UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA

INVERSO MEDIANTE GPU PARA IMAGEN ELECTROCARDIOGRÁFICA. RESOLUCIÓN DEL PROBLEMA

Autor: Diego Paracuellos de los Santos

Tutor: Miguel Rodrigo Bort

Co-tutor: Alejandro Liberos Mascarell

Trabajo Fin de Master presentado en el Departamento de Ingeniería Electrónica de la Universitat Politècnica de València, para la obtención del Master Universitario en Ingeniería de Sistemas Electrónicos.

Curso 2018-19

Valencia, julio de 2019

Resumen:

Las enfermedades cardiovasculares son un problema de creciente repercusión, siendo las arritmias cardiacas una de las más comunes en los casos de ingreso hospitalario y en la población en general. La gran carga social y económica que suponen estas enfermedades cardiacas propicia que se desarrollen nuevos métodos de diagnóstico y tratamiento que ayuden a paliar el problema. Uno de ellos es la Imagen Electrocardiográfica (ECGI), proceso con un alto grado de procesamiento de señal que permite reconstruir de manera no invasiva la actividad eléctrica cardíaca, y con ello realizar un diagnóstico y tratamiento más preciso. No obstante, el alto grado de procesamiento requerido supone que esta técnica no su utilice en la práctica clínica, por entre otros motivos el largo tiempo de espera para obtener resultados.

Dentro de lo que es el ECGI, se pueden extraer dos procesos de especial carga computacional. El primero es el Problema Inverso (IP), que permite reconstruir de manera no invasiva los Electrogramas (EGM) en la superficie cardiaca a partir de Electrocardiogramas (ECG). El segundo es el cálculo de Frecuencias Dominantes (DFs), que permite la detección de frecuencias de activación cardíacas en los potenciales cardiacos reconstruidos y con ello la caracterización electrofisiológica de las arritmias.

En las últimas décadas, han surgido diferentes dispositivos con capacidades computacionales superiores a los de las CPUs, las Unidades de Procesamiento Gráfico (GPU), que permiten acelerar el procesado mediante su ejecución paralela mediante la cual se segmenta un algoritmo en unidades menores que se ejecutan en paralelo formando una solución equivalente al algoritmo original.

En este estudio, se analizan las posibilidades de aplicar este concepto para acelerar la ejecución de los algoritmos de IP y DFs con el fin de igualar y/o mejorar las prestaciones de dichos algoritmos en cuanto a tiempo de ejecución.

Bajo este contexto, se pretende emplear el framework de CUDA para ello. Bajo el mismo, y empleando Visual Studio y MATLAB como apoyos para el desarrollo, se han realizado las implementaciones ligadas a los algoritmos citados, ofreciendo este documento un análisis básico de los procesos empleados para dicha implementación.

El resultado de estas implementaciones ha sido un aumento general en las prestaciones temporales de los algoritmos, con mejoras medias en los tiempos en GPU respecto de CPU de entre 3x y 18x, en función del algoritmo. Ello ha sido posible sin introducir un error significativo en el cálculo, caracterizándose la actividad electrofisiológica con una precisión de entre el 100% y el 98.39% respecto al procesado CPU.

Palabras Clave: Procesos no-invasivos, Paralelización de algoritmos, ECG, EGM, GPU, CUDA.

Resum:

Les malalties cardiovasculars són un problema de creixent repercussió, sent les arítmies cardíaques una de les més comunes en els casos d'ingrés hospitalari i en la població en general. La gran càrrega social i econòmica que suposen estes malalties cardíaques propícia que es desenrotllen nous mètodes de diagnòstic i tractament que ajuden a pal·liar el problema. Un d'ells és la Imatge Electrocardiográfica (ECGI), procés amb un alt grau de processament de senyal que permet reconstruir de manera no invasiva l'activitat elèctrica cardíaca, i amb això realitzar un diagnòstic i tractament més precís. No obstant això, l'alt grau de processament requerit suposa que esta tècnica no el seu utilitze en la pràctica clínica, per entre altres motius el llarg temps d'espera per a obtindre resultats.

Dins del que és l'ECGI, es poden extraure dos processos d'especial càrrega computacional. El primer és el Problema Invers (IP), que permet reconstruir de manera no invasiva els Electrogramas (EGM) en la superfície cardíaca a partir d'Electrocardiogrames (ECG). El segon és el càlcul de Freqüències Dominants (DFs), que permet la detecció de freqüències d'activació cardíaques en els potencials cardíacs reconstruïts i amb això la caracterització electrofisiològica de les arítmies.

En les últimes dècades, han sorgit diferents dispositius amb capacitats computacionals superiors als de les CPUs, les Unitats de Processament Gràfic (GPU), que permeten accelerar el processat per mitjà de la seua execució paral·lela per mitjà de la qual se segmenta un algoritme en unitats menors que s'executen en paral·lel formant una solució equivalent a l'algoritme original.

En este estudi, s'analitzen les possibilitats d'aplicar este concepte per a accelerar l'execució dels algoritmes d'IP i DFs a fi d'igualar y/o millorar les prestacions dels dits algoritmes quant a temps d'execució. Davall este context, es pretén emprar el framework de CUDA per a això.

Davall el mateix, i emprant Visual Studio i MATLAB com a suports per al desenrotllament, s'han realitzat les implementacions lligades als algoritmes esmentats, oferint este document una anàlisi bàsica dels processos empleats per a la dita implementació.

El resultat d'estes implementacions ha sigut un augment general en les prestacions temporals dels algoritmes, amb millores mitges en els temps en GPU respecte de CPU d'entre 3x i 18x, en funció de l'algoritme. Això ha sigut possible sense introduir un error significatiu en el càlcul, caracteritzant-se l'activitat electrofisiològica amb una precisió d'entre el 100% i el 98.39% respecte al processat CPU.

Paraules Clau: Processos no-invasius, Paralelización d'algoritmes, ECG, EGM, GPU, CUDA.

Abstract:

Cardiovascular diseases are a problem of increasing impact, with arrhythmias being more important in cases of hospital admission and in the general population. The great social and economic burden posed by these heart diseases leads to the development of new methods of diagnosis and treatment that help to alleviate the problem. One of them is the Electrocardiographic Image (ECGI), a process with a high degree of signal processing that allows the cardiac electrical activity to be reconstructed in a non-invasive way, and with this, a diagnosis and a more precise treatment. However, in clinical practice, for other reasons, in the long waiting time to obtain results.

Within what the ECGI is, two special computational load processes can be extracted. The first is the Inverse Problem (IP), which allows a non-invasive operation on electrograms (EGM) on the cardiac surface from Electrocardiograms (ECG). The second is the calculation of the dominant frequencies (DFs), which allows the detection of cardiac activation frequencies in reconstructed cardiac potentials and with it the electrophysiological characteristic of arrhythmias.

In the latest news, different devices with computational capabilities superior to CPUs have arisen, the Graphical Processing Units (GPU), which allow you to pass the process through parallel execution through which an algorithm is segmented in the units that are executed in parallel, building an equivalent solution to the original algorithm.

In this study, we analyze the possibilities of applying this concept to accelerate the execution of the IP and DF algorithms in order to equalize and / or improve the performance of these algorithms in terms of execution time.

Under this context, it is intended to use the CUDA framework for this. Under the same, and using Visual Studio and MATLAB as support for development, implementations linked to the mentioned algorithms have been made, it is a basic analysis of the processes used for this function.

The result of these implementations has been a general increase in the temporal performance of the algorithms, with improvements in GPU times regarding CPUs between 3x and 18x, depending on the algorithm. This has been possible without introducing a significant error in the process, characterizing the electrophysiological activity with a precision of between 100% and 98.39% with respect to the CPU process.

Keywords: Non-invasive processes, Parallelization of algorithms, ECG, EGM, GPU, CUDA.

Compendio de abreviaturas empleadas.

AP: Action Potential

BLAS: Basic Linear Algebra Subroutines

CPU: Central Processing Unit.

CUDA: Compute Unified Device Architecture.

DFs: Dominant Frequency.

DLL: Dynamic-Link Library.

DSP: Digital Signal Processor.

ECG: Electrocardiogram

ECGI: Electrocardiographic/Electrocardiogram Imaging

EGM: Electrogram

GPU: Graphic Processing Unit.

GPGPU: General-Purpose Processing on Graphic Processing Unit.

HDL: Hardware Description Language.

INTOPS: Integer Operations.

IP: Inverse Problem

FLOPS: Floating Point Operations.

FFT: Fast Fourier Transform.

FPGA: Field Programmable Gate Array.

HOPS: Half-Precision Operations.

MAC: Multiply and Acumulate.

VHDL: VHSIC + HDL.

VHSIC: Very High Speed Integrated Circuit.

Contenido

Capítulo 1. Introducción

1.1 Introducción

Las arritmias cardiacas son una de las enfermedades cardiovasculares más extendidas e importantes en el entorno socio-sanitario. Actualmente, entorno al 65% de los ingresos hospitalarios por arritmia están relacionados con la Fibrilación Auricular (AF) [1]. Se estima que aproximadamente el 4.4% de la población está en riesgo de sufrir Arritmia Cardiaca [2], siendo uno de los marcadores de riesgo la edad, aunque no el único.

Estos ingresos conllevan unos costes sanitarios que están relacionados directamente con los procedimientos empleados y la duración de la estancia clínica. En ocasiones, dichos procedimientos conllevan el uso de técnicas invasivas, como el cateterismo, que consiste en la visualización del corazón de manera directa, introduciendo un catéter hacia el corazón por vía intravenosa.

Existen alternativas, de manera no invasiva, como la medición de Electrocardiogramas (ECGs) en el entorno torácico y mediante algoritmos computacionales, transformar esos ECGs y mapear las señales que se desean medir en la superficie cardiaca. Un ejemplo de esta técnica es el Mapeo de Potenciales en la Superficie Corporal, en inglés *Body Surface Potential Mapping (BSPM)*. Mediante este proceso, y empleando un algoritmo denominado Problema Inverso (IP), es posible reconstruir las señales de potencial en el entorno cardiaco o Electrogramas (EGMs), sin necesidad de realizar intrusiones en el cuerpo ni sedación. El conjunto de medición y procesado de señales abarca parte de lo que se conoce como Imagen Electrocardiográfica, del inglés *Electrocardiographic Imaging (ECGI)*. Esta técnica, el ECGI, permite reconstruir de manera no invasiva las señales cardiacas internas (EGM) a partir del potencial eléctrico registrado en la superficie del torso (ECG) y de la anatomía del torso y del corazón del paciente.

Por otro lado, con las señales reconstruidas mediante ECGI, es posible emplear técnicas algorítmicas para procesarlas, y encontrar marcadores que permitan caracterizar las diferentes arritmias cardiacas. Una de estas técnicas es la detección de las frecuencias dominantes (DFs), que permiten extraer la frecuencia de activación de las señales calculadas mediante el IP.

No obstante, estos procedimientos, en ocasiones conllevan un coste computacional que difícilmente es asumible, tanto a nivel de adquisición, de aproximadamente 30-45 minutos, como de procesado, de aproximadamente 10-20 minutos, que sumados pueden frenar las posibilidades de implementación o desarrollo de dichas técnicas en la práctica clínica.

La programación secuencial ha desarrollado un incremento progresivo en sus capacidades computacionales desde su origen, aumentando las frecuencias de operación y la densidad de integración de puertas lógicas, pero, en la actualidad, nos encontramos con la problemática de un estancamiento en sus capacidades, debido a problemas varios como las frecuencias de operación, la potencia consumida y la capacidad de integración a nivel de puerta de los mismos.

Por este motivo, en las últimas décadas, aparecieron una vertiente en temas de computación con un enfoque diferente, conocida como computación paralela, que consiste básicamente de realizar todas aquellas acciones posibles, de manera simultánea, aplicando diferentes técnicas para transformar los cálculos secuenciales en cálculos paralelos.

Bajo estos precedentes, este estudio pretende implementar y comprobar el funcionamiento de los algoritmos del IP y DFs bajo un entorno paralelo en particular, las Unidades de Procesamiento Gráfico (GPUs) al ser un entorno que permite el procesado paralelo con una aplicación directa.

1.2 Objetivos

La finalidad del siguiente estudio, es analizar las posibilidades de aceleración de los algoritmos de reconstrucción del Problema Inverso (IP) y del cálculo de frecuencias dominantes (DFs) en un entorno controlado de paralelización de algoritmos base, en este caso una GPU.

Por ello, el objetivo fundamental de este trabajo puede ser definido como:

- Reducir el tiempo de ejecución de los algoritmos IP y DFS.

Para alcanzar este objetivo, se han marcado las siguientes tareas que se desarrollaran en este estudio:

- Implementar los algoritmos en un entorno paralelo.
- Realizar una comparación de los resultados del modelo paralelo y el original.
- Realizar una comparación temporal de los modelos.
- Realizar una optimización base de dichos algoritmos.

Para ello, se ha fijado un flujo base a la hora de implementar y comprobar los algoritmos en cuestión.

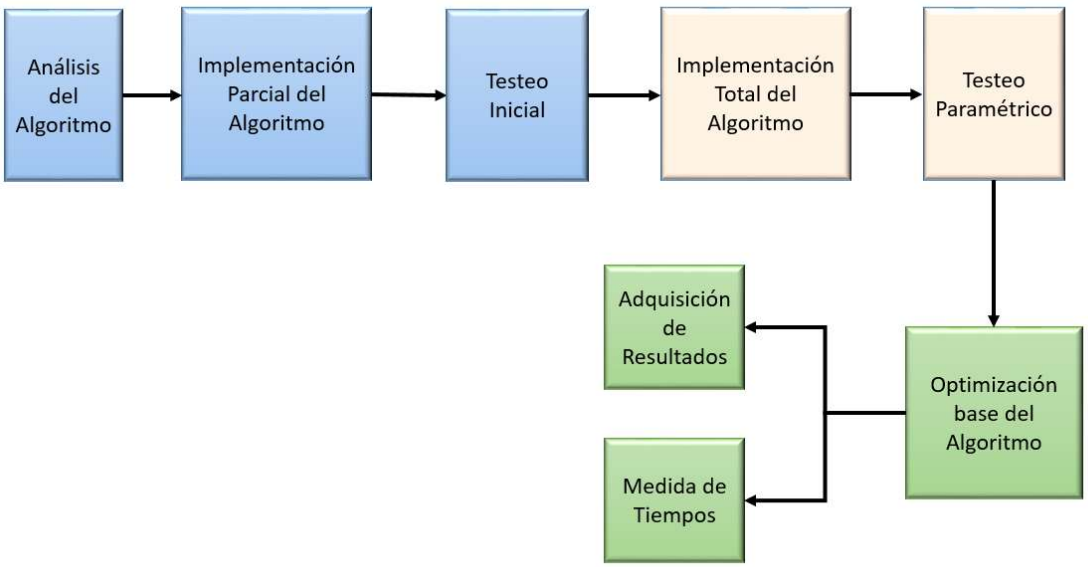


Figura 1. Flujo de trabajo

Este flujo de trabajo, puede definirse en tres pasos básicos:

1- Implementación base del algoritmo, resaltada en azul, donde se estudiaría el algoritmo y se realizaría una implementación funcional básica del mismo junto a una comprobación inicial.

2- Implementación general del algoritmo, en naranja, en la que se generalizaría el algoritmo, y se realizaría un testeo a nivel general.

3- Optimización del algoritmo, resaltado en verde, con el fin de optimizar el algoritmo para intentar reducir los tiempos de ejecución.

Mediante estas tres etapas, en un principio, se debería observar alguna mejora en los tiempos de computación de los algoritmos, siempre y cuando sea posible.

Capítulo 2. Marco Teórico

2.1. Métodos diagnósticos no invasivos en cardiología clínica

2.1.1. Introducción al comportamiento eléctrico del corazón y a las arritmias cardiacas

El corazón es una entraña muscular, localizada dentro de la cavidad torácica del cuerpo de un ser vivo, cuya función principal es el bombeo de sangre para la distribución de oxígeno y nutrientes al resto del cuerpo.

Este bombeo se produce por la contracción provocada por el conjunto de impulsos de carácter eléctrico dados de manera individualizada en las células miocárdicas, que reciben el nombre de Potenciales de Acción (AP).

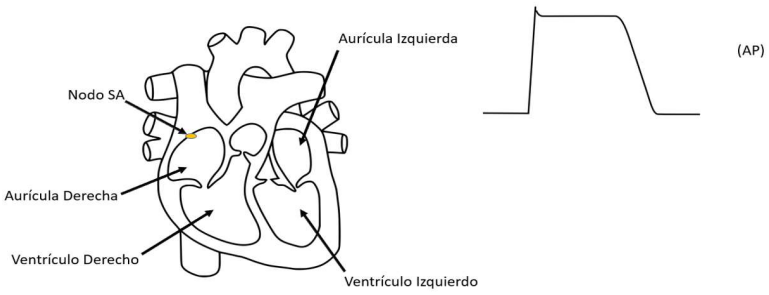


**Imagen original extraída de Pixabay*

Figura 2. Corazón y Potenciales

De manera general, en un corazón sano, la activación de los AP viene fijada por la intervención del nodo sinoauricular (SA) que actúa de marcapasos biológico, con una frecuencia base entre 60 y 100 activaciones por minuto, siendo el caso en el que los AP no siguen la tendencia fijada del SA lo que se conoce como Arritmia cardiaca.

2.1.2. Electrocardiograma

Se conoce como electrocardiograma al método de representación gráfica de los potenciales eléctricos del corazón en función del tiempo.

