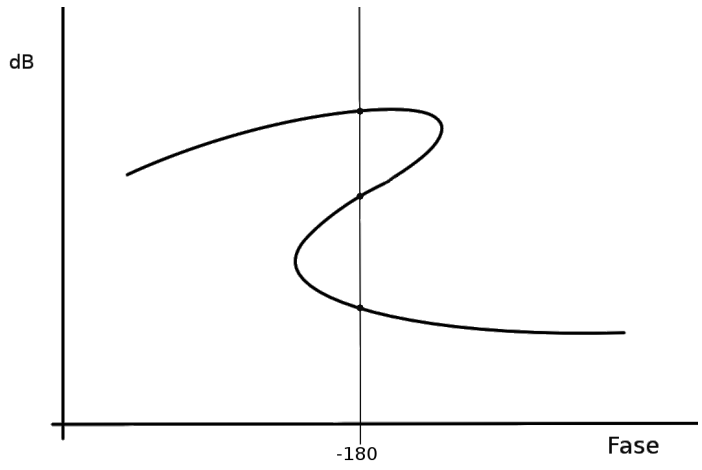

Figura 4: Explicación frontera multivaludada.

Una vez calculadas las fronteras y dibujadas sobre el diagrama de Nichols, para cada especificación y cada frecuencia de diseño (figura 5), son agrupadas de modo que finalmente exista una sola curva para cada frecuencia (figura 6), la fusión de las restricciones impuestas para cada frontera.

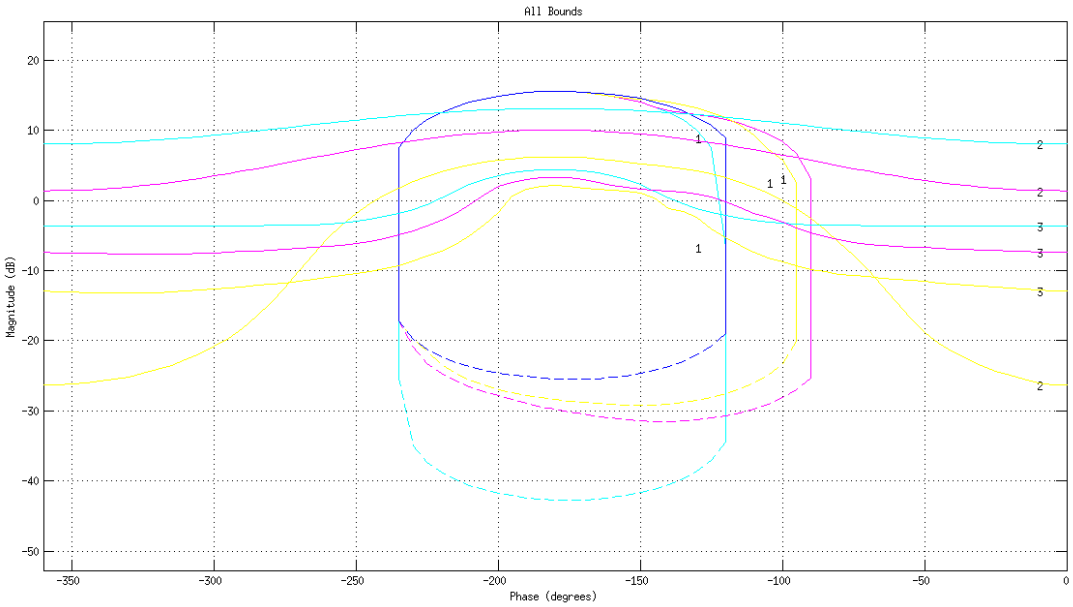

Figura 5: Fronteras sin agrupar para la planta (4).

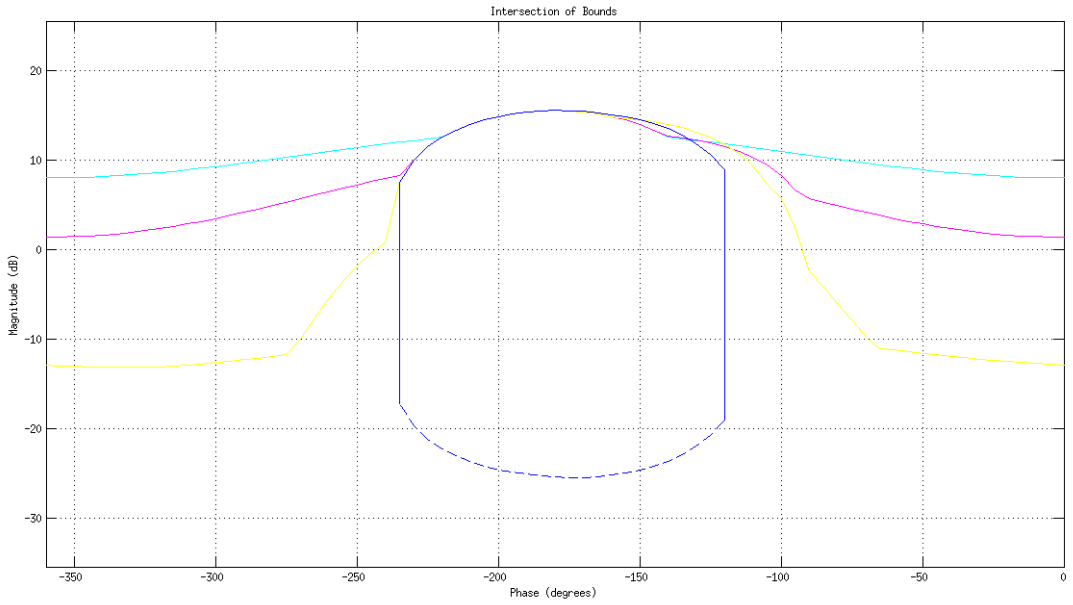

Figura 6: Fronteras agrupados para la planta (4).

1.2.6 Síntesis del controlador

Sin duda, la fase más difícil en *QFT*, y la que plantea más cuestiones abiertas, es el diseño de la función de ganancia del lazo nominal $L_0 = C(s)P_0(s)$. La síntesis consiste en el moldeo de la función $L_0(j\omega)$ en el *plano de Nichols*, de forma que se cumplan las especificaciones de diseño, lo que se traduce en que para cada frecuencia de diseño, el número complejo $L_0(j\omega)$ debe de estar fuera de la región prohibida definida por las fronteras calculadas previamente.

Una vez dibujados las fronteras en el *diagrama de Nichols*, y superpuesta la función $L_0 = P_0C$ ($C = 1$, inicialmente), el controlador $C(j\omega)$ se diseña añadiendo polos y ceros al mismo hasta que la función L_0 se encuentre fuera de las regiones prohibidas. El controlador óptimo (según la función objetivo clásica en QFT consistente en reducir al mínimo la ganancia de alta frecuencia) será, según demostraron *Gera y Horowitz**** [Gera y Horowitz [1980]], aquel que teniendo la mínima ganancia consiga que L_0 descanse sobre los contornos a cada frecuencia de interés.

Si nos centramos en el ejemplo que se ha usado a lo largo de esta sección, se puede observar el seguimiento de L_0, que para el controlador ejemplo (11) se obtiene la figura (7) como resultado.

$$C_2(s) = \frac{379(\frac{s}{42} + 1)}{\frac{s^2}{272^2} + \frac{s}{247} + 1} \tag{11}$$

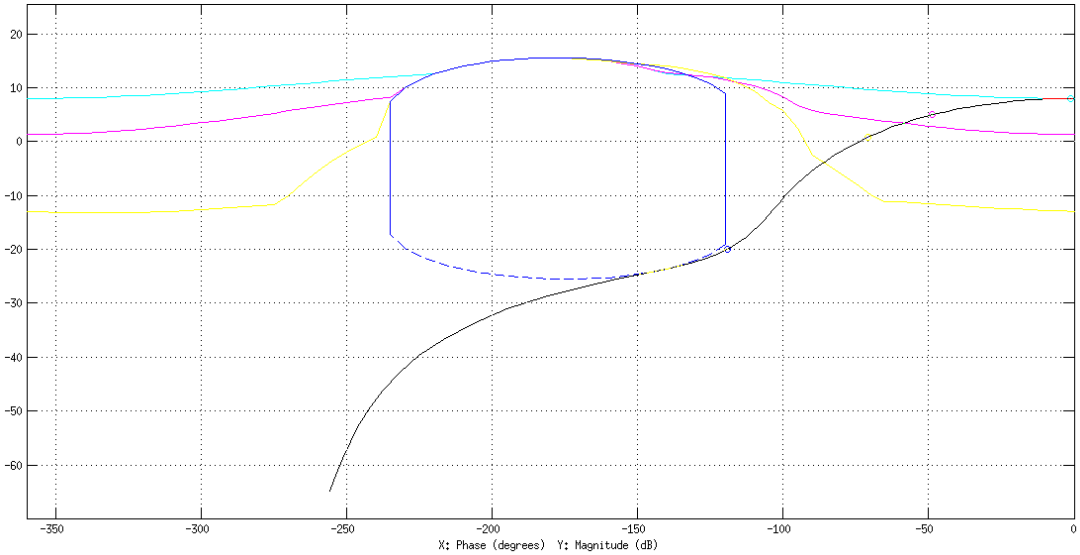

Figura 7: Gráfica de síntesis del controlador basándose en la ecuación (11).

1.2.7 Síntesis del pre-filtro (opcional)

En algunos sistemas de control se requieren especificaciones de seguimiento de referencia (*tracking*). En estos sistemas será necesario también el diseño de un pre-filtro $F(j\omega)$. Así, mientras $C(j\omega)$ se encarga de reducir la incertidumbre y alcanzar los objetivos señalados anteriormente, el objetivo del pre-filtro es el cumplimiento de las especificaciones de tracking junto con el controlador.

1.2.8 Validación del diseño

Una vez que se ha finalizado el diseño de $C(j\omega)$ (y también de $F(j\omega)$, si procede) es necesario analizar su comportamiento frente a las especificaciones deseadas, tanto en el

dominio de la frecuencia como en el dominio del tiempo. No se ha conseguido expresar de forma matemática una relación directa entre el cumplimiento de las especificaciones en el dominio de la frecuencia y el cumplimiento de las especificaciones en el dominio del tiempo. Sin embargo, pocas veces aparece la paradoja de cumplirse las especificaciones frecuenciales asociadas y no las temporales. Por lo tanto, el principal objetivo de esta fase es validar el diseño realizado en un grupo amplio de especificaciones frecuenciales y de tiempo, por supuesto incluyendo los más desfavorables de la incertidumbre de la planta.*** Mirar corrección

1.3 Introducción al Paralelismo

El paralelismo es una forma de computación en la cual varios cálculos pueden realizarse simultáneamente gracias al uso de más de una unidad hardware de cálculo, basado en el principio de dividir los problemas grandes para obtener varios problemas pequeños, que son posteriormente solucionados en paralelo. El paralelismo ha sido empleado durante muchos años, sobre todo para la Computación de Alto Rendimiento.

El aumento de la frecuencia de reloj fue la razón dominante de las mejoras en el rendimiento de las computadoras desde mediados de 1980 hasta el año 2004. El tiempo de ejecución de un programa es igual al número de instrucciones multiplicado por el tiempo promedio por instrucción. Manteniendo todo lo demás constante, el aumento de la frecuencia de reloj reduce el tiempo medio que tarda en ejecutarse una instrucción, por tanto un aumento en la frecuencia reduce el tiempo de ejecución de los programas.

Sin embargo, el consumo de energía de un chip está dado por la ecuación $P = C \cdot V^2 \cdot F$, donde P es la potencia eléctrica, C es el cambio de capacitancia por ciclo de reloj —proporcional al número de transistores cuyas entradas cambian—, V es la tensión, y F es la frecuencia del procesador (ciclos por segundo). Un aumento en la frecuencia aumenta la cantidad de energía utilizada en un procesador. El aumento del consumo de energía del procesador llevó a *Intel* en mayo del 2004 a la cancelación de sus procesadores *Tejas* y *Jayhawk*, este hecho generalmente se cita como el fin del escalado de frecuencia como el paradigma dominante de arquitectura de computadores.

A partir de 2004 se empezó a implantar en los ordenadores personales los procesadores multinúcleo, siento muy habituales este tipo de procesadores hoy en día, que suelen tener entre 2 y 8 núcleos. Este hecho abre las puertas a la computación paralela en ordenadores personales, lo cual es un fenómeno interesante y desconocido hasta la fecha.

Existen diferentes tipos diferentes de paralelismo: nivel de bit, nivel de instrucción, de datos y de tarea. A continuación se explican cada uno de ellos:

- **El paralelismo a nivel de bit** es una forma de computación paralela basada en el aumento de tamaño de procesador de palabra, dicho aumento reduce el número de instrucciones que el procesador debe ejecutar con el fin de realizar una operación con variables cuyos tamaños son mayores que la longitud de la palabra.
- **El paralelismo a nivel de instrucción** es una forma de computación paralela basada en aumentar el número de instrucciones que un procesador ejecuta en paralelo.

Duplicando las unidades funcionales hardware un procesador es capaz de ejecutar varias instrucciones en paralelo, teniendo que resolver para ello diferentes problemas que surgen, como pueden ser la dependencia entre instrucciones o las operaciones de salto.

- **El paralelismo a nivel de datos** un paradigma de la programación concurrente que consiste en subdividir el conjunto de datos de entrada, de manera que a cada procesador le corresponda un subconjunto de esos datos. Cada procesador efectuará la misma secuencia de operaciones que los otros procesadores sobre su subconjunto de datos asignado. Este tipo de paralelismo es también muy adecuado para la utilización de la tarjeta gráfica, dado que al ser un hardware de propósito específico tiene una serie de limitaciones, la principal de ellas es que todos los procesadores deben ejecutar las mismas instrucciones a la vez. Este nivel de paralelismo es el aplicado en este trabajo fin de máster.

- **El paralelismo a nivel de tarea** es un paradigma de la programación concurrente que consiste en asignar distintas tareas a cada uno de los procesadores de un sistema de cómputo. En consecuencia, cada procesador efectuará su propia secuencia de operaciones.

La tarjeta gráfica es un coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante, para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos y o aplicaciones 3D interactivas. Estas capacidades de cálculo en coma flotante se pueden aprovechar para realizar algoritmos de carácter generalista que sean intensivos en cálculo, o incluso dividir la carga de trabajo entre la *GPU* y la *CPU*. La arquitectura de una *GPU* está pensada para ejecutarse de forma masivamente paralela, para ello dispone de una gran cantidad de procesadores de poca potencia que actúan en paralelo ejecutando todos ellos la misma instrucción.

2 Estado del arte

El estado del arte de este proyecto consta de dos partes fundamentales, la primera de ellas tiene como motivación el objetivo marcado a largo plazo cuando se inició este proyecto el curso pasado. Dicho objetivo es crear un software abierto que permita a la comunidad científica poder adaptarlo a nuevas investigaciones, así como aportar nuevas características para ser compartidas como software libre. En el marco de ese proyecto a largo plazo se estudió qué paquetes software existentes sobre *QFT* cumplían una finalidad similar, obviamente, este objetivo planteado es a largo plazo.

La segunda parte del estado del arte se centra en el proyecto realizado de este curso, el objetivo ha sido continuar con la mejora del prototipo iniciado el curso pasado, estudiando la paralelización de los algoritmos ya implementados utilizando *OpenMP* y *CUDA*. Por tanto, el estado del arte ha sido estudiar qué aportaciones existentes hasta la fecha tratan la temática de la paralelización de algoritmos de *QFT*.

En primer lugar, para poner en contexto, se hace una introducción al prototipo implementado el curso pasado ([Martínez [2013]]). El proyecto del año pasado se planteo como un primer prototipo inicial, en ese marco se implementaron las tres primeras fases de *QFT* consistentes en la fase de introducción de datos por parte del usuario, tanto planta como frecuencias de diseño. La fase de cálculo de plantillas, con el desarrollo de un algoritmo por fuerza bruta y el algoritmo *e-hull* para el cálculo del contorno de dichas plantillas. Y para terminar se trabajo en la fase de cálculo de fronteras, donde se implemento un algoritmo que soporta fronteras multivaluadas para la especificación de estabilidad.

También se implementó las interfaces gráficas necesarias para las fases de *QFT* estudiadas, todas ellas utilizado las librerías *Qt*[2] sobre *C++*. Con la vista puesta en la ampliación del prototipo hacia un sistema más grande, se implementó un sistema de almacenamiento basado en el patrón *DAO*[3] con el fin de poder modificar el tipo de almacenamiento en el futuro sin necesidad de reprogramar las clases afectadas. Al ser un programa orientado a la investigación es fundamental poder guardar los datos calculados para que puedan ser usados en el futuro, por ello se implemento un sistema de guardado en ficheros utilizando *XML*[4] que permite guardar el estado del sistema y recuperarlo en un momento posterior. Para favorecer la compatibilidad con otros sistemas se implemento el exportado de datos a texto plano, de esta manera los datos pueden ser utilizados en otros software similares, como por ejemplo *Matlab*. Para terminar, y pensado en su ampliación futura se escribió documentación basada en *Doxygen*[5] de todas las funciones públicas del sistema, facilitando así la ampliación y modificación del prototipo por terceros.

A continuación de estudian los software existentes hasta la fecha que ofrecen herramientas para el desarrollo informatizado de *QFT*:

- **QFT toolbox** [Borghesani et al. [2003]]: Se considera el toolbox clásico de

[2]Página oficial http://qt-project.org/ Último acceso (20/07/2014)
[3]Explicación http://es.wikipedia.org/wiki/Data_Access_Object Último acceso (20/07/2014)
[4]Página oficial http://www.w3.org/XML/ Último acceso (20/07/2014)
[5]Página oficial http://www.stack.nl/~dimitri/doxygen/ Último acceso (20/07/2014)

QFT, fue distribuido durante algunos años por *The MathWorks Inc.*® como un toolbox de *Matlab*[6]. Posteriormente, en el año 2000, fue distribuido por una empresa independiente, *Terasoft Inc.*®[7], también como toolbox de Matlab. Finalmente, en 2013, fue puesto a libre disposición para descarga[8], si bien no fue publicado como software libre. Los ficheros fuente están disponibles, de modo que es posible inspeccionar el funcionamiento de los algoritmos.

- **Qsyn** [Gutman [1996a]] [Gutman [1996b]]: También distribuido como toolbox de *QFT*, pero de forma independiente. Es software privativo y no está disponible para descarga gratuita. Introduce importantes mejoras en relación al *QFT toolbox*, como son las herramientas para cálculo de plantillas o la posibilidad de almacenar datos calculados (plantillas, fronteras, etc.) en ficheros. Está fuertemente basado en la introducción de la información en formato texto.

- **SISO-QFTIT** [Dormido et al. [2004]]: Una herramienta basada en el entorno de programación interactiva *Sysquake*[9], de la empresa *Calerga*®, que pone énfasis en la interactividad de cara a su uso como herramienta docente. Para aumentar la velocidad de respuesta de la herramienta, de cara a su propósito académico, se hacen algunas simplificaciones (conservativas) durante el proceso de diseño. La herramienta está disponible para descarga gratuita[10], pero no es libre.

- **QFTCT** [Garcia-Sanz et al. [2009]]: Disponible para descarga gratuita[11] como toolbox de *Matlab*, si bien ni es software libre ni los ficheros fuente están disponibles. Añade, en relación al *QFT* toolbox original, características de interactividad muy interesantes, como es el hecho de tener diferentes informaciones gráficas a la vista de forma simultánea. También es interesante la adición de la introducción interactiva de plantas con diferentes formatos para la incertidumbre, incluyendo distribuciones de probabilidad.

Si nos centramos en la segunda parte del estado del arte, la relacionada con la paralelización de algoritmos relacionados con *QFT* tanto en *CPU* como en *GPU*. Se ha realizado una búsqueda por Internet, tanto en artículos científicos (*Google scholar*) como una búsqueda generalista (*Google*), utilizando las palabras clave siguientes:

- "qft parallel"
- "qft parallel agorithm"
- "qft algorithm cuda"
- "qft cuda"
- "qft optimization"

[6]Página oficial: http://www.mathworks.es/ Último acceso(8/07/2014)
[7]Página oficial: http://www.terasoft.com/ Último acceso(8/07/2014)
[8]Página descarga: http://pcsl.ecs.umass.edu/cbs/software.html Último acceso(8/07/2014)
[9]Página oficial: http://www.calerga.com/products/Sysquake/ Último acceso(8/07/2014)
[10]Página oficial: http://ctb.dia.uned.es/asig/qftit/principal.html Último acceso(8/07/2014)
[11]Página oficial: http://cesc.case.edu/OurQFTCT.htm Último acceso(8/07/2014)

- “qft OpenMP”

No encontrando información relacionada que pudiera aportar datos de otros investigadores que hayan estudiado el tema con anterioridad. Por tanto se puede determinar que no existen trabajos relacionados con la implementación paralela de algoritmos de QFT que tenga la suficiente repercusión como para aparecer en los buscadores, tanto académicos como generalistas.

3 Análisis de objetivos y metodología

3.1 Objetivo

El objeto principal de este trabajo fin de máster es el estudio, mejora, ampliación y paralelización de los algoritmos ya implementados en el prototipo durante el curso pasado. Los algoritmos implicados en el desarrollo de este trabajo son tres, que corresponden con dos de las fases de *QFT* (1.2), cálculo de plantillas y cálculo de fronteras. En esta sección se hace un estudio detallado del objetivo que se persigue conseguir, cual es la motivación para plantear dicho objetivo y que beneficios traerá conseguirlo.

En la disciplina de *QFT* tradicionalmente se ha basado en la prueba y error, las distintas fases que componen la técnica se desarrollan de la forma en la cual el investigador cree que va a dar los resultados adecuados, introduciendo para ello los parámetros de configuración de los distintos algoritmos. Es la última de las fases, en la validación del diseño, donde se comprueba si el resultado obtenido es el adecuado, si no lo es, se vuelve a fases anteriores y se repite el proceso modificando los parámetros de entrada.

