Aceleración de Time-Series sismográficas en Python

Francisco López[1], Thomas Grass[1], Rafael Asenjo[2], Angeles Navarro[2]

Resumen— **Python se ha convertido en un lenguaje de programación muy popular, pero también es uno de los menos eficientes en términos de prestaciones y consumo energético. Este artículo describe el proceso que hemos seguido para acelerar una aplicación Python de tratamiento masivo de datos orientada a las Time-Series sismográficas, de manera que al usuario final se le sigue ofreciendo la productiva interfaz Python que tanta aceptación tiene. Este proceso se ha desplegado siguiendo una estrategia en tres fases. En la primera fase se ha aplicado un cambio algorítmico cuyo objetivo ha sido reducir la complejidad computacional del principal kernel (hot-spot) del código: las correlaciones cruzadas. Para ello se ha optado por implementar dichas correlaciones aplicando el Teorema de la Convolución. En la segunda fase se ha aplicado un cambio de modelo de programación que ha consisitido en la implementación en C++ del kernel, lo que nos ha permitido la utilización de la muy optimizada biblioteca FFTW. En la tercera fase, gracias al cambio del modelo de programación, aplicamos optimizaciones conscientes de la arquitectura, entre ellas OpenMP, para aprovechar los nodos multicore de nuestro sistema, o ArrayFire que nos permite hacer uso de aceleradores gráficos (con soporte en CUDA y OpenCL). Tras este proceso de optimización hemos obtenido una aceleración de 6121x sobre la aplicación original de partida.**

Palabras clave— **Time-series, procesado de señal, computación de altas prestaciones, OpenMP, ArrayFire, FFTW.**

I. Introducción

LAS aplicaciones de tratamiento masivo de datos están a la orden del día. La gran cantidad de sensores y otros dispositivos, como *smartphones* y componentes IoT, registran múltiples magnitudes que varían a lo largo del tiempo. Dentro de este marco, surge el término de Time-Series, que consisten en señales discretas que representan los distintos valores que una magnitud toma con el transcurso del tiempo. Por ello, invertir tiempo y esfuerzo, no sólo en el desarrollo de aplicaciones que extraigan información relevante, sino también en optimizar las ya existentes, se erige como una necesidad imperiosa por varios motivos.

El primer motivo es el tiempo que el usuario de estas aplicaciones tiene que esperar hasta que obtiene resultados con los que poder trabajar. Cuanto menos tarden nuestras aplicaciones en obtener los mismos resultados a partir de un mismo conjunto de datos, más comparaciones, estudios y análisis se podrán realizar. El segundo motivo es la energía consumida por los equipos que realizan el cómputo. En términos generales, cuanto menos tiempo tarde en completarse nuestra ejecución, menos energía tendremos que invertir. El tercer motivo es poder hacer frente a conjuntos de datos de tamaños mayores, muchas veces necesarios para poder aumentar la precisión de los resultados. Este motivo está relacionado con el primero, pero no siempre es únicamente una cuestión de tiempo, sino también de otros recursos, como memoria.

Por todos estos motivos, nos decidimos por optimizar una aplicación que trabaja con Time-Series sismográficas, aplicando conocimientos de procesado de señal. La estrategia de optimización que ilustramos en este trabajo es perfectamente extrapolable a Time-Series que representen cualquier otro tipo de magnitudes, pues nuestra aproximación es independiente del dominio de aplicación.

El problema en cuestión consiste principalmente en realizar correlaciones cruzadas, que constituyen el núcleo computacional del principal kernel del código. Para optimizar este kernel se ha utilizado una estrategia en tres fases. En la primera fase se ha hecho uso del Teorema de la Convolución [1], que nos ha permitido reducir la complejidad computacional del kernel mediante un cambio de algoritmo. En la segunda fase se ha aplicado un cambio de modelo de programación que ha consisitido en la implementación en C++ de dicho kernel. El cambio de modelo de programación nos ha permitido explotar tres factores:

i Reducir la sobrecarga que supone en Python la ejecución interpretada y la resolución de tipos de datos en tiempo de ejecución.

ii Habilitar la invocación de librerías optimizadas para ciertos dominios de aplicación, en nuestro caso hemos podido explotar la muy optimizada biblioteca FFTW [2]. Aunque invocamos esta librería en C++, desarrollaremos un interfaz que hará posible al usuario seguir trabajando desde Python en todo momento.

iii Habilitar el uso de frameworks de optimización conscientes de la arquitectura.