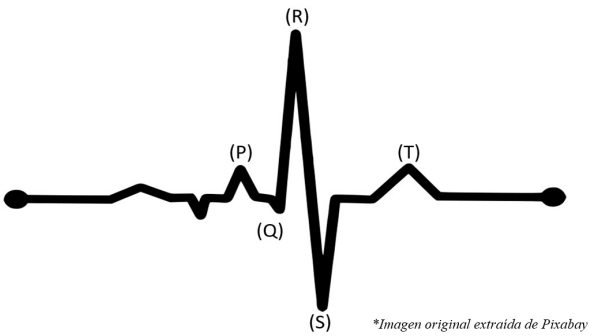
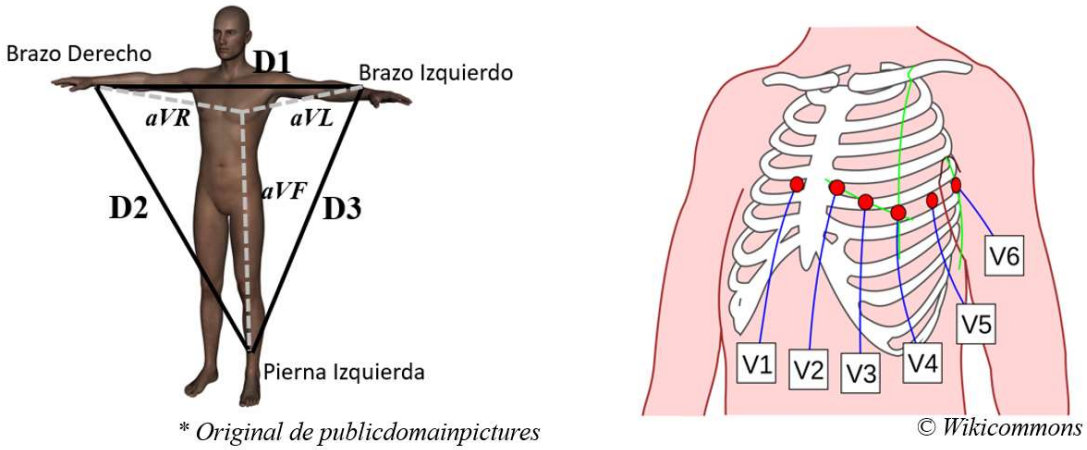


**Imagen original extraída de Pixabay*

Figura 3. Electrocardiograma

Aunque existen diferentes métodos de obtención de la señal, de manera general, se realiza una serie de operaciones sobre los potenciales recibidos en ambas extremidades superiores junto al encontrado en la pierna izquierda, en las que se colocan tres electrodos y se calcularan las derivaciones I, II y III, también conocidas como D1, D2 y D3 o como Derivaciones de Einthoven, correspondientes a la diferencia de potencial medido entre los electrodos, tal y como se muestra en la siguiente figura.



(a) Derivaciones Principales y Aumentadas (b) Derivaciones Precordiales

Figura 4. Métodos de adquisición típicos

La técnica clásica de registro de Electrocardiograma emplea 12 derivaciones para una mayor precisión y/o reconstrucción parcial del potencial en el entorno cardiaco para la detección de determinadas afecciones cardiacas. Se realiza una medición desde cada electrodo extremo a la forma compuesta del resultado de los otros dos electrodos, obteniendo tres derivaciones adicionales denominadas aVR, aVL y aVF, como se muestra en la imagen superior (4), recibiendo comúnmente el nombre de derivaciones aumentadas.

Adicionalmente, se emplean otros seis electrodos alrededor de la cavidad torácica, formando seis derivaciones adicionales, desde V1 a V6, comúnmente denominadas derivaciones precordiales.

2.1.3. Registro eléctrico de superficie (BSPM) y problema inverso

Aunque para muchas de las afecciones cardiacas la técnica de 12 derivaciones puede ser suficiente, existen arritmias cardiacas para cuya diagnosis y/o caracterización se ha demostrado que el uso de más derivaciones superficiales es potencialmente beneficioso [3].

Como solución a este problema, se propuso un método de registro cardiaco con una alta resolución espacial, bautizado como registro eléctrico de superficie, del inglés, *Body Surface Potential Mapping (BSPM).*

El BSPM consiste en la obtención del potencial eléctrico en un numero entre 32 y 256 de puntos repartidos alrededor de la superficie torácica del cuerpo, que podríamos denominar señales del Electrocardiograma (ECG), con el fin de, tras tratar dichas señales, obtener las señales en la superficie cardiaca (EGM) mediante el cálculo del problema inverso. Este procedimiento permite, de manera no invasiva, realizar una reconstrucción de alta precisión de la región cardiaca, hasta el punto de estar siendo utilizada para guiar procedimientos de ablación.

La siguiente imagen [4], muestra dos posibles distribuciones de electrodos en el torso de un paciente.

Figura 5. Ejemplos de posicionamiento de electrodos para BSPM

Existe una relación entre las señales ECG y EGM, que permiten el cálculo de ECG a partir de EGMs y viceversa, lo que se conoce como problema directo y problema inverso del electrocardiograma.

Problema Directo: ECG = M * EGM

Problema Inverso: $icEGM = M^{-1} * ECG$

Donde, *M* es la matriz de transferencia entre el torso y la aurícula, y ECG/ECM son las señales en torso y aurícula respectivamente.

En este estudio, se implementará la parte relacionada con el problema inverso para el cálculo de los potenciales en la superficie de la aurícula a partir de los potenciales en la superficie del torso.

2.1.4. Desarrollo matemático del problema inverso

El punto de partida de este estudio, desde un punto de vista de algorítmica, es lo que se ha definido como problema inverso del electrocardiograma. Este método se basa en la obtención del potencial eléctrico cardiaco (icEGM) a partir de las señales eléctricas superficiales registradas (ECG) y de la matriz de transferencia (M). El cálculo de la matriz de transferencia se realiza a partir de los modelos anatómicos de torso y aurícula del paciente [5].

En este caso, de una serie de potenciales en la superficie y la relación de los mismos con los potenciales en el entorno cardiaco, reconstruir el potencial en este último.

$$icEGM = M^{-1} * ECG$$

Por lo general, el problema inverso se considera mal condicionado, al no cumplir las normas de los problemas bien definidos que son la existencia, exclusividad y estabilidad de las potenciales soluciones a los problemas, siendo esta última la que por lo general se vulnera. Para ello, se puede recurrir a la introducción de un parámetros de regularización (λ) para el problema como medio de reformular el problema en forma discreta*[5, 6]*.

$$icEGM(\lambda) = (M^T * M + \lambda * I * I^T)^{-1} * M^T * ECG$$

.

Donde $I * I^T$ puede ser simplificado por las propiedades de la matriz identidad a una I equivalente.

$$icEGM(\lambda) = (M^T * M + \lambda * I)^{-1} * M^T * ECG$$

Este método, sin embargo, puede causar un sobreajuste en la solución al problema, lo que anularía la finalidad del método proporcionando en este caso la solución a un problema diferente al inicial ($M \approx I$).

Para evitar esta problemática, se realiza un barrido del parámetro de regularización para encontrar mediante la denominada curva L [7], siendo el valor máximo de su pendiente, que corresponde al valor optimo del parámetro de regularización, en el que se minimizan los dos errores involucrados en el problema inverso: el mal condicionamiento y el sobreajuste.

$$curvatura(\lambda) = \frac{\frac{dx}{d\lambda} * \frac{d^2y}{d\lambda^2} - \frac{dy}{d\lambda} * \frac{d^2x}{d\lambda^2}}{\left|\left(\frac{d^2x}{d\lambda^2} + \frac{d^2y}{d\lambda^2}\right)\right|^{\frac{3}{2}}}$$

Siendo;

$$x = \log\|\mathrm{M} \cdot \mathrm{icEGM}(\lambda) - \mathrm{ECG}\|^2$$

$$y = \log\|\mathrm{I} \cdot \mathrm{icEGM}(\lambda)\|^2$$

En el que el valor óptimo de regularización corresponde al máximo de la curvatura de esta curva, que minimiza el error cometido en la inversión de la matriz de transferencia.

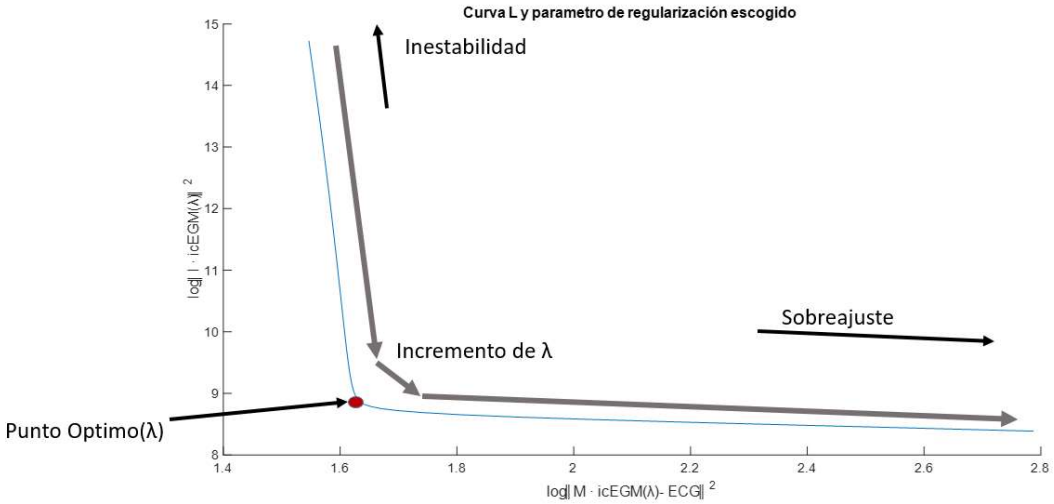


Figura 6. Curva L

2.1.5. Métodos de post-procesado del problema inverso (DF)

Uno de los puntos importantes una vez tenemos un conjunto de señales de potencial en la superficie cardiaca, es el conocer la frecuencia de activación cardiaca de cada una de las regiones del corazón mapeadas. Para ello, existen varias técnicas, como puede ser la detección de activaciones en el dominio temporal, para aproximar la frecuencia, que, aunque son computacionalmente rápidas, pueden incurrir en grandes errores de estimación *[8]*.

En nuestro caso, hemos acudido a el desarrollo frecuencial mediante el promediado de FFTs en lo que se conoce como periodograma o método de Welch. El mismo, consiste en segmentar la señal original en un numero entero de sub-señales del mismo tamaño, con un porcentaje de solapamiento entre sus valores para posteriormente realizar la FFT enventanada de cada uno de

esos segmentos y promediar el valor cuadrático del módulo de esos sub-segmentos, siguiendo las siguientes ecuaciones *[5, 9, 10, 11]*:

$$x_i(n) = x(n + i * D);$$

$$x_{iw}(n) = x_i(n) * w(n)$$

$$P_{x_i,M}(w_k) = \frac{1}{LU}|FFT_{NFFT}(x_{iw(n)})|^2 \; ; \; U = \frac{\Sigma|w(n)|^2}{N_W}$$

$$S_x^w(w_k) = \frac{1}{K}\Sigma_{i=0}^{K-1} P_{x_{i,M}}(w_k)$$

Para este caso, además, se incluye un promediado de Lomb-Scargle en el que no se analiza la señal original, sino lo que se opera es la señal sustrayéndole su media. El resultado así, será un conjunto de valores correspondientes a un rango de frecuencias entre $[0, 2\pi]$ en el que visualizaríamos el espectro enventanado de la señal.

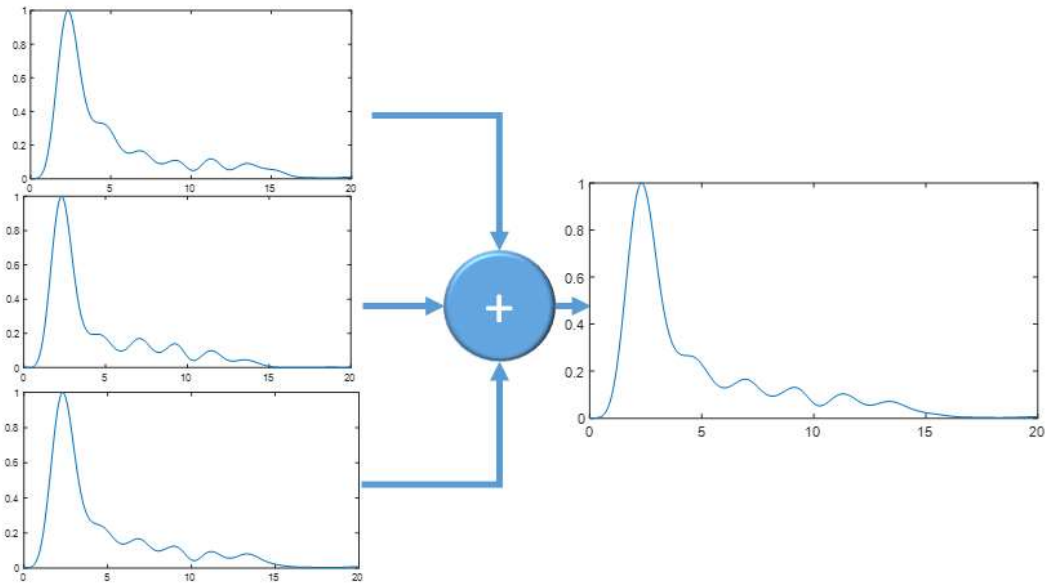


Figura 7. Ejemplo de Periodograma para 3 señales

Puesto que la señal resultante posee simetría par respecto a la frecuencia 0, y conocemos la frecuencia de muestreo f_s de la señal original, podríamos obviar la parte correspondientes a frecuencias negativas obteniendo un rango entre $[0, \pi]$ y re-escalar dicho rango a $[0, \frac{f_s}{2}]$.

Sin embargo, la información que nos interesa no estará en este rango completo, sino entre una f_{min} y una f_{max} en el que se analizara el valor máximo de dicho sub-rango y esa será la frecuencia de pulsación buscada. Esto es debido a que solo se buscan frecuencias de activación en el rango de lo fisiológicamente posible, entre unos 0.5 y 12 Hercios o activaciones por segundo.

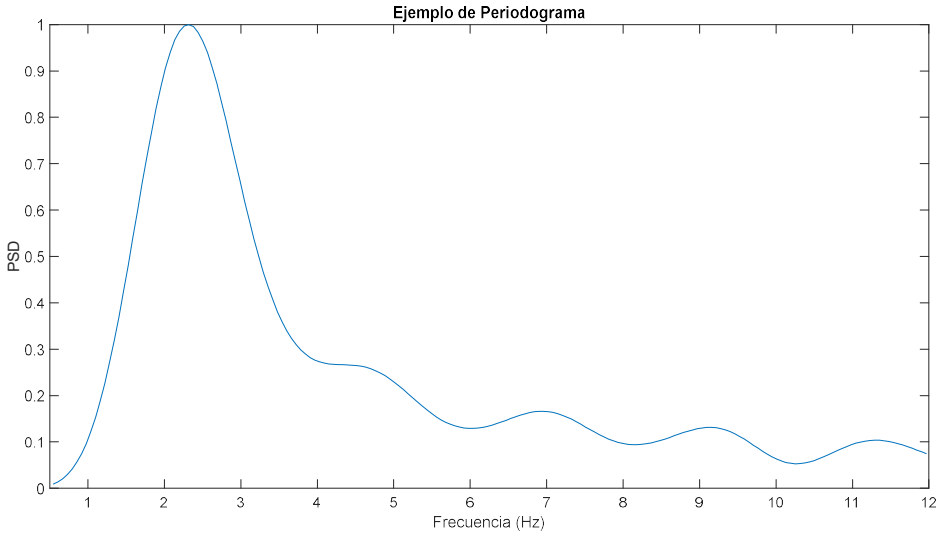


Figura 8. Aumento de una zona de interés donde se aprecia el máximo detectado.

2.2. Procesado GPGPU

2.2.1. Procesado Paralelo

Como ya se ha introducido, de manera tradicional la computación ha seguido un esquema secuencial, es decir la ejecución de rutinas una tras otra sin solaparse temporalmente y siguiendo una secuencia fija.

Sin embargo, conforme la cantidad de información a procesar y/o la complejidad de la misma ha ido aumentando, y han surgido necesidades de carácter temporal, la necesidad de nuevos modelos de computación se hizo evidente. Es en este punto donde se concibió la necesidad de realizar muchas operaciones en el tiempo que hasta entonces se realizaba una, creándose el concepto de procesado paralelo.

El procesado paralelo se podría definir, en este pretexto, como la ejecución simultánea de diversas instrucciones dentro de un entorno dado. Su funcionalidad es, de manera básica, acelerar la ejecución de procesos a coste de aumentar la cantidad de hardware empleado, en general en término de procesadores y memoria.

Para que un proceso sea paralelizable, estrictamente, debe cumplir un requisito básico.

- Que sea segmentable en subprocesos de menor complejidad.

No obstante, en muchas ocasiones, es posible modificar los algoritmos para transformarlos en un proceso aparentemente paralelizable. Para estos casos se suele incluir un requisito adicional relacionado con el primer requisito descrito anteriormente:

- Si la complejidad del conjunto de los cálculos paralelos supera a la complejidad del problema secuencial, se considera el algoritmo de Tiempo Polinomial no Determinista (NP).

Esto quiere decir que, si el problema paralelo es de mayor complejidad que el problema secuencial, la ganancia en tiempo de la implementación paralela respecto a la secuencial (Speed-Up) puede ser limitada, nula o inversa debido al aumento de la complejidad de cálculos. Un ejemplo de este tipo de algoritmos son los relacionados con Problemas de Evaluación de Circuitos (CVP), en los que para calcular la salida de una puerta conocemos su tipo y sus entradas. En este caso, para conocer las entradas posteriores en una estructura en árbol, es necesario conocer todas

las salidas anteriores, por lo que, en función de la profundidad del árbol y distribución de los elementos, la paralelización puede no aportar ninguna mejora.

Dentro del concepto de ejecución simultánea se pueden definir diferentes niveles de paralelismo:

- A nivel de bit, en el que dentro de una instrucción se paraleliza el procesado de la misma, como puede ser en una suma o resta.
- A nivel de instrucción, en el que las instrucciones se reordenan y paralelizan siguiendo una secuencia ordenada, siendo el caso del denominado pipelining.
- A nivel de datos, En el que un conjunto de datos se distribuye entre varias unidades de procesamiento, siempre que no exista dependencia de datos entre sus operaciones.
- A nivel de tareas, en el que diferentes sub-conjuntos de instrucciones pueden paralelizarse dentro de un procesador.