Por ejemplo, el usuario introduce una especificaciones que cree que son las adecuadas para conseguir el objetivo que se plantea, después de realizar diversas pruebas se da cuenta de que las especificaciones elegidas son demasiado restrictivas en algunos campos y tiene que volver a fases anteriores para cambiarlas, ello implicaría recalcular todos los datos. Otro ejemplo pueden ser los parámetros que configuran la ejecución de cada uno de los algoritmos, como se elijan estos parámetros afecta al resultado final. Un ejemplo de esto sería el parámetro *epsilon* de algoritmo *e-hull* que determina cual es la distancia base entre los puntos de contorno. Cuando más pequeña sea esta distancia más puntos tendrá dicho contorno, la necesidad de que el contorno sea más detallado o menos depende del caso concreto con el que se esté trabajando. Una elección de *epsilon* que no sea adecuada puede provocar que se tenga que realizar la ejecución del algoritmo variando el valor de *epsilon*.

En conclusión, se debe partir de la idea de que *QFT* no se aplica de forma lineal, con un principio, un fin, y un recorrido claro entre las fases de diseño. Tiene muchas vueltas atrás, pruebas, decisiones empíricas que pueden no ser las adecuadas, etc. Tal que disponer de un software que facilite la interactividad con el usuario sería muy adecuado para trabajar con *QFT*. En la figura (8) se puede observar el diagrama de actividad[12] del sistema hasta el cálculo de fronteras, en dicho diagrama se observan las interacciones que pueden resultar entre el usuario y el programa.

Esta forma de realizar el proceso es muy costosa actualmente, dado que los paquetes de software disponibles tienen unos tiempos de ejecución de los algoritmos altos con respecto a la interactividad. Por ejemplo, la demo número 1 del *toolbox* de *QFT* en *MATLAB* para un ejemplo sencillo el cálculo de plantillas tarda 6 segundos y el cálculo de fronteras tarda 9 segundos.

[12]Página con información http://es.wikipedia.org/wiki/Diagrama_de_flujo Último acceso (22/07/2014)

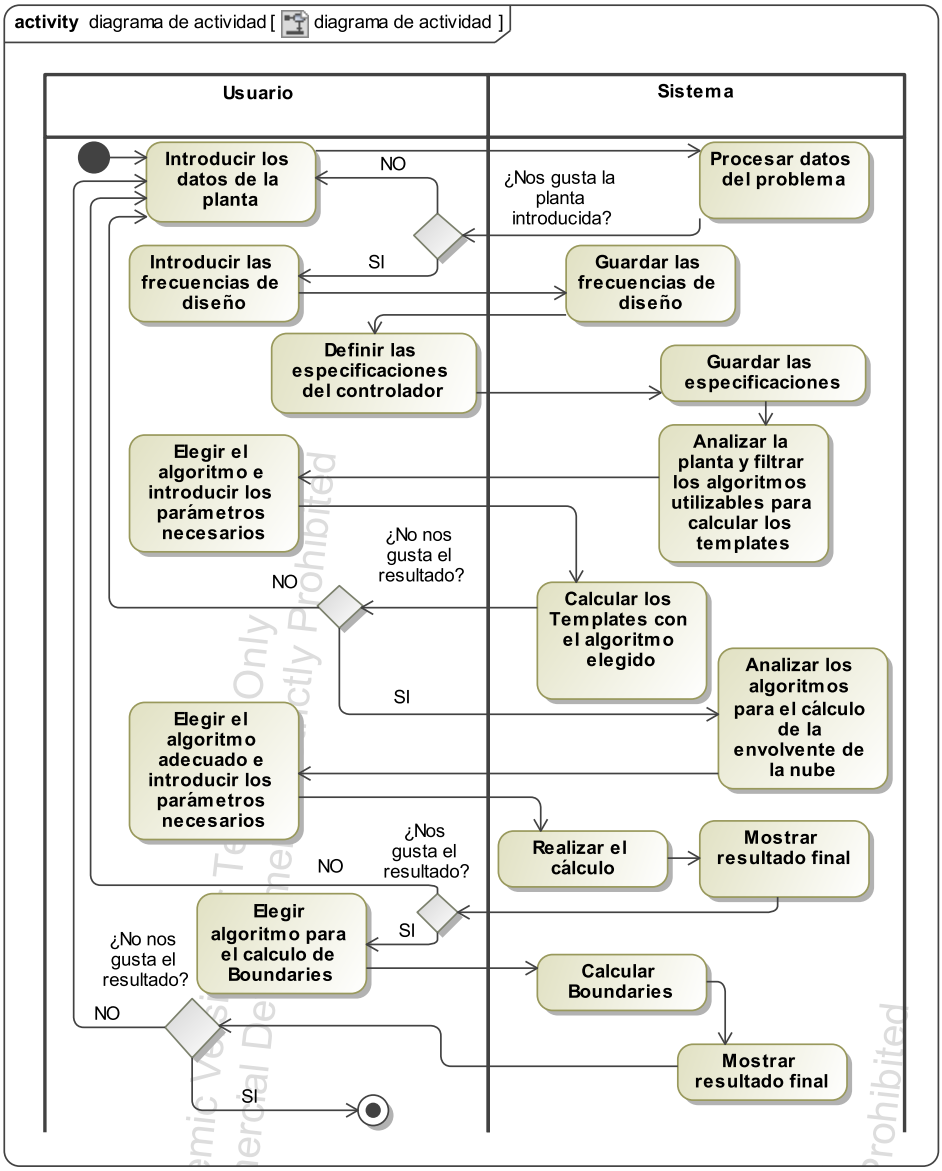

Figura 8: Diagrama de actividad del sistema.

Dado que lo normal es necesitar ejecutar cada uno del os algoritmos repetidas veces hasta obtener el resultado adecuado. Además, al ser una técnica en la cual las fases de diseño se aplican en orden, dependiendo las fases posteriores de lo calculado en las fases predecesoras, si se cambia un parámetro en una fase es necesario volver a ejecutar todos los algoritmos posteriores y por tanto, sumar los tiempos de ejecución de todos los algoritmos.

El objetivo que se persigue conseguir a largo plazo es conseguir construir un software que proporcione la interactividad suficiente al usuario para poder trabajar con *QFT* sin necesidad de tener que esperar la ejecución de los algoritmos, es decir, que cambiar los parámetros de entrada en cierta fase y recalcular todos los algoritmos posteriores de forma imperceptible en tiempo de ejecución para el usuario.

3.2 metodología

Una vez estudiados los objetivos, se plantea una metodología para poder conseguirlos, para cada uno de los algoritmos seleccionados se hace un estudio de que herramientas se pueden aplicar sobre ellos para conseguir la finalidad planteada:

- En primer lugar se estudiará el algoritmo de cálculo de plantillas por fuerza bruta, sobre dicho algoritmo se plantea una revisión general del código para reducir el número de operaciones que realiza, así como estudiar la paralelización del código con *OpenMP*.
- En segundo lugar se estudiará el algoritmo *e-hull* para el cálculo del contorno de las plantillas. Para este algoritmo en primer lugar se plantea una revisión general a su código, optimizando la versión secuencial del código para que realice menos operaciones. A continuación se implementarán dos versiones, la primera de ellas una versión paralela con *OpenMP* para la *CPU*, y una segunda versión implementada con *CUDA* para su ejecución en la *GPU*.
- El último de los algoritmos que se plantea revisar corresponde al cálculo de fronteras. La primera acción sobre este algoritmo, al igual que el anterior, será una revisión y mejora del código, estudiando que optimizaciones se pueden aplicar sobre el algoritmo para reducir el número de operaciones ejecutadas. A continuación se paralelizará el código con *OpenMP*, siendo este más complicado que el anterior. Una vez realizado este trabajo, se plantea amplir la funcionalidad ofrecida por este algoritmo, el cual solo tiene implementada la especificación de estabilidad robusta (6), siendo ampliada esta funcionalidad a todas las especificaciones recogidas en las ecuaciones de la (5) a la (10) completando con ello las especificaciones clásicas de *QFT*. Para terminar se programará una versión del algoritmo con *CUDA* para ejecutarse en la *GPU*.

Estas son las acciones que se proponen para conseguir el objetivo marcado, para llevar a cabo estas acciones son necesarias una serie de herramientas que se explica a continuación, en la sección (3.3).

3.3 Herramientas utilizadas

Las acciones planteadas para realizar en el marco de este trabajo fin de máster se centran en el estudio y paralelización de los algoritmos pertenecientes a las 3 primeras fases de *QFT*. Para ello se plantea estudiar como acelerar los algoritmos desde distintos puntos de vista.

En primer lugar, antes de paralelizar, se estudió el código implementado para cada alguno de los algoritmos para optimizarlo lo máximo posible en lo que a instrucciones se refiere, actuaciones como reutilizar operaciones matemáticas, eliminar comparaciones innecesarias y reformar bucles para que funcionen de forma más optimizada.

La segunda de las actuaciones fue estudiar las posibles formas de paralelización de los algoritmos en la *CPU*, en este campo hay dos posibilidades principales, paralelización *fork-join* con memoria compartida o paralelización de procesos independientes con paso

de mensajes.

La segunda de las opciones, paso de mensajes, se descartó dado que el prototipo está pensado para su utilización a nivel de usuario en ordenadores personales, donde el paso de mensajes no tiene sentido que se use, dado que en los ordenadores personales los distintos cores de un procesador comparten la memoria y por tanto no es necesario una comunicación vía paso de mensaje entre ellos. Por ello se decidió utilizar la librería *OpenMP* para la paralelización de los algoritmos. En la sección (4.2.1) se hace una explicación detallada de esta librería, así como, la justificación de esta elección frente a otras de características similares

La tercera y última de las actuaciones realizadas es aprovechar la potencia de cálculo en punto flotante y masivamente paralelo que disponen las tarjetas gráficas que son comunes en los ordenadores personales. Para ello se utilizó el lenguaje de programación de tarjetas gráficas de *NVIDIA*, *CUDA*, para portar a este lenguaje dos de los algoritmos implementados. La decisión de utilizar este lenguaje, así como una explicación detallada del funcionamiento de una tarjeta gráfica de *NVIDIA* junto con las opciones de configuración que ofrece el lenguaje *CUDA* se puede consultar en la sección (4.1.2 y 4.2.2).

4 Diseño y resolución del trabajo realizado

Durante el curso pasado de implementaron tres algoritmos en el marco de las tres primeras fases de *QFT*, los algoritmos fueron implementados para que fueran funcionales sin estudiar en mucha profundidad el rendimiento de los mismos. El planteamiento para el trabajo de este año, tal y como se explica en la sección de objetivos (3.1), es conseguir el rendimiento suficiente de los algoritmos para poder ejecutar los algoritmos en cadena si el usuario cambia los parámetros de entrada. Con ese objetivo como finalidad se estudiaron los tres algoritmos en profundidad para ver que mejoras se le podían aplicar a cada uno de ellos.

4.1 Elección de herramientas

En esta sección se van a estudiar las herramientas que se han utilizado para la realización del proyecto, como se ha introducido la sección anterior (3.3) y la justificación de la elección de cada una de ellas.

4.1.1 Elección de la librería para paralelización en *CPU*

Las dos opciones principales para la paralelización en *CPU* es gestionar los hilos de forma manual o utilizar una librería que automatice el proceso.

La primera opción es la que más libertad ofrece al programador, al poder manipular a su antojo los hilos creados y que código ejecutar en cada uno de ellos. Esta gestión manual de los hilos es muy conveniente cuando se desea realizar paralelismo de grano grueso[13] a nivel de tarea, donde se puede llegar incluso a dividir el programa entero en varias partes que se ejecuten por separado. Para conseguir esta finalidad la librería más extendida es POSIX Threads, el estándar POSIX para la creación de hilos.

Por otro lado, como ya se indicó en la introducción (1.3) el paralelismo principal que tenemos en los algoritmos a estudiar es a nivel de datos, es decir, queremos paralelizar el la operaciones sobre los datos, aplicar sobre todos ellos las mismas operaciones dividiendo el trabajo entre los distintos procesadores disponibles.

Para realizar este tipo de paralelismo la librería más extendida es *OpenMP*, de hecho es prácticamente un estándar, tanto en la industria como en la investigación. Como el paralelismo que se va a desarrollar en este trabajo está claro, la librería finalmente seleccionada es *OpenMP*, en la sección (4.2.1) se hace una introducción a dicha tecnología.

4.1.2 Elección del lenguaje de programación para *GPU*

Para programar tarjetas gráficas existen diferentes lenguajes diseñados específicamente para ello, en esta sección se van a estudiar los cuatro principales que se usan en la

[13] Se considera paralelismo de grano grueso si las diferentes tareas deben comunicarse pocas veces por segundo.

actualidad y que mejor soporte tienen. Se dividen en dos tipos de lenguajes, los orientados a programación gráfica y los orientados a programación generalista.

Los dos principales lenguajes del primer grupo son *GLSL* y *HLSL*. Por otro lado, *GLSL* es parte de la API de *OpenGL* pero orientado al sombreado gráfico, como principal factor a favor tiene que está muy extendido y que es software libre mantenido activamente por la comunidad. Por otro lado, el segundo lenguaje es *HLSL*, con la misma finalidad que el anterior pero esté está construido por *Microsoft* y es parte de *Direct3D*[14], es el lenguaje más usado en su ámbito pero con el inconveniente de que no está orientado a la programación general de algoritmos.

Estos lenguajes al ser de propósito gráfico tiene menos cabida en una programación generalista como la que se desea realizar, por ello se analizan a continuación los dos lenguajes principales en este ámbito, que serán entre los cuales se hará la elección definitiva, *CUDA* y *OpenGL*.

OpenCL consta de una interfaz de programación de aplicaciones y de un lenguaje de programación. Juntos permiten crear aplicaciones con paralelismo a nivel de datos y de tareas que pueden ejecutarse tanto en *CPU* como en *GPU*. El lenguaje está basado en *C99*[15], eliminando cierta funcionalidad y extendiéndolo con operaciones vectoriales. Está soportado por los principales fabricantes de tarjetas gráficas y es software libre, facilitando con ello la programación para todo tipo de tarjetas independiente del modelo y del fabricante.

La denominación *CUDA* hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por *NVIDIA* que permiten a los programadores usar una variación del lenguaje de programación *C* para codificar algoritmos en *GPUs* de *NVIDIA*. Este lenguaje no es portable entre diferentes fabricantes de tarjetas gráficas como sí que lo es *OpenCL*.

Una vez analizados los dos lenguajes generalistas la principal diferencia entre ellos es que *CUDA* solo se puede ejecutar en arquitecturas *NVIDIA* mientras que *OpenCL* está soportado por todos los fabricantes de tarjetas. A cambio, *CUDA* permite explotar mejor la arquitectura de la *GPU*, sí se tiene un conocimiento amplio sobre *CUDA* y sobre la tarjeta que se quiere trabajar se puede hacer una estructura lógica y un uso de la jerarquía de memoria que favorezca aplicaciones más rápidas que en *OpenCL* ([Bernabé et al. [2012]]). Según podemos observar en la siguiente tabla para la transformada rápida y 3D *Wavelet*.

Code Version	512 x 512	1K x 1K	2K x 2 K
CPU optimal	156.09	655.33	2843.43
CUDA	29.21 (5.3x)	100.61 (6.5x)	381.58 (7.4x)
OpenCL	87.12 (1.8x)	276.39 (2.4x)	1011.47 (2.8x)

[14] API para la programación de gráficos 3D perteneciente a las librerías *DirectX*.

[15] Explicación del lenguaje de programación gráfica *C99* http://en.wikipedia.org/wiki/C99 (Último acceso 17/07/2014).

En el siguiente documento ([Karimi et al. [2010]]) se estudia la comparación entre *CUDA* y *OpenCL* utilizando el algoritmo *Adiabatic Quantum* (*AQUA*) que es una simulación de *Monte Carlo*, para diferentes tamaños de problema. En este documento se parte de un algoritmo implementado en *CUDA* y se hace una traducción "directa" a *OpenCL*, es decir, se modifica lo mínimo posible el código, cambiando principalmente las palabras reservadas por el compilador. Tal y como se indica en el documento, el algoritmo utilizado en *OpenCL* no sería compatible con el resto de tarjetas gráfica debido a que el código estaría adaptado a las tarjetas *NVIDIA* y habría que hacerlo más generalista para que funcionara en todas. Se explica también, que debido a esta peculiaridad las diferencias de rendimiento entre *CUDA* y *OpenCL* son pocas, si se optara por una implementación generalista que no adaptada al tipo de tarjeta donde se está ejecutando el algoritmo los tiempos de ejecución aumentarían de forma similar a lo indicado en el documento anterior ([Bernabé et al. [2012]]).

Se puede observar el resultado en la siguiente tabla:

Code Version	8	16	32	48	72	96	128
CUDA	1.97	3.87	7.71	13.75	26.04	61.32	101.07
OpenCL	2.24	4.75	9.05	19.89	42.32	72.29	113.95

En estos datos se puede observar como el rendimiento de *CUDA* es superior al de *OpenCL*, con un margen relativamente pequeño pero que para problemas grandes se puede convertir en una diferencia importante.

El último de los documentos estudiados ([Fang et al. [2011]]) se inclina por estudiar lo que denominan una comparación "justa" entre los dos lenguajes de programación. Su argumentación se basa en que para diferentes algoritmos se deben usar diferentes formas de medida distintas del tiempo de ejecución (o tiempo de *GPU*), para cada tipo de los algoritmos que se evalúan en la publicación propone una forma de medición del rendimiento y bajo estos preceptos sale un rendimiento similar entre *CUDA* y *OpenCL*.

Se elije finalmente el lenguaje *CUDA*, dado que para algoritmos concretos y si se explota de forma correcta la arquitectura de la tarjeta gráfica permite un rendimiento en tiempo de ejecución superior a *OpenCL*. Pero no se descarta la opción de utilizar *OpenCL* en vías futuras debido a las ventajas que aporta sobre *CUDA*, principalmente la capacidad de generar código compatible.

4.2 Antecedentes

4.2.1 Introducción a *OpenMP*

OpenMP es una interfaz de programación de aplicaciones (API) para la programación multiproceso de memoria compartida en múltiples plataformas. Permite añadir concurrencia a los programas escritos en *C*, *C++* y *Fortran* sobre la base del modelo de ejecución fork-join[16]. Está disponible en muchas arquitecturas, incluidas las plataformas de *Unix* y de *Microsoft Windows*. Se compone de un conjunto de directivas de compilador, rutinas de

[16]Método basado en la bifurcación de un proceso en diferentes hilos.

biblioteca, y variables de entorno que influyen el comportamiento en tiempo de ejecución.

La memoria compartida es aquel tipo de memoria que puede ser accedida por múltiples programas, ya sea para comunicarse entre ellos o para evitar copias redundantes. La memoria compartida es un modo eficaz de pasar datos entre procesos. Dependiendo del contexto, los programas pueden ejecutarse en un mismo procesador o en procesadores separados. Como *OpenMP* funciona con el sistema fork-join de creación de hilos dentro del mismo programa en ejecución, la compartición de memoria se puede realizar de forma privada sin necesidad de participación del sistema operativo.

OpenMP se basa en directivas para la gestión de los procesos, que tienen la siguiente forma:

```
1 # pragma omp <directiva> [clusula [ , ...] ...]
```

Esta directivas son interpretadas por el compilador para generar el código fuente final, este código se adapta al tipo de procesador en el cual se está ejecutando el código. El número de hilos creados no solo será en función de los datos a procesar, si no también, del número de procesadores del que se disponga.

4.2.2 Introducción a *CUDA*

CUDA son las siglas de *Compute Unified Device Architecture* (Arquitectura Unificada de Dispositivos de Cómputo) que hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por *nVidia* que permiten a los programadores usar una variación del lenguaje de programación *C* para codificar algoritmos en *GPU* de *nVidia*.

CUDA intenta explotar las ventajas de las *GPU* frente a las *CPU* de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten el lanzamiento de un altísimo número de hilos simultáneos. Por ello, si se consigue programar una aplicación de manera que se ejecute de forma masivamente paralela, la utilización de la *GPU* será muy adecuada para la obtención de un rendimiento superior.

CUDA intenta aprovechar el gran paralelismo, y el alto ancho de banda de la memoria en las *GPU* en aplicaciones con un gran coste aritmético, el procesador de propósito específico que contiene una tarjeta gráfica está especialmente diseñado para el cálculo aritmético en punto flotante. En el caso de los algoritmos de esta tesis, sus datos son principalmente números complejos de longitud doble o números reales de longitud doble. Es por ello que la utilización de la *GPU* es muy adecuada en este tipo de algoritmos.

El modelo de programación de *CUDA* está diseñado para que se creen aplicaciones que de forma transparente y se escalen su paralelismo para poder incrementar el número de núcleos computacionales usados. Este diseño contiene tres puntos claves, que son la jerarquía de grupos de hilos, las memorias compartidas y las barreras de sincronización.

La estructura que se utiliza en este modelo está definido por un *grid*, dentro del cual hay bloques de hilos que están formados por como máximo 512 hilos distintos. Cada hilo tiene un identificador único dentro de su bloque, que se accede con la variable *threadIdx*.

Esta variable es muy útil para repartir el trabajo entre distintos hilos. *threadIdx* tiene 3 componentes (x, y, z), coincidiendo con las dimensiones de bloques de hilos. Así, cada elemento de una matriz, por ejemplo, lo podría tratar su homólogo en un bloque de hilos de dos dimensiones. La función que contiene código *CUDA* tradicionalmente se la conoce como *kernel*. En la imagen (9) se puede observar un esquema de la estructura de una *GPU*.