Precisamente, en la tercera fase de nuestra estrategia, gracias al cambio del modelo de programación, aplicamos optimizaciones conscientes de la arquitectura, entre ellas OpenMP [3] - para aprovechar los nodos multicore de nuestro sistema - o ArrayFire [4] - que nos permite hacer uso de aceleradores gráficos (con soporte en CUDA y OpenCL)-.

La estructura del artículo es la siguiente: en la Sección II se explicarán en profundidad las peculiaridades del problema y se refrescarán los conocimientos sobre el Teorema de la Convolución. A continuación,

[1]RWTH Aachen University, Institute for Communication Technologies and Embedded Systems. e-mail: {francisco.lopez, thomas.grass}@ice.rwth-aachen.de.

[2]Universidad de Málaga, Andalucía Tech, Depto. de Arquitectura de Computadores. e-mail: {asenjo,angeles}@ac.uma.es.

la Sección III describe como aplicamos la metodología en tres fases, mencionada previamente. Para cada una de las fases e implementaciones presentadas se incluye la evaluación experimental sobre la mejora que se consigue respecto a la versión Python original. Finalmente, la Sección IV resume algunos trabajos relacionados y la Sección V expone las conclusiones y las líneas de trabajo futuro.

II. Situación de partida

A. Visión general

El principal propósito del trabajo realizado es acelerar, en la medida de lo posible, el rendimiento de una aplicación de tratamiento masivo de datos. La versión original del código fue desarrollada por el personal del *School of GeoSciences*, de la *University of Edinburgh*, para tratar datos recogidos por sensores de actividad sísmica en las Islas Galápagos. Dicha versión original, que estaba escrita única y exclusivamente en Python, consiste en varias etapas de procesamiento; sin embargo, la mayor parte del tiempo de ejecución corresponde a la primera fase, en la que se han concentrado nuestros esfuerzos.

Esta primera etapa consiste en tomar los datos procedentes de los sensores, que han sido almacenados previamente, y realizar correlaciones cruzadas dos a dos. De cada una de estas señales de correlación se extraen los elementos máximos y mínimos, así como los desplazamientos para los que se producen, dentro de un rango que se puede introducir como parámetro de entrada. De este modo, los resultados que se extraen, originalmente, son cuatro matrices donde se pueden encontrar los máximos, mínimos y desplazamientos necesarios para ambos casos, de la correlación cruzada de cualquier par de Time-Series de entrada. En la Figura 1 puede observarse que a partir de un conjunto de datos con un determinado número de Time-Series se extraen cuatro matrices cuadradas, con tantos elementos en cada fila y columna como Time-Series de entrada se tengan. También queda denotada la peculiaridad triangular de la estructura de datos de salida, así como la variabilidad en las longitudes de las Time-Series que conforman el conjunto de datos de entrada.

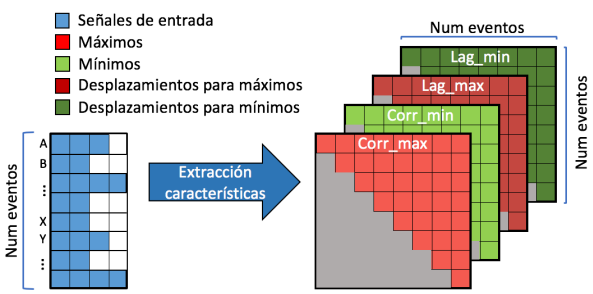

Fig. 1. Visión general del problema.

Una vez obtenidas estas matrices, se aplica un algoritmo de *clustering* para identificar familias de terremotos. Tanto la duración como el tipo de terremoto que conforman las familias ofrecen información sobre el sistema físico que los genera [5], [6].

El parámetro de entrada, *shift*, fija el rango en torno al centro de la señal de correlación cruzada donde se van a buscar los máximos y mínimos. Además, es reseñable indicar que se considera como centro de la correlación el valor resultado de alinear los elementos centrales de ambas Time-Series.