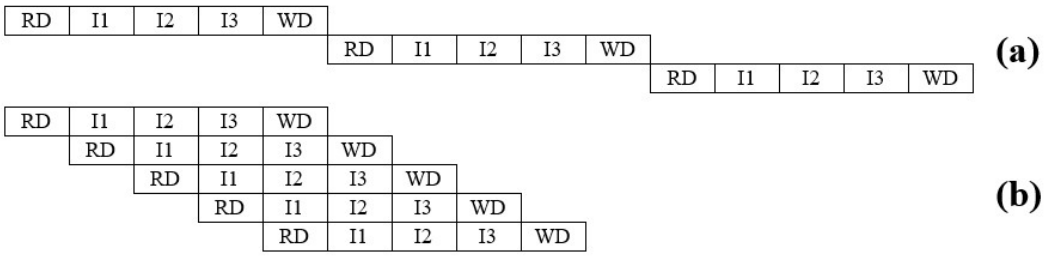


Figura 9. Arquitecturas

En la figura (8) se muestran los diferentes tipos de paralelización, correspondiendo al caso (a) la arquitectura secuencial estándar, en la que una determinada funcionalidad tardaría cinco ciclos en ejecutarse, al caso (b) a una paralelización a nivel de instrucción, en el que la cadencia del programa se ha reducido a un ciclo.

La capacidad de paralelización, no obstante, estará ligada a las capacidades del dispositivo a emplear, los que, en adelante si poseen dichas capacidades, denominaremos procesadores paralelos.

2.2.2. Procesadores paralelos

Como ya se ha introducido, existen una serie de procesadores computacionales con la capacidad de paralelizar, en mayor o menor grado, las operaciones que realizan.

Dentro de este concepto, podemos encontrar principalmente cuatro grandes familias de dispositivos con las que es posible lograr dicha paralelización. Estos son las CPU multinúcleo, las unidades de procesamiento Gráfico (GPU), los procesadores digitales de señal (DSP) y las matrices de puerta programable (FPGA).

Existen diferencias notables en diversos aspectos, como su potencia computacional alcanzable, los niveles de abstracción y complejidad de su programación o el consumo potencial, que ha conseguido un posicionamiento de alguno respecto a los demás en diferentes campos. A continuación, se recopila la información disponible de algunos de estos dispositivos para su posterior comparación:

Tipo	Dispositivo	Computo[1] (TFLOPS)	Tecnología (nm)	Memoria (Mb)	Tipo	Consumo	Lenguaje	Coste
FPGA	Stratix X [12]	10*	16	310	M20K	Moderado	Verilog/VHDL	23.000€
FPGA	Ultrascale [13]	4.9	20	115	RAM	Moderado	Verilog/VHDL	6.000€
FPGA	Virtex 7 [14]	3*	28	68	RAM	Moderado	Verilog/VHDL	2.500€
FPGA	Cyclone 10 [15]	0.173	20	13	M20K	Bajo	Verilog/VHDL	100€
CPU	9900k[2] [16]	1.23	14	16	Cache L3	Moderado	C++/Assembler	400 €
CPU	Xeon 8180 [17]	8	14	38	Cache L3	Moderado	C++/Assembler	11.000€
CPU	2700[2] [18]	1.67	7	16	Cache L3	Moderado	C++/Assembler	750 €
DSP	ADSP-TS101S [19]	1.8	28	6	SRAM	Bajo	C++/Assembler	231€
DSP	66AK2H14 [20]	0.198	28	6	SRAM	Bajo	C++/Assembler	700€
GPU	RTX 2080 [21]	10.5*	12	8192	GDDR6	Alto	C++/Python	700€
GPU	GTX 960 [22]	2.4	28	4096	GDDR4	Alto	C++/Python	250€
GPU	P6000 [23]	12	16	24576	GDDR5X	Alto	C++/Python	4.000 €
GPU	RTX 6000 [24]	16*	12	24576	GDDR6	Alto	C++/Python	4.000€

Tabla 1. Comparativa de productos para computación Paralela. Información extraída generalmente de sus referentes páginas de información

[1] *La capacidad de cómputo de algunos dispositivos no incluye operaciones adicionales. La capacidad real se puede considerar superior. Marcados con *.*

[2] *En el 2700 de AMD y el 9900k de Intel, referente a la GPU Integrada. No existe información fiable.*

- La capacidad de cómputo de esta tabla está referida a la capacidad teórica de pico en precisión simple.
- El coste de esta tabla indica precios estándar comerciales.
- Se ha considerado solo la memoria de mayor capacidad.

La tabla anterior, recopila la información de los dispositivos tal y como podrían adquirirse de por sí. La gran mayoría de los mismos, requerirán de un conjunto adicional de hardware (sistema) a su alrededor para funcionar, ya sea una PCB con componentes como memoria externa y puertos I/O, como puede ser con las FPGAs y DSPs, o un servidor o estación de trabajo con las CPUs y GPUs.

En ese sentido no se han contabilizado dichos elementos, puesto que añadirían un punto de complejidad innecesaria a este análisis. Además, se han obviado temas como el tamaño del sistema final al considerarse irrelevantes para el estudio y la aplicación.

De la tabla anterior, podemos sacar una métrica con interés para este análisis:

Tipo	Dispositivo	Potencia (TFlops)	Precio (€)	Potencia/Precio (GFlops/€)	Precio/Potencia (€/GFlops)
FPGA	Stratix X	10	23.000	0.43478261	2.3
FPGA	Ultrascale	4.9	6.000	0.75	1.33
FPGA	Virtex 7	3	2.500	1.2	0.83
FPGA	Cyclone 10	0.173	100	1.73	0.57
CPU	9900k	1.23	400	3.07	0.32
CPU	Xeon 8180	8	11.000	0.72	1.38
CPU	2700	1.67	750	2.22	0.45
DSP	ADSP-TS101S	1.8	231	7.79	0.12
DSP	66AK2H14	0.198	700	0.28	3.57
GPU	RTX 2080	10.5*	700	15	0.06
GPU	GTX 960	2.4	250	9.6	0.11
GPU	P6000	12	4.000	3	0.33
GPU	RTX 6000	16*	4.000	4	0.25

Tabla 2. Ratio Potencia/Precio.

**La capacidad de cómputo de algunos dispositivos no incluye operaciones adicionales.*

Observamos que, en este sentido, la GPU RTX 2080 es la que ofrece mejor ratio potencia/precio de entre todos estos dispositivos, siendo la siguiente mejor opción, fuera de las GPUs, el DSP ADSP-TS101S con aproximadamente la mitad de potencia/precio.

Además, en algunos casos, la potencia solo contabiliza las operaciones de coma flotante (Flops) siendo posible, en algunos de estos dispositivos, paralelizar operaciones de Enteros (Intops) y/o media precisión (Hops) en un mismo núcleo/ciclo, como pueden ser el caso de las RTX. A la vista de este análisis, se postula el sector de GPU como el potencial mayor para lograr los objetivos previstos para este estudio, siendo la RTX 2080 el dispositivo en el que se basarán los resultados de este documento.

No obstante, indicar que, tanto las FPGAs como los DSPs, presentan otra serie de ventajas con respecto a las GPUs, como puede ser una mayor eficiencia energética [3] y/o posibilidades de interoperabilidad con determinados tipos de I/O, que les da un valor añadido en otros campos.

2.2.3. GPGPU

De manera clásica, las GPUs se han compuesto de pequeñas unidades simples de procesado (núcleos) cuya finalidad era realizar operaciones relativamente simples sobre los píxeles de una imagen.

Conforme dichas operaciones fueron ganando complejidad, fue necesario que dichos núcleos crecieran ligeramente en potencialidad y especialmente en cantidad [22], lo que, a la larga, ha incentivado el interés del empleo de este tipo de dispositivos fuera del ámbito gráfico.

Como ya se ha indicado, se ha elegido el sector de las GPU como punto base para el desarrollo de este estudio precisamente por estos motivos, núcleos con capacidades relativamente básicas y en gran cantidad. El término empleado para este sector, fuera de su ámbito estándar, la computación gráfica, es el de Computación de Propósito General en Unidades de Procesamiento Gráfico (GPGPU), y consiste en el uso de estas Unidades de Procesamiento Gráfico (GPU) para implementar algoritmos a nivel general.

La idea básica es emplear la amplia cantidad de núcleos disponibles, para realizar acciones computacionales simples de manera paralela para acelerar el tiempo de procesado de un determinado algoritmo y liberar a la CPU de parte de la carga computacional.

Las primeras ideas de utilización de este tipo de dispositivos surgieron a principios de los años 2000, marcando el año 2007 el principio de la era del GPGPU con el lanzamiento de CUDA [26]. Posteriormente, en 2010, surgiría el primer producto comercial, en el sector automovilístico, que integraría esta tecnología.

Dentro del mundo del GPGPU, existen dos vertientes diferenciadas, OpenCL como un estándar para el uso de GPGPU en cualquier GPU compatible, y CUDA, solución propietaria solamente compatible con las GPU de Nvidia. En nuestro caso, emplearemos CUDA al ofrecer un mayor número de herramientas para la implementación general del código.

En general, existen una serie de creencias y comentarios, en el uso de este tipo de dispositivos siendo las más sonadas las siguientes:

- Pérdida de precisión respecto a otros dispositivos, que actualmente es, como se demostrará, despreciable siempre y cuando se respeten los procesos.
- Arquitecturas cambiantes, aunque siempre en pos de mejorar este tipo de dispositivos para este tipo de aplicaciones.
- El uso de la tecnología fuera del sector destino, aunque en los últimos tiempos se ha adaptado dicha tecnología para abarcar un mayor rango de sectores.
- Potencia bruta, siendo hoy en día equiparables tanto las gamas de nivel doméstico como profesional a los dispositivos específicos empleados tradicionalmente.

Una ventaja adicional del empleo de GPUs es la facilidad de implementar sistemas Multi-GPU mediante la tecnología de Scalable Link Interface (SLI), que permite interconectar diversas GPUs de la misma familia para incrementar el rendimiento general. Actualmente el máximo de unidades interconectadas es de 4 unidades de GPU en paralelo en determinados modelos, aunque las configuraciones más comunes de soporte son de 2 o 3 unidades gráficas.

2.2.4. Arquitecturas GPU

Dentro de las múltiples arquitecturas presentes en el mundo de las GPUs, vamos a analizar de manera general su estructuración y en especial se explicará una de las ultimas implementadas, denominada Turing, integrada en los últimos modelos tanto domésticos como profesionales de las familias de Nvidia.

El elemento fundamental de un sistema CUDA se denomina CUDA Core, que es básicamente una pequeña MAC (Multiply & Accumulation Operation) de 16 bits con una serie de registros de acceso inmediato, en ocasiones denominada memoria de hilo. Estas pequeñas MACs se agrupan en los denominados Streaming Multiprocessors (SM), que son conjuntos de CUDA Cores con canales de transmisión dedicados y una determinada cantidad de memoria de acceso rápido, conocida como memoria compartida. Una GPU por lo general presenta varios SM junto a una memoria mayor conectados con canales de acceso medio-rápido, conocida como memoria global.

No obstante, en la arquitectura Pascal, se realizó un re-estructurado de los SM, agrupándolos en Graphics Processing Clusters (GPC) siendo la base de un GPC para esta misma arquitectura una unidad de rasterizado y hasta cuatro SM.

A partir de la Arquitectura Volta, con proceso de fabricación de 12 nm, se implementaron los Tensor Cores, que son básicamente MACs de 4x4 elementos en forma matricial con entrada de 16 bits y salida de 32 bits cuya finalidad es el entrenamiento de redes neuronales y/o las operaciones de matrices de grandes dimensiones.

En la última arquitectura, Turing con un proceso de fabricación de 12 nm, se implementaron además los denominados Ray Tracing Cores, que son unidades específicas empleadas en rutinas gráficas, y actualmente sin uso fuera de la computación gráfica. En esta última arquitectura, se realizó una mejora en la distribución de los CUDA Core por SM aumentando el número de SM y reduciendo el número de CUDA Core por SM. Además, en esta arquitectura, se ha redefinido los CUDA Cores, permitiendo el cálculo paralelo de operaciones de punto flotante y enteros, así como de los cálculos propios de los Tensor Core, dentro de cada CUDA Core.

La siguiente imagen muestra la arquitectura Turing TU102 presente en las tarjetas RTX 2080 TI, RTX 6000 y RTX 8000.

Figura 10. Arquitectura Turing TU102 con ampliación del SM. © Nvidia Corporation (modificado)

2.2.5. CUDA

CUDA podría definirse como una API que permite a los desarrolladores un acceso directo al conjunto virtual de instrucciones de la GPU. Esta API definiría una capa software, que, mediante otros lenguajes de programación, siendo los más empleados C/C++, Fortran y Python, facilitaría la implementación y utilización de código a nivel de GPU con respecto a utilidades convencionales como Direct3D o OpenGL. En otras palabras, es el conjunto de utilidades que permite utilizar una GPU compatible para procesar código.

Para ello, aparte de las herramientas básicas de desarrollo, ofrece una serie de librerías, de ámbitos variados, con instrucciones comunes ya implementadas y optimizadas. Algunas de estas librerías son: cuBLAS, cuFFT, cuLASS, cuRAND, cuSPARCE y cuSOLVER.

Su sintáctica es equivalente al lenguaje empleado, permitiendo una conversión pseudo-directa del código a GPU, mientras que su utilización sigue un régimen jerárquico, siendo la CPU la unidad directora de la GPU.

En relación a su funcionamiento, la CPU es la encargada de decirle a la GPU qué instrucciones/rutinas (conocidas como kernels) debe de ejecutar que son lanzados en uno o varios CUDA cores. Los kernels, deben trabajar con la memoria propia de la GPU, siendo imposible acceder de manera directa a la memoria GPU desde la CPU y viceversa.

El proceso típico de funcionamiento, sigue un patrón directo, asignando en primer lugar la memoria necesaria en GPU, luego enviando los datos de CPU a GPU, en tercer lugar, lanzando los kernels necesarios para el procesado de esos datos, que una vez procesados son devueltos a memoria CPU para liberar la memoria GPU.

Es por estos motivos que, se le podría considerar una herramienta de co-procesado en el que se relegan las tareas más pesadas en un sentido computacional, o con altos requisitos de acceso a memoria, y que pueden ser divididas en pedazos de simple resolución, en otras palabras, en procesos con un alto grado de paralelización.

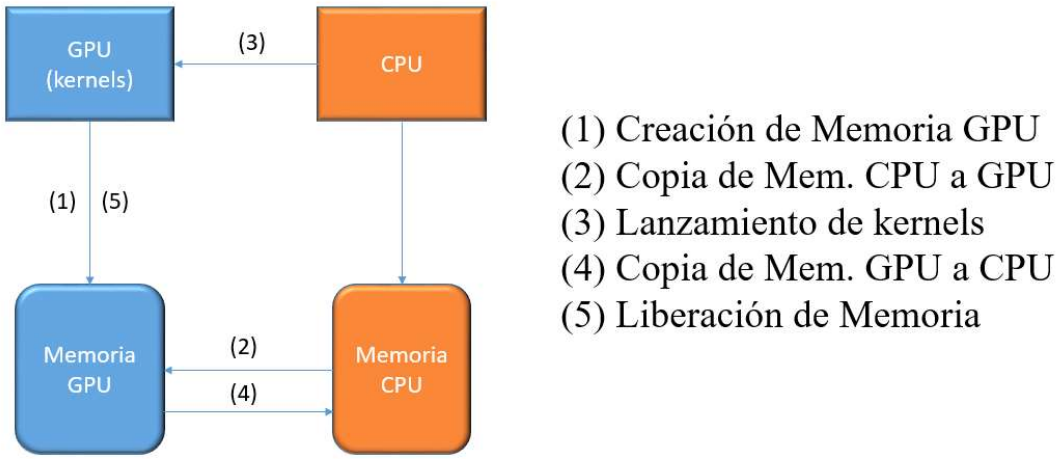


Figura 11. Secuencia base de un programa CUDA

Capítulo 3. Métodos

3.1. Herramientas y Modelos

3.1.1. MATLAB

Una de las herramientas base empleadas, es el entorno de trabajo MATLAB en sus versiones 2018b y 2019a con licencia en Ámbito Educativo. Este programa es empleado en el entorno del cálculo técnico para el desarrollo y obtención de soluciones a problemas específicos.

En este caso, la aplicación clínica base está escrita en MATLAB, por lo que era deseable que los códigos CUDA generados fueran igualmente compatibles con MATLAB, con el fin de sustituir los fragmentos originales por los fragmentos acelerados por paralelización.

3.1.2. Visual Studio

El segundo programa empleado es el entorno de programación y depuración Visual Studio, en nuestro caso en su versión 2017 Community. Este programa facilita una serie de utilidades para programar, compilar y depurar el código de manera efectiva.

En este caso, se hará uso del mismo con fines de programación, en el sentido de implementación de los algoritmos en CUDA, y como vinculo al compilador propio de CUDA, trabajando con los templates base que se crean con la instalación de CUDA.

3.1.3. CUDA y Librerías

Como ya se ha explicado, CUDA es una plataforma de computación paralela. Para facilitar la implementación de los algoritmos, se ha acudido a una serie de librerías básicas de CUDA:

- cuBLAS (CUDA Basic Linear Algebra Subroutines): Librería con operaciones básicas de operaciones entre matrices y/o vectores.
- cuSOLVER (CUDA Solver): Librería con utilidades y/o métodos para la resolución de problemas algebraicos.
- cuFFT (CUDA Fast Fourier Transform): Librería con métodos para calcular y paralelizar FFTs.

Estas librerías han sido empleadas en su versión 10.1.168 incluida en la versión 10.1 del toolkit de CUDA.

3.1.4. Equipo

Como equipo de trabajo, se ha empleado el siguiente equipo, de ámbito doméstico:

- CPU: Intel i5 4460 3.5 GHz.
- Memoria: 16 GB DDR3 1.666 MHz CL11.
- GPU: Nvidia RTX 2080 1.8 GHz 8 GB GDDR6 2944 CUDA Cores.
- Almacenamiento 1: SSD 128 GB.
- Almacenamiento 2: HDD 3TB 7200 rpm clase comercial.