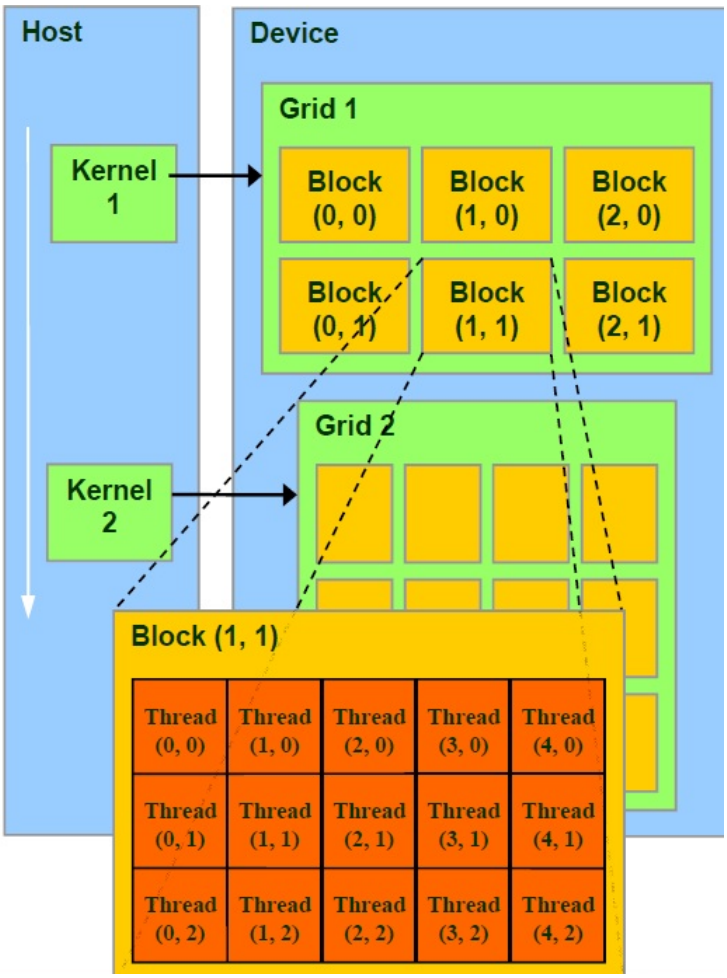

Figura 9: Esquema de la estructura de hilos de una *GPU*

Al igual que los hilos, los bloques se identifican mediante *blockIdx* (en este caso con dos componentes x e y). Otro parámetro útil es *blockDim*, para acceder al tamaño de bloque.

La jerarquía de memoria disponible en *CUDA* es bastante compleja, cada hilo dispone de distintas memorias que se clasifican por la velocidad que ofrecen de lectura y escritura, y el ámbito de acceso que tienen cada una de ellas.

Desde el punto de vista de un hilo tiene disponible las siguientes memorias:

- Leer y escribir en los registros locales para cada hilo.
- Leer y escribir en la memoria local para cada hilo.

- Leer y escribir en la memoria compartida para cada bloque de hilos.
- Leer y escribir en la memoria global para toda la *GPU*.
- Leer memoria de constantes para cada grid.
- Leer memoria de constantes en el grid.

Cada uno de los tipos de memoria nombradas se suelen utilizar para unas funciones concretas, principalmente dependiendo de su grado de compartición y de su latencia.

- Memoria global:
  - Se utiliza principalmente para comunicar por lectura y escritura la *GPU* y la *CPU*.
  - Su contenido es visible y modificable por todos los hilos de la *GPU*.
  - Tiempo de acceso lento, varios cientos de ciclos.
- Memoria de texturas:
  - Su contenido es inicializado por la *GPU*.
  - Es de solo lectura por los hilos.
  - Tiene un tiempo de latencia lento, aunque no tanto como la memoria global.
- Banco de registros:
  - Acceso local para cada hilo.
  - Número limitado de registros, si se sobrepasa el límite los hilos deberán esperar.
  - Tiempo de acceso muy pequeño, el más rápido de todos.
- Memoria compartida:
  - Memoria compartida por los hilos de cada bloque.
  - Se puede declarar de forma constante dentro del kernel o en tiempo de ejecución antes de lanzar el kernel.
  - Tiempo de acceso muy rápido, similar a los registros, se suele utilizar como memoria caché.
  - Si se sobrepasa el límite de memoria de la *GPU* saltará un error en la ejecución.
- Memoria local:
  - Memoria privada para cada hilo para la pila y las variables locales.
  - Se debe de declarar su tamaño de forma constante.
  - Tiempo de acceso similar a la memoria global.
- Memoria constante:
  - Compartida por todos los hilos de un bloque y de acceso simultaneo.

- Solo admite operaciones de lectura.
- Tiempo de acceso similar a los registros.

En la figura (10) se puede observar un esquema de la jerarquía de memoria de una *GPU*.

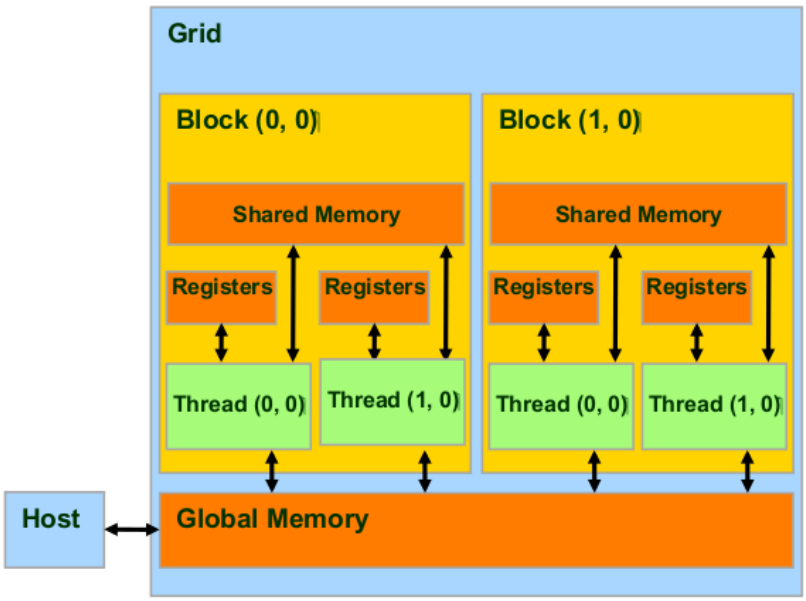

Figura 10: Esquema de la jerarquía de memoria de una *GPU*

4.3 Análisis teórico de los algoritmos estudiados

4.3.1 Algoritmo de fuerza bruta para el cálculo de las plantillas

Para el cálculo de las plantillas, la solución más sencilla es calcular todas las soluciones de la planta junto con las frecuencias de diseño y obtener las nubes de puntos correspondientes, es decir, realizar la combinatoria completa de todas las variables. Las plantillas son utilizadas para el cálculo de fronteras, y realmente sólo es necesario el contorno de dicha nube de puntos, por lo tanto, calcular todas las posibilidades de la planta es la forma más lenta computacionalmente hablando. Pero también es la única forma que nos asegura encontrar siempre la solución correcta, si se conoce al detalle la planta a estudiar se puede utilizar otro tipo de algoritmos que permiten "atajos" para encontrar la solución. Por ejemplo, existen algoritmos específicos para plantillas con determinadas formas, o plantas a las cuales se les puede aplicar una operación matemática y determinar el contorno de su plantilla directamente.

La dificultad planteada para implementar el algoritmo de fuerza de bruta, es que el número de variables participantes en la resolución es indeterminado en tiempo de compilación, depende de la entrada del usuario, con lo cual no es posible resolver utilizando bucles anidados. Cada variable tiene una serie de valores posibles, generalmente un rango con un espaciado (lineal o logarítmico) para sus valores intermedios. La solución implementada utiliza un vector de enteros donde cada variable tiene una posición asignada, el contenido de dicha posición indica cuál de los valores posibles de cada variable se tiene que utilizar. Dicho valor se actualiza conforme va avanzado el algoritmo.

En primer lugar, en la figura (11), se muestra un diagrama donde se indican los distintos bucles que tiene el algoritmo, dentro de cada bucle se indica que tipo de operaciones contiene y para terminar se indica para cada bucle la longitud del mismo a partir de variables que posteriormente se determinarán estudiando de forma individual cada uno de los bucles. Este es el primer paso para analizar donde es más adecuado y de que forma paralelizar utilizando *OpenMP*.

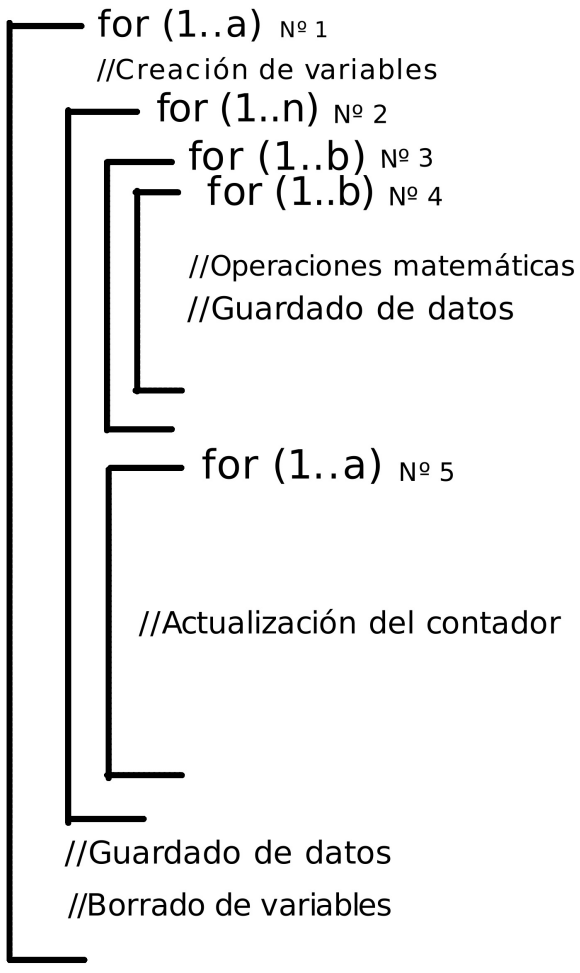

Figura 11: Diagrama de bloques del algoritmo de cálculo de plantillas.

Las entradas de este algoritmo, que afectan a su orden de complejidad son las siguientes:

- ω, las frecuencias de diseño.
- p, número de parámetros de la planta, lo constituye el numerador, denominador, la ganancia y el retardo.
- Número de puntos por parámetro, será un vector de valores para cada variable.

El bucle for nº 1 corresponde al bucle que recorre las frecuencias de diseño, este bucle se encuentra en los tres algoritmos implementados, dado que todos los cálculos se deben repetir para cada una de las frecuencias de diseño. La longitud de este bucle es muy variable, dado que depende de la entrada del usuario, y dicha entrada es muy dependiente también del problema con el que se esté trabajando. Generalmente se han usado pocas frecuencias de diseño, entre 5 y 15, para luego ya, una vez resuelto el problema, hacer una validación del diseño final con unas frecuencias de diseño más amplias.

El segundo de los bucles es el que realiza la combinatoria de todos los valores de la planta, la cantidad de variables de la planta, y la cantidad de valores para cada una de las variables puede ser muy diferentes dependiendo de la entrada del usuario, a su vez, los bucles 3 y 4 son una extensión con dos variables más del segundo, por tanto se pueden unir en el mismo orden de complejidad sumándole dos al número de parámetros de la planta. Por tanto los bucles números 2,3 y 4 se pueden caracterizar de la siguiente manera:

$$p' = p + 2$$
$$comb = p' \cdot n$$

Donde:

p es el número de parámetros de la planta.

n es el número de puntos por cada una de los parámetros de la planta, se asume el mismo número para todas los parámetros.

El bucle número 5 se puede considerar que tiene tiempo despreciable y por tanto operaciones constantes dentro del bucle número 2.

Con los bucles caracterizados se puede calcular el tiempo de ejecución en función de ω y $comb$ de la siguiente manera:

$$t(\omega, comb) = a \sum_{i=1}^{\omega} \left(b + \sum_{j=1}^{comb} c \right) = a + \sum_{i=1}^{\omega} (b + c \cdot comb) = a + (b + c \cdot comb) \cdot \omega$$

Donde:

- a son las operaciones constantes predecesoras al bucle número 1.
- b son las operaciones constantes dentro del bucle número 1.
- c son las operaciones constantes dentro de los bucles 2, 3 y 4.

De esta manera ya se tiene determinado el tiempo de ejecución del algoritmo en función de los parámetros de entrada proporcionados por el usuario, donde el orden de complejidad vendrá determinado por:

$$O(\omega, comb) = \omega \cdot comb$$

4.3.2 Algoritmo e-hull para el cálculo del contorno de las plantillas

Como se indicó en la sección (4.3.1) cuando el algoritmo de cálculo de fronteras utiliza las plantillas, solo necesita su contorno, aunque admitiría trabajar con la plantilla completa generada por el algoritmo de fuerza bruta, el tiempo de *CPU* aumentaría enormemente. Por ello es conveniente utilizar algún tipo de algoritmo que calcule el contorno de cada plantilla. El algoritmo implementado durante el curso pasado fue $\epsilon-hull$ [Nordin [1993]] utilizando como base el algoritmo implementado por [Montoya [1998]] en el lenguaje *MATLAB*, el funcionamiento del algoritmo se explica a continuación.

La entrada del algoritmo es la nube de puntos calculada por el algoritmo de fuerza bruta, y el objetivo es obtener el contorno de dicha nube de puntos. En primer lugar el algoritmo requiere como parámetro un número real que actuará como distancia (ϵ), tiene que ser indicada por el usuario y su valor adecuado depende de la forma de la nube. En la sección de vías futuras (6.2) se hace un estudio sobre las posibilidades de investigación para la elección automática de este parámetro. El algoritmo se aplica para cada plantilla, correspondiendo cada una de ellas a una frecuencia de diseño.

En la figura (12) se puede observar el diagrama de bucles del algoritmo. El bucle número 1 recorre el vector de las frecuencias de diseño introducidas por el usuario, la longitud de este bucle es igual al explicado en la sección (4.4.1) al ser el mismo bucle con los mismos datos a recorrer.

El bucle número 2 corresponde al primer paso del algoritmo, que es encontrar el primer punto del contorno, dicho punto es el número complejo con la parte real de mayor valor, por lo tanto hay que hacer un recorrido por todos los números del vector de complejos. La operación clave del bucle es la comparación de cada uno de los números con el mayor encontrado hasta el momento, por tanto, tiene dependencias insalvables para la paralelización a menos que se implemente un código que haga un tratamiento por bloques de los datos. Dado que el tiempo de *CPU* que tarda este bucle es irrelevante no se estima necesario profundizar en su paralelización.

A continuación, el bucle número 3 corresponde a la búsqueda del 2º número del contorno, dicho número se calcula a partir del punto número uno ya encontrado. El cálculo a realizar son diversas operaciones matemáticas sobre cada uno de los puntos que entran dentro de la distancia *epsilon*, con respecto al primer punto, de la plantilla. El objetivo

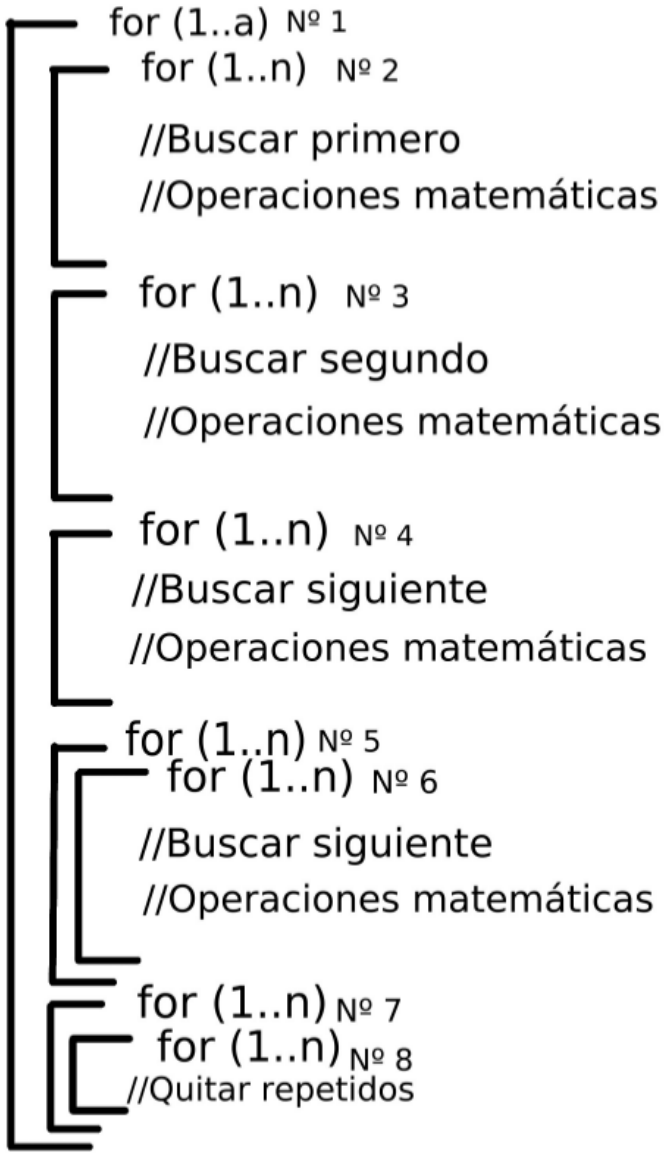

Figura 12: Diagrama de bloques del algoritmo *e-hull*.

es determinar el punto de menor ángulo con respecto al primer punto, para ello hay que comparar los resultados de las operaciones matemáticas sobre los puntos del radio *epsilon*. Para estudiar el orden de complejidad de este bucle hay que hacer un estudio del mejor y peor caso. El número de iteraciones del bucle no cambia para los distintos casos, pero si el número de operaciones que se realizan si, depende del valor escogido de *epsilon*, pero de forma no predecible, es decir, dado un valor a *epsilon* a priori no se puede determinar como afectará a la ejecución del bucle. Para poder determinar que operaciones corresponden a cada uno de los casos se muestra el código del algoritmo con las líneas numeradas:

```
1 qint32 Templates::buscarSegundo(qint32 b1, QVector<complex<qreal> > *cv, qreal
      epsilon){
2
```

```
3       qreal dist = 0;
4       complex <qreal> primero = cv->at(b1);
5
6       qreal fmin = numeric_limits<qreal>::infinity();
7       qint32 pmin = -1;
8       qreal aco = 0;
9       qreal dmax = 0;
10
11      complex <qreal> cvActual;
12
13      qreal fas = 0;
14
15      for (qint32 i = 0; i < cv->size(); i++){
16
17          cvActual = cv->at(i);
18          dist = abs(primero - cvActual);
19
20          if (dist > 0 && dist <= epsilon){
21
22              fas = arg (cvActual - primero);
23
24              if (fas < 0)
25                  fas += 2*M_PI;
26
27              aco = qAcos(dist / epsilon);
28              fas -= aco;
29
30              if (fas < fmin){
31                  fmin = fas;
32                  pmin = i;
33                  dmax = dist;
34
35              }else if (fas == fmin && dist > dmax){
36                  pmin = i;
37                  dmax = dist;
38              }
39          }
40      }
41
42      return pmin;
43  }
```

El orden de complejidad de este bucle es el siguiente:

- **Mejor caso:**

$$t_{m1}(n) = a + \sum_{i=0}^{n} b = a + b \cdot n$$

Donde:

– *n:* Corresponde al número de puntos de la plantilla original.

- *a:* Es una constante que corresponde a las operaciones antes del bucle (de la línea 3 a la 9).
- *b:* Es una constante que corresponde a las operaciones en el mejor caso (las líneas 17 y 18).

$t_{m1}(n)$ corresponde al mejor caso en una iteración cuando el punto de la plantilla que se está estudiando no entra dentro del radio de *epsilon*, esto podría pasar en todas las iteraciones si el valor de *epsilon* fuera lo bastante pequeño.

- **Peor caso:**

$$t_{M1}(n) = a + \sum_{i=0}^{n} c = a + c \cdot n$$

Donde:

- *n:* Corresponde al número de puntos de la plantilla.
- *a:* Es una constante que corresponde a las operaciones antes del bucle (de la línea 3 a la 9).
- *c:* Es una constante que corresponde a las operaciones en el mejor caso (de la línea 17 a la 37).

$t_{M1}(n)$ corresponde al peor caso en una iteración cuando el punto de la plantilla que se está estudiando entra dentro del radio de *epsilon*, esto podría pasar en todas las iteraciones si el valor de *epsilon* lo bastante grande.