En la Figura 2 puede observarse un ejemplo de la operación de correlación cruzada que ha de realizarse entre cada par de Time-Series. Se observa que el *shift* toma un valor de tres, por lo que se buscarán los máximos y mínimos en tantos elementos alrededor del elemento central. Puede comprobarse que el elemento central es el resultado de realizar el producto escalar (*dot product*, en este caso igual a $X \cdot Y^T$) entre las dos Time-Series y que el resto de elementos se hallan desplazando una de las dos Time-Series. Así, los elementos que caigan fuera de la multiplicación se tomarán como nulos. En el ejemplo, cuando Y se desplaza se introducen ceros por el lado contrario hacia el que se realiza el desplazamiento. Todos los valores de CC son calculados con la siguiente expresión:

$$CC[i] = X[n] \cdot Y[n-i], i \in (-N, N)$$

donde N representa la longitud de las Time-Series, que suponemos del mismo número de elementos. Al contrario de lo que pueda considerarse en primera instancia, los valores finales de la correlación cruzada no dependen únicamente del número de multiplicaciones válidas sino también de los valores a multiplicar. Con esto, puede observarse que, en el ejemplo, el máximo no se halla en el elemento central, sino en el $lag\_max = 1$; por su parte, el mínimo se halla en $lag\_min = -1$. El máximo y mínimo correspondientes a estos desplazamientos son 117 y 58, respectivamente. Estos valores siempre cumplirán las ecuaciones siguientes:

$$CC[lag\_max] = max(CC[i]) = corr\_max$$

$$CC[lag\_min] = min(CC[i]) = corr\_min$$

Con todo lo mencionado previamente, se presenta ante nosotros un problema de carácter triangular, dada la simetría que desprenden las matrices de salida. Esto va a condicionar, en gran medida, la forma en que se implementarán las distintas versiones paralelas del código. No menos importante es el hecho de que al crear no sólo una, sino 4 matrices, donde la mitad de los elementos finales sabemos que serán ceros, estamos desperdiciando prácticamente la mitad de la memoria consumida por el programa. Este problema, como se tratará más adelante, ha sido abordado satisfactoriamente, de forma que únicamente se emplee la memoria que realmente sea necesaria.

En la Figura 3 se muestra el código Python de partida, que se irá optimizando en diferentes etapas, tal y como se describe en las siguientes secciones. En las líneas 2-7 se halla el número de eventos (Time-Series) y, en función de dicho valor, se crean las cuatro matrices de salida: `Corr_max`, `Corr_min`, `Lag_max`

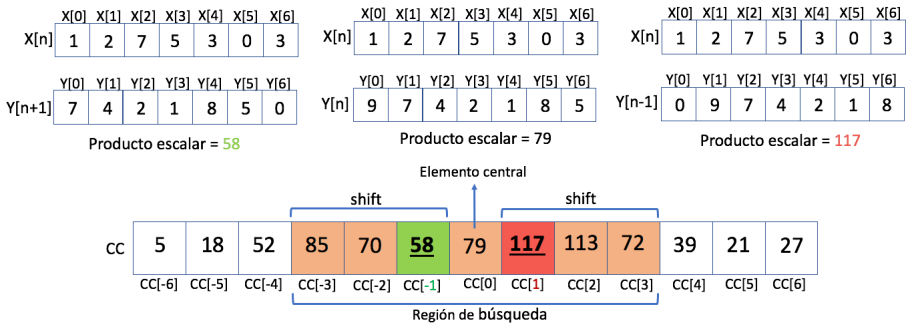

Fig. 2. Ejemplo de correlación cruzada de dos Time-Series

```
1  def XCM2(events):
2      n_events = len(events)
3
4      Corr_max = zeros((n_events,n_events))
5      Lag_max = zeros((n_events,n_events))
6      Corr_min = zeros((n_events,n_events))
7      Lag_min = zeros((n_events,n_events))
8
9      for i in range(n_events):
10         for j in range(i, n_events):
11             xcorrij = xcorr(events[i], events[j], 250, full_xcorr=True)
12             #Returns index and CC[i] for max(abs(CC[i])) --including negative values--
13             Lag_min[i,j] = xcorrij[0]
14             Corr_min[i,j] = xcorrij[1]
15             if xcorrij[1]<0.:
16                 #Return highest positive CC[i] and index (xcorrij[2] contains CC)
17                 Lag_max[i,j], Corr_max[i,j] = xcorr_max(xcorrij[2],abs_max=False)
18             else:
19                 Lag_max[i,j] = xcorrij[0]
20                 Corr_max[i,j] = xcorrij[1]
21
22     return Corr_max, Lag_max, Corr_min, Lag_min
```

Fig. 3. Código original en python.

y `Lag_min`, tal y como se ha explicado previamente. A continuación, comienza el doble bucle que recorre el triángulo superior de las matrices de salida. En la línea 11 se lanza el cómputo de la correlación cruzada correspondiente. Posteriormente, en las líneas 13-20 para cada señal de correlación se busca el máximo, el mínimo y los desplazamientos correspondientes. Sin embargo, es peculiar el hecho de que, si el valor absoluto del máximo es mayor que el valor absoluto del mínimo (que suele ser negativo), éste último tomará el valor del primero. Esto es así por motivos derivados del sentido físico de las señales sísmicas y las Time-Series que las representan.