Esto situaría el equipo en una posición media-alta en cuanto a potencia de procesado.

Debido a la naturaleza del equipo, algunos resultados y conclusiones pueden desvirtuarse de lo posiblemente encontrado en equipos de ámbito profesional, en especial por la disparidad generacional entre la GPU y la CPU, no obstante, en el sentido del estudio, la capacidad de las CPU de ámbito doméstico tampoco ha sufrido cambios de potencia extremadamente relevantes como para que esta disparidad sea realmente extrema.

Como dato, a continuación, se presenta una tabla con dos dispositivos CPU y GPU de la misma gama entre sí, para comparar el incremento en potencia computacional:

Comparativa de Dispositivos				
	Modelo	Año	Potencia (GFlops)	Precio
CPU 1	I5 4460	Q2 – 2014	15.62	180€
CPU 2	I5 9400	Q1 – 2019	27.66	200€
GPU 1	GTX 780	Q1 – 2014	3200	649€
GPU 2	RTX 2080	Q4 – 2018	10500	700€

Tabla 3. Comparativa GPU y CPU. Información extraída de Setiathome y Nvidia

Nota: Comparación realizada en relación a los dispositivos empleados en este trabajo.

3.2. Conexión con MATLAB

A la hora de plantear la pasarela hacia Matlab del potencial código desarrollado, existen diferentes alternativas de uso. Para este caso, se ha acudido al uso de librerías dinámicas (.dll).

De todos los métodos consultados, es el que más se adaptaba a las necesidades de este estudio, al proporcionar las siguientes ventajas:

- Fácil inserción de los datos mediante punteros.
- Certeza de que el código compilado es independiente de MATLAB.
- Fácil depuración.

La compilación del código de esta manera se realizaría en Visual Studio, y MATLAB actuaría en este caso como plataforma de lanzamiento.

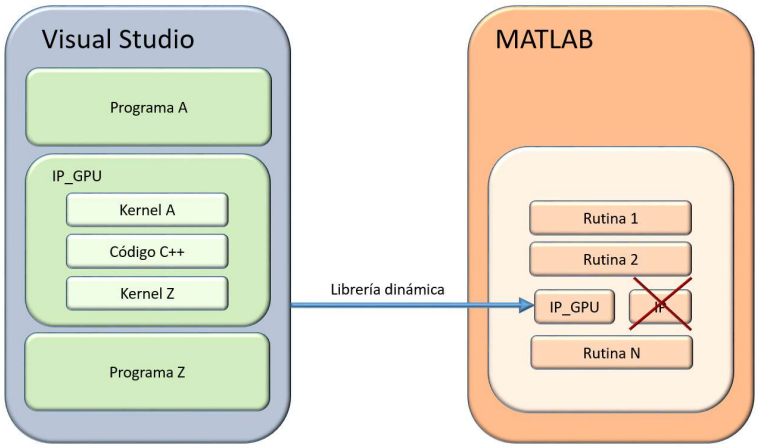


Figura 12. Entornos de Trabajo y pasarela

3.3. Resolución del Problema Inverso

3.3.1. Entorno

El primer algoritmo con el que se trabajara, es el denominado problema inverso, explicado en la parte teórica de esta memoria. Dicho algoritmo, facilitado como una función de MATLAB en forma secuencial, podía plantearse de dos formas diferentes que se muestras a continuación:

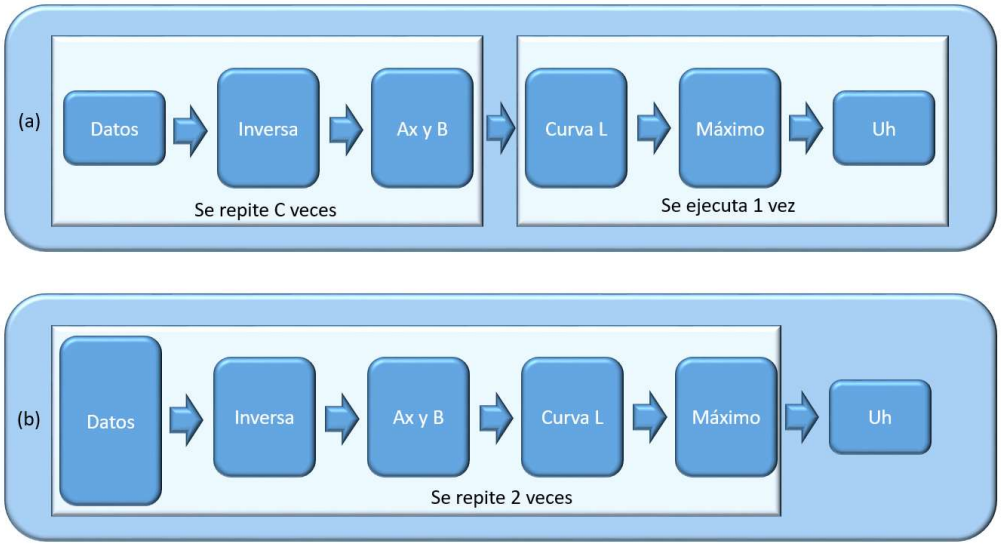


Figura 13. Diagramas del algoritmo original y modificado

En su forma original (a), el algoritmo calcula la inversa y los parámetros Ax y B para cada valor del parámetro de regularización (λ), el cual varía entre un valor máximo y un valor mínimo siguiendo una exponencial, realizando un numero C de iteraciones igual al número de parámetros de regularización de lo que se disponga.

Una vez calculados todos los Ax y B, se calcula la Curva L y su máximo, siendo este máximo el valor óptimo para la solución del problema, para posteriormente, calcular la solución Uh con ese λ óptimo.

En su forma (b), solución sugerida a posteriori, en lugar de calcular todos los valores de λ, se calculan un número menor de los mismos con una separación p entre ellos en una primera iteración de manera que se abarque todo el rango de valores con menor precisión.

Del máximo de esta primera iteración, se iniciaría una segunda iteración alrededor de ese máximo y se hallaría así la solución final, ahorrando, de manera teórica, ciclos de computación.

Además, en lugar de calcular los valores de manera secuencial, se reservan una serie de zonas amplias de memoria en la que se mapean los datos correspondientes a una misma iteración, y en vez de calcular los resultados de manera secuencial, se calcularan de manera paralela, mediante una técnica conocida como Batching. Esencialmente, la técnica de Batching se aplica a nuestro problema al resolver de forma simultanea mediante GPU la solución correspondiente a 2 o más parámetros de regularización.

Por ello, en adelante, a estos bloques de memoria se les denominaran Batch, y se empleara junto a un número que indicara la cantidad de parámetros de regularización que se procesaran de manera paralela.

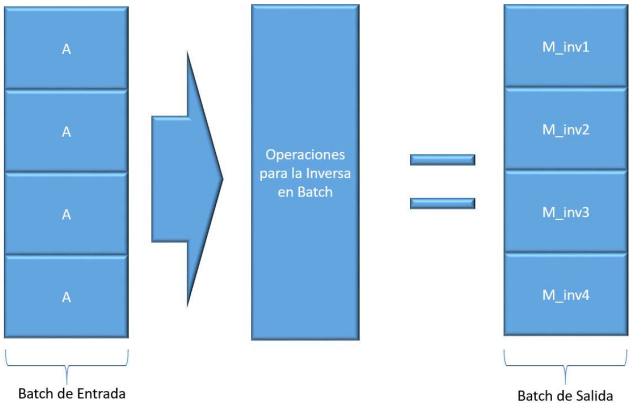


Figura 14. Ilustración de Batch

En este caso, estaríamos hablando de un Batch de tamaño 4, que contendría cuatro subsecciones de memoria con réplicas de una teórica matriz A de tamaño [M x N], siendo el espacio de memoria reservado para A de 4 x M x N x sizeof(double) bytes de memoria. De igual forma. M_inv, con unas dimensiones teóricas de [M x N] ocupara 4 x M x N x sizeof(double) bytes en memoria. De esta forma, y de manera teórica, se obtendrá un mejor aprovechamiento de la GPU, a costa de aumentar los requisitos de memoria GPU del algoritmo a implementar.

A continuación, se muestra una gráfica de la Curva L con los valores que toma λ en la primera implementación, marcados en azul, los de la primera iteración de la segunda implementación, en rojo, y los de la segunda iteración de la segunda implementación, en morado. Como se puede observar, la resolución del problema inverso conlleva una búsqueda del parámetro de regularización en un rango muy grande de valores posibles, normalmente de varios órdenes de magnitud. En este caso se muestra un ejemplo en el que la búsqueda contempla 10 órdenes de magnitud (de 1e-15 a 1e-5), en el que el parámetro de regularización óptimo se obtuvo para λ = 5.62e-8.

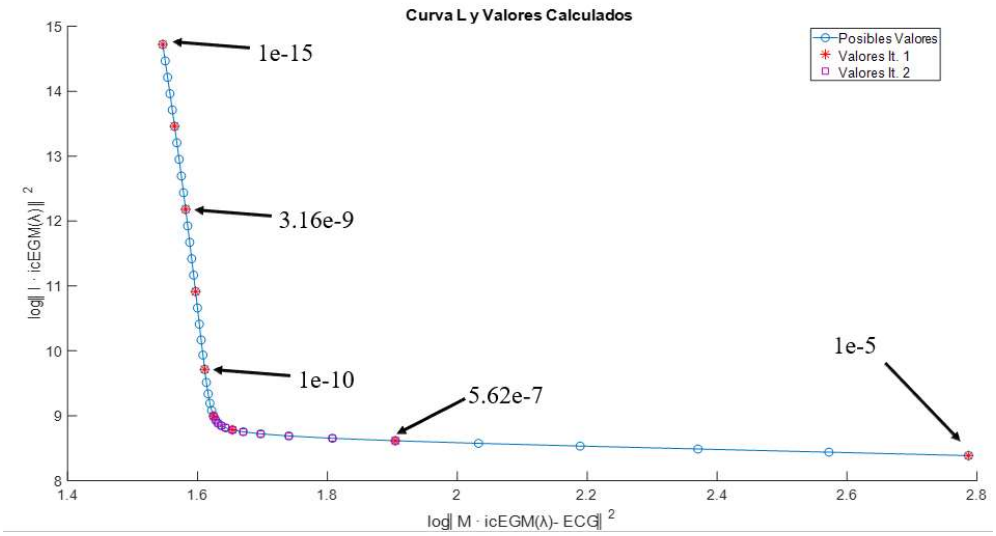


Figura 15. Curva L y valores elegidos.

Otro punto importante a indicar, es que la matriz a la que se le aplica el coeficiente de regulación, es $A_x = A' * A$ que, para este caso, A_x se convierte en una matriz hermitiana, también conocida como hermítica, con una serie de propiedades que permiten un cálculo de su inversa de forma presta y computacionalmente eficiente.

3.3.2. Métodos de resolución del problema inverso

El método en cuestión, posee tres entradas principales de datos:

- A: Matriz de transferencia del paciente con dimensiones [M x N], siendo M el número de señales potencial en la superficie torácica y N el número de señales potencial en la superficie cardiaca.
- Ub: Matriz de potenciales superficiales [M x S], siendo S el número de muestras temporales.
- t: Vector correspondiente a los parámetros de regularización λ con dimensión K, siendo este la cantidad de parámetros de regularización.

Que tras su procesamiento se deberán de obtener dos salidas:

- Uh: Matriz con los potenciales procesados con dimensiones [N x S].
- Topt: Valor óptimo de λ.

En lo relativo al proceso, primeramente, se realizará un acondicionamiento del entorno, inicializando los espacios de memoria requeridos y las librerías empleadas.

En este punto, es donde las implementaciones Secuencial y en Batch difieren.

En el primer caso, se iterará un numero K de veces realizando las multiplicaciones y transformaciones de matrices oportunas para obtener la inversa y el cálculo de Ax y B. Después se guardarán los resultados de Ax y B obtenidos para formar la curva L y de la misma obtener la máxima pendiente.

Tras esto, se calculará la Uh correspondiente al valor de λ óptimo repitiendo el proceso para obtener la inversa. Esto supone que esta implementación realiza K + 1 veces el mismo calculo.

En el segundo caso, se realizarán un máximo de dos iteraciones, realizando todas las operaciones correspondientes a una iteración en forma de bloque en paralelo, lo que denominamos un Batch, siguiendo el mismo patrón de valores que en la implementación secuencial.

En la primera de esas iteraciones se tomarán puntos equidistantes con una distancia mayor que 1 entre los valores de índice de λ. De estos primeros valores se calculará su Curva L y se obtendrá su máximo. Este máximo se guardará y se preparará el espacio para la segunda iteración, esta vez con los valores alrededor de dicho máximo.

Tras la segunda iteración, en esta ocasión como ya disponemos de la inversa almacenada en memoria, el cálculo de Uh es directo sin requerir del cálculo adicional.

Con este método realizamos un número de operaciones equivalente a $2 * N_{batch}$ siendo $N_{batc} < K$ por lo que supondrá de manera teorica una mejora notable tanto en utilización como en tiempo de procesado.

La secuencia aproximada de operaciones realizadas para el cálculo de la inversa es en ambos casos la siguiente, donde los índices I, II, III describen los sub-conjuntos de operaciones relacionados con un cálculo concreto:

Calculo de la Inversa (I)

1- R= A' * A + (λ I)
2- R = L*L' (Cholesky)
3- H = $(L^{-1})' * (L^{-1})$ o H = R \ (Ub')

Cálculo de Ax y B (II):

1- $A'_x = H * Ub$
2- A''_x = A*A'_x+ Ub
3- $Ax_i = |A''_x|^2$; (Norma de Frobenius)
4- $|B'_x|^2 = I * A'_x$
5- $B_i = |B'_x|^2$; (Norma de Frobenius)

Cálculo de la curva L y su máximo (III):

1- x | y = log10 (Ax | b)
2- Derivada primera y segunda de x e y (dx | dy | ddx | ddy)
3- Calculo de la Curva L.
4- Búsqueda de máximo.

Esto contemplaría a grandes rasgos las secuencias de instrucciones empleadas para la implementación, obviando las operaciones de copia de datos y acondicionamiento de variables.

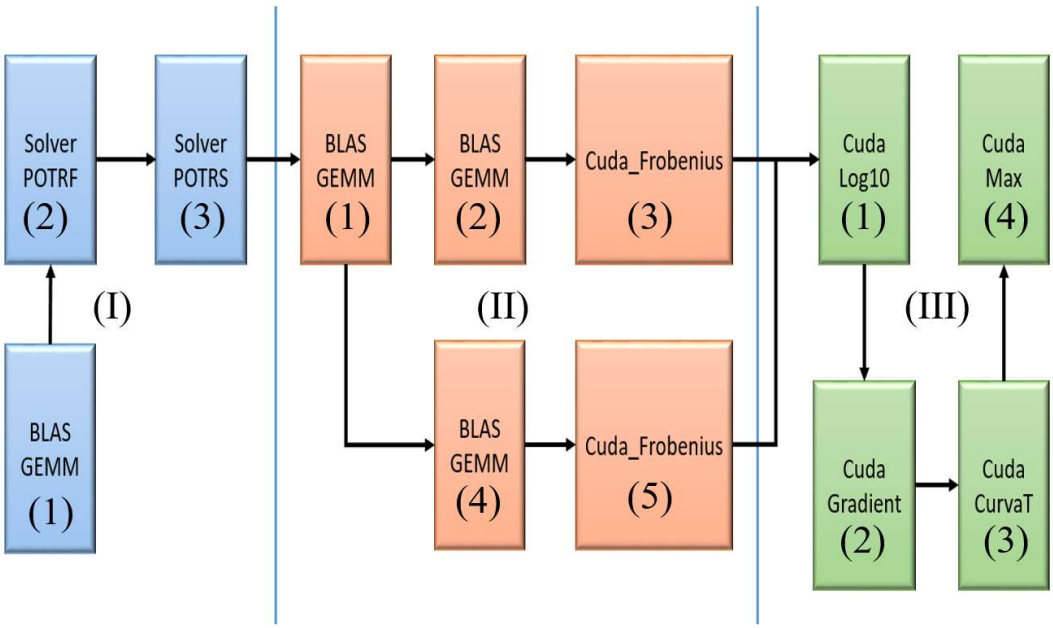


Figura 16. Flujo de Operaciones de IP

En la figura anterior, se pueden apreciar tres identificadores principales, BLAS, Solver y CUDA, que hacen referencia a las funciones de cuBLAS, cuSOLVER y los kernel creados por el autor de esta memoria correspondientemente.

Los bloques BLAS, contienen el sub-identificador GEMM, correspondiente a la instrucción de nivel 3 de cuBLAS GEMM, la cual implementa una multiplicación y suma de matrices en la forma $C = \alpha * A * B + \beta C$, función de tipo in-place si $\beta \neq 0$ [27] .

En los bloques Solver, se aprecian dos identificadores, POTRF y POTRS, correspondientes al cálculo de la factorización Cholesky para matrices hermitianas, y al cálculo de la inversa de una matriz Hermitiana que previamente ha sido factorizada [28].

En cuanto a los bloques CUDA, encontramos cinco tipos:

- Frobenius: Para el cálculo de la norma de Frobenius $\left(y_0 = \sum\sum\left(a_{ij}^2\right)\right)$ mediante una algoritmo de reduccion.
- Log10: Para el cálculo de $y_k = 10 * \log_{10} x_k$.
- Gradient: Para el cálculo de las derivadas parciales siguiendo la siguiente formula:

$$y_k = \begin{cases} x_{k+1} - x_k, & if\ k = 0 \\ x_k - x_{k-1}, & if\ k = n-1 \\ \dfrac{x_{k+1} - x_{k-1}}{2 * paso}, & resto \end{cases}$$

- CurvaT: Algoritmo para el cálculo de los puntos de la Curva L siguiendo la ecuación $y_k = \frac{dx_k * ddy_k - ddx_k * dy_k}{\left(dx_k^2 * dy_k^2\right)^{3/2}}$.
- Max: Algoritmo de búsqueda iterativa del máximo siendo su salida el índice correspondiente a indexación basada en 0.