El siguiente de los bucles corresponde a la búsqueda del punto siguiente, en el diagrama de bloques corresponde a los bucles 4 y 6, son una llamada a la misma función. Este bucle es muy similar al número 3, las diferencias se basan en que la búsqueda del punto siguiente se basa en los dos puntos anteriores, lo que añade algunas operaciones adicionales. El cálculo del orden de complejidad es muy similar, se añade a continuación el código de la función para poder identificar las operaciones para el peor y mejor caso:

```
1  qint32 Templates::buscarSiguiente(qint32 punto_previo, qint32 punto_actual,
2                                    QVector<complex<qreal> > *cv, qreal epsilon){
3     complex <qreal> actual = cv->at(punto_actual);
4     complex <qreal> anterior = cv->at(punto_previo);
5
6     qreal aco2 = qAcos(abs(anterior-actual) / epsilon);
7
8     qreal fasActual = 0;
9
10    qreal aco1Actual = 0;
11    qreal dmax = 0;
12
13    qreal psiActual = 0;
14
15
16    qreal psiMinActual = numeric_limits<qreal>::infinity();
17    qint32 posPsiMin = -1;
18
```

```
19      complex <qreal> cvActual;
20      qreal absResta;
21
22      for (qint32 i = 0; i < cv->size(); i++){
23
24          cvActual = cv->at(i);
25          absResta = abs(cvActual - actual); //Calculamos el valor absoluto de la
                distancia del punto al punto actual.
26
27
28          if (absResta > 0 && absResta <= epsilon && cvActual != actual &&
                cvActual != anterior){
29
30              //-----------------------------------------
31
32              //calculamos la fase entre los dos puntos normalizada.
33              fasActual = arg((cvActual - actual) / (anterior - actual));
34
35              if(fasActual < 0) // si es menor que cero le sumamos 2*PI.
36                  fasActual = fasActual + 2*M_PI;
37
38              //-------------------------------------------------------------
39
40              aco1Actual = qAcos(absResta / epsilon );
41
42              //-------------------------------------------------------------
43
44              if(fasActual == 0){
45                  psiActual = 2*M_PI - aco1Actual - aco2;
46              }else if (fasActual > 0 && fasActual < aco2){
47                  psiActual = fasActual + aco1Actual- aco2;
48              }else{
49                  psiActual = fasActual - aco1Actual - aco2;
50              }
51
52              if (psiActual < 0)
53                  psiActual += 2*M_PI;
54
55              if (psiActual < psiMinActual) {
56                  psiMinActual = psiActual;
57                  posPsiMin = i;
58                  dmax = absResta;
59              }else if (psiActual == psiMinActual &&
60                        (absResta > dmax)){
61                  posPsiMin = i;
62                  dmax = absResta;
63              }
64          }
65      }
66
```

```
67      return posPsiMin; //retornamos el mnimo
68
69  }
```

El orden de complejidad de este bucle número 4 (y 6) es el siguiente:

- **Mejor caso:**

$$t_{m2}(n) = a + \sum_{i=0}^{n} b = a + b \cdot n$$

  Donde:

  - *n:* Corresponde al número de puntos de la plantilla original.
  - *a:* Es una constante que corresponde a las operaciones antes del bucle (de la línea 3 a la 20).
  - *b:* CEs una constante que corresponde a las operaciones antes del bucle (las líneas 24 y 25).

  $t_{m2}(n)$ corresponde al mejor caso en una iteración cuando el punto de la plantilla que se está estudiando no entra dentro del radio de *epsilon*, esto podría pasar en todas las iteraciones si el valor de *epsilon* fuera lo bastante pequeño.

- **Peor caso:**

$$t_{M2}(n) = a + \sum_{i=0}^{n} c = a + c \cdot n$$

  Donde:

  - *n:* Corresponde al número de puntos de la plantilla.
  - *a:* Es una constante que corresponde a las operaciones antes del bucle (de la línea 3 a la 20).
  - *c:* Es una constante que corresponde a las operaciones antes del bucle (de la línea 24 a la 62).

  $t_{M2}(n)$ corresponde al peor caso en una iteración cuando el punto de la plantilla que se está estudiando entra dentro del radio de *epsilon*, esto podría pasar en todas las iteraciones si el valor de *epsilon* fuera lo bastante grande.

El siguiente de los bucles es el número 5, el cual no tiene un número de iteraciones fija, depende de la entrada y de como se desarrolle la ejecución del resto de bucles. Este caso también se va a estudiar conforme al peor y al mejor caso:

- **Mejor caso:**

$$t_{m3}(n) = \sum_{i=1}^{3} a + t_{m2}(n) = 3 \cdot t_2(n)$$

- **Peor caso:**

$$t_{M3}(n) = \sum_{i=1}^{3 \cdot n} a + t_{M2}(n) = 3n \cdot (t_2(n))$$

- Donde:
  - n: Es el número de puntos de la plantilla.
  - a: Son las operaciones constantes dentro del bucle.
  - $t_{X2}(n)$: Es el tiempo de ejecución del bucle número 6.

El último de los bucles, el número 7 es un clásico doble bucle para quitar los valores repetidos en un vector, su tiempo de ejecución puede calcularse a partir del parámetro n que corresponde al número de valores del vector y que coincide con el número de iteraciones del bucle número 5, el cual tendría mejor y peor caso, siendo el mejor caso el número mínimo de puntos para formar un contorno (3), y el pero caso serían todos los puntos de la plantilla (n):

$$t_4(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n} (n - (i+1) + 1) = \frac{1}{2} n^2$$

Al final, el orden de complejidad del algoritmo uniendo los cálculos parciales realizados a lo largo de toda la sección es el siguiente para el mejor y el peor caso:

- **Mejor caso:**

$$\begin{aligned} t_m(n, \omega) &= a + \sum_{i=1}^{\omega} (b + t_{m1}(n) + t_{m2}(n) + t_{m3}(n) + t_4(n)) \\ &= a + (b + t_{m1}(n) + t_{m2}(n) + t_{m3}(n) + t_4(u)) \cdot \omega \end{aligned}$$

- *Peor caso:*

$$\begin{aligned} t_M(n, \omega) &= a + \sum_{i=1}^{\omega} (b + t_{M1}(n) + t_{M2}(n) + t_{M3}(n) + t_4(n)) \\ &= a + (b + t_{M1}(n) + t_{M2}(n) + t_{M3}(n) + t_4(u)) \cdot \omega \end{aligned}$$

Con este cálculo definido tendríamos el tiempo de ejecución del algoritmo completo, donde a corresponde a las operaciones constantes antes del primer bucle y b a las operaciones constantes dentro del bucle número 1.

4.3.3 Algoritmo para el cálculo de fronteras

El algoritmo implementado está propuesto en [Moreno et al. [2006]], dicho algoritmo presenta una forma de calcular las fronteras que soporta fronteras multivaluadas. Para poder utilizar este algoritmo se necesitan una serie datos calculados previamente: la planta nominal, es decir, la planta calculada con los valores nominales de cada variable y las plantillas de la planta calculadas para cada frecuencia de diseño.

El funcionamiento del algoritmo es similar para las especificaciones definidas en las ecuaciones de la (5) a la (10), en el documento [Moreno et al. [2006]] se muestra el

desarrollo matemático que hay que realizar con cada una de las ecuaciones para obtener el resultado deseado. Para ejemplificar el proceso se escoge la especificación de estabilidad al ser la primera que se implementó y una de las más usadas. La ecuación tiene la siguiente forma:

$$\left| \frac{L(j\omega)}{1 + L(j\omega)} \right|_{dB} \leqslant \lambda \qquad \forall \omega > 0 \tag{12}$$

Donde $L(j\omega)$ representa al lazo abierto,

$$L(j\omega) = P(j\omega)C(j\omega) \tag{13}$$

y a su vez:

- $P(j\omega) = P_0(j\omega)\Delta P(j\omega)$ es el conjunto de posiciones que adopta la planta P a la frecuencia ω debido a la presencia de incertidumbre sobre la planta, incertidumbre que se puede expresar como un conjunto de variaciones $\Delta P(j\omega)$ respecto a la planta nominal $P_0(j\omega)$.
- $C(j\omega)$ es el controlador a diseñar. Precisamente la misión de las fronteras es definir, de forma indirecta, a través de restricciones sobre L, restricciones en forma de regiones prohibidas del *plano de Nichols* tales que los controladores $C(j\omega)$ que las violen serán considerados no admisibles.

El algoritmo basa su funcionamiento en reescribir la ecuación (13) de este modo:

$$L(j\omega) = P_0(j\omega)\Delta P(j\omega)C(j\omega) = L_0(j\omega)\Delta P(j\omega) \tag{14}$$

donde $L_0(j\omega)$ representa la posición del lazo nominal (el que corresponde a la planta $P_0(j\omega)$ de la plantilla) para un cierto controlador $C(j\omega)$.

La restricción dada por la ecuación (12) se puede interpretar ahora como: para una posición elegida del *plano de Nichols* que ocupe $L_0(j\omega)$, para cualquier L dada por este valor de $L_0(j\omega)$ y $\Delta P(j\omega)$ aplicando (14), se debe cumplir (12). Esto puede reescribirse como $D <= \lambda$, con

$$D = \text{máx} \left| \frac{L_0(j\omega)}{\frac{P_0(j\omega)}{P(j\omega)} + L_0(j\omega)} \right|_{dB} \tag{15}$$

La determinación de qué puntos del plano de Nichols son admisibles o no para $L_0(j\omega)$ se realiza estableciendo un grid en el eje de fases y otro en el de magnitudes (en dB.). Para cada punto de ese grid se calcula el valor de D. Esto genera una superficie de valores que se define como una función $f : \mathbb{R}^2 \to \mathbb{R}$ con dominio en el plano de $\mathbb{R}^2$ e imagen en la recta real $\mathbb{R}$, tal que $f(x, y) = h$ para todo (x, y) perteneciente a $\mathbb{R}^2$ y h perteneciente a la imagen de f. Dado que la superficie es un *plano de nichols* de números complejos, también se puede definir como una función $f : \mathbb{C} \to \mathbb{R}$ con dominio en el conjunto de los números complejos e imagen en la recta real, tal que $f(z) = h$, para todo z perteneciente

a $\mathbb{C}$ y h perteneciente a la imagen de f, siendo h un número real que determina una altura para un punto (x, y). Estas dos definiciones son equivalentes dado que $\mathbb{R}^2$ y $\mathbb{C}$ son isomorfos, siendo $\mathbb{C}$ el conjunto de los números complejos.

A continuación se utiliza la función *contour* para dar un "corte" a la superficie de valores a la altura h, esta función se define como recogedora del conjunto de soluciones de las ecuaciones $f(x, y) = h_i$ con h_i perteneciente a un conjunto J. Es decir, es una función $g : J \to 2^{\mathbb{R}^2}$ tal que para todo h_i perteneciente a J da un subconjunto de $\mathbb{R}^2$ como imagen de la función. O dicho de otra forma, la función *Contour* recoge el conjunto de curvas de nivel $f(x, y)$ que son una familia de funciones de la forma: $f(x, y) = h_i$ para cada h_i perteneciente J.

Problemática con el punto $(-180°, 0dB)$**:**

Si el punto $(-180°, 0dB)$ está contenido en la región definida por la plantilla cuando su nominal se sitúa en L_0 en el cálculo de D en (15) se generan problemas numéricos en el cálculo de dicha ecuación:

$$M = |L|_{dB} - 10 \log \left(1 + 2 \cdot 10^{\frac{|L|_{dB}}{20}} \cos \theta + 10^{\frac{|L|_{dB}}{10}}\right)$$

Para el punto $(-180°, 0dB)$:

$$M = -10 \log (1 + 2 \cos(-180) + 1) = -10 \log 0 = \infty$$

Lo que puede expresarse como:

$$\boxed{\lim_{(|L|_{dB} \to 0. \theta \to -180°)} M = \infty}$$

Con lo calculado en las ecuaciones predecesoras se concluye que si la plantilla acoge en su interior el mencionado punto, automáticamente es descartado como zona válida. Por lo tanto el procedimiento a seguir será comprobar antes de calcular la fórmula (15) si la plantilla contiene al punto $(-180°, 0dB)$.

Una vez explicado el funcionamiento del algoritmo se explica el código que se implementó el curso pasado para ver que puntos son los adecuados para su paralelización. En la figura (13) se puede observar el diagrama de bloques del algoritmo.

El bucle número 1 es el correspondiente a las frecuencias de diseño y es idéntico al explicado en los dos algoritmos anteriores (4.3.1, 4.3.2).

En primer lugar se estudian los bucles internos, dado que son los únicos que contienen operaciones de tiempo constantes en su interior, el resto contiene operaciones que no tiene coste constante. El bucle número 4 corresponde al bucle que recorre la plantilla de la planta para calcular los valores de esta con su desplazamiento. La cantidad de iteraciones de este bucle depende de la cantidad de valores que tenga la plantilla calculada en algoritmos anteriores, no se puede estimar una cantidad de valores usuales dado que son muy dependientes del problema que se esté estudiando, el tiempo que consume el algoritmo se

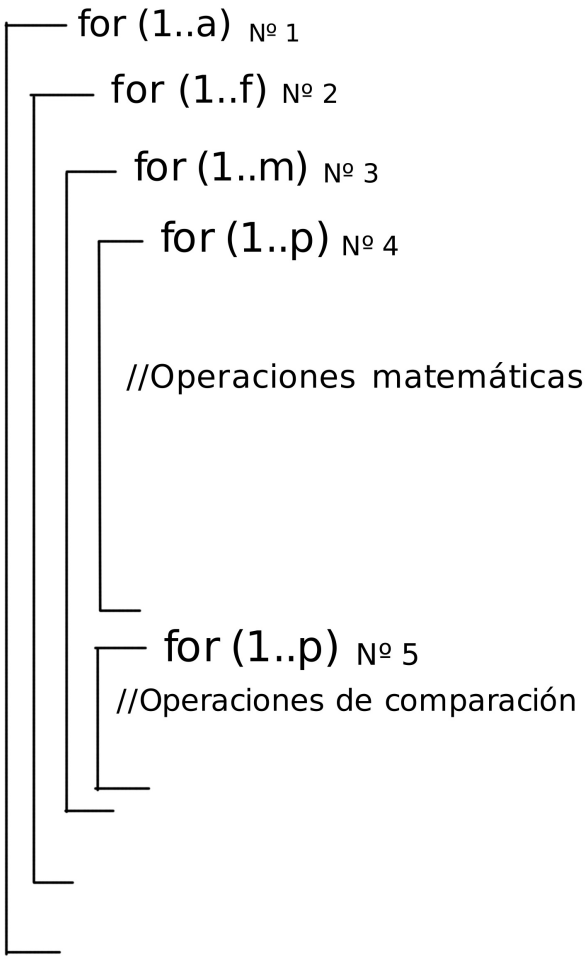

Figura 13: Diagrama de bloques del algoritmo de cálculo de fronteras.

calcula en función de ese parámetro.

$$t_1(p) = a + \sum_{i=1}^{p} b = a + b \cdot p$$

- a: Son las operaciones de tiempo constantes fuera del bucle.
- b: Son las operaciones de tiempo constantes dentro del bucle.
- p: Es la cantidad de valores de la plantilla de la planta, calculado por algoritmos anteriores.

El siguiente bucle, el número 5, corresponde a la función que determina si el punto $(0, -180)$ está dentro de la plantilla desplazada, recorre todos los puntos de dicha plantilla haciendo comparaciones, por tanto el tiempo de ejecución del bucle depende de este parámetro, de forma idéntica al anterior.

$$t_2(p) = a + \sum_{i=1}^{p} b = a + b \cdot p$$

- a: Son las operaciones de tiempo constantes fuera del bucle.
- b: Son las operaciones de tiempo constantes dentro del bucle.
- p: Es la cantidad de valores de la plantilla de la planta, calculado por algoritmos anteriores.

El siguiente de los bucles interiores, el número 3, corresponde al vector de magnitudes, los valores de este vector son dependientes de la entrada del usuario, los valores típicos suelen estar entre 60 y -60 decibelios. El orden de complejidad depende del parámetro introducido por el usuario:

$$t_3(p, m) = a + \sum_{i=1}^{m}(b + t_1(p) + t_2(p)) = a + (b + t_1(p) + t_2(p)) \cdot m$$

Donde:

- a: Son las operaciones de tiempo constantes fuera del bucle.
- b: Son las operaciones de tiempo constantes dentro del bucle.
- m: Es la cantidad de valores del vector de magnitudes elegidas por el usuario.

El segundo bucle exterior, el número 2, recorre el vector de fases elegido por el usuario, los valores típicos del vector suelen estar entre 0 y -360 grados, aunque los puntos no tienen porque estar a una equidistancia de 1, podría ser más o menos, depende de la elección del usuario. El orden de complejidad se puede calcular de la siguiente manera:

$$t_4(f, p, m) = a + \sum_{i=1}^{f}(b + t_3(p, m)) = a + (b + t_3(p, m)) \cdot f$$

Donde:

- a: Son las operaciones de tiempo constantes fuera del bucle.
- b: Son las operaciones de tiempo constantes dentro del bucle.
- f: Es la cantidad de valores del vector de fases elegidas por el usuario.

Una vez analizados los bucles del algoritmo desde los más interiores a los más exteriores podemos calcular el orden de complejidad del algoritmo completo:

$$t(\omega, f, p, m) = a + \sum_{i=1}^{\omega}(b + t_4(f, p, m)) = a + (b + t_4(f, p, m)) \cdot f$$

Donde:

- a: Son las operaciones de tiempo constantes fuera del bucle.
- b: Son las operaciones de tiempo constantes dentro del bucle.
- ω: Es la cantidad de valores del vector de frecuencias de diseño elegidas por el usuario.

4.4 Estudio de los algoritmos para su paralelización en la *CPU*

4.4.1 Estudio de la paralelización del algoritmo de cálculo de plantillas en la *CPU*

En esta sección se explican las decisiones tomadas con respecto a la paralelización del algoritmo de cálculo de plantillas en la *CPU* con *OpenMP*. Una vez estudiado el orden de complejidad, se puede observar como el parámetro ω es el que domina la ejecución del bucle exterior del algoritmo al ser el que determina su número de iteraciones. Como se ha indicado en la sección anterior (4.3.1) el parámetro ω suele tener entre 5 ó 15 valores, es decir, suele tener un tamaño pequeño. Como este programa se ha construido pensando en ordenadores personales, que suelen tener entre 2 y 4 cores (8 los más potentes), paralelizar con *OpenMP* a nivel del bucle número 1 proporcionaría suficiente hilos para ocupar todos los cores del procesador. En cuanto al resto de bucles se puede observar que todos ellos presentan dependencias que habría que resolver, lo cual dada la ocupación de los cores que proporciona el bucle número 1 y sobretodo el poco tiempo de ejecución que tiene este algoritmo no se estudia como opción. En el anejo (B.1) se puede consultar el código del algoritmo implementado.

Este algoritmo se decidió no implementarlo en la *GPU* al tener un tiempo de ejecución muy bajo, como se podrá observar en la sección de estudio experimental (5), además, los bucles interiores al número 1 presentan todos dependencias entre las iteraciones, lo cual complicaría su resolución para una implementación paralelamente masiva.

4.4.2 Estudio de la paralelización del algoritmo *e-hull* en la *CPU*

El bucle número uno es idéntico al bucle que recorre las frecuencias de diseño en el algoritmo de cálculo de plantillas (4.4.1), este bucle de hecho está presente en los tres algoritmos. Al igual que en el caso anterior este bucle es idóneo para paralelizarlo con *OpenMP* dado que sus iteraciones son completamente independientes, como ya se ha explicado el número de valores del vector de frecuencias de diseño suele estar entre 5 y 15, lo cual es ideal para tener ocupados los cores de la *CPU* de un ordenador personal.

El bucle número 2 que se encarga de buscar el primero de los puntos del contorno, tiene dependencia entre las iteraciones, dado que se objetivo es encontrar el número complejo que tenga la parte real de mayor valor.

Los bucles números 3, 4 y 6 son muy parecidos, cambian algunas operaciones matemáticas dentro de ellos pero en lo demás son idénticos, dichos bucles se encargan de aplicar una serie de operaciones matemáticas sobre los puntos de la plantilla para luego compararlos entre ellos. Por tanto sus iteraciones tienen dependencias.

El bucle número 5 corresponde a la búsqueda del contorno completo de la plantilla, cada iteración del bucle ejecutara el bucle número 6 en su interior. En este caso, cada ejecución de los dos bucles depende de los valores calculados en las dos iteraciones anteriores, por tanto es casi imposible de paralelizar dado que la ejecución debe ser continua para todo el contorno, es decir, no se puede calcular el contorno dividiendo la ejecución

en diferentes partes.

Los dos últimos bucles (el 7 y el 8) corresponden a la parte del algoritmo que quita los valores repetidos de un vector, el funcionamiento que tiene es comparar cada uno de los valores del bucle con todos los que son predecesores de él y así determinar si es un valor está repetido o no. Como se puede observar sus iteraciones tienen dependencias entre ellas debido a las operaciones de comparación necesarias. Se puede consultar el código del algoritmo en el anejo (B.2).

4.4.3 Estudio de la paralelización del algoritmo de cálculo de fronteras en la *CPU*

El primero de los bucles del algoritmo, al igual que en algoritmos anteriores, se puede paralelizar sin dificultad al ser el de las frecuencias de diseño. El siguiente de los bucles corresponde al vector de fases y el siguiente al de magnitudes, estos dos bucles recorren un grid en dos dimensiones, donde para cada una de las celdas de dicho grid hay que realizar una serie de cálculos matemáticos. Los cálculos para cada una de las celdas son independientes, por tanto estos dos bucles son perfectamente paralelizables, tal y como se ha hecho.

El siguiente de los bucles, el número 4, recorre la plantilla de la planta para la frecuencia de diseño y calcula el desplazamiento de la plantilla a la posición correspondiente de fase y magnitud. A su vez se calcula también el valor de las especificaciones seleccionadas por el usuario para cada una de las celdas del grid. Dependiendo de la especificación concreta, para todos los valores calculados en cada celda hay que seleccionar el que tenga el valor más alto, o la resta del mayor y el menor (según el tipo de frontera de que se trate). Por tanto, este bucle tiene dependencia entre las iteraciones, así que no interesa paralelizarlo.

El siguiente de los bucles, el número 5, es similar al anterior en cuanto al número de iteraciones se refiere. En este caso para cada una de las iteraciones del bucle se hacen diversas comparaciones y se invierte el valor de un booleano dependiendo del resultado de las comparaciones, por tanto tiene dependencia entre sus iteraciones. Se puede consultar el código del algoritmo en el anejo (B.4).

4.5 Estudio de la implementación de los algoritmos en la *GPU*

4.5.1 Estudio de la implementación del algoritmo *e-hull* en la *GPU*

La segunda de las actuaciones realizadas sobre el algoritmo para el cálculo del contorno de las plantillas *e-hull* ha sido su programación para ejecutarse en la *GPU*. La principal cuestión a tener en cuenta a la hora de desarrollar un algoritmo que se ejecute en la *GPU*, es que esta es masivamente paralela, tal como se describió en la sección (4.2.2), todos los hilos que se creen dentro del mismo bloque de la *GPU* deben ejecutar las mismas instrucciones, así como hay que resolver las dependencias entre los hilos.