B. *Teorema de la convolución*

El Teorema de la convolución supone un pilar fundamental sobre el que descansan, en gran medida, los avances conseguidos. Éste teorema afirma que la convolución puede realizarse en el dominio de la frecuencia como una simple multiplicación de las señales. Sin embargo, el propósito del código original no es computar convoluciones, sino correlaciones cruzadas, para lo cual se hará uso de la teoría de señales y sistemas lineales e invariantes en tiempo. Así, para obtener una convolución de dos Time-Series la cadena de operaciones consiste en realizar las transformadas de Fourier de cada Time-Series por separado, realizar la multiplicación elemento a elemento y, finalmente, computar la transformada de Fourier inversa del resultado. La única diferencia con la correlación cruzada estriba en tener que invertir el eje de tiempos de una de las señales antes de realizar la transformada de Fourier, tal y como se muestra en las ecuaciones siguientes, donde x e y denotan dos Time-Series de entrada:

$$Conv(x,y) = IFFT[FFT[x[n]] * FFT[y[n]]]$$

$$Cross{-}corr(x,y) = IFFT[FFT[x[n]] * FFT[y[-n]]]$$

No son pocas las implementaciones de algoritmos de comparación de Time-Series que hacen uso de esta propiedad; sin embargo, ha de tenerse en cuenta la longitud de las señales de entrada, es decir, el número de elementos que componen cada una de ellas. Así, para Time-Series compuestas por menos de un cierto número de elementos, no es beneficioso realizar la convolución en frecuencia. En nuestro caso, al pasar del dominio del tiempo al dominio de la frecuencia es posible reducir la complejidad del algoritmo de correlación cruzada de $\mathcal{O}(n^2)$ a $\mathcal{O}(n\log(n))$, pues, además, realizaremos un *padding* sobre las señales de entrada para que el número de elementos alcance la siguiente potencia de dos. Con este *padding* conseguiremos obtener una mejor eficiencia al calcular las FFT e IFFT.

C. *Setup experimental*

A lo largo del proyecto llevado a cabo se ha hecho uso del *clúster* MinoTauro [7] del Barcelona Super-

computing Center (BSC). Éste está compuesto por 38 servidores Bullx R421-E4, donde cada uno, a su vez, ofrece:

- CPU: 2 procesadores Intel(R) Xeon E5-2630 v3 @ 2,4 GHz con 20 MB de cache L3.
- GPU: 2 tarjetas NVIDIA(R) K80.
- 128 GB de memoria principal.
- *Peak Performance*: 250,94 TFlops
- Sistema operativo: RedHat Linux 6,7

Pra evaluar las distintas implementaciones, trabajaremos con un *dataset* formado por 6659 Time-Series, compuestas por 1501 puntos. Éste será el conjunto de datos de referencia. Originalmente, el tiempo de ejecución del código presentado en la Figura 3, con conjuntos de datos similares en tamaño, se ubica en 111774,44 segundos (31 horas, 2 minutos y 54 segundos), empleando el clúster MinoTauro.

Como se demostrará en la siguiente sección, el margen de mejora posible sobre el código original es bastante significativo. Varios motivos propician esta situación, tanto el hecho de realizar todas las correlaciones cruzadas en el dominio temporal como el emplear un lenguaje interpretado de alto nivel como es Python. Además, el problema es altamente paralelo (*embarrassingly parallel*), pues todos los elementos de las matrices de salida a calcular son independientes entre ellos. Así, cada uno de los elementos de las matrices de salida únicamente compartirá una de las señales de entrada con los elementos que se encuentren en la misma fila o columna. Esta propiedad la podremos considerar cuando apliquemos optimizaciones conscientes de la arquitectura.

III. Mejoras aplicadas

En las siguientes subsecciones se expondrán las sucesivas mejoras que se han implementado. Sin embargo, es importante tener en mente que todas ellas comparten una interfaz desde Python, pues el objetivo es que el problema pueda seguir computándose con una simple llamada a una función desde la terminal, pero en un tiempo mucho menor. De aquí en adelante todos los tiempos mostrados serán el resultado de ejecutar la computación en el servidor MinoTauro del BSC, en aras de poder realizar una justa comparación de los tiempos obtenidos.