3.4. Detección de la frecuencia dominante

3.4.1. Entorno

Como ya se ha explicado en el apartado teórico, para este algoritmo podríamos simplificar su explicación al hecho de tener una serie de señales en el dominio del tiempo y buscar su equivalente promediada en frecuencia mediante FFTs para posteriormente buscar el máximo valor dentro de un determinado rango de frecuencias.

En este estudio, se ha definido los siguientes parámetros base para trabajar con las señales:

- Tamaño de FFTs: NFFT = 8192 puntos.
- Numero de segmentos: 8
- Cantidad de solapamiento: 50%.
-

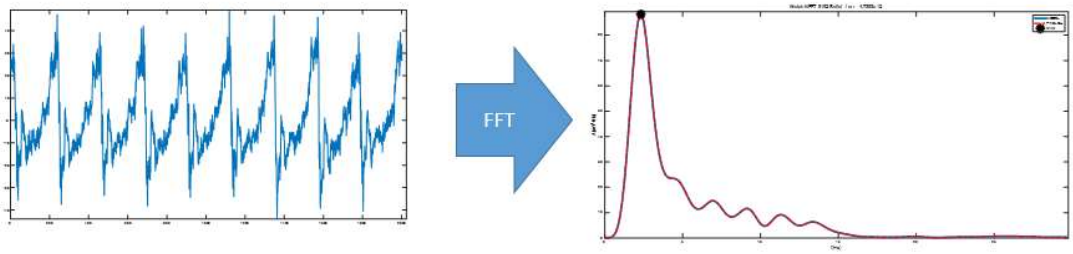


Figura 17. Ejemplo del Periodograma

Puesto que el cálculo de las FFTs es la parte más pesada en relación a cálculo, es la parte que se ha procurado implementar en CUDA.

Para ello, se ha acudido a una estructura típica de implementación en la que, tras la inicialización del entorno, construiremos todas las señales correspondientes a una iteración en una matriz y aplicaremos el enventanado seleccionado de manera paralela a todas esas sub-señales. Posteriormente, se calculará la FFT de dichas señales y se aplicarán los módulos necesarios para su tratamiento.

3.4.2. Métodos del cálculo de Frecuencias Dominantes

El método desarrollado para este algoritmo constaría de dos entradas y una salida.

Las entradas, A y window, corresponden al conjunto de señales a analizar y al tipo de ventana que se va a aplicar respectivamente, siendo posible en el caso de la ventana elegir entre Hanning, Hamming o Blackman. La salida Output corresponde a un espacio de dimensiones N x NFFT/2 para almacenar los resultados.

Además, el método cuenta con dos entradas adicionales, N y S, para indicar el tamaño de la matriz A, así como de la entrada de habilitación (Verbose) de la ventana de información.

Como se adelantaba en el apartado teórico, la implementación efectiva dependerá de la cantidad de señales y la cantidad de segmentos a calcular. En este caso al ser la segunda muy inferior a la primera, se va a realizar la implementación paralelizando los segmentos de las señales, realizando 8 iteraciones de cálculo y acumulación.

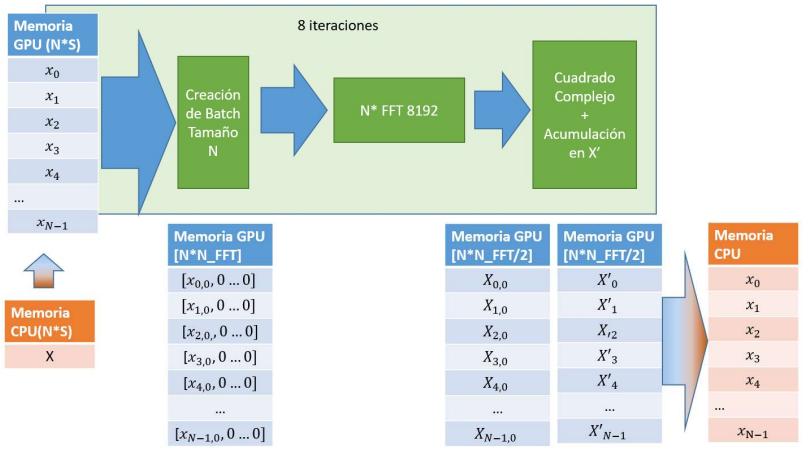


Figura 18. Esquema de Memoria de DFS

Este proceso se puede descomponer en una secuencia de seis instrucciones básicamente:

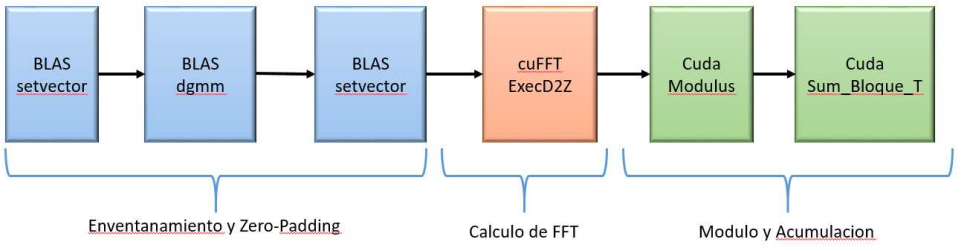


Figura 19. Flujo de Operaciones de DFS

En la que se emplea la librería BLAS para el acondicionamiento de señales y el cálculo de las señales enventanada, cuFFT para el cálculo de las FFTs en Batch [26] y una serie de kernels propios para el cálculo del módulo complejo y la acumulación en Batch.

En este esquema, se ha marcado en azul la secuencia necesaria para la creación de la matriz y el enventanamiento de las sub-señales que la componen. En rojo, se resalta el cálculo de todas las

FFT en paralelo, y en verde la aplicación de los kernels correspondientes al cálculo del módulo complejo y la acumulación del resultado entre iteraciones.

El resultado de la acumulación al finalizar las iteraciones, será insertado en las direcciones reservadas por Output.

3.5. Experimentos realizados

3.5.1. Pruebas para IP

Para comprobar el funcionamiento de los algoritmos implementados, se procederá a montar una batería de pruebas en MATLAB desde el cual se lanzarán las dos implementaciones GPU y el algoritmo base en CPU y se realizarán las comprobaciones oportunas en cuanto a tiempos y valores obtenidos.

Dichas comprobaciones se realizarán para un número determinado de lanzamientos y una variación de los parámetros N y K (correspondientes al número de señales en el torso y a la cantidad de parámetros de regularización) entre 771 a 32 y 41 a 161 cantidades de valores respectivamente.

Además, se realizarán dos comprobaciones separadas para dos tamaños de Batch en particular, 9 y 15, siendo el primero el estimado para que la implementación del Batch sea numéricamente lo más estable posible y el segundo un valor lo suficientemente diferente como para apreciar cambios relevantes en el funcionamiento de ambas pruebas.

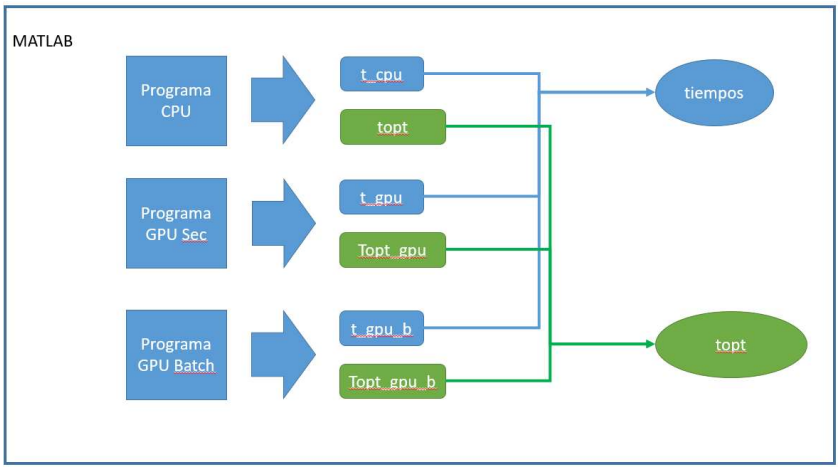


Figura 20. Pruebas de IP

Adicionalmente, se mostrarán los resultados de las Uh obtenidas en los tres casos para la implementación estándar (N = 771 y K=41) para poder comprobar el error cometido en el cálculo final de las Uh.

3.5.2. Pruebas para DFS

De igual manera que en el algoritmo anterior, se realizará un banco de pruebas en MATLAB, comparando en esta ocasión la implementación base con la de GPU y extrayendo tres conjuntos de datos, correspondientes a tiempos de ejecución, las frecuencias dominantes encontradas y las densidades de potencia normalizadas.

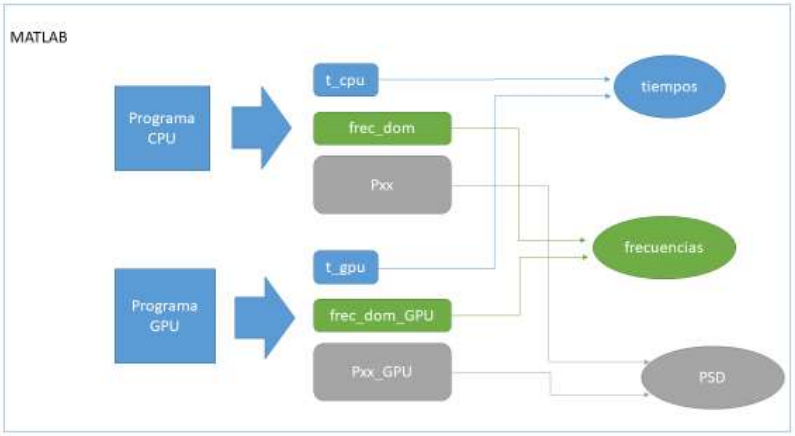


Figura 21. Pruebas de DFS

En la realización, además, se realizará una variación de dos parámetros, el número de señales de entrada y el orden de la FFT variando entre 500 a 2046 y 8192 a 32768 respectivamente.

3.5.3. Implementación IP + DFS

Con el fin de complementar el desarrollo de este estudio, se ha decidido realizar la implementación de ambos métodos de manera conjunta. Esta implementación se realizará de dos formas, siendo una la ejecución de ambos métodos por separado (IP + DFS) y la otra la integración de ambos métodos en una nueva función (IP_GPU) en el que se realizará la vinculación directa sin pasar por MATLAB.

Tras la obtención de ambos métodos, se realizará una comprobación temporal y a nivel de resultado final entre ambos módulos y la ejecución total en CPU.

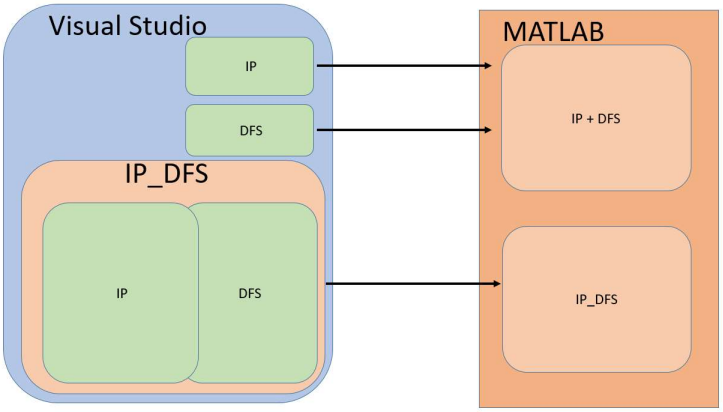


Figura 21. Diagrama Implementación IP + DFS

Capítulo 4. Resultados

4.1. Problema Inverso

Como ya se adelantaba en el apartado de pruebas, se ha procedido a realizar cuatro análisis diferentes, en función del parámetro variado y del tamaño de la implementación paralela.

4.1.1. Implementación Paralela de 9 puntos

4.1.1.1. Variación del número de señales en superficie

Tras lanzar el banco de pruebas descrito en la sección de pruebas, se han obtenido los siguientes resultados en relación al valor óptimo del parámetro de regularización encontrado.

Comparativa del valor Optimo Obtenido [Valor (# índice)]			
Número de señales de superficie	Valor obtenido con CPU	Valor obtenido con GPU Secuencial	Valor obtenido con GPU Batch (N=9)
771	5.6234e-08 (#32)	5.6234e-08 (#32)	5.6234e-08 (#32)
512	5.6234e-08 (#32)	5.6234e-08 (#32)	5.6234e-08 (#32)
256	3.1623e-08 (#31)	3.1623e-08 (#31)	3.1623e-08 (#31)
128	1.7783e-08 (#30)	1.7783e-08 (#30)	1.7783e-08 (#30)
64	1.0000e-08 (#29)	1.0000e-08 (#29)	1.0000e-08 (#29)
32	1.0000e-08 (#29)	1.0000e-08 (#29)	5.6234e-09 (#28)

Tabla 3. Resultados del Valor Optimo variando el número de señales en superficie. Resaltado los Valores erróneos.

Se puede observar, que a nivel general se obtiene el mismo resultado, exceptuando un ligero error de un índice en el caso de la implementación paralela con respecto al resto, en el caso de 32 señales en superficie, subrayado en la tabla anterior.

Esta diferencia podría ser aceptable, si consideramos que los subconjuntos menores de datos son recortes del original, y para este tamaño de implementación en particular se produzca algún tipo de inestabilidad numérica provocada por una dependencia adquirida en las operaciones.

A continuación, se presentan los resultados de la comparación temporal de este conjunto de resultados:

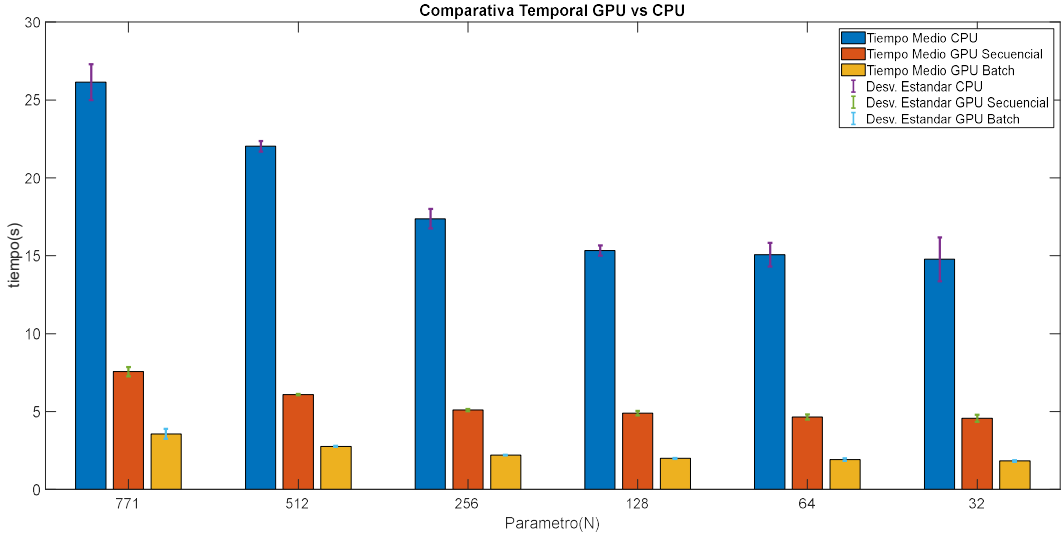


Figura 22. Tiempos de Ejecución entre CPU/GPU/GPU Paralelo para variaciones en N

Se observa una tendencia descendente relacionada con el tamaño de señales, estabilizándose alrededor de 128 señales.

Además, se puede ver una clara diferencia entre los tiempos de ejecución, ya no solo a nivel de tiempo medio, sino a nivel de su desviación típica, siendo el caso de GPU de algunos milisegundos en comparación a segundos que posee la de CPU, debido a la necesidad de la misma de ser compartida con otros procesos externos.

En relación a los tiempos medios, si realizamos una comparativa de los respectivos tiempos GPU contra los de CPU podemos sintetizar la siguiente tabla:

Speed-Up		
Numero de señales de superficie	CPU/GPU Secuencial	CPU/GPU Batch (9)
771	3.4x	7.3x
512	3.6x	7.9x
256	3.4x	7.9x
128	3.1x	7.7x
64	3.2x	7.9x
32	3.2x	8.1x
Speed-Up Medio	3.3x	7.8x

Tabla 4. Mejora del tiempo de procesado variando el nº de señales en superficie

Esta tabla muestra la mejora promedio obtenida en cada conjunto de ejecuciones realizadas. Observamos que, para el equipo empleado, se obtiene una mejora de aproximadamente 3.3 veces mejores tiempos en la implementación secuencial en GPU con respecto a CPU.

En el caso de la implementación paralela en GPU, dicha mejora es de aproximadamente 7.8x veces menor el tiempo en GPU que en CPU.

4.1.1.2. Variación del número de parámetros de regularización

El siguiente punto es el análisis de un segundo testeo realizado, en el que ahora lo que se variara es la cantidad de parámetros de regularización con los que trabajamos (parámetro K). De nuevo, lo primero que analizaremos es la exactitud del parámetro de regularización optimo encontrado.

Comparativa del valor Optimo Obtenido [Valor (# índice)]			
Nº de parámetros de regularización	CPU	GPU Secuencial	GPU Batch (9)
41	5.6234e-8 (#32)	5.6234e-8 (#32)	5.6234e-8 (#32)
81	5.6234e-8 (#63)	5.6234e-8 (#63)	5.6234e-8 (#63)
161	4.8697e-8 (#124)	4.8697e-8 (#124)	4.217e-8 (#123)

Tabla 5. Resultados del Valor Optimo variando el nº de parámetros de regularización

Observamos un error en la implementación paralela cuando el número de parámetros de regularización es de 161. Este error, a diferencia del caso anterior, posiblemente venga producido por una falta de rango en la cantidad de valores con los que se calcula la implementación paralela.