Se decidió implementar en la *GPU* las funciones de búsqueda de los puntos, es decir, los tres bucles que tienen orden n, siendo n el número de puntos de la plantilla. Se eligieron estos tres bucles y no otros dado que la cantidad de datos a tratar suele ser lo suficientemente grande como para poder llenar todos los cores de la tarjeta gráfica. Hay que tener en cuenta que estas suelen tener entre 256 y 512 cores las instaladas en ordenadores personales. El bucle correspondiente a las frecuencias de diseño es difícil que alcance tan envergadura y el bucle principal del algoritmo, el número 5, es muy difícil de predecir cuantas iteraciones tendrá y al mismo tiempo el número habitual no suele ser tan grande (como 256).

Como se ha indicado en la sección anterior (4.4.2), estos bucles tienen dependencias entre sus iteraciones, por tanto hay que resolverlas antes de proceder a su paralelización. La solución ha sido dividir el problema en dos partes, la primera de ellas se encarga de operar sobre los valores de la plantilla, calcular los datos necesarios y guardarlos en un vector. A continuación, la segunda parte del algoritmo se encarga de encontrar entre esos valores el mínimo o el máximo, dependiendo del caso concreto.

Con el diseño elegido se van a realizar muchas llamadas a las funciones implementadas en *CUDA*, el tiempo que requieren cada una de estas llamadas es poco, la clave está en copiar datos del *host* a la *GPU* es lento, por tanto hay que evitar en lo máximo posible las copias de datos. Para ello se ha optado por reutilizar los datos ya copiados a la *GPU* entre las distintas llamadas, por ejemplo, el vector que contiene la plantilla que se debe utilizar en todas las llamadas se copia solo una vez en la primera de ellas, y el resto se reutiliza los mismos datos.

La primera de las funciones implementadas en *CUDA* ha sido la de "buscarprimero", dicha función tiene que seleccionar entre todos los valores de la plantilla quien punto tiene la parte real de mayor valor, es obvio que este bucle tiene dependencia entre sus iteraciones. La forma clásica de resolver este tipo de bucles, conocidos como "reducciones"[17] sobre un vector, es ir comparando dos a dos los valores del vector y seleccionar el que cumple las características deseadas, ya sea el menor, el mayor, la suma de ambos, etc. De esta manera cada una de las operaciones dos a dos se puede hacer de forma paralela.

Para poder hacer esta reducción sobre el vector se han utilizados las librerías *Thrust*[18] construidas por *NVIDIA* que proporcionan operaciones sobre vectores de forma paralela en la *GPU*, concretamente se han usado las operaciones para buscar el mínimo y el máximo en vectores. En el siguiente código se puede observar como se ha implementado esta funcionalidad:

```
1 int buscar_primero(host_vector<double> vector_buscar_primero_h){
2    //Copiamos los valores al vector del device para buscar el primer numero
3    device_vector<double> vector_buscar_primero_d = vector_buscar_primero_h;
4
5    //Buscamos el maximo.
6    device_vector<double>::iterator iter = max_element(vector_buscar_primero_d.
```

[17] Ejemplo de reducción http://cms.ac.uma.es/GPU/index.php/codigos-a-desarrollar/suma-por-reduccion Último acceso (22/04/2014)

[18] Web oficial: http://docs.nvidia.com/cuda/thrust (Último acceso 19/07/2014).

```
        begin(), vector_buscar_primero_d.end());
7
8       //Recuperamos la posicion del complejo primero.
9       int posicion_primero = iter - vector_buscar_primero_d.begin();
10
11      return posicion_primero;
12  }
```

En la instrucción de la línea 3 se hace la copia de datos del *host* a la *GPU*, en la instrucción de la línea 6 se hace la llamada a la función que busca el número mayor en el vector y devuelve un iterador con la posición de dicho valor, a continuación solo queda restar el iterador con la primera posición del vector y retornar el valor.

Se puede observar que utilizar estas librerías es muy sencillo, el principal problema que tienen radica en que, por un lado solo disponen de las principales operaciones que se realizan sobre vectores y, por otro lado, su integración con un código implementado en *CUDA* es complicado. Principalmente porque utilizan tipos de datos de alto nivel para manejar la copia de datos entre el *host* y la *GPU* algo de lo que no se dispone en un código implementado en *CUDA*. La librería está pensada para ser utilizada como "caja negra" tanto en lo que se refiere a la copia de datos entre el *host* y el *dispositivo*, como en las operaciones a realizar.

A continuación se analiza como se ha implementado la función "buscarsegundo" en la *GPU*, como se ha indicado anteriormente se ha divido el algoritmo en dos partes principales, la primera de ellas realiza los cálculos matemáticos sobre los puntos de la plantilla, en el siguiente código se puede observar como se ha implementado:

```
1   static __global__ void buscar_segundo_kernel(Complex primero, Complex * puntos
        , float epsilon,
2                                                double * retorno, int n_puntos)
3   {
4       int i = blockDim.x * blockIdx.x + threadIdx.x;
5
6       if (i < n_puntos){
7
8           double dist = abs(puntos[i] - primero);
9
10          if (dist > 0 && dist <= epsilon){
11
12              retorno[i] = arg(puntos[i] - primero);
13
14              if (retorno[i] < 0){
15                  retorno[i] += 2 * pi;
16              }
17
18              retorno[i] -= acos(dist / epsilon);
19          } else {
20              retorno[i] = inf;
21          }
22      }
```

```
23 }
```

Los datos de entrada que utiliza la función son tres principalmente, el punto primero calculado por la función "buscarprimero", el vector con todos los puntos de la plantilla, para esta función es la única vez que se copia este vector a la memoria de la *GPU*, en el resto de ocasiones se utiliza sin volverse a copiar, y la variable *epsilon* proporcionada por el usuario.

Para guardar los cálculos parciales realizados en la función se ha utilizado el mismo vector que será salida del procedimiento, de esta manera no es necesario utilizar variables auxiliares. Esta función se llama para todos los puntos de la plantilla de forma paralela. Y retorna el ángulo que existe entre cada uno de los puntos con respecto al punto primero, a continuación, para seleccionar el punto definitivo hay que recupera el valor mínimo del vector, que se convertirá en el punto segundo del contorno.

Para buscar el valor mínimo en un vector se valoró la posibilidad de implementar la búsqueda de dicho valor mínimo de forma manual, pero después de implementar parte de la funcionalidad y estudiar los pros y los contras que ofrecía se optó por utilizar las librerías *Thrust*, con el problema añadido de la compatibilidad. En este caso el vector donde buscar el valor mínimo ya está ubicado en la memoria de la *GPU* por tanto el objetivo a conseguir es no copiar esos datos al *host* para volver a copiarlos a la *GPU*.

Para ello en vez de utilizar los tipos de datos clásicos de la librería como son el *host_vector*[19] y el *device_vector*[20] se ha utilizado el *device_ptr* que crea el vector a partir de un puntero a los datos ya almacenados. Utilizar este tipo de vector ha supuesto utilizar otro tipo de funciones para poder llamar a la librería, pero así se ha conseguido evitar la copia de datos entre las dos funciones que se ejecutan en la tarjeta gráfica. A continuación se puede observar la parte del código donde está implementado la llamada a kernel de cuda y la posterior búsqueda el valor mínimo del vector que retorna:

```
1  buscar_segundo_kernel <<<blocksPerGrid, threadsPerBlock>>> (complejo_primero,
2                                  puntos_d, epsilon,retorno_segundo_d,
                                       numElements);
3
4     device_ptr<double> d_vec (retorno_segundo_d);
5     device_ptr<double> d_vec_final = d_vec + numElements;
6     device_ptr<double> result = min_element(d_vec, d_vec_final);
7
8     unsigned int posicion_segundo = result - d_vec;
9
10    double min_val = *result;
```

La operación de la línea número uno corresponde a la llamada de la función de *CUDA* que resuelve la primera parte del problema, a continuación las líneas de la 4 a la 6 corresponde la preparación del vector y la llamada a la función *min_element*, y ya por último se retorna la posición del valor buscado.

[19]Vector el cual los datos que contiene están situados en la memoria del procesador.
[20]Vector el cual los datos que contiene están situados en la memoria de la tarjeta gráfica.

La última de las funciones, "buscarsiguiente", tiene una estructura idéntica a la ya explicada "buscarsegundo", la única diferencia radica en que en esta función se realizan una serie de operaciones matemáticas adicionales. Se puede consultar el código completo del algoritmo en el anejo (B.3).

4.5.2 Estudio de la implementación del algoritmo de cálculo de fronteras en la *GPU*

En esta sección se explica como se ha implementado el algoritmo de cálculo de fronteras en la *GPU*, los bucles que se estimaron adecuados para implementarlos en paralelo fueron el que recorre el vector de fases y el que recorre el vector de magnitudes, dado que son los que generan el mayor número de iteraciones del algoritmo. Estos dos bucles se asemejan en sus recorridos a iterar sobre una matriz de dos dimensiones, por lo tanto su paralelización no sería directa como en el caso anterior.

Existen muchos ejemplos de como configurar la estructura lógica de la tarjeta gráfica para poder recorrer una matriz de forma paralela, se estudió la posibilidad de implementarlo de esta manera pero se descarto por la necesidad que tiene el algoritmo de utilizar muchas variables y también gran cantidad de memoria, evitando de esta manera desbordar tanto el número de registros que tiene la tarjeta gráfica como la memoria de la que dispone. Se determinó que era más adecuado lanzar menos hilos a la vez, siendo este número aun así superior al número de procesadores de la *GPU*, que aumentar el número de hilos y que se agotara la memoria de registros disponible y con ello se retrasara la ejecución de algunos hilos. Por tanto la decisión final fue paralelizar el algoritmo por el bucle que recorre las fases, de esta manera se crean suficientes hilos como para tener los cores de la tarjeta gráfica ocupada y no se crean demasiados como para llenar alguna de las memorias utilizadas.

En el código del algoritmo, que se adjunta en el anejo (B.5), se puede observar como en este caso concreto era necesario almacenar datos temporales que una vez utilizados no era necesario guardar, en primer lugar se optó por utilizar memoria compartida, al ser está la más rápida de las memorias disponibles, pero una vez comprobado que el tamaño de esta memoria era muy inferior al necesitado se descartó su uso y se optó por utilizar memoria global, que a pesar de ser más lenta, tiene un tamaño mucho mayor.

La siguiente problemática que surgió fue la gran cantidad de datos que el algoritmo debe calcular y retornar a la *CPU*, para cada uno de los hilos calcula un vector por cada especificación del tamaño del número de magnitudes elegidas por el usuario, que junto con la utilización de la memoria global para guardar datos temporales para los ejemplos grandes puede desbordar la memoria y hacer que el resultado sea indeterminado. Para solucionar esto se ha propuesto desviar la ejecución de los ejemplos más grandes a la *CPU* donde se dispone de memoria suficiente y se podría ejecutar de forma paralela al cálculo en la *GPU*. Esta propuesta va en sintonía con lo explicado en vías futuras (6.2) de dividir la carga computacional entre la *CPU* y la *GPU*.

5 Estudio experimental de los algoritmos implementados

5.1 Introducción

En esta sección se van a exponer los resultados obtenidos para las diferentes pruebas de rendimiento realizadas sobre los tres algoritmos optimizados. Se han seleccionado tres ejemplos diferentes para realizar las pruebas y se ha planteado para cada uno de los ejemplos diferentes casos, casos con diferente complejidad computacional en relación al tamaño y la forma de los datos de entrada..

Para cada uno de los ejemplos se pueden variar dos parámetros fundamentales, el número de frecuencias de diseño que el usuario introduce, o el número de puntos para cada variable de la plantilla, afectando de manera diferente a las diversas versiones de los algoritmos. Por ejemplo, aumentar el número de frecuencias de diseño favorece el rendimiento del algoritmo paralelo en *CPU* al poder aprovechar los diferentes núcleos del procesador. En cambio, aumentar el número de puntos de cada variable favorece al algoritmo implementado en *GPU* dado que aumenta el número de datos por cada frecuencia de diseño, y por tanto, el número de datos con los que la *GPU* debe de trabajar.

Si nos centramos en las frecuencias de diseño se han planteado las siguientes pruebas, en primer lugar se ha seleccionado una sola frecuencia de diseño, para comparar los algoritmos de calculo de plantillas y calculo del contorno de plantillas como si no existiera paralelismo en la *CPU*, para el cálculo de fronteras sí que se utilizarían todos los núcleos del procesador al estar el algoritmo paralelizado por diferentes bucles. A continuación se han seleccionado 4 frecuencias de diseño, que coincide con el número de cores del procesador, de esta manera se explota el máximo paralelismo en *CPU*. Para terminar se ha seleccionado un número de frecuencias de diseño que no sea múltiplo del número de núcleos del procesador, 5 ó 6 dependiendo del ejemplo concreto.

En cuanto al número de puntos de cada variable se han planteado dos casos diferentes, diez puntos por variable como caso sencillo y veinte puntos por variable, como prueba más costosa computacionalmente. De esta manera se puede observar como responden los diferentes algoritmos al tener que procesar más y menos datos.

El equipo donde se han realizado las pruebas es un *Asus A55V*[21] que tiene un procesador *Intel Core i7-3610QM a 2.30GHz*[22] y una tarjeta gráfica *NVIDIA GEFORCE 610M*[23]. Todas las pruebas han sido medidas en milisegundos tomando para cada una de las ejecuciones 6 mediciones de tiempo, descartando las 2 primeras y haciendo la media con las 4 siguientes.

[21]Página del producto http://www.asus.com/my/Notebooks_Ultrabooks/A55VD/ Último acceso (22/07/2014)

[22]Página del producto http://ark.intel.com/products/64899/Intel-Core-i7-3610QM-Processor-6M-Cache-up-to-3_30-GHz Último acceso (22/07/2014)

[23]Página del producto http://www.geforce.com/hardware/notebook-gpus/geforce-610m Último acceso (22/07/2014)

5.2 Ejemplo número uno

El ejemplo número uno coincide con el también ejemplo uno del *toolbox* de *QFT* de *MATLAB* y consta de los siguientes datos:

$$P(s) = \frac{k}{(s+a)(s+b)}$$

Donde:

$k \in [1, 10]$

$a \in [1, 5]$

$b \in [20, 30]$

5.2.1 Una frecuencia de diseño y diez puntos por variable

En primer lugar se plantea una ejecución con una sola frecuencia de diseño y 10 puntos por variable, un ejemplo pequeño para observar como se comportan los algoritmos para estos casos.

Estos son los resultados obtenidos en las diferentes ejecuciones:

Impl. / Alg.	Alg. plantillas	Alg. *e-hull*	Alg. fronteras
CPU secuencial	0.75 ms	9.5 ms	983 ms
CPU paralelo	1.75 ms	5.75 ms	974 ms
CUDA	-	11 ms	93.25 ms

Como se puede observar en los resultados para el algoritmo de cálculo de plantillas el sobrecoste que proporciona usar las librerías *OpenMP* no compensa al utilizar solo un core, por ello para ejemplos tan pequeños la implementación secuencial obtiene mejor resultados.

En cuanto al algoritmo *e-hull* las mejoras introducidas en el código en la nueva versión justifican la mejora de rendimiento de la versión paralela en *CPU* con respecto a la versión secuencial a pesar de utilizar un solo core. En cambio la versión implementada en la *GPU* obtiene el peor resultado de todos debido que al ser un ejemplo tan pequeño no compensa con el coste de inicializar la tarjeta gráfica.

Para terminar, el algoritmo de cálculo de fronteras es mucho más pesado computacionalmente que el resto, para las dos implementaciones en *CPU* obtiene unos resultados similares, en cambio para la *GPU* se obtienen unos resultados mucho mejores, se aprovecha la potencia de cálculo que ofrece la tarjeta gráfica.

5.2.2 Una frecuencia de diseño y veinte puntos por variable

En esta segunda ejecución se aumentan el número de puntos por variable, de esta manera se incrementa notablemente el número de operaciones aritméticas necesarias en cada algoritmo, de esta manera se puede observar como responden los algoritmos al realizar cálculo intensivo con números complejos.

Estos son los resultados obtenidos en las diferentes ejecuciones:

Impl. / Alg.	Alg. plantillas	Alg. *e-hull*	Alg. fronteras
CPU secuencial	7.5 ms	181.5 ms	1918 ms
CPU paralelo	3.25 ms	65 ms	1986.75 ms
CUDA	-	38.25 ms	187 ms

Para este ejemplo donde se aumenta el número de operaciones aritméticas se puede observar como los algoritmos que mejor se desenvuelven con este tipo de operaciones incrementan su diferencia con respecto a los otros algoritmos. Así por ejemplo en el algoritmo de cálculo de plantillas la versión secuencial tarda el doble que la versión paralela, eso a pesar de que se está utilizando un solo core, es debido a las mejoras de código aplicadas sobre la versión paralela con respecto a la secuencial. En cambio para el algoritmo de la *e-hull* las diferencias entre las dos implementaciones en la *CPU* aumentan de forma significativa y la implementación en la *GPU* se sitúa con el mejor tiempo al haberse incrementado el número de operaciones aritméticas.

En cuanto al algoritmo de calculo de fronteras las diferencias se mantienen entre todas las implementaciones y los tiempos de ejecución se doblan con respecto a la prueba anterior al tener que realizar un número mayor de operaciones aritméticas.

5.2.3 Cuatro frecuencias de diseño y diez puntos por parámetro

En esta ejecución se vuelve a proponer menos puntos por parámetro y se incrementa el número de frecuencias de diseño, para observar que rendimientos se obtienen en la versión paralela con *OpenMP* siendo esta la ejecución ideal al coincidir el número de cores con el número de frecuencias de diseño.

Estos son los resultados obtenidos en las diferentes ejecuciones:

Impl. / Alg.	Alg. plantillas	Alg. *e-hull*	Alg. fronteras
CPU secuencial	3 ms	673.75 ms	5130.25 ms
CPU paralelo	3.75 ms	253 ms	2142 ms
CUDA	-	239 ms	470 ms

Para el algoritmo de cálculo de plantillas se puede observar como, que a pesar de haber utilizado todos los cores del procesador el tiempo de la versión secuencial es más rápido por poca diferencia, esto se debe a que como el algoritmo tiene muy poca carga computacional cada uno de los hilos realiza muy pocas operaciones y no compensa crear los hilos, por ello la versión secuencial tarda menos.

Centrándonos en el algoritmo *e-hull* se puede observar como los tiempos de la versión secuencial se disparan al no poder competir con las otras dos implementaciones. Para las versiones paralelas pasa una situación similar a la del algoritmo anterior, a pesar de que se utilizan todos los cores de la *CPU* como la cantidad de carga de trabajo para cada uno de los cores es pequeña la ejecución en la tarjeta gráfica es algo más rápida a pesar de contar solo con una unidad funcional.

Para terminar en el algoritmo de cálculo de fronteras la situación para la versión secuencial se asemeja al algoritmo anterior, no es capaz de competir en rendimiento con las versiones paralelas, esta tónica se repetirá para las pruebas posteriores. En cambio las diferencias en las versiones se mantienen similares, en este caso la ejecución en la *GPU* pierde ventaja con respecto a la versión paralela en *CPU* al poder aprovechar dicho algoritmo todos los cores del procesador.

5.2.4 Cuatro frecuencias de diseño y veinte puntos por parámetro

En este apartado se propone utilizar las mismas frecuencias de diseño que en el anterior pero aumentando el número de puntos por variable, con el fin de observar como se comportan los algoritmos en esta nueva situación donde el paralelismo en *CPU* puede aprovechar todos los cores del procesador y la tarjeta gráfica puede aprovechar el aumento de las operaciones a realizar.

Estos son los resultados obtenidos en las diferentes ejecuciones:

Impl. / Alg.	Alg. plantillas	Alg. *e-hull*	Alg. fronteras
CPU secuencial	23 ms	666.75 ms	9110.5 ms
CPU paralelo	2.5 ms	102 ms	3837 ms
CUDA	-	171 ms	567 ms

Para el primero de los algoritmos, el de cálculo de plantillas, se puede observar como la versión paralela obtiene mucho mejor rendimiento que la secuencial, cada uno de los cores realiza más operaciones y por tanto las diferencias con la versión secuencial aumentan.

En el algoritmo de *e-hull* se puede observar como los tiempos de ejecución de las dos versiones paralelas han disminuido con respecto a la prueba anterior, a pesar de haber aumentado el número de puntos, esto puede ser debido a la propia forma de ejecución del algoritmo, que para la nueva distribución de los puntos se aproveche la localidad espacial y temporal de la memoria caché y los datos se sirvan más rápido.

En cuanto al algorito de cálculo de fronteras las diferencias de tiempos sigue aumentando, se puede observar como el aumento de puntos apenas afecta a la versión en *GPU*, en cambio a las dos versiones en *CPU* afecta de manera notable.

5.2.5 Cinco frecuencias de diseño y diez puntos por variable

En este apartado se aumentan las frecuencias de diseño para que no sea un número múltiplo de del número de cores, para ver así que sobrecarga introduce la frecuencia de

diseño extra.

Estos son los resultados obtenidos en las diferentes ejecuciones:

Impl. / Alg.	Alg. plantillas	Alg. *e-hull*	Alg. fronteras
CPU secuencial	4 ms	694 ms	6130 ms
CPU paralelo	2.75 ms	265 ms	2496 ms
CUDA	-	247 ms	524 ms

Para el algoritmo de cálculo de plantillas se puede observar como al haber aumentado el número de datos a calcular, en este caso por las frecuencias de diseño, la versión secuencial aumenta el tiempo que tarda con respecto a la versión paralela.

En cuanto al algoritmo *e-hull* se obtienen unos tiempos algo mas altos que con 4 frecuencias de diseño y 10 puntos por variable, pero la diferencia entre los distintos algoritmos no varia.

Para el algoritmo de cálculo de fronteras la situación es similar, los tiempos de ejecución han aumentado pero la diferencias se mantienen estables, la variación de una sola frecuencia de diseño no afecta de manera determinante a la paralelización en la *CPU*.