En este caso, tenemos un paso de $\frac{K}{N_{batch}} = \frac{161}{9} = 17$ puntos entre valores en la primera iteración. En la implementación paralela, a la hora de realizar la segunda iteración, solo se revisa, para este caso, el valor máximo obtenido y cuatro valores hacia cada dirección, por ejemplo, si en la primera iteración encontrase el índice #119 como máximo, su rango en la segunda iteración seria de [115 - 123], siendo claramente insuficiente para incluir los valores intermedios entre dos puntos de la primera iteración, contando que el siguiente índice seria #136 con un rango en la segunda iteración entre [131-140]. Los valores entre 124 y 130 en esta implementación se pierden al no ser alcanzables en ningún caso, siendo este un caso de valores/Batch incompatible.

Si realizamos la comparación temporal para este caso:

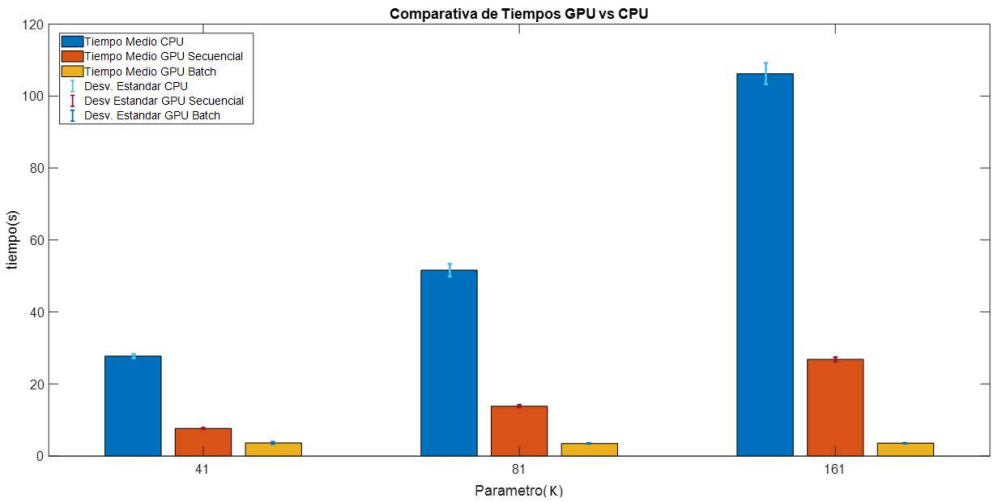


Figura 23. Comparación temporal entre GPU/CPU/Batch GPU para variaciones de K

Lo primero que se puede observar, es que el tiempo en CPU se dispara, siendo aproximadamente el doble al doblar el número de valores. En el caso de la implementación secuencial en GPU, se producen también incrementos proporcionales al número de puntos. No obstante, la

implementación paralela en GPU, es prácticamente invariante al número de valores de regularización, al realizar siempre el mismo número de cálculos y estando únicamente relacionado con el tamaño asignado de Batch en este sentido.

Si, de nuevo, realizamos una comparación de tiempos entre CPU y las implementaciones GPU, obtenemos la siguiente tabla:

Speed-Up		
Número de valores del parámetro de regularización	CPU/GPU Secuencial	CPU/GPU Batch (9)
41	3.62x	7.6x
81	3.72x	14.9x
161	3.96x	29.8x

Tabla 6. Mejora de tiempos de procesado variando el nº de puntos del parámetro de regularización

Como se adelantaba, el incremento de tiempos entre CPU y GPU secuencial no es proporcional entre ellos, obteniendo una mejora creciente en el tiempo de procesado conforme aumenta el número de puntos. En el caso de la implementación paralela, ciertamente se obtienen márgenes de mejora comparativamente enormes, no obstante, el hecho de perder valores, resta importancia a dicha mejora, siendo algo anecdótico más que un resultado real.

Tras realizar las pruebas con un tamaño de Batch de 9 puntos, a continuación, se va a proceder a mostrar los resultados obtenidos para las mismas pruebas con un tamaño de Batch de 15 puntos.

4.1.2. Implementación Paralela de 15 puntos

4.1.2.1. Variación del número de señales en superficie

El siguiente análisis, seguirá el mismo orden que en el caso anterior, empezando por los resultados en el valor optimo al variar el número de señales de superficie.

Comparativa del valor Optimo Obtenido [Valor (# índice)]			
Numero de señales de superficie	CPU	GPU Secuencial	GPU Batch (15)
771	5.6234e-08 (#32)	5.6234e-08 (#32)	5.6234e-8 (#32)
512	5.6234e-08 (#32)	5.6234e-08 (#32)	5.6234e-8 (#32)
256	3.1623e-08 (#31)	3.1623e-08 (#31)	3.1623e-08 (#31)
128	1.7783e-08 (#30)	1.7783e-08 (#30)	1.7783e-08 (#30)
64	1.0000e-08 (#29)	1.0000e-08 (#29)	1.0000e-08 (#29)
32	1.0000e-08 (#29)	1.0000e-08 (#29)	1.0000e-08 (#29)

Tabla 7. Resultados del Valor Optimo variando el nº de señales en superficie (2)

Se observa que, para este tamaño de la implementación, el error en el caso de 32 señales ha desaparecido. Esto nos podría seguir indicando una dependencia numérica para trabajar con un tamaño de Batch concreto en función del modelo real del procedimiento, pudiendo solventar esta inestabilidad numérica ajustando el tamaño del Batch.

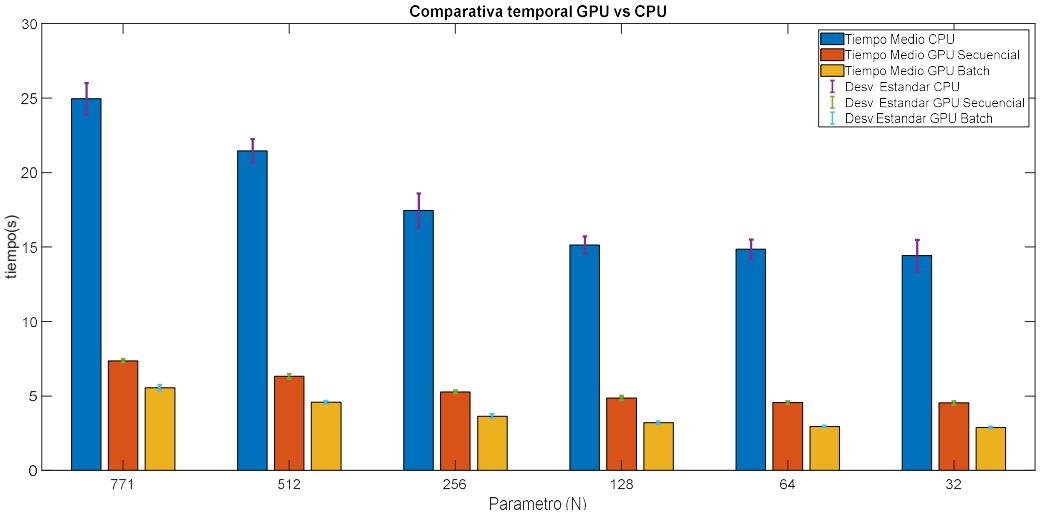


Figura 24. Comparativa temporal GPU vs CPU vs GPU Batch variando nº de señales de superficie.

Se denota una reducción considerable en los tiempos de procesado independiente al tamaño del parámetro, en todos los casos con tendencia decreciente, y estabilizándose aparentemente a partir de 128 puntos como en el caso anterior. De nuevo, la desviación estándar en CPU es mucho mayor que la de GPU, por la carga adicional en la CPU.

Si realizamos la comparativa de tiempos entre CPU y las implementaciones GPU, obtenemos los siguientes valores de Speed-Up:

Speed-Up		
Numero de señales de superficie	CPU/GPU Secuencial	CPU/GPU Batch (15)
771	3.39x	4.49x
512	3.39x	4.68x
256	3.11x	4.78x
128	3.11x	4.72x
64	3.25x	5.03x
32	3.18x	5.00x
Speed-Up Medio	3.27x	4.79x

Tabla 8. Mejora de tiempos de ejecución en función del nº de señales de superficie.

Se observa que la mejora podría considerarse constante con una tendencia creciente en el caso de la implementación en paralelo, siendo aproximadamente tres veces más rápida la implementación secuencial y hasta cinco veces la paralela en las condiciones del entorno de trabajo.

4.1.2.2. Variación del número de parámetros de regularización

Si ahora, analizamos los valores del parámetro de regularización en función del número de puntos de dicho parámetro:

Comparativa del valor Optimo Obtenido [Valor (#Índice)]			
Número de valores del parámetro de regularización	CPU	GPU Secuencial	GPU Batch (15)
41	5.6234e-8 (#32)	5.6234e-8 (#32)	5.6234e-8 (#32)
81	5.6234e-8 (#63)	5.6234e-8 (#63)	5.6234e-8 (#63)
161	4.8697e-8 (#124)	4.8697e-8 (#124)	4.8697e-8 (#124)

Tabla 9. Resultados del Valor Optimo variando la cantidad de parámetros de regularización.

Se puede ver una equivalencia exacta entre los valores de las tres implementaciones.

En este caso, para un valor teórico de la primera iteración de 120 su rango seria de [113 - 127] siendo el del siguiente punto, 135, de [128-142], no quedando valores sin usarse y podemos, aparentemente, afianzar la teoría de que existe una correlación entre el tamaño del Batch y la exactitud del parámetro de regularización, siendo 15 suficientes para el caso de 161 parámetros de regularización.

Si ahora visualizamos los tiempos, obtendríamos la siguiente tabla.

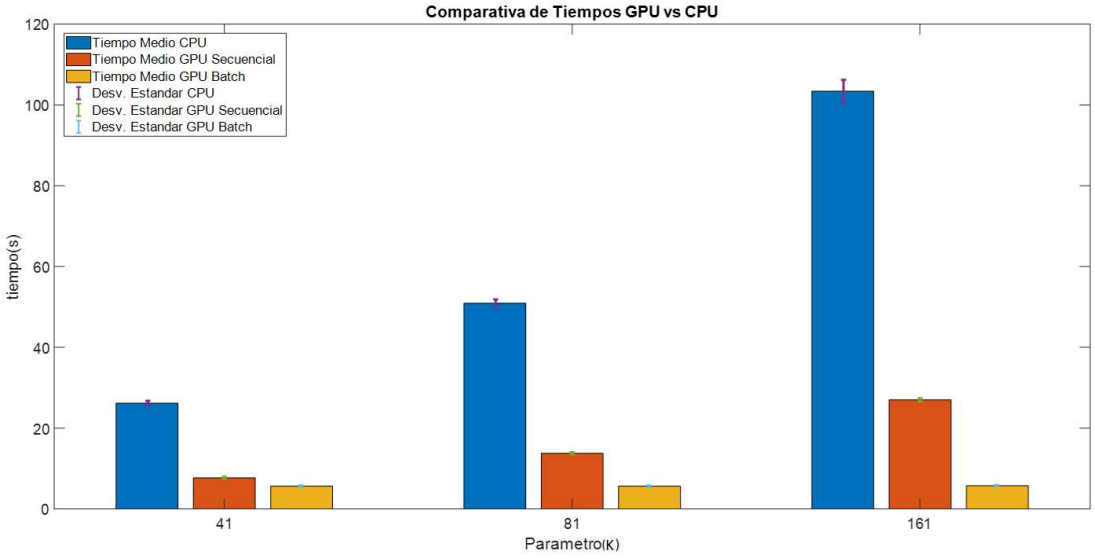


Figura 25. Comparación de Tiempos de procesado en función del nº de parámetros de regularización

Como pasaba con la implementación de 9 puntos, el tiempo en CPU y la implementación GPU secuencial se ve incrementado por el número de parámetros, no siendo el caso de la implementación paralela por las razones ya dadas.

Si procesamos para comparar los tiempos medios de ejecución entre implementaciones, en esta ocasión obtenemos la siguiente tabla:

Speed-Up		
Número de valores del parámetro de regularización	CPU/GPU Secuencial	CPU/GPU Batch (15)
41	3.4x	4.5x
81	3.7x	9x
161	3.8x	18x

Tabla 10. Mejora de tiempos de ejecución en función de la cantidad de parámetros de regularización.

Como ya se ha explicado, aunque los tiempos de la implementación secuencial y CPU aumentan, dichos aumentos no son equivalentes, produciéndose una mejora con el número de puntos. En esta ocasión, la mejora en la implementación paralela en GPU ya no es algo anecdótico, sino real, llegando a obtener una mejora de hasta 18 veces mejores tiempos para el caso de 161 puntos.

4.1.3. Resultados a nivel de señal

Para finalizar este apartado, a continuación, se muestra la comparación en el resultado de una de las señales de potencial cardiaco reconstruidas mediante IP, obtenidas en CPU y GPU respectivamente, para un caso en el que la resolución mediante GPU proporcionó el mismo valor del parámetro de regularización que la solución CPU

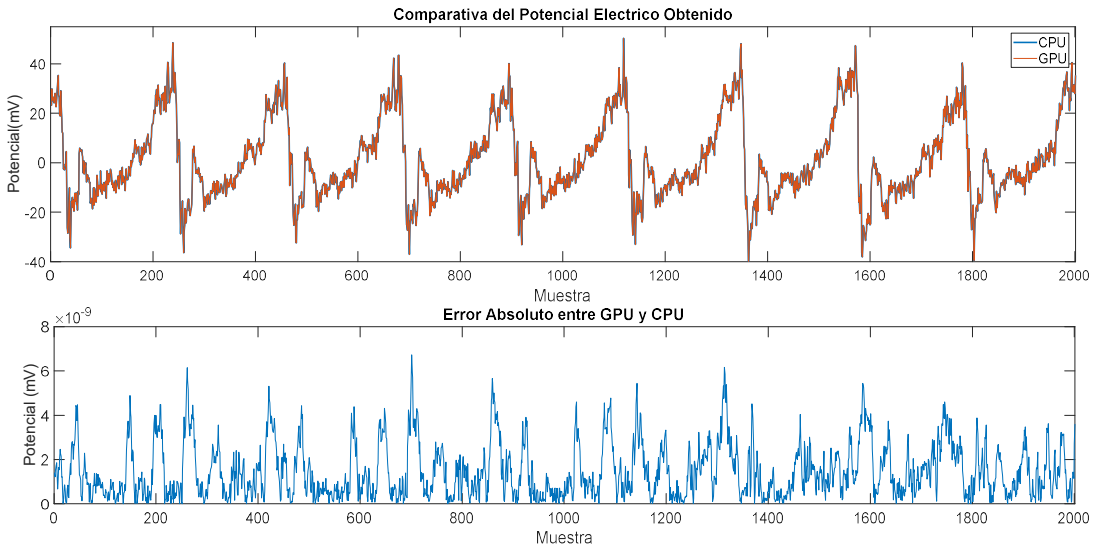


Figura 26. Comparativa de la señal potencial entre GPU y CPU

Como se observa en la figura superior, las señales son bastante similares, siguiendo la misma forma de onda, existe un pequeño error en la señal de hasta 7e-9 mV.

Por último, se va a visualizar las señales y el error cometido en el caso de que los índices no coincidan, tomando como caso el de 32 señales para el tamaño de Batch 9 [Tabla 3].

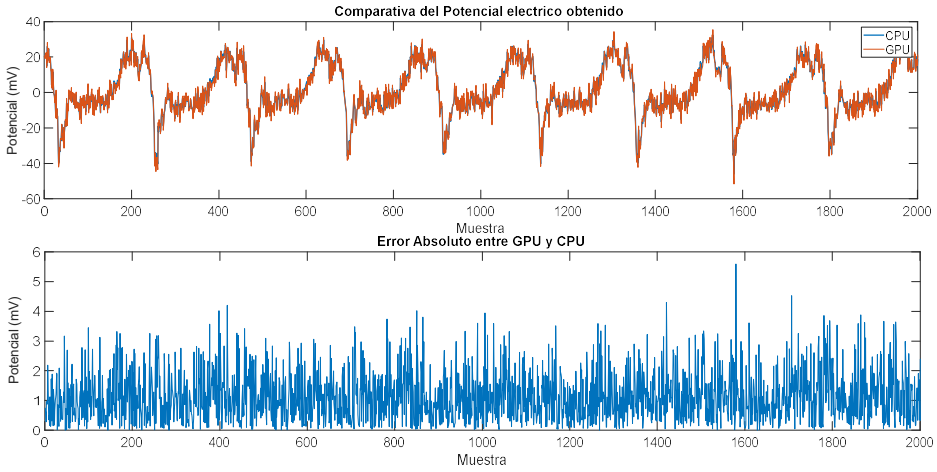


Figura 27. Comparativa de la señal potencial entre GPU y CPU con un índice de diferencia

En este caso, con los parámetros con índices 1e-8 (GPU) y 5.62e-9 (CPU), las diferencias ya apreciables, aunque, de manera aproximada, la tendencia de la señal se mantiene. Dicha diferencia no debería afectar de manera considerable al análisis de la señal, ya que se su ritmo de activación o la forma de onda se preserva prácticamente inalterada.

4.2. Detección de Frecuencia Dominante

Los resultados presentes a continuación, forman parte de la implementación del algoritmo de cálculo de frecuencias, realizando una comparación entre los resultados en GPU y CPU.

4.2.1. Variación del orden de la FFT.

El primero de los análisis a realizar, es el cómo afecta el orden de la FFT a el tiempo de procesado del algoritmo. Para ello, como ya se ha explicado en el apartado de Pruebas, se realizarán tres bancos independientes de 8192, 16384 y 32768 puntos de FFT y se analizarán los tiempos de computo en los tres casos.

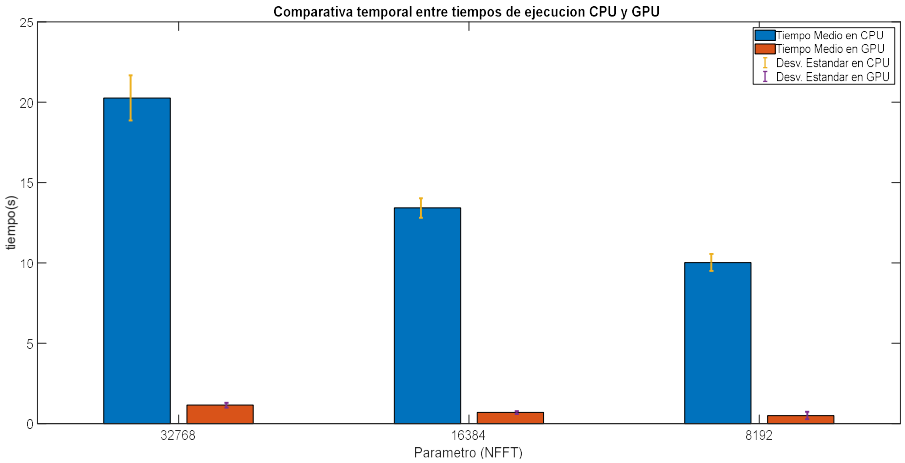


Figura 28. Comparativa Temporal CPU y GPU con orden de FFT variante

Observamos, a nivel general, una tendencia creciente ligada al crecimiento de datos a calcular.

Si comparamos los tiempos de cada prueba entre CPU y GPU, se puede ver un decrecimiento sustancial del tiempo de cómputo a nivel general en el caso de GPU. Dicha mejora está recogida en la siguiente tabla:

Speed-Up (Numero de Señales = 2048)	
Parámetro(NFFT)	CPU/GPU
8192	19.43x
16384	19.14x
32768	17.94x

Tabla 11. Mejora de tiempos de ejecución en función del orden de las FFTs.

Podemos deducir una mejora en un factor entre 18-19 veces del tiempo de GPU respecto al de CPU.

4.2.2. Variación del número de señales de entrada.

El segundo análisis a realizar para este algoritmo es el cómo varia el tiempo de procesado en función del número de señales a procesar. La siguiente figura muestra los resultados realizando una variación entre 4.000 y 500 de dicho número.

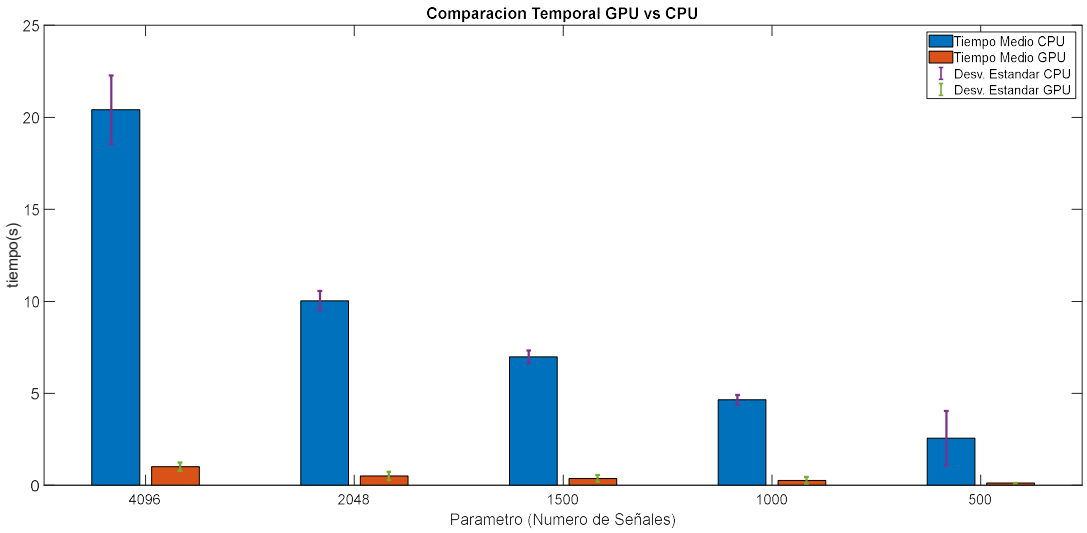


Figura 29. Comparación Temporal GPU vs CPU con Numero de Señales variable.

De nuevo, existe una tendencia directamente ligada al número de señales procesadas y una reducción drástica en el tiempo de procesado de GPU respecto a CPU, tal y como se muestra en la siguiente tabla.

Speed-Up (FFTs 8192 puntos)	
Parámetros (Numero de Señales)	CPU/GPU
4096	20.05x
2048	19.66x
1500	18.39x
1000	17.22x
500	20.48x
Speed-Up Medio	19.16x

Tabla 12. Mejora de tiempos de ejecución en función del número de señales.

En este caso, no existe una tendencia clara de aumento o decrecimiento del rendimiento de la implementación, pudiendo fijar un valor promedio de alrededor de 19 veces mejoría en GPU respecto a la CPU empleada.

4.2.3. Resultados a nivel de señal.

En esta apartado, se mostrarán dos señales diferentes, resultado del procesamiento de una de las matrices de potenciales obtenidas en los procedimientos.

En el primer caso, se mostrará la señal número 689 para una FFT de 8192 puntos en el conjunto base, comparando las salidas GPU y CPU.

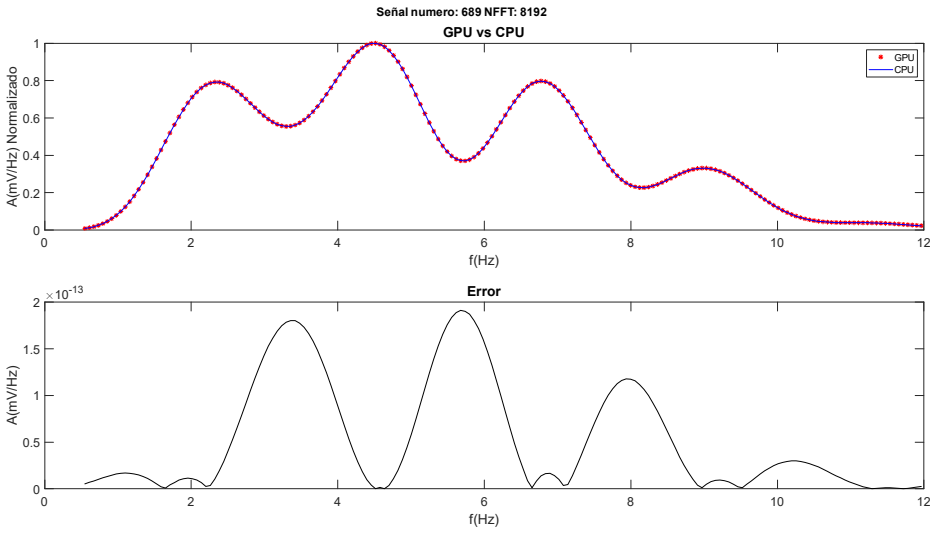


Figura 30. PSD en GPU y CPU

Como era de esperar, las densidades espectrales de potencia normalizadas en ambos casos siguen la misma forma discreta. También, encontramos un pequeño error en las señales, de hasta 2e-13 unidades respecto al valor base, error más que aceptable para la mejora de velocidad de procesado obtenida.

El siguiente caso, representa la señal número 40 para una FFT de 8192 valores.

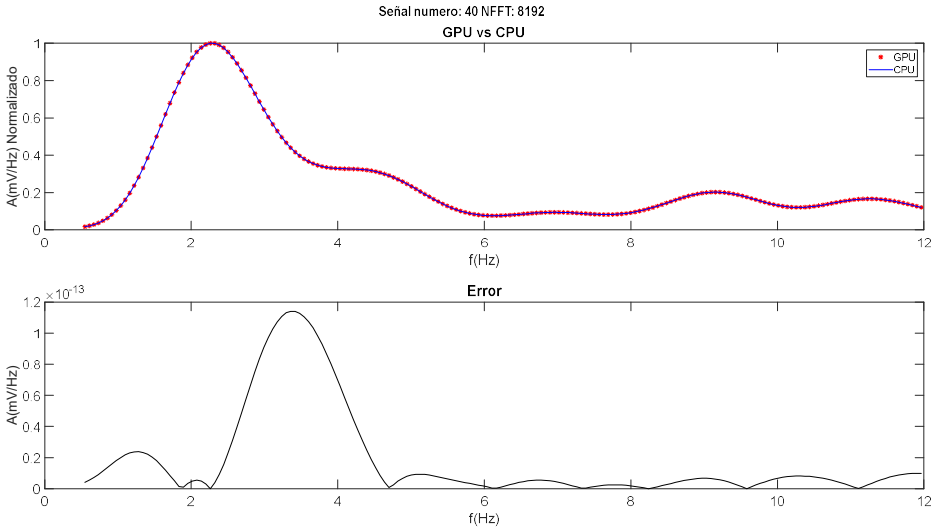


Figura 31. PSD en GPU y CPU

De nuevo las densidades de potencia coinciden de manera discreta y se produce un error dentro de unos márgenes aceptables.

No obstante, puesto que el interés de este algoritmo no es tanto el cálculo de los periodogramas, sino el resultado de las frecuencias dominantes encontradas, la siguiente grafica representa el conjunto de frecuencias dominantes encontradas para ambas implementaciones.

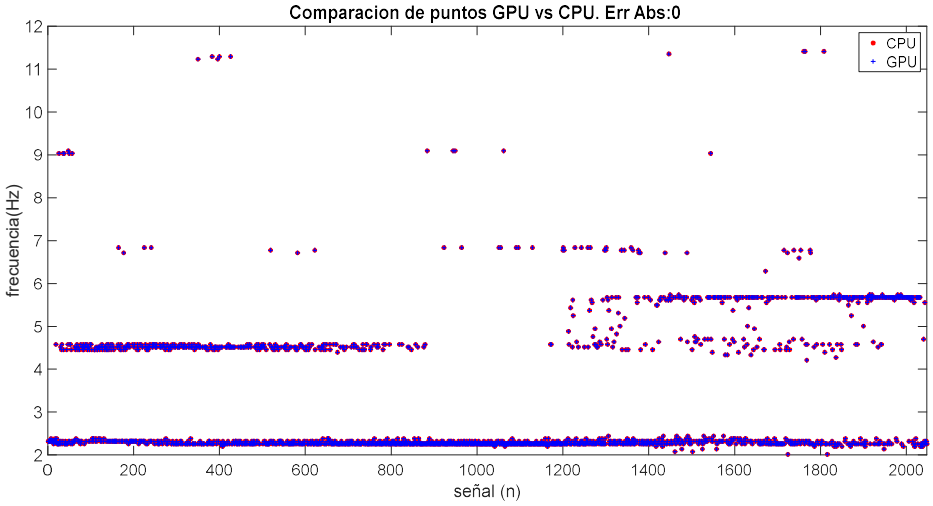


Figura 32. Comparativa de resultados frecuenciales en CPU y GPU

En esta nube de puntos, a simple vista, no se observa ninguna discrepancia entre los valores de ambas implementaciones.

A continuación, se muestra un aumento de una zona aleatoria del conjunto.

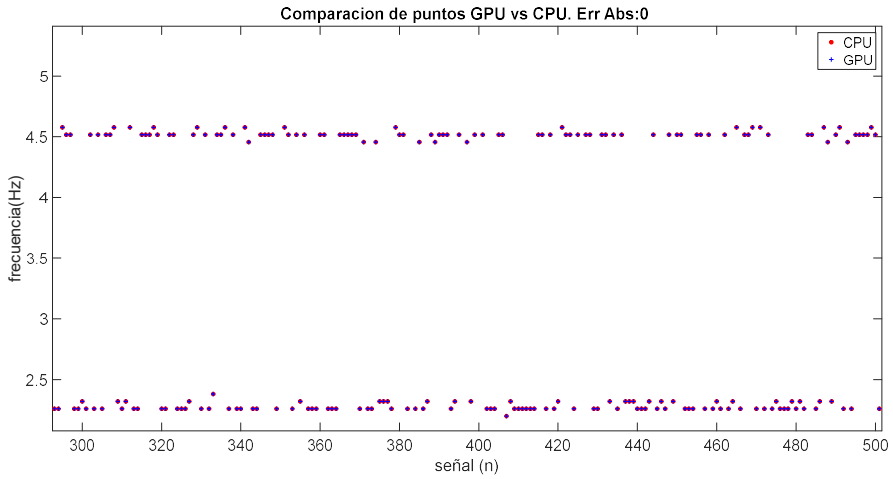


Figura 33. Aumento Figura 32

De nuevo, no se aprecian diferencias entre los puntos CPU y GPU, estando ubicados exactamente en la misma posición, o, en otras palabras, el error justo en este punto es de exactamente 0, no se producen fallos o discrepancias en el reconocimiento de los máximos.

4.3. Implementación IP + DFS

Como ya se adelantaba, una de las pruebas realizadas es la comprobación de ambos algoritmos funcionando juntos. Para ello, se han formado dos implementaciones diferentes, una directa, modificando el código base en *VS* para vincular las salidas y entradas correspondientes directamente en GPU, que llamaremos "*IP_DFS*"; y otra indirecta, en que la vinculación se realizara por medio de *MATLAB*, redirigiendo por Memoria Host los datos del módulo de IP al de DFS, que llamaremos "*IP+DFS*". También, mediremos el tiempo de ejecución para el caso de realizar ambos algoritmos en CPU y los resultados.

A continuación, se muestran los resultados de las implementaciones del conjunto de algoritmos desarrollados.

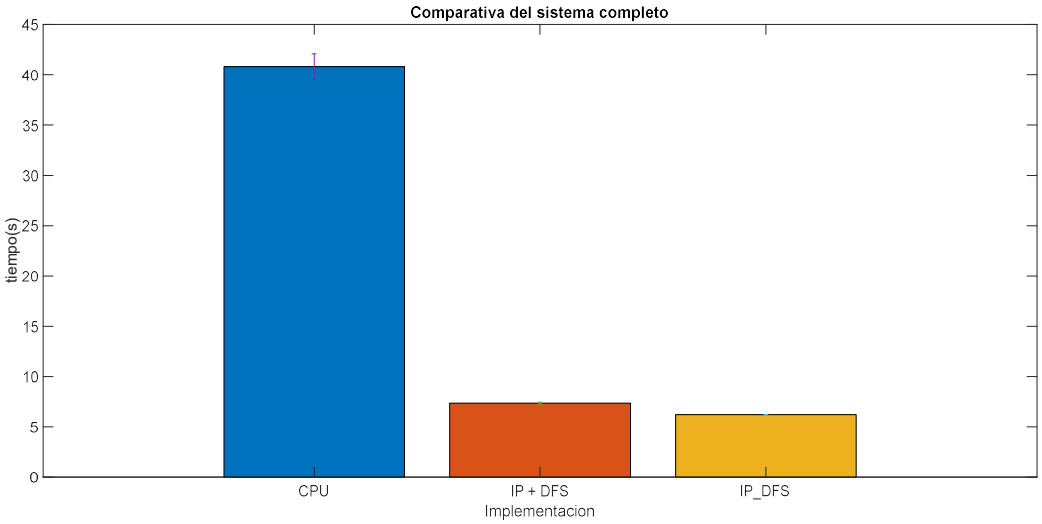


Figura 34. Comparación temporal

En este caso, la configuración pura en GPU, *IP_DFS*, resulta más eficiente temporalmente que la *IP + DFS*, estando está vinculada a través de *MATLAB*, con una diferencia media de 1.1 segundos, atribuibles por una parte por el hecho de ya no pasar por la memoria host, y por otra a una leve modificación necesaria del código para calcular la media directamente en la GPU.

Con esto, podemos fijar una serie de marcadores de speed-up como base para futuras implementaciones:

Speed-up		
CPU/(IP+DFS)	CPU/IP_DFS	IP + DFS / IP_DFS
5.54x	6.56x	1.18x

Mediante la implementación *IP_DFS*, obtendríamos un factor de aceleración final para ambos algoritmos de 6.56. Esto contrasta con los resultados de los módulos por separados con aproximadamente 4.5 y 19.5 en los respectivos módulos.

La explicación más plausible, es que, por un lado, los procesos secuenciales en este punto están limitando la capacidad total del sistema, y por otro que algunos segmentos del código pueden y deberán ser optimizados.

No obstante, en este caso, se han conseguido reducir los 41 segundos de la ejecución paralela a aproximadamente 6.2 segundos, de los cuales 5.6 corresponderían al problema inverso y 0.5 segundo aproximadamente serian del cálculo de periodogramas.

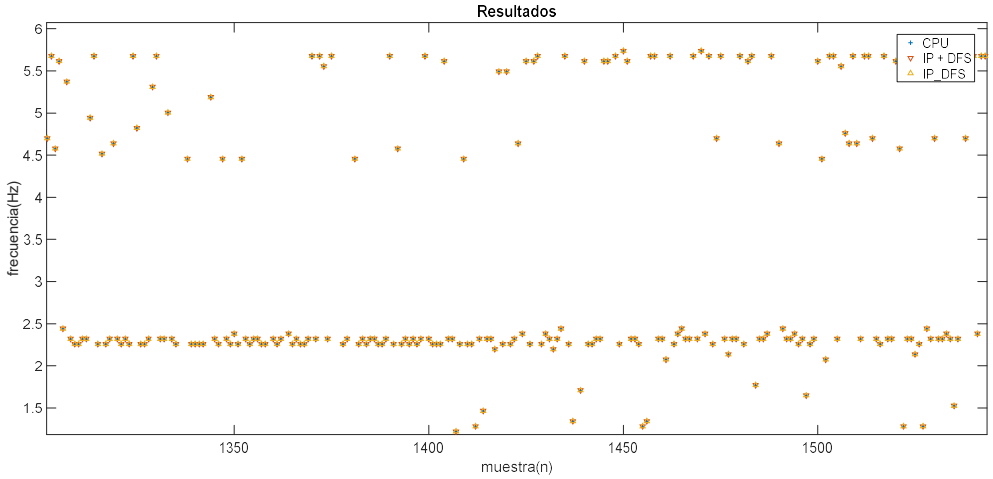


Figura 35. Comparación de resultados de las implementaciones.

En este caso, volvemos a comprobar que la exactitud es del 100% en todas y cada una de las señales, lo que sumado a la ganancia de tiempos de la implementación puramente en GPU, asegura una mejora general en el proceso completo.