5.2.6 Cinco frecuencias de diseño y 20 puntos por variable

En esta prueba se desea comprobar como afecta que el número de frecuencias de diseño no sea múltiplo del número de cores del procesador cuando el número de operaciones a realizar aumenta notablemente.

Estos son los resultados obtenidos en las diferentes ejecuciones:

Impl. / Alg.	Alg. plantillas	Alg. *e-hull*	Alg. fronteras
CPU secuencial	28 ms	665 ms	11040 ms
CPU paralelo	3 ms	132 ms	4332.5 ms
CUDA	-	207 ms	755 ms

Para el algoritmo de cálculo de plantillas se puede observar como el tiempo para la versión paralela se mantiene estable, en cambio para la versión secuencial aumenta algo la diferencia, el aumento del número de operaciones, al ser un algoritmo tan rápido para este ejemplo no afecta al tiempo de ejecución.

En el segundo algoritmo, *e-hull*, sucede un fenómeno similar que para las pruebas con cuatro frecuencias de diseño, el aumento de puntos por variable disminuye el tiempo de ejecución, puede deberse a un mejor aprovechamiento de la localidad espacial y temporal en la memora caché así como a la forma de ejecución del algoritmo, que con la nueva distribución de los puntos realiza menos operaciones.

Para el último algoritmo, el del cálculo de fronteras, se puede observar como todos los tiempos de ejecución aumentan pero la implementación paralela en la *GPU* tiene un menor incremento que el resto, aprovechando la capacidad de cálculo de la tarjeta gráfica.

5.3 Ejemplo número dos

Para este ejemplo número 2 se ha seleccionado una función de transferencia más compleja computacionalmente, se ha aumentado el número de variables presentes para así aumentar el número de operaciones necesarias que debe resolver cada algoritmo. De esta manera se podrá comprobar que resultados se obtienen para cada algoritmo cuando el número de datos a trata aumenta. Se puede observar en los resultados como esta planta tiene unas plantillas especialmente complejas para calcularse su contorno, y por ello el algoritmo *e-hull* necesita muchas operaciones para obtener el resultado.

La planta seleccionada es la siguiente:

$$P(s) = K\frac{a}{bs^2 + cs + d}$$

Donde:

$k \in [1, 10]$

$a \in [1, 5]$

$b \in [5, 20]$

$c \in [20, 30]$

$d \in [1, 10]$

5.3.1 Una frecuencia de diseño y diez puntos por variable

Para esta primera prueba se ha seleccionado una sola frecuencia de diseño, para al igual que en el ejemplo anterior, comprobar como funcionan los algoritmos al no existir paralelismo en *CPU*, en este caso con pocos puntos por variable para comprobar, en primer lugar, su comportamiento al realizar pocas operaciones.

Estos son los resultados obtenidos en las diferentes ejecuciones:

Impl. / Alg.	Alg. plantillas	Alg. *e-hull*	Alg. fronteras
CPU secuencial	165.5 ms	1172 ms	1395 ms
CPU paralelo	25 ms	510 ms	1438 ms
CUDA	-	185 ms	128.25 ms

Para el primero de los algoritmos, el del cálculo de plantillas, a pesar de que el algoritmo no ha podido explotar el paralelismo en *CPU* ha obtenido un mejor resultado que la versión secuencial, esto es debido a las mejoras introducidas en el código en la nueva versión del algoritmo

En el segundo de los algoritmos, el de *e-hull*, las diferencias entre resultados son notables, mientras que la versión secuencial tiene la ejecución más lenta de todas, el algoritmo implementado con *OpenMP* tarda algo menos de la mitad y el algoritmo en *GPU* es mucho más rápido. Que la versión paralela en *GPU* obtenga tan buen resultado con respecto a la versión paralela en *CPU* es debido a que solo se está usando un core del procesador

y no se puede explotar el paralelismo en la aplicación.

Para finalizar, en el algoritmo de cálculo de fronteras, los tiempos de ejecución obtenidos son muy similares para las dos implementaciones en la *CPU*, incluso la versión paralela es más lenta que la versión secuencial, eso es debido a que al realizar pocas operaciones y no utilizar paralelismo la versión secuencial tiene menos coste computacional. En cambio la versión implementada en la *GPU* obtiene mejores resultados al explotar la potencia de cálculo de la tarjeta gráfica.

5.3.2 Una frecuencia de diseño y veinte puntos por variable

Para esta prueba se ha aumentado el número de operaciones a realizar por cada algoritmo, para comprobar de esta manera como se comportan sin tener disponible el paralelismo en la *CPU* y aumentando el número de operaciones aritméticas a realizar.

Estos son los resultados obtenidos en las diferentes ejecuciones:

Impl. / Alg.	Alg. plantillas	Alg. *e-hull*	Alg. fronteras
CPU secuencial	3198 ms	74655 ms	2621 ms
CPU paralelo	415 ms	29804 ms	2749 ms
CUDA	-	9726 ms	251 ms

Para esta prueba se puede observa como en el algoritmo de cálculo de plantillas el resultado obtenido es mucho mejor en la versión paralela a pesar de estar utilizando un solo core, al haber aumentado el número de operaciones a realizar, las mejoras introducidas en el código de la nueva versión obtienen un rendimiento superior.

Para el algoritmo *e-hull*, como se ha dicho anteriormente es un algoritmo complicado computacionalmente, se puede observar como los tiempos de ejecución son altos, principalmente el algoritmo secuencial obtiene el pero rendimiento con diferencia frente, en primer lugar al algoritmo paralelo, que a pesar de usar un solo core, y de forma similar al anterior algoritmo, las mejoras introducidas en el código dan un resultado mejorado, en segundo lugar para la *GPU* obtiene el mejor rendimiento al no tener que competir con paralelismo en la *CPU*.

Para el último de los algoritmos, el del cálculo de fronteras, las dos versiones implementadas en la *CPU* obtiene un rendimiento similar, a no disponer de paralelismo en la *CPU* el algoritmo secuencial incluso es algo más rápido. En cambio, para el algoritmo en *GPU* el resultado es mucho mejor, al poder aprovechar la potencia de cálculo de la tarjeta gráfica.

5.3.3 Cuatro frecuencias de diseño y diez puntos

Para esta prueba se ha seleccionado que el número de frecuencias de diseño coincida con el número de cores del procesador, para observar como se comporta el algoritmo paralelo en *CPU* a poder explotar al máximo el paralelismo.

Estos son los resultados obtenidos en las diferentes ejecuciones:

Impl. / Alg.	Alg. plantillas	Alg. *e-hull*	Alg. fronteras
CPU secuencial	363 ms	4597 ms	5170.25 ms
CPU paralelo	52 ms	689 ms	1959 ms
CUDA	-	620 ms	439.5 ms

Para el primero de los algoritmos, el del cálculo de plantillas, se puede observar como el rendimiento de la versión paralela es muy superior, al haber utilizado todos los cores del procesador se obtienen mejores tiempos de ejecución.

Para el algoritmo *e-hull* la versión secuencia es con diferencia la que peores tiempos obtiene, no puede competir con el paralelismo en *CPU* y *GPU*. Para las dos versiones paralelas implementadas obtienen un resultado muy similar, siendo la versión implementada en la tarjeta gráfica algo más rápida, eso es debido a que al ser un ejemplo que requiere más operaciones para ser resuelto, la tarjeta gráfica se impone.

En el último de los algoritmos, el del cálculo de fronteras, al igual que en las anteriores pruebas, el algoritmo secuencial obtiene los peores resultados, a continuación le sigue la versión paralela en *CPU* que mejora notablemente con respecto a la versión secuencial, pero que al ser un problema con muchas operaciones aritméticas no puede competir con la versión en *GPU* que obtiene los mejores resultados.

5.3.4 Cuatro frecuencias de diseño y veinte puntos por variable

Para esta prueba se han aumentado el número de puntos por variable, para observar como se comportan los algoritmos al aumentar el número de operaciones a realizar.

Estos son los resultados obtenidos en las diferentes ejecuciones:

Impl. / Alg.	Alg. plantillas	Alg. *e-hull*	Alg. fronteras
CPU secuencial	9270 ms	295599 ms	10175 ms
CPU paralelo	1224 ms	31394 ms	3837 ms
CUDA	-	32907 ms	876 ms

Para el primero de los algoritmos, el del cálculo de plantillas, se puede observar como la diferencia de tiempos para las dos versiones aumenta, obteniendo la versión secuencial un resultado mucho mayor.

En el algoritmo *e-hull* la versión secuencial obtiene el peor tiempo con diferencia, siendo las dos versiones paralelas las que mejor tiempo obtienen y de forma similar, siendo la implementación para la *GPU* algo más lenta.

Para terminar, en el algoritmo de cálculo de fronteras, se obtiene unas diferencias similares a las anteriores, la implementación para la *GPU* obtiene el mejor resultado con diferencia, seguido de la versión paralela en *CPU* y terminando con el peor resultado para la versión secuencial.

5.3.5 Seis frecuencias de diseño y diez puntos por variable

Para esta prueba se ha fijado un número de frecuencias de diseño que no sea múltiplo del número de cores, para observar así que resultados se obtiene para en la versión paralela en la *CPU*.

Estos son los resultados obtenidos en las diferentes ejecuciones:

Impl. / Alg.	Alg. plantillas	Alg. *e-hull*	Alg. fronteras
CPU secuencial	490 ms	6915 ms	7770 ms
CPU paralelo	63 ms	740.25 ms	1952 ms
CUDA	-	923 ms	670 ms

Para el primero de los algoritmos, el del cálculo de plantillas, el resultado es similar a las pruebas anteriores, la explotación del paralelismo hacer que la versión implementada con *OpenMP* obtenga mucho mejores resultados que la versión secuencial.

En el algoritmo *e-hull* la versión secuencia obtiene el peor resultado, de forma similar a las anteriores pruebas, al igual que las dos versiones paralelas obtiene un resultado casi igual siendo la versión en *CPU* algo más rápida que la versión en *GPU*.

Para terminar, el algoritmo de cálculo de fronteras, se obtiene un resultado similar a prueba anteriores, obteniendo la versión paralela en *CPU* un mejor resultado al poder aprovechar mejor el paralelismo.

5.3.6 Seis frecuencias de diseño y veinte puntos por variable

Para esta prueba se ha mantenido un número no múltiplo del número de cores para las frecuencias de diseño y se ha aumentado el número de operaciones a realizar.

Estos son los resultados obtenidos en las diferentes ejecuciones:

Impl. / Alg.	Alg. plantillas	Alg. *e-hull*	Alg. fronteras
CPU secuencial	13185 ms	440810 ms	16789.75 ms
CPU paralelo	1497.5 ms	45624 ms	3941 ms
CUDA	-	47315 ms	1295 ms

En la última de las pruebas se puede observar que para el algoritmo de cálculo de plantillas los resultados son similares a pruebas anteriores, la versión secuencial obtiene unos tiempos de ejecución mucho más altos que la versión paralela.

Para el algoritmo *e-hull* la versión secuencial obtiene unos tiempos de ejecución muy altos comparado con las otras dos implementaciones que obtienen unos tiempos de ejecución parecidos, comportándose de forma similar a las pruebas anteriores.

Para el último de los algoritmos, el del cálculo de fronteras, la versión secuencial del algoritmo obtiene el peor de los resultados, a continuación la versión paralela en *CPU* obtiene un resultado mucho mejor manteniendo de forma estable las diferencias con la versión implementada en *GPU*.

5.4 Conclusiones del estudio experimental

Una vez realizado el estudio experimental de las diferentes versiones de los algoritmos, se puede estudiar que conclusiones se obtienen con respecto a los tiempos de ejecución recogidos.

En primer lugar, si nos centramos en el algoritmo de cálculo de plantillas, se puede observar como la versión paralela en la *CPU* obtiene en general mejores tiempos de ejecución salvo que el tamaño del problema sea muy pequeño, en cualquier caso, para este tipo de problemas el tiempo de ejecución de este algoritmo es irrelevante al ser inferior a 5 milisegundos.

Para el segundo de los algoritmos, el algoritmo *e-hull* que calcula el contorno de las plantillas, en primer lugar hacer notar que su ejecución es dependiente de las plantillas calculadas por el algoritmo anterior, de la densidad de puntos de dichas plantillas pero también de su forma. Por tanto no se puede predecir que tipo de ejecución va a tener dicho algoritmo conociendo exclusivamente la planta y las frecuencias de diseño, es necesario conocer también las plantillas de la planta.

Una vez analizados los tiempos de ejecución, se puede observar que para una sola frecuencia de diseño, la ejecución secuencial y paralela en *CPU* obtienen unos resultados similares, siendo la paralela mejor, dado que aunque no puede explotar el paralelismo en *CPU* las mejoras introducidas en el código con respecto a la versión secuencial, le hacen obtener mejores resultados. En cambio, la versión implementada en la *GPU* obtiene mejores resultados conforme se aumenta el número de operaciones a realizar dado que de esta manera puede explotar de forma más adecuada la potencia de cálculo de la tarjeta gráfica.

Si se aumentan las frecuencias de diseño, la versión paralela en *CPU* mejora sus tiempos de ejecución con respecto a la versión en la *GPU*, esto es debido a que está utilizando los cuatro cores del procesador y realizando los cálculos en paralelo, mientras que la versión en *GPU* solo dispone de una unidad funcional. En cualquier caso, se puede observar como este algoritmo no es adecuado para ejecutarse en la *GPU*, los resultados de tiempo obtenidos en general son iguales o algo peores que para la versión en la *CPU*, esto es debido, a que es un algoritmo intensivo en cálculo, pero también intensivo en tratamiento de datos, algo en lo cual la tarjeta gráfica no destaca. La justificación para haber implementado este algoritmo en la *GPU* a pesar de no obtener mejores tiempos en la versión paralela en la *CPU* es la posibilidad, que se indica como vía futura (6.2), de poder aumentar el paralelismo utilizando de forma simultanea los cores del procesador y la tarjeta gráfica, dividiendo adecuadamente los datos que deben tratar cada uno de los procesadores.

Para el último de los algoritmos, el algoritmo de cálculo de fronteras, destaca el resultado obtenido para la implementación en la *GPU* dado que siempre obtiene los mejores tiempos, independientemente del ejemplo que se esté tratando. En primer lugar hacer notar que incluso para los ejemplos más pequeños este algoritmo tiene unos tiempos de ejecución altos, esto es debido a que el algoritmo debe realizar siempre una serie de cálculos mínimos independientemente del tamaño del problema. Este algoritmo es mucho más intensivo en cálculo que el resto de los planteados, es por ello que explota de forma más adecuada la potencia de cálculo de la tarjeta gráfica. Este algoritmo es también muy

adecuado para aprovechar la capacidad masivamente paralelas de la *GPU* dado que las operaciones aritméticas a realizar no están influenciadas por ningún condicional y por tanto todos los hilos ejecutarán las mismas instrucciones siendo esta ejecución mucho más rápida que en el caso del algoritmo *e-hull* donde el número de operaciones a realizar depende de un condicional al principio de la función.

6 Conclusiones y vías futuras

6.1 Conclusiones

En este proyecto se ha abordado la paralelización y optimización de los algoritmos relacionados con las tres primeras fases de *QFT*, el objetivo a conseguir era mejorar los tiempos de ejecución de dichos algoritmos con la finalidad de proporcionar al usuario una herramienta interactiva para poder desarrollar la técnica *QFT* de la mejor manera posible. Para ello en primer lugar se planteo una metodología analizando el objetivo a conseguir en profundidad (3.1) y proponiendo que acciones concretas iban en la dirección adecuada para acercarse al objetivo planteado (3.2). Se decidió que, en primer lugar, que se paralelizarían los algoritmos en la *CPU* utilizando un sistema *fork-join* de memoria compartida, para a continuación, implementar una versión de los algoritmos *e-hull* y cálculo de fronteras para ejecutarse en la tarjeta gráfica.

A continuación se estudiaron que herramientas eran las adecuadas para utilizar en cada uno de los algoritmos, para ello se hizo un estudio en profundidad de las herramientas disponibles y poder así seleccionar la más conveniente para el objetivo a conseguir (3.3 y 4.2).Una vez hecho el estudio se decidió que para la paralelización en la *CPU* se utilizaría la librería *OpenMP* y para la implementación en la *GPU* se utilizaría el lenguaje de programación de tarjetas gráficas *CUDA*.

Una vez seleccionadas las herramientas a utilizar el siguiente paso consistió en analizar de forma teórica cada uno de los algoritmos y averiguar de esta forma cual era el procedimiento más adecuado para abordar su optimización (4.3). Estudiando para cada uno de ellos su índice de complejidad, para los peores y mejores casos, así como, que tipo de operaciones y de bucles contenían, para de esta forma poder decidir las acciones correctas a tomar para su paralelización.

Una vez realizado estos estudios el trabajo principal consistió en primer lugar en revisar el código secuencial de cada algoritmo para mejorarlo en lo posible y reducir así el número de operaciones a ejecutar. A continuación se implementaron las dos versiones paralelas planteadas, tanto en la *CPU* (4.4) como en la *GPU* (4.5), siendo esta la parte principal del proyecto y llevándose la mayor parte del tiempo dedicado al TFM.

Una vez que las diferentes implementaciones funcionaban correctamente se hizo un estudio experimental para ver que resultados se obtenían para cada uno de los algoritmos (5). El objetivo de este estudio experimental fue determinar el comportamiento de las diferentes versiones de los algoritmos para diferentes ejemplos y configuraciones, y por supuesto, determinar que mejora de tiempo se había conseguido con respecto a la versión secuencial. En las conclusiones situadas en dicho apartado (5.4) se hace un estudio de los resultados obtenidos, explicando para cada uno de los algoritmos y para las diferentes configuraciones el porque de los tiempos obtenidos.

6.2 Vías futuras

En esta sección se van a explicar las diferentes vías futuras relacionadas con este TFM, al ser un proyecto de investigación dichas vías futuras son muy amplias. A continuación se van a explicar de forma concreta cuales son las más importantes:

- Las implementaciones paralelas de los algoritmos trabajados en el proyecto corresponden a una primera versión, aunque los tiempos de ejecución han mejorado notablemente con respecto a la versión secuencial, aún se pueden mejorar continuando con el estudio de cada uno de los algoritmos. Principalmente las versiones implementadas en la *GPU* utilizando *CUDA* tienen mucho margen de mejora, tal y como se ha explicado en la sección (4.1.2) la elección de *CUDA* estaba basada en poder aprovechar la arquitectura concreta de la tarjeta gráfica con respecto a una implementación generalista, pero para ello hay que conocer en profundidad el lenguaje de programación, algo que para esta primera versión no se ha podido conseguir.

- Relacionado con el párrafo anterior se contempla la posibilidad como vía futura de hacer una implementación de los algoritmos utilizando *OpenCL* para poder así realizar un estudio experimental en profundidad. Así como poder ofrecer dos versiones de las implementaciones en la *GPU*, una que sea específica para tarjetas *Nvidia* y otra más generalista para todas las tarjetas. Por ejemplo para el algoritmo de cálculo de fronteras, aunque la versión en *OpenCL* sea más lenta que la versión en *CUDA* será mucho más rápida que las versiones en la *CPU*.

- Este TFM se ha centrado en mejorar los tiempos de ejecución de algoritmos ya diseñados, una vía futura muy importante es la posibilidad de estudiar en profundidad dichos algoritmos y plantear diseños que sean más rápidos que los actuales, esto obviamente entra dentro del campo de la investigación.

- Una vía futura que puede dar resultados muy interesantes se combinar la ejecución de los algoritmos para utilizar simultáneamente la *CPU* y la *GPU*, esto se puede realizar de forma sencilla utilizando *OpenMP* y estimando que porción de datos debe ejecutar cada una de las dos unidades funcionales.

- Una vía futura de aplicación inmediata sería la utilización del patrón *Observer*[24] para el sistema de recálculo de valores al cambiar los datos de entrada, es decir, para recalcular los resultados de todos los algoritmos en cadena. De esta forma se podría proporcionar de forma completa la interactividad con el usuario, el cual si desea cambiar cierto parámetro en cualquiera de las fases de *QFT* vea como los datos se calculan para el nuevo parámetro seleccionado.

- Una vía de futuro fundamental es estudiar la siguiente fase de *QFT*, el diseño automático del lazo, la cual está en fase de investigación con muchas partes aún no resueltas de manera satisfactoria, es una vía de investigación futura muy importante.

- Otra vía futura de investigación sería estudiar en profundidad el algoritmo *e-hull* e investigar métodos para una elección automática del parámetro *epsilon*, algo que hasta ahora no se ha conseguido.

[24] Página de información http://danielggarcia.wordpress.com/2014/06/02/patrones-de-comportamiento-vi-patron-observer/ Último acceso (22/07/2014)

A Anejo 1: GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers,

but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component,

but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

   (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   (d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

(a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

(b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

(c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

(d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

(e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

(a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

(b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works

containing it; or

(c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

(d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your

rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale,

import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree

to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

    If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

    Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY

WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

End of Terms and Conditions

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>

This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.