Como adelantábamos unos apartados atrás, un error en la detección del índice óptimo es posible que produzca fallos en las posteriores etapas.

Para ilustrar este caso, a continuación, se muestra el resultado del error cometido en la detección de la frecuencia al introducir exactamente el caso de 32 señales para un Batch de tamaño 9, cuyos índices eran 1e-8 (GPU) y 5.62e-9 (CPU), para una longitud de FFT de 8192 en el caso de CPU/GPU Secuencial y GPU Batch respectivamente.

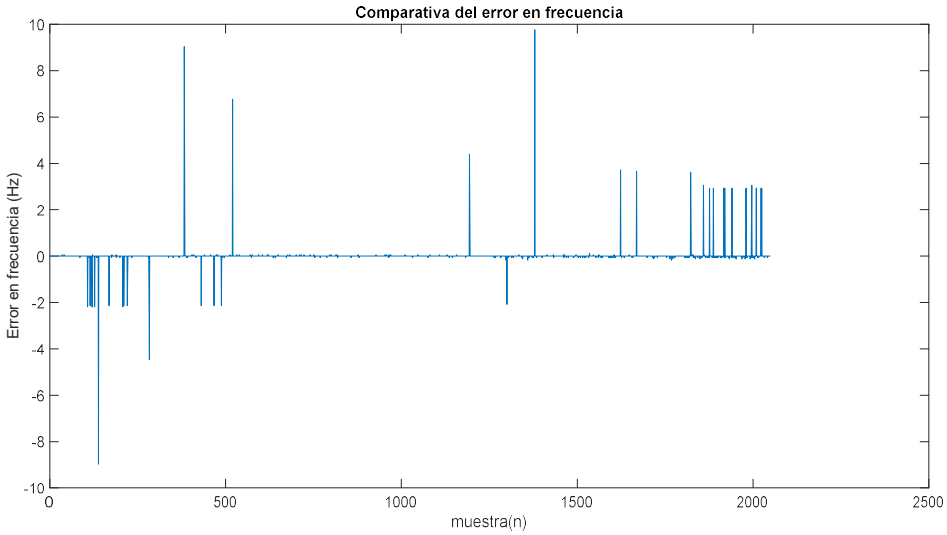


Figura 36. Error relativo para índices distintos.

Se aprecia que existe una diferencia de hasta 10 Hz en determinadas señales, aunque no existe una tendencia definida en ese error.

Para este caso, si contabilizáramos el número de muestras en los que la diferencia absoluta supera un determinado umbral, podemos sacar la siguiente tabla:

	Aciertos	Fallos (Globales)	% Aciertos
1e-12	1812	236	88.48%
0.01	1812	236	88.48 %
0.1	1999	49	97.61 %
0.5	2015	33	98.39 %

Tabla 13. Discretización del error cometido.

Observamos que, aproximadamente el 98% de las frecuencias entrarían dentro de un intervalo menor de 0.5 Hz de error, teniendo el 88.48% un error nulo.

Si analizamos algunas de las señales que producen un error importante, y visualizado por separado en el espectro obtenemos las siguientes gráficas, que representan los dos casos principales detectados que podrían causar el error.

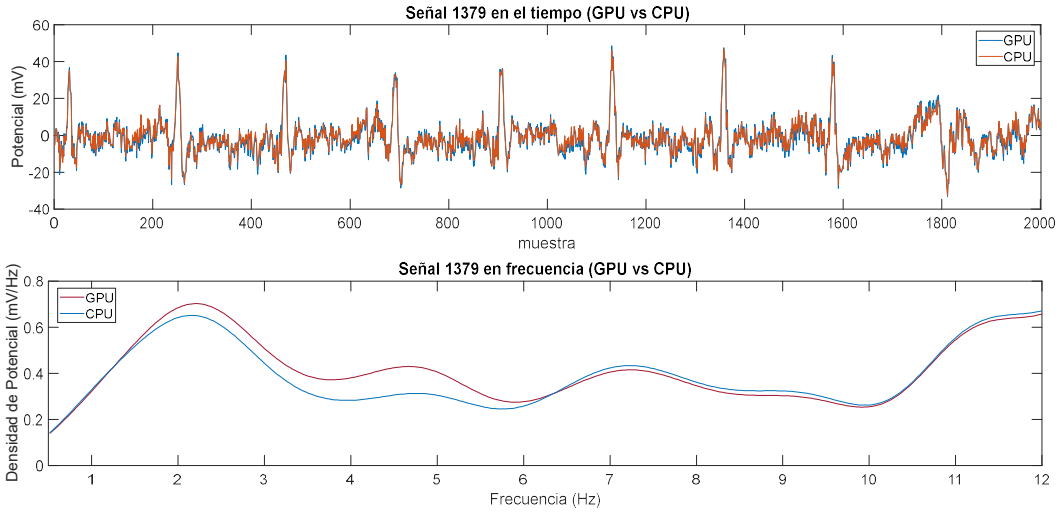


Figura 37. Comparación de los espectros con error 1

En este caso, observamos una discrepancia extrema correspondiente a la señal 1379. La misma representa el caso en el que el máximo de ambas señales esta fuera del rango analizado, debido a la gran presencia de ruido de alta frecuencia. Debido a la fluctuación de la señal, se detecta el máximo de la CPU en el extremo del rango analizado (12 Hz), mientras que en GPU encuentra su máximo local en 2.1973 Hz, valor correcto de la frecuencia de activación de la señal cardíaca.

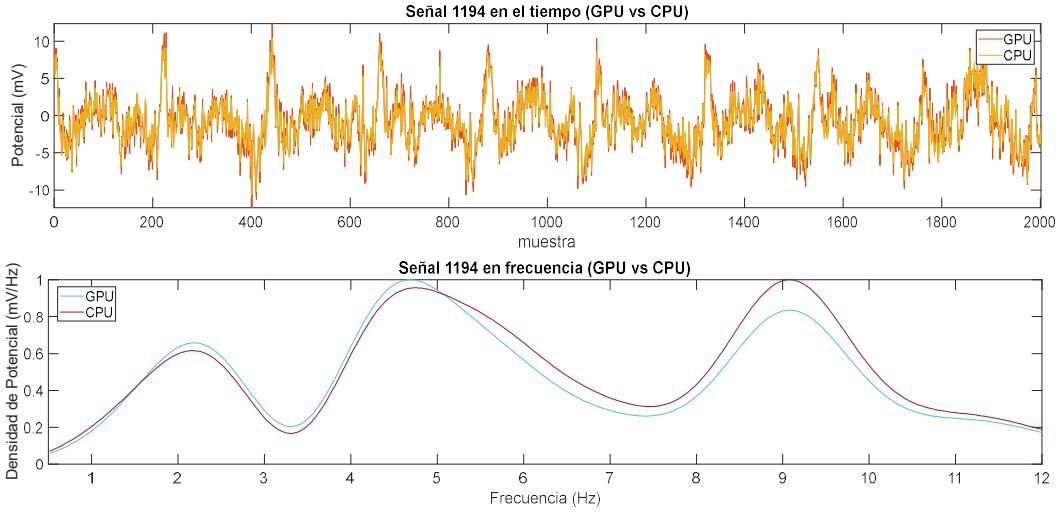


Figura 38. Comparación de los espectros con error 2

Este es otro de los posibles casos de discrepancia, encontrada en los valores de aproximadamente 3-5 Hz de diferencia. En este caso, perteneciente a la señal 1194, ambos máximos se encuentran dentro del rango analizado, y abarcarían los casos en las que pequeñas diferencias en el contenido espectral proporcionan grandes divergencias en la detección. En este caso, el error es debido a la presencia de dos picos espectrales de similar amplitud, a 4,5 y 9 Hz respectivamente, cuyas amplitudes varían ligeramente en las soluciones GPU/CPU y hacen que en cada solución sea mayor uno de los 2 picos.

Capítulo 5. Conclusiones

Como se ha comprobado a lo largo de este documento, la posibilidad y utilidad del empleo de dispositivos GPU para acelerar algoritmos es una realidad. En este caso, se han obtenido mejoras sustanciales en el tiempo de cómputo, sacrificando algo de precisión en los pasos intermedios, pero sin que la misma aparezca reflejada en el resultado final, a nivel general.

No obstante, este análisis no debe ser tomado como una prueba absoluta de la capacidad sino como el inicio de un estudio mayor, puesto que solo se ha comparado una pareja de dispositivos, siendo necesario una comparación más exhaustiva, no solo con dispositivos de otro tipo, sino con más dispositivos dentro de las respectivas familias.

Por otro lado, cabe indicar que, aunque los márgenes de mejora son aceptables, existiría la posibilidad de aumentarlos, ya no con hardware más potente, sino con una implementación diferente y/o más optimizada de las diferentes partes que componen los algoritmos, siendo la actual, una de las posibles interpretaciones que se le puede dar, y siendo algunas de las posibles mejoras, la optimización en el acceso a memoria mediante técnicas de lectura/escritura asíncrona ,aumentar el grado de paralelización por independencia de datos a costa de aumentar la memoria utilizada o la implementación distribuida del algoritmo en un entorno multi-GPU.

En lo relativo a los objetivos y tareas fijados:

- *Reducir el tiempo de ejecución de los algoritmos IP y DFS.* Se ha conseguido una mejora sustancial en los tiempos de ejecución a nivel general en ambos algoritmos.
- *Implementar los algoritmos en un entorno paralelo.* Se ha comprobado que dichos algoritmos son posibles de implementar de manera paralela, ya sea a nivel de cálculo y/o de síntesis. En el caso del algoritmo del IP, se realizó primeramente una paralelización a nivel de cálculo y posteriormente a nivel de síntesis, mientras que en el DFS se realizó una implementación directa a nivel de síntesis.
- *Realizar una comparación de los resultados del modelo paralelo y el original.* Se ha comprobado que, aunque se sufre una pequeña perdida de precisión en el cálculo en el modelo en GPU, siempre y cuando los datos sean matemáticamente estables la solución final es equivalente.
- *Realizar una comparación temporal de los modelos.* Se ha comprobado que la paralelización en los sistemas GPU permite una reducción notable sobre los modelos de computación CPU.
- *Realizar una optimización base de dichos algoritmos*. Siendo, en el caso del IP, la implementación paralela por salto de índice que permite reducir el tiempo de procesado a costa de condicionalidad, y en el caso de la implementación conjunta, una reducción ligada a los tiempos de acceso al integrar ambos algoritmos en una macrofunción.

Con esto, se espera que esta implementación, ayude e incentive el desarrollo e implantación de este tipo de técnicas, dentro de las posibilidades de diagnóstico e intervención en procesos médicos, en este caso en diagnostico cardiaco. Se ha demostrado que la implementación es posible, existiendo ya los medios y herramientas necesarios para ello, aunque será necesario aun un estudio más exhaustivo para verificar de manera empírica la validez de estas implementaciones, con el fin principal de realzar su atractivo de cara a su uso.

Capítulo 6. Proyectos Futuros

Como se planteaba en el apartado de conclusiones, este trabajo seria solo el comienzo de lo que podríamos considerar una serie de proyectos de cara al estudio del procesado en GPU, surgiendo así una serie de líneas de investigación potenciales, como pueden ser:

- Comparativa de diversos dispositivos dentro del campo de las GPU.
- Comparativa de diversos dispositivos paralelizables (GPU/FPGA/DSP).
- Co-procesamiento en GPU y CPU.
- Rendimiento y rentabilidad de diferentes dispositivos.

Estas líneas, apuntarían, en general, a realizar una valoración en mayor profundidad de la utilidad del sector de las GPUs para labores de GPGPU, más específicamente en el sector de la salud, con el fin de, a largo plazo, ayudar a lo que son ambos sectores, a comprender lo que son las necesidades fuera de un contexto mayoritariamente teórico, y su desarrollo dentro del ámbito cotidiano, convirtiéndose no solo en una posibilidad, sino en el modelo a seguir.

Por otro lado, se podrían definir otro conjunto de líneas de investigación diferentes, más ligadas a lo que serían los algoritmos en sí, con el fin de solventar las diferencias ligadas a la implementación GPU de estos y optimizar su rendimiento.

- Análisis y Re-estructuración del algoritmo IP en un entorno paralelo.
- Análisis y Re-estructuración del algoritmo DFs en un entorno paralelo.

Estas líneas, irían dirigidas a realizar una verdadera estructuración paralela de los algoritmos, puesto que, en el fondo, la implementación actual es básicamente una traducción directa del algoritmo base a un entorno paralelo, sin entrar en un análisis exhaustivo que garantizase la completa utilización de dicho entorno.

Capítulo 7. Referencias

[1] J. Montes-Santiago et al, Características y costes de los pacientes ingresados por arritmias cardiacas en España, Revista Clínica Española*, 213(5):235-239*

[2] JJ Gomez Doblas, J Muñiz, JJ Alonso, G. Rodriguez-Roca et al, Prevalencia de fibrilación auricular en España. Resultados del estudio OFRECE, Revista Española de Cardiología, *67(4):259:269*

[3] M. Haisaguerre et al, Driver domains in persistent atrial fibrillation. Circulation 2014*; 130:530-538*

[4] MS Guillem, AM Climent, M Rodrigo, Alejandro Liberos et al, Non-invasive identification of atrial fibrillation drivers, Computing in Cardiology Conference (CinC), *121-124*

[5] J. Pedron-Torrecilla, Noninvasive estimation of epicardial dominant highfrequency regions during atrial fibrillation. J Cardiovasc Electrophysiol 2016; *27:435-442.*

[6] Tikhonov AN. On the solution of incorrectly posed problems and the method of regularization. Sov Math Dokl 1963; *4:1035-1038*

[7] Horácek BM, Clements JC. The inverse problem of electrocardiography: a solution in terms of single- and double-layer sources of the epicardial surface. Math Biosci 1997; *144:119-154.*

[8] Dutacheau J, Sacher F et al, Performance and limitations of noninvasive cardiac activation mapping, Hearth Rhythm *16(3):435-442*

[9] Guillem MS, Climent AM, Millet J, Arenal A, Fernández-Avilés F, Jalife J, Atienza F, Berenfeld O. Noninvasive localization of maximal frequency sites of atrial fibrillation by body surface potential mapping. Circ Arrhythm Electrophysiol *2013; 6:294-301****.***

[10] M Rodrigo, AM Climent, A Liberos, F Fernández-Avilés, O Berenfeld et al Highest dominant frequency and rotor positions are robust markers of driver location during noninvasive mapping of atrial fibrillation: a computational study. Heart rhythm *14 (8), 1224-1233*

[11] H. Rani, R. Medra et al, Power Spectrum Estimation using Welch Method for various Window Techniques, *IJSRET, ISSN 2278 –0882*

[12] Stratix X Overview, Intel Corporation,
https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10.html, visitado por última vez el 08 de Julio de 2019

[13] Understanding Peak Floating-Point Performance Claims, Intel Corporation, Michael Parker*,*
https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01222-understanding-peak-floating-point-performance-claims.pdf ,visitado por última vez el 24 de Junio de 2019.

[14] GPU vs FPGA Performance Comparison, Berten DSP,
http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf, visitado por última vez el 08 de Julio de 2019

[15] Cyclone 10 Overview, Intel Corporation,
https://www.intel.com/content/www/us/en/programmable/documentation/grc1488182989852.html, visitado por última vez el 08 de Julio de 2019

[16] CPU comparison, lanoc,
https://lanoc.org/review/cpus/7870-intel-i9-9900k?showall=1 *, visitado por última vez el 08 de Julio de 2019*

[17] A Survey and Benchmarks of Intel® Xeon® Gold and Platinum Processors, colfaxresearch,
https://colfaxresearch.com/xeon-2017/ , visitado por última vez el 09 de Julio de 2019

[18] Ryzen Family, Wikipedia,
https://es.wikipedia.org/wiki/Ryzen, visitado por última vez el 09 de Julio de 2019

[19] ADSP-TS101 Datasheet, Analog Devices,
https://www.analog.com/media/en/technical-documentation/data-sheets/ADSP-TS101S.pdf, , visitado por última vez el 08 de Julio de 2019

[20] 66AK2H14 Datasheet, Texas Instrument,
http://www.ti.com/lit/ds/symlink/66ak2h14.pdf, visitado por última vez el 08 de Julio de 2019

[21] RTX 2080 Overview, NVIDIA CORPORATION,
https://www.nvidia.com/es-es/geforce/graphics-cards/rtx-2080/, visitado por última vez el 08 de Julio de 2019

[22] GTX 960 Specs, TechPowerUp,
https://www.techpowerup.com/gpu-specs/geforce-gtx-960.c2637, visitado por última vez el 08 de Julio de 2019

[23] Quadro P6000 Specs, NVIDIA CORPORATION,
https://www.nvidia.com/object/quadro-graphics-with-pascal.html, visitado por última vez el 08 de Julio de 2019

[24] Quadro RTX 6000 Overview, NVIDIA CORPORATION,
https://www.nvidia.com/es-es/design-visualization/quadro/rtx-6000/, visitado por última vez el 08 de Julio de 2019

[25] Ian Buck, The Evolution of GPUs for General Purpose Computing, NVIDIA Corporation,
https://www.nvidia.com/content/gtc-2010/pdfs/2275_gtc2010.pdf, visitado por última vez el 25 de Junio de 2019.

[26] Mark Harris, A Brief History of GPGPU, NVIDIA Corporation,
http://cs.unc.edu/xcms/wpfiles/50th-symp/Harris.pdf visitado por última vez el 24 de Junio de 2019.

[27] cuBLAS Toolkit Documentation, NVIDIA CORPORATION,
https://docs.nvidia.com/cuda/cublas/index.html, visitado por última vez el 22 de junio de 2019.

[28] cuSolver Toolkit Documentation, NVIDIA CORPORATION,
https://docs.nvidia.com/cuda/cusolver/index.html, visitado por última vez el 22 de junio de 2019.

[26] cuFFT Toolkit Documentation, NVIDIA CORPORATION,
https://docs.nvidia.com/cuda/cufft/index.html, visitado por última vez el 10 de junio de 2019.