B Anejo 2: Código de los algoritmos estudiados

B.1 Algoritmo de cálculo de plantillas

```
1
2  QVector<QVector<complex<qreal> > * > * Templates::calcularTemplate
3 (Planta *planta, QVector<qreal> *omega){
4
5     QVector <qreal> * denominador = new QVector <qreal> ();
6     QVector <qreal> * numerador = new QVector <qreal> ();
7     QVector<QVector<qreal> * > * variables = new QVector <QVector <qreal> * >
          ();
8
9     qint32 lonNume = planta->getNumerador()->size();
10    qint32 lonDeno = planta->getDenominador()->size();
11
12    qint32 lonOmega = omega->size();
13
14    variables->reserve(lonNume + lonDeno);
15    numerador->reserve(lonNume);
16    denominador->reserve(lonDeno);
17
18    combinaciones = 1;
19    qint32 i;
20
21    for (i = 0; i < lonNume; i++){
22        QVector <qreal> * vector = getVariables(planta->getNumerador()->at(i));
23        combinaciones *= vector->size();
24        variables->insert(i,vector);
25        numerador->insert(i,vector->at(0));
26    }
27
28    qint32 salto = i;
29
30    for (qint32 i = 0; i < lonDeno; i++){
31        QVector <qreal> * vector = getVariables(planta->getDenominador()->at(i)
              );
32        combinaciones *= vector->size();
33        variables->insert(i + salto,vector);
34        denominador->insert(i,vector->at(0));
35    }
36
37    QVector <qreal> * k = getVariables(planta->getK());
38    QVector <qreal> * ret = getVariables(planta->getRet());
39
40    qint32 combT = combinaciones * k->size() * ret->size();
41    QVector<QVector<complex<qreal> > * > * temCompleto = new QVector <QVector<
          complex<qreal> > * > ();
42    temCompleto->reserve(lonOmega);
43
```

```
44      nueva_omega = new QVector <qreal> ();
45      nueva_omega->reserve(lonOmega);
46
47      #pragma omp parallel for
48      for (qint32 u = 0; u < lonOmega; u++){
49
50          QVector <qreal> * deno = new QVector <qreal> (*denominador);
51          QVector <qreal> * nume = new QVector <qreal> (*numerador);
52          QVector <qint32> * contador = new QVector <qint32> (lonDeno + lonNume +
                1, 0);
53
54          qreal w = omega->at(u);
55
56          QVector <complex<qreal>> * templateParcial = new QVector <complex<qreal
               >> ();
57          templateParcial->reserve(combT);
58
59          for (qint32 i = 0; i < combinaciones; i++){
60
61              foreach (const qreal &kT, *k) {
62                  foreach (const qreal &retT, *ret) {
63                      templateParcial->append(planta->getPunto(nume,deno,w,kT, retT
                            ));
64                  }
65              }
66
67              contador->replace(0, contador->first()+1);
68
69              bool salir = false;
70
71              for (qint32 j = 0; j < lonNume+lonDeno && salir == false;j++){
72
73                  if (contador->at(j) >= (variables->at(j)->size())){
74                      contador->replace(j,0);
75                      contador->replace(j+1, contador->at(j+1) +1);
76                  }else {
77                      salir = true;
78                  }
79
80                  if (j < lonNume){
81                      nume->replace(j,variables->at(j)->at(contador->at(j)));
82
83                  }else {
84                      deno->replace(j-lonNume,variables->at(j)->at(contador->at(j))
                            );
85
86                  }
87              }
88
89          }
```

```
90
91          #pragma omp critical
92          {
93              nueva_omega->append(w);
94              temCompleto->append(templateParcial);
95          }
96
97          deno->clear();
98          nume->clear();
99          contador->clear();
100     }
101
102     variables->clear();
103     numerador->clear();
104     denominador->clear();
105
106     return temCompleto;
107 }
```

B.2 Algoritmo $e-hull$ para el cálculo de contornos de plantillas en la *CPU*

```
1  bool Templates::calcularContorno(bool cuda){
2
3      contorno = new QVector <QVector <complex <qreal> > * > ();
4
5      bool correcto = true;
6      qint32 lon = templates->size();
7
8
9      if (!cuda){
10         QVector <qreal> * nueva_omega2 = nueva_omega;
11         nueva_omega = new QVector <qreal> ();
12
13         #pragma omp parallel for
14         for (qint32 i = 0; i < lon; i++){
15
16             QVector <complex <qreal> > * cont = e_hull(templates->at(i), epsilon
                   );
17
18             if (cont == NULL){
19                 #pragma omp critical
20                 correcto = false;
21             }
22             #pragma omp critical
23             {
24                 contorno->append(cont);
25                 nueva_omega->append(nueva_omega2->at(i));
26             }
27         }
```

```
28
29          delete nueva_omega2;
30      } else {
31          for (qint32 i = 0; i < lon; i++){
32
33              vector <complex <double> > aux = e_hull_cuda(templates->at(i)->
                    toStdVector(), epsilon);
34              QVector <complex <qreal> > aux2 = QVector<complex <qreal> >::
                    fromStdVector(aux);
35              QVector <complex <qreal> > * cont = new QVector <complex <qreal> > (
                    aux2);
36
37              if (aux.size() == 0){
38                  correcto = false;
39              }
40              contorno->append(cont);
41          }
42      }
43
44      if (!correcto){
45          contorno = NULL;
46          menerror("No se ha podido calcular el contorno de los Templates", "
                Calculo de Templates");
47          return false;
48      }
49
50      contornoCalculado = true;
51
52      return true;
53  }
54
55
56  QVector <complex <qreal> > * Templates::e_hull(QVector<complex<qreal> > *temp,
        qreal epsilon){
57
58      qint32 numDatos = temp->size();
59      qint32 MAXP = 3 * numDatos;
60
61      qint32 b1 = 0;
62      qreal numDe = -numeric_limits<qreal>::infinity();
63
64      for(qint32 i = 0;i < numDatos ; i++){ //calculamos que punto esta mas a la
            derecha y ese sera el primer punto.
65          qreal realC = real(temp->at(i));
66          if (realC > numDe){
67              b1 = i;
68              numDe = realC;
69          }
70      }
71
```

```
72      qint32 b2 = buscarSegundo(b1, temp, epsilon); //buscamos el segundo punto.
73
74      if (b2 < 0)
75          return NULL;
76
77      QVector <qint32> * resultado = new QVector <qint32>();
78
79      qint32 punto_previo = b1;
80      qint32 punto_actual = b2;
81
82      resultado->append(b1);
83      resultado->append(b2);
84
85      qreal punto_sig = buscarSiguiente(b1,b2,temp,epsilon); //a partir de aqui
            buscamos los demas puntos
86      if (punto_sig < 0)
87          return NULL;
88
89      qint32 contador = 2;
90
91
92      while (b1 != punto_actual  b2 != punto_sig){
93
94          resultado->append(punto_sig);
95          contador++;
96
97          if (contador > MAXP)
98              break;
99
100         punto_previo = punto_actual;
101         punto_actual = punto_sig;
102
103
104         punto_sig = buscarSiguiente(punto_previo, punto_actual, temp, epsilon);
105
106         if (punto_sig < 0){
107             return NULL;
108         }
109     }
110
111
112     QVector <qint32> * aux = new QVector <qint32> ();
113     aux->reserve(resultado->size());
114
115     bool isRepetido = false;
116
117     foreach (const qint32 var1, *resultado) { //Quitamos los valores repetidos
            del vector
118         foreach (const qint32 var2, *aux) {
119             if (var1 == var2){
```

```
120                 isRepetido = true;
121             }
122         }
123         if (!isRepetido){
124             aux->append(var1);
125         }
126     }
127 
128 
129     QVector <complex <qreal> > * devolver = new QVector <complex <qreal> > ();
130     devolver->reserve(aux->size());
131 
132     foreach (const qint32 var, *aux) {
133         devolver->append(temp->at(var));
134     }
135 
136     resultado->clear();
137     aux->clear();
138 
139     return devolver;
140 }
141 
142 qint32 Templates::buscarSegundo(qint32 b1, QVector<complex<qreal> > *cv, qreal
        epsilon){
143 
144     qreal dist = 0;
145     complex <qreal> primero = cv->at(b1);
146 
147     qreal fmin = numeric_limits<qreal>::infinity();
148     qint32 pmin = -1;
149     qreal aco = 0;
150     qreal dmax = 0;
151 
152     complex <qreal> cvActual;
153 
154     qreal fas = 0;
155 
156     for (qint32 i = 0; i < cv->size(); i++){ //recorremos todo el vector de
            puntos.
157 
158         cvActual = cv->at(i);
159         dist = abs(primero - cvActual); //calculamos el valor absoluto de la
                resta.
160 
161         if (dist > 0 && dist <= epsilon){ //nos quedamos con los mas pequeos.
162 
163             fas = arg (cvActual - primero); //calculamos la phase del angulo de
                    la resta de complejos
164 
165             if (fas < 0) // si dicha fase es menor que cero le sumamos 2*PI
```

```
166                 fas += 2*M_PI;
167
168             aco = qAcos(dist / epsilon); //calculamos el arcotangente del punto
                    actual dividido por epsilon
169             fas -= aco; //a la fase actual le restamos el arcotangente.
170
171             if (fas < fmin){ //vamos cual es la fase minima y la guardamos
172                 fmin = fas;
173                 pmin = i;
174                 dmax = dist;
175
176             }else if (fas == fmin && dist > dmax){ //si son iguales guardamos
                    aquella que su distancia sea mayor.
177                 pmin = i;
178                 dmax = dist;
179             }
180         }
181     }
182
183     return pmin; //retornamos el mnimo
184 }
185
186 qint32 Templates::buscarSiguiente(qint32 punto_previo, qint32 punto_actual,
187                                   QVector<complex<qreal> > *cv, qreal epsilon){
188     complex <qreal> actual = cv->at(punto_actual);
189     complex <qreal> anterior = cv->at(punto_previo);
190
191     qreal aco2 = qAcos(abs(anterior-actual) / epsilon);
192
193     qreal fasActual = 0;
194
195     qreal aco1Actual = 0;
196     qreal dmax = 0;
197
198     qreal psiActual = 0;
199
200
201     qreal psiMinActual = numeric_limits<qreal>::infinity();
202     qint32 posPsiMin = -1;
203
204     complex <qreal> cvActual;
205     qreal absResta;
206
207     for (qint32 i = 0; i < cv->size(); i++){
208
209         cvActual = cv->at(i);
210         absResta = abs(cvActual - actual); //Calculamos el valor absoluto de la
                distancia del punto al punto actual.
211
212
```

```
213         if (absResta > 0 && absResta <= epsilon && cvActual != actual &&
                cvActual != anterior){
214         // Si la distancia es mayor que cero y menor que epsilon.
215
216
217             //calculamos la fase entre los dos puntos normalizada.
218             fasActual = arg((cvActual - actual) / (anterior - actual));
219
220             if(fasActual < 0) // si es menor que cero le sumamos 2*PI.
221                 fasActual = fasActual + 2*M_PI;
222
223             aco1Actual = qAcos(absResta / epsilon ); //calculamos la
                    arcotangente de dicho valor absoluto
224
225             if(fasActual == 0){ // Vamos a calcular psi la fase, hay tres tipos
                    de calculos distintos
226                 psiActual = 2*M_PI - aco1Actual - aco2;
227             }else if (fasActual > 0 && fasActual < aco2){
228                 psiActual = fasActual + aco1Actual- aco2;
229             }else{
230                 psiActual = fasActual - aco1Actual - aco2;
231             }
232
233             if (psiActual < 0) // Si alguno es negativo se sumamos 2*PI
234                 psiActual += 2*M_PI;
235
236             if (psiActual < psiMinActual) { //Buscamos el psi minimo
237                 psiMinActual = psiActual; //como puede haber iguales nos
                        quedamos con el de la
238                 posPsiMin = i; //distancia mas grande.
239                 dmax = absResta;
240             }else if (psiActual == psiMinActual &&
241                       (absResta > dmax)){
242                 posPsiMin = i;
243                 dmax = absResta;
244             }
245         }
246     }
247
248     return posPsiMin; //retornamos el mnimo
249
250 }
```

B.3 Algoritmo $e-hull$ para el cálculo de contornos de plantillas en la *GPU*

```
1  vector <complex <double> > e_hull_cuda(vector<complex<double> > puntos, float
       epsilon){
2
3     // Error code to check return values for CUDA calls
```

```
4     cudaError_t err = cudaSuccess;
5
6     vector <int> indice_contorno;
7
8     // Creamos los tamaos de los distintos vectores
9     int numElements = puntos.size();
10    size_t size_complex = numElements * sizeof(Complex);
11    size_double = numElements * sizeof(double);
12    blocksPerGrid = (numElements + threadsPerBlock - 1) / threadsPerBlock;
13
14    //Creamos el vector de complejos
15    Complex * puntos_h = (Complex *) malloc(size_complex);
16
17    //Creamos el vector del host para buscar el primer nmero
18    host_vector<double> vector_buscar_primero_h (numElements);
19
20    for (int i = 0; i < numElements; i++){
21        puntos_h[i].x = puntos.at(i).real();
22        puntos_h[i].y = puntos.at(i).imag();
23
24        vector_buscar_primero_h[i] = puntos.at(i).real();
25    }
26
27    int posicion_primero = buscar_primero(vector_buscar_primero_h);
28
29    indice_contorno.push_back(posicion_primero);
30
31    //Guardamos el primero nmero complejo del contorno
32    Complex complejo_primero;
33    complejo_primero.x = puntos.at(posicion_primero).real();
34    complejo_primero.y = puntos.at(posicion_primero).imag();
35
36    Complex * puntos_d = NULL;
37    err = cudaMalloc((void **)&puntos_d, size_complex);
38
39    if (err != cudaSuccess)
40    {
41        cerr << "Error al reservar memoria en CUDA (error code " <<
              cudaGetErrorString(err) << ")!" << endl ;
42        exit(EXIT_FAILURE);
43    }
44
45    //Copiamos a la memoria de CUDA los datos.
46    cudaMemcpy(puntos_d, puntos_h, size_complex, cudaMemcpyHostToDevice);
47
48    int posicion_segundo = buscar_segundo(puntos_d, complejo_primero, epsilon,
          numElements);
49
50    if (posicion_segundo < 0){
51        cudaFree(puntos_d);
```

```
52          free(puntos_h);
53          cout << "tres" << endl;
54
55          vector <complex <double> > vector_nulo;
56          return vector_nulo;
57      }
58
59      indice_contorno.push_back(posicion_segundo);
60
61      //Preparamos el complejo segundo.
62      Complex complejo_segundo;
63      complejo_segundo.x = puntos.at(posicion_segundo).real();
64      complejo_segundo.y = puntos.at(posicion_segundo).imag();
65
66
67      int punto_previo = posicion_primero;
68      int punto_actual = posicion_segundo;
69
70
71      //Creamos las estructuras necesarias para guardar el retorno del kernel
72      double * retorno_siguiente_d = NULL;
73      cudaMalloc((void **)&retorno_siguiente_d, size_double);
74
75      int punto_siguiente = buscar_siguiente(complejo_primero, complejo_segundo,
            puntos_d,
76                                             epsilon, numElements,
                                                   retorno_siguiente_d);
77
78      if (punto_siguiente < 0){
79          cudaFree(retorno_siguiente_d);
80          cudaFree(puntos_d);
81          free(puntos_h);
82
83          vector <complex <double> > vector_nulo;
84          return vector_nulo;
85      }
86
87      int contador = 2;
88
89      int MAXP = numElements;
90
91      while (posicion_primero != punto_actual  posicion_segundo !=
            punto_siguiente){
92
93          indice_contorno.push_back(punto_siguiente);
94          contador++;
95
96          if (contador > MAXP){
97              break;
98          }
```

```
99
100         punto_previo = punto_actual;
101         punto_actual = punto_siguiente;
102
103         //Preparamos los nmeros complejos anteriores
104         Complex complejo_previo;
105         complejo_previo.x = puntos.at(punto_previo).real();
106         complejo_previo.y = puntos.at(punto_previo).imag();
107
108
109         Complex complejo_actual;
110         complejo_actual.x = puntos.at(punto_actual).real();
111         complejo_actual.y = puntos.at(punto_actual).imag();
112
113         punto_siguiente = buscar_siguiente(complejo_previo, complejo_actual,
                puntos_d,
114                                            epsilon, numElements,
                                                retorno_siguiente_d);
115
116
117         if (punto_siguiente < 0){
118             cudaFree(retorno_siguiente_d);
119             cudaFree(puntos_d);
120             free(puntos_h);
121
122             vector <complex <double> > vector_nulo;
123             return vector_nulo;
124         }
125
126
127     }
128
129
130
131     cudaFree(retorno_siguiente_d);
132     cudaFree(puntos_d);
133     free(puntos_h);
134
135     vector <int> aux;
136     bool isRepetido = false;
137
138     for (int i = 0; i < indice_contorno.size(); i++) {
139         for (int j = 0;j < aux.size(); j++) {
140             if (indice_contorno.at(i) == indice_contorno.at(j)){
141                 isRepetido = true;
142                 break;
143             }
144         }
145         if (!isRepetido){
146             aux.push_back(indice_contorno.at(i));
```

```
147         }
148     }
149 
150     vector <complex <double> > puntos_contorno;
151 
152     for (int i = 0; i < aux.size(); i++){
153         puntos_contorno.push_back(puntos.at(aux.at(i)));
154     }
155 
156     return puntos_contorno;
157 }
158 
159 int buscar_primero(host_vector<double> vector_buscar_primero_h){
160     //copiamos los valores al vector del device para buscar el primer nmero
161     device_vector<double> vector_buscar_primero_d = vector_buscar_primero_h;
162 
163     //Buscamos el mximo.
164     device_vector<double>::iterator iter = max_element(vector_buscar_primero_d.
        begin(), vector_buscar_primero_d.end());
165 
166     //Recuperamos la posicin del complejo primero.
167     int posicion_primero = iter - vector_buscar_primero_d.begin();
168 
169     return posicion_primero;
170 }
171 
172 int buscar_segundo (Complex * puntos_d, Complex complejo_primero, float epsilon
    , int numElements){
173 
174     //Asignamos memoria el vector de fases de CUDA
175     double * retorno_segundo_d = NULL;
176     cudaMalloc((void **) &retorno_segundo_d, size_double);
177 
178     // Lanzamos la ejecucin del kernel
179 
180     buscar_segundo_kernel <<<blocksPerGrid, threadsPerBlock>>> (
        complejo_primero, puntos_d, epsilon,
181                                                                 retorno_segundo_d,
                                                                    numElements);
182     cudaGetLastError();
183 
184     device_ptr<double> d_vec (retorno_segundo_d);
185     device_ptr<double> d_vec_final = d_vec + numElements;
186     device_ptr<double> result = min_element(d_vec, d_vec_final);
187 
188     //Recuperamos la posicin del valor mnimo.
189     unsigned int posicion_segundo = result - d_vec;
190 
191     double min_val = *result;
192 
```

```
193     cudaFree(retorno_segundo_d);
194 
195     if (min_val == inf){
196         return -1;
197     }
198 
199     return posicion_segundo;
200 }
201 
202 inline int buscar_siguiente (Complex complejo_primero, Complex complejo_segundo
        , Complex * puntos_d, float epsilon, int numElements, double *
        retorno_siguiente_d){
203     // Lanzamos la ejecucin del kernel
204 
205     buscar_siguiente_kernel <<<blocksPerGrid, threadsPerBlock>>> (
            complejo_primero, complejo_segundo, puntos_d, epsilon,
206                                                                 retorno_siguiente_d
                                                                        , numElements);
207 
208     cudaGetLastError();
209 
210     device_ptr<double> d_vec (retorno_siguiente_d);
211     device_ptr<double> d_vec_final = d_vec + numElements;
212     device_ptr<double> result = min_element(d_vec, d_vec_final);
213 
214     //Recuperamos la posicin del valor mnimo.
215     unsigned int posicion_siguiente = result - d_vec;
216 
217     double min_val = *result;
218 
219     if (min_val == inf){
220         return -1;
221     }
222 
223     return posicion_siguiente;
224 }
225 
226 static __global__ void buscar_segundo_kernel(Complex primero, Complex * puntos,
         float epsilon,
227                                              double * retorno, int n_puntos)
228 {
229     int i = blockDim.x * blockIdx.x + threadIdx.x;
230 
231     if (i < n_puntos){
232 
233         double dist = abs(puntos[i] - primero);
234 
235         if (dist > 0 && dist <= epsilon){
236 
237             retorno[i] = arg(puntos[i] - primero);
```

```
238
239             if (retorno[i] < 0){
240                 retorno[i] += 2 * pi;
241             }
242
243             retorno[i] -= acos(dist / epsilon);
244         } else {
245             retorno[i] = inf;
246         }
247     }
248 }
249
250 static __global__ void buscar_siguiente_kernel(Complex primero, Complex segundo
    , Complex * puntos, float epsilon,
251                                                double * retorno, int n_puntos)
252 {
253     int i = blockDim.x * blockIdx.x + threadIdx.x;
254
255     if (i < n_puntos){
256
257         double dist = abs(puntos[i] - segundo);
258
259         if (dist > 0 && dist <= epsilon && puntos[i] != primero){
260
261             double fase = arg((puntos[i]- segundo) / (primero - segundo));
262
263             if (fase < 0){
264                 fase += 2 * pi;
265             }
266
267             double aco1 = acos(dist / epsilon);
268             double aco2 = acos(abs((primero - segundo)) / epsilon);
269
270             if (fase == 0){
271                 retorno[i] = 2 * pi - aco1 - aco2;
272             } else if (fase < aco2){
273                 retorno[i] = fase + aco1 - aco2;
274             } else{
275                 retorno[i] = fase - aco1 - aco2;
276             }
277
278             if (retorno[i] < 0){
279                 retorno[i] += 2 * pi;
280             }
281
282         } else {
283             retorno[i] = inf;
284         }
285     }
286 }
```

```

//Funciones para operar con nmeros complejos.
static __device__ __host__ inline double abs(Complex a)
{
    return hypot(a.x, a.y);
}

static __device__ __host__ inline Complex operator-(Complex a, Complex b)
{
    a.x = a.x - b.x;
    a.y = a.y - b.y;
    return a;
}

static __device__ __host__ inline double arg(Complex a)
{
    return atan2(a.y, a.x);
}

static __device__ __host__ inline Complex operator/(Complex a, Complex b)
{
    Complex c;
    double i = pow(b.x,2) + pow(b.y,2);
    c.x = (a.x * b.x + a.y * b.y)/i;
    c.y = (a.y * b.x - a.x * b.y)/i;
    return c;
}

static __device__ __host__ inline bool operator!=(Complex a, Complex b)
{
    if ((a.x != b.x) && (a.y != b.y))
        return true;
    return false;
}
```

B.4 Algoritmo de cálculo de fronteras en *CPU*

```
void Boundaries::bnd(QVector<qreal> *omega, Planta *planta, QVector <QVector <
    complex <qreal> > *>
                * templates, QPointF * datosFas, qint32 puntosFas,
                QPointF * datosMag, qint32 puntosMag, qreal infinito)
{
    // Se genera la rejilla base del algoritmo.
    QVector <qreal> * fases = linspace(datosFas->x(), datosFas->y(), puntosFas)
        ;
    QVector <qreal> * mag = linspace(datosMag->x(), datosMag->y(), puntosMag);

    qreal inf;

    //Si el valor de infinito ha sido introducido por el usuario.
    if (infinito < 0){
```

```
13          inf = numeric_limits<qreal>::infinity();
14      }else {
15          inf = infinito;
16      }
17
18      //Se recorren las frecuencias de diseo.
19      #pragma omp parallel for
20      for (qint32 i = 0; i < omega->size(); i++){
21
22          //Se crean las variables necesarias
23          complex <qreal> p0;
24
25          //Se obtiene cada plantilla.
26          QVector <complex <qreal> > * p = templates->at(i);
27          //Se resuelve la planta con las frecuencias de diseo.
28          p0 = planta->getPunto(omega->at(i));
29
30          //Una sbana por cada frecuencia de diseo.
31          QVector <QVector <qreal> * > * sabanaEstabilidadRuido = new QVector <
                QVector <qreal> * > ();
32          sabanaEstabilidadRuido->reserve(fases->size());
33
34          QVector <QVector <qreal> * > * sabanaSeguimiento = new QVector <QVector
                <qreal> * > ();
35          sabanaSeguimiento->reserve(fases->size());
36
37          QVector <QVector <qreal> * > * sabanaRPS = new QVector <QVector <qreal>
                * > ();
38          sabanaRPS->reserve(fases->size());
39
40          QVector <QVector <qreal> * > * sabanaRPE = new QVector <QVector <qreal>
                * > ();
41          sabanaRPE->reserve(fases->size());
42
43          QVector <QVector <qreal> * > * sabanaEC = new QVector <QVector <qreal>
                * > ();
44          sabanaEC->reserve(fases->size());
45
46          //Se recorre la rejilla
47          #pragma omp parallel for
48          for (qint32 j = 0; j < fases->size(); j++){ // f
49
50              QVector <qreal> * vectorEstabilidadRuido = new QVector <qreal> ();
51              vectorEstabilidadRuido->reserve(mag->size());
52
53              QVector <qreal> * vectorSeguimiento = new QVector <qreal> ();
54              vectorSeguimiento->reserve(mag->size());
55
56              QVector <qreal> * vectorRPS = new QVector <qreal> ();
57              vectorRPS->reserve(mag->size());
```

```
58
59          QVector <qreal> * vectorRPE = new QVector <qreal> ();
60          vectorRPE->reserve(mag->size());
61
62          QVector <qreal> * vectorEC = new QVector <qreal> ();
63          vectorEC->reserve(mag->size());
64
65          #pragma omp parallel for
66          for (qint32 k = 0; k < mag->size(); k++){ //l
67
68              //variables necesarias
69              qreal l = mag->at(k);
70              qreal f = fases->at(j);
71
72              //Se pasa la magnitud a lineal
73              qreal maglineal = pow(10,l/20);
74              //Se calcula el nmero complejo correspondiente a la posicin de
                    la rejilla.
75              //Se pasa de Nichols a lineal.
76              complex <qreal> L = complex<qreal> (maglineal * cos (f * M_PI
                    /180),
77                                                  maglineal * sin (f * M_PI /180))
                                                      ;
78
79              QVector <QPointF * > * templateMovido = new QVector <QPointF * >
                     ();
80
81              qreal valorAnterior = -0.00001;
82
83              //Se recorre la plantilla y se calcula la plantilla
84              //movida por el punto de la rejilla.
85
86              qreal dEstabilidadRuido = -numeric_limits<qreal>::infinity();
87              qreal dRPS = dEstabilidadRuido;
88              qreal dRPE = dEstabilidadRuido;
89              qreal dEC = dEstabilidadRuido;
90
91
92              qreal dSeguimiento = numeric_limits<qreal>::infinity();
93
94              qreal maX = dEstabilidadRuido, maY = dEstabilidadRuido, meX =
                    dSeguimiento, meY = dSeguimiento;
95
96              for (qint32 h = 0; h < p->size(); h++) { // temp
97                  //Clculo principal.
98                  complex <qreal> complejoMovido = p->at(h) * L / p0;
99                  //Fase del ngulo del nmero complejo en grados.
100                 qreal fase = arg(complejoMovido)* 180 / M_PI;
101                 //Se llama a la funcin unwrap.
102                 fase = unwrap(valorAnterior, fase);
```

```
103
104                 valorAnterior = fase;
105
106                 //Se guardan los puntos de la plantilla movida.
107                 QPointF * par = new QPointF(20*log10(abs(complejoMovido)),
                        fase);
108                 templateMovido->append(par);
109
110                 //Clculo necesario que se guarda en el template
111
112                 complex <qreal> p_actual = p->at(h);
113
114                 //Estabilidad y ruido del sensor
115                 qreal dTempEstabilidadRuidoSeguimiento = 20 * log10 (abs (L /
                        ((p0 / p_actual) + L)));
116
117                 //Rechazo de perturbaciones a la salida de la planta
118                 qreal dTempRPS = 20 * log10 (abs ((p0 / p_actual) / ((p0 /
                        p_actual) + L)));
119
120                 //Rechado de perturbaciones a la entrada de la planta
121                 qreal dTempRPE = 20 * log10 (abs (p0 / (p0 / p_actual + L)));
122
123                 //Esfuerzo de control
124                 qreal dTempEC = 20 * log10 (abs ((L / p_actual) / (p0 /
                        p_actual + L)));
125
126                 if (dTempEstabilidadRuidoSeguimiento > dEstabilidadRuido){
127                     dEstabilidadRuido = dTempEstabilidadRuidoSeguimiento;
128                 } else if (dTempEstabilidadRuidoSeguimiento < dSeguimiento){
129                     dSeguimiento = dTempEstabilidadRuidoSeguimiento;
130                 }
131                 if (dTempRPS > dRPS){
132                     dRPS = dTempRPS;
133                 }
134                 if (dTempRPE > dRPE){
135                     dRPE = dTempRPE;
136                 }
137                 if (dTempEC > dEC) {
138                     dEC = dTempEC;
139                 }
140
141                 if (par->x() > maX){
142                     maX = par->x();
143                 }else if (par->x() < meX){
144                     meX = par->x();
145                 }
146
147                 if (par->y() > maY){
148                     maY = par->y();
```

```
149                 }else if (par->y() < meY){
150                     meY = par->y();
151                 }
152             }
153 
154             if ((((-180 < maX && -180 > meX) && (0 < maY && 0 > meY))
155                     ((180 < maX && 180 > meX) && (0 < maY && 0 > meY))) &&
156                     pnpoly(templateMovido, new QPointF(0, -180))){
157                 #pragma omp critical
158                 {
159                     vectorEstabilidadRuido->append(inf);
160                     vectorSeguimiento->append(inf);
161                     vectorRPS->append(inf);
162                     vectorRPE->append(inf);
163                     vectorEC->append(inf);
164                 }
165 
166             }else{
167                 #pragma omp critical
168                 {
169                     vectorEstabilidadRuido->append(dEstabilidadRuido);
170                     vectorSeguimiento->append(dEstabilidadRuido -
                            dSeguimiento);
171                     vectorRPS->append(dRPS);
172                     vectorRPE->append(dRPE);
173                     vectorEC->append(dEC);
174                 }
175             }
176             templateMovido->clear();
177         }
178 
179         #pragma omp critical
180         {
181             sabanaEstabilidadRuido->append(vectorEstabilidadRuido);
182             sabanaSeguimiento->append(vectorSeguimiento);
183             sabanaRPS->append(vectorRPS);
184             sabanaRPE->append(vectorRPE);
185             sabanaEC->append(vectorEC);
186         }
187     }
188 
189     #pragma omp critical
190     {
191         vectorSabanasEstabilidadRuido->append(sabanaEstabilidadRuido);
192         vectorSabanasSeguimiento->append(sabanaSeguimiento);
193         vectorSabanasRPS->append(sabanaRPS);
194         vectorSabanasRPE->append(sabanaRPE);
195         vectorSabanasEC->append(sabanaEC);
196     }
197 }
```

```
198 }
199
200 bool Boundaries::pnpoly(QVector<QPointF *> *vector, QPointF *punto)
201 {
202     qint32 i, j;
203     qint32 lon = vector->size();
204     bool c = false;
205     for (i = 0, j = lon-1; i < lon; j = i++) {
206         if (((vector->at(i)->y() > punto->y()) != (vector->at(j)->y() > punto->
            y())) &&
207             (punto->x() < (vector->at(j)->x()) - vector->at(i)->x() *
208              (punto->y()-vector->at(i)->y()) / (vector->at(j)->y()-vector->
                 at(i)->y()) +
209              vector->at(i)->x()) )
210             c = !c;
211     }
212
213     delete punto;
214     return c;
215 }
216
217 //Se normaliza a valores [-180,180):
218 qreal Boundaries::constrainAngle(qreal x){
219     x = fmod(x + M_PI,2 * M_PI);
220     if (x < 0)
221         x += 2 * M_PI;
222     return x - M_PI;
223 }
224
225 //Se convierte a valores [-360,360]
226 qreal Boundaries::angleConv(qreal angle){
227     return fmod(constrainAngle(angle),2 * M_PI);
228 }
229 qreal Boundaries::angleDiff(qreal a,qreal b){
230     qreal dif = fmod(b - a + M_PI,2 * M_PI);
231     if (dif < 0)
232         dif += 2 * M_PI;
233     return dif - M_PI;
234 }
235 qreal Boundaries::unwrap(qreal previousAngle,qreal newAngle){
236     //Hay que pasar el ngulo actual y el anterior.
237     return previousAngle - angleDiff(newAngle,angleConv(previousAngle));
238 }
```

B.5 Algoritmo de cálculo de fronteras en *GPU*

```
1 double * bnd_cuda (vector <complex <double> >templates, complex <double> p0,
      double infinito,
2                    vector <double> fas, vector <double> mag){
3
4     int nElementos = templates.size();
```

```
5
6     size_t tamano = nElementos * sizeof (Complex);
7     size_t tamano_double = sizeof (double);
8     int fas_size = fas.size();
9     int mag_size = mag.size();
10
11    double infinito_sabana;
12
13    if (infinito < 0){
14        infinito_sabana = numeric_limits<double>::infinity();
15    }else {
16        infinito_sabana = infinito;
17    }
18
19    //Creamos los vectores para pasar a cuda.
20    Complex * p_h = (Complex *) malloc (tamano);
21    double * fas_h = fas.data();
22    double * mag_h = mag.data();
23
24    for (int i = 0; i < nElementos; i++){
25        p_h[i].x = templates.at(i).real();
26        p_h[i].y = templates.at(i).imag();
27    }
28
29    //Copiamos la memoria del dispositivo los datos necesarios.
30    cudaError_t err = cudaSuccess;
31
32    Complex * p_d = NULL;
33    err = cudaMalloc((void **) &p_d, tamano);
34
35    if (err != cudaSuccess)
36    {
37        cerr << "Error al reservar memoria en CUDA (error code " <<
              cudaGetErrorString(err) << ")!" << endl ;
38        exit(EXIT_FAILURE);
39    }
40
41    cudaMemcpy(p_d, p_h, tamano, cudaMemcpyHostToDevice);
42
43    double * fas_d = NULL;
44    cudaMalloc((void **) &fas_d, fas_size * tamano_double);
45    cudaMemcpy(fas_d, fas_h, fas_size * tamano_double, cudaMemcpyHostToDevice);
46
47    double * mag_d = NULL;
48    cudaMalloc((void **) &mag_d, mag_size * tamano_double);
49    cudaMemcpy(mag_d, mag_h, mag_size * tamano_double, cudaMemcpyHostToDevice);
50
51    int sabana_size = fas_size * mag_size * tamano_double;
52
53    double * sabana_d = NULL;
```

```
54      cudaMalloc((void **) &sabana_d, sabana_size);
55
56      int tamano_memoria = fas_size * nElementos * tamano_double;
57
58      Complex * memoria_d = NULL;
59      cudaMalloc((void **) &memoria_d, tamano_memoria);
60
61
62
63      Complex p0_d; p0_d.x = p0.real(), p0_d.y = p0.imag();
64
65      int threadsPerBlock = 128;
66      int blocksPerGrid = (fas_size + threadsPerBlock - 1) / threadsPerBlock;
67
68      bnd_kernel <<<blocksPerGrid, threadsPerBlock>>>(fas_d, mag_d,
            infinito_sabana, mag_size,
69                                                  fas_size, p_d, p0_d , nElementos
                                                        , sabana_d, memoria_d);
70
71      cudaGetLastError();
72
73      double * sabana_h = (double *) malloc (sabana_size);
74      memset(sabana_h, 0 , sabana_size);
75
76      cudaMemcpy(sabana_h, sabana_d, sabana_size, cudaMemcpyDeviceToHost);
77
78      cudaFree(p_d);
79      cudaFree(fas_d);
80      cudaFree(mag_d);
81      cudaFree(sabana_d);
82      cudaFree(memoria_d);
83
84      free(p_h);
85
86      return sabana_h;
87  }
88
89
90  static __global__ void bnd_kernel(double *fases, double *mag, double infinito,
        int puntos_mag, int puntos_fas,
91                                    Complex * p, Complex p0, int nElementos, double
                                          * sabana, Complex * memoria){
92
93      int i = blockDim.x * blockIdx.x + threadIdx.x;
94
95      int contador, contador2;
96      double fase_anterior, fase;
97      Complex L, complejo;
98      double dTemp;
99      double d;
```

```
100     double maX, maY, meX, meY;
101
102
103     if (i < puntos_fas) {
104         for (contador = 0; contador < puntos_mag; contador++){
105
106             L.x = pow(10.,mag[contador]/20.) * cos (fases[i] * M_PI /180.);
107             L.y = pow(10.,mag[contador]/20.) * sin (fases[i] * M_PI /180.);
108
109             fase_anterior = -0.1;
110             d = -infinito;
111             maX = infinito, maY = infinito, meX = -infinito, meY = -infinito;
112
113             for (contador2 = 0; contador2 < nElementos; contador2++){
114
115
116                 complejo = p[contador2] * L / p0;
117
118                 fase = unwrap(fase_anterior, arg(complejo) * 180. / M_PI);
119
120                 complejo.x = 20. * log10(abs(complejo));
121                 complejo.y = fase;
122
123                 memoria [i * nElementos + contador2] = complejo;
124
125                 dTemp = 20. * log10 (abs(L / ((p0 / p[contador2]) + L)));
126
127                 if (dTemp > d){
128                     d = dTemp;
129                 }
130
131                 if (complejo.x > maX){
132                     maX = complejo.x;
133                 } else if (complejo.x < meX){
134                     meX = complejo.x;
135                 }
136
137                 if (complejo.y > maY){
138                     maY = complejo.y;
139                 } else if (complejo.y < meY){
140                     meY = complejo.y;
141                 }
142
143             }
144
145             if ((((-180 < maX && -180 > meX) && (0 < maY && 0 > meY))
146                  ((180 < maX && 180 > meX) && (0 < maY && 0 > meY))) &&
147                     pnpoly(memoria, nElementos, i * nElementos)){
148                 sabana [i * puntos_mag + contador] = infinito;
149             } else {
```

```
150                 sabana [i * puntos_mag + contador] = d;
151             }
152         }
153     }
154 }
155
156 static __device__ __host__ inline bool pnpoly(double2 memoria [], int lon, int
        pivote)
157 {
158     int i, j;
159     bool c = false;
160     double2 punto;
161     punto.x = 0;
162     punto.y = -180;
163     for (i = 0, j = lon-1; i < lon; j = i++) {
164         if (((memoria [pivote + i].y > punto.y) != (memoria [pivote + j].y >
            punto.y)) &&
165             (punto.x < (memoria [pivote + j].x) - memoria [pivote + i].x *
166              (punto.y-memoria [pivote + i].y) / (memoria [pivote + j].y-
                 memoria [pivote + i].y) +
167              memoria [pivote + i].x) )
168             c = !c;
169     }
170
171     return c;
172 }
173
174 //Funciones para operar con nmeros complejos.
175 static __device__ __host__ inline double abs(Complex a)
176 {
177     return hypot(a.x, a.y);
178 }
179
180 static __device__ __host__ Complex inline operator*(Complex a, Complex b)
181 {
182     Complex c;
183     c.x = (a.x * b.x - a.y * b.y);
184     c.y = (a.x * b.y + a.y * b.x);
185     return c;
186 }
187
188 static __device__ __host__ inline Complex operator/(Complex a, Complex b)
189 {
190     Complex c;
191     double i = pow(b.x,2) + pow(b.y,2);
192     c.x = (a.x * b.x + a.y * b.y)/i;
193     c.y = (a.y * b.x - a.x * b.y)/i;
194     return c;
195 }
196
```

```
197 static __device__ __host__ inline double arg(Complex a)
198 {
199     return atan2(a.y, a.x);
200 }
201 
202 static __device__ __host__ inline Complex operator+(Complex a, Complex b)
203 {
204     a.x = a.x + b.x;
205     a.y = a.y + b.y;
206     return a;
207 }
208 
209 //Se normaliza a valores [-180,180):
210 static __device__ __host__ inline double constrainAngle(double x){
211     x = fmod(x + M_PI,2 * M_PI);
212     if (x < 0)
213         x += 2 * M_PI;
214     return x - M_PI;
215 }
216 
217 //Se convierte a valores [-360,360]
218 static __device__ __host__ inline double angleConv(double angle){
219     return fmod(constrainAngle(angle),2 * M_PI);
220 }
221 
222 static __device__ __host__ inline double angleDiff(double a,double b){
223     double dif = fmod(b - a + M_PI,2 * M_PI);
224     if (dif < 0)
225         dif += 2 * M_PI;
226     return dif - M_PI;
227 }
228 
229 static __device__ __host__ inline double unwrap(double previousAngle,double
        newAngle){
230     //Hay que pasar el ngulo actual y el anterior.
231     return previousAngle - angleDiff(newAngle,angleConv(previousAngle));
232 }
```

Bibliografía

[Bailey y Hui [1989]] F. N. Bailey y C. H. Hui [1989]. *A fast algorithm for computing rational functions.* 34. IEEE.

[Bernabé et al. [2012]] G. Bernabé, G. D. Guerrero y J. Fernandez. CUDA and OpenCL implementations of 3D Fast Wavelet Transform. 4. Universidad de Murcia [2012].

[Bode [1945]] H. W. Bode [1945]. *Network analysis and feedback amplifier design / H.W. Bode*. Princeton, N.J. : Van Nostrand.

[Borghesani et al. [2003]] C. Borghesani, Y. Chait y O. Yaniv [2003]. *The QFT Frequency Domain Control Design Toolbox For Use with MATLAB*. Terasoft, Inc. Último acceso (2/08/2013).

[Chen y Ballance [1998]] W. Chen y D. J. Ballance. Automatic loop-shaping in qft using genetic algorithms. Inf. téc. [1998].

[Dormido et al. [2004]] S. Dormido, J. Aranda y J. Díaz [2004]. *An Interactive Software Tool for Learning Robust Control Design Using Quantitative Feedback Theory Methodology*. International Journal of Engineering Education.

[Fang et al. [2011]] J. Fang, A. L. Varbanescu y H. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. 216–225. IEEE [2011].

[Garcia-Sanz et al. [2009]] M. Garcia-Sanz, A. Mauch y C. Philippe [2009]. *QFT Control Toolbox: an Inter-active Object-Oriented Matlab CAD tool for Quantitative Feedback Theory*. 6th IFAC Symposium on Robust Control Design.

[Gera y Horowitz [1980]] A. Gera y I. Horowitz [1980]. Optimization of the Loop Transfer Function. *Int. J. of Control* **31**.

[Gutman [1996a]] P. Gutman [1996a]. *Qsyn - the Toolbox for Robust Control Systems Design for use with Matlab, Reference Guide*. El-Op Electro-Optics Industries Ltd, Rehovot, Israel.

[Gutman [1996b]] P. Gutman [1996b]. *Qsyn - the Toolbox for Robust Control Systems Design for use with Matlab, User's Guide.*. El-Op Electro-Optics Industries Ltd, Rehovot, Israel.

[Horowitz y Sidi [1972]] I. Horowitz y M. Sidi [1972]. *Synthesis of feedback systems with large plant ignorance for prescribed time-domain tolerances*. Int. J. Control 2 ed$^{\underline{ón}}$.

[Horowitz [1959]] I. M. Horowitz [1959]. *Fundamental theory of automatic linear feedback control systems*. Automatic Control, IRE Transactions on.

[Horowitz [1963]] I. M. Horowitz [1963]. *Synthesis of feedback systems*. Academic Press, New York.

[Karimi et al. [2010]] K. Karimi, N. G. Dickson y F. Hamze [2010]. A Performance Comparison of CUDA and OpenCL. *CoRR*.

[Martínez [2013]] I. Martínez. *QFTbx, herramienta de diseño QFT: especificación de requisitos y prototipado*. Proyecto Fin de Carrera Universidad de Murcia [2013].

[Montoya [1998]] F. J. Montoya. *Diseño de sistemas de control no lineales mediante QFT: análisis computacional y desarrollo de una herramienta CACSD* . Tesis Doctoral [1998].

[Moreno et al. [2006]] J. C. Moreno, A. Baños y M. Berenguel [2006]. Improvements on the computation of boundaries in QFT. *International Journal of Robust and Nonlinear Control* **16**, nº 12: 575–597.

[Nordin [1993]] M. Nordin. *Uncertain systems with backslash: modeling, identification and synthesis*. Proyecto Fin de Carrera Royal Institute of Technology [1993].