A. Fase 1: Optimización algorítmica. Paso al dominio de la frecuencia

El código inicial hace uso de funciones de *ObsPy* [8], que se ha establecido como un *framework* de referencia para procesamiento de datos sismológicos. Su objetivo principal es facilitar un desarrollo cómodo y sencillo de aplicaciones de este ámbito.

Tal y como se ha explicado previamente, se puede conseguir una gran mejora en el tiempo de computación si las correlaciones se realizan en el dominio de la frecuencia, pues reduciremos la complejidad computacional pasando de $\mathcal{O}(n^2)$ a $\mathcal{O}(n\log(n))$. Por ello, se comenzará por emplear una función que pertenece a una versión más reciente de la propia librería *Obspy*: *correlate*. Esta función realiza una llamada a la función *fftconvolve* de *SciPy*, si se va a computar en el dominio de la frecuencia. Sin embargo, el dominio en el que queremos que se realice la correlación puede ser fijado de antemano; dado que a partir de señales de 100 elementos se fija el umbral para que sea beneficioso trabajar en el dominio de la frecuencia. Nosotros especificaremos que sea éste el dominio en el que queremos que se compute.

Con este simple cambio ya se obtiene una primera mejora evidente respecto a la versión original. El tiempo de ejecución es de 125,62 minutos, frente a las más de 31 horas de la versión original (speedup de 14,83x). En esta ocasión no se está explotando toda el potencial del Teorema de la Convolución porque las Time-Series con las que se está trabajando no tienen un número de elementos potencia de dos. Aún así, para haber realizado un cambio tan sutil como sustituir una llamada de una función por otra, el resultado es más que esperanzador.

B. Fase 2: Optimización del modelo de progrmación

B.1 Compilación con *Cython*

El siguiente paso para reducir el tiempo de ejecución del código Python consiste en evitar la ejecución interpretada y la resolución de los tipos de datos en tiempo de ejecución (tipado dinámico). Existen distintas alternativas para conseguir ese objetivo, pero la más inmediata se apoya en usar compiladores fuente-fuente. En este trabajo, hemos usado Cython que es a su vez un lenguaje de programación que extiende a Python y un compilador de Cython a C/C++ que elimina gran parte del overhead de ejecución de Python.

En su vertiente como lenguaje, Cython permite especificar el tipo de las variables de un programa Python (int, double, etc). Esto permite eliminar casi en su totalidad el tiempo que Python tiene que dedicar a consultar el tipo de datos de cada variable que tiene que leer o escribir. En su vertiente como compilador fuente-fuente, la traducción de Python a C/C++, y la posterior compilación del código C/C++ para generar una librería dinámica, elimina el overhead de interpretación de código Python.

Estas dos ventajas (reducción del tipado dinámico y eliminación del intérprete de Python) no llevan aparejadas una merma en la productividad del desarrollador ya que sólo se requiere modificar ligeramente el código Python para especificar el tipo de las variables y usar el tool-chain de Cython para generar la libería dinámica. Además, de cara al usuario final, esa librería se carga como un módulo Python convencional (desde cualquier código o intérprete Python) y las funciones Cython ya compiladas en la librería pueden ser invocadas como cualquier función Python. La única diferencia es que la ejecución es más rápida ya que la llamada a la función Cython está finalmente ejecutando código nativo de la librería dinámica.

Empleando Cython se consigue un tiempo de ejecución de 8381,21 segundos (2 horas, 19 minutos y 41

segundos) y, por tanto, un speedup de 13,34x sobre el *baseline*. Vemos que en este código, la implementación en Cython no mejora a la versión que llama a la función *correlate* de la librería *Obspy*. Aunque no hemos podido confirmarlo experimentalmente, pensamos que, dado que *correlate* ya está implementado en C dentro de la librería *Obspy* y que la llamada a esta función es la que más tiempo consume, el uso de Cython no está consiguiendo una implementación mejor que la que ya incluye la librería *Obspy*.

B.2 Ctypes y C++

Dado que nuestro objetivo es conseguir ejecutar el problema en el menor tiempo posible, y tras evidenciar la capacidad de aceleración en la ejecución al computar las correlaciones en el dominio frecuencial, el siguiente paso es crear una versión en un lenguaje completamente compilado que explote esta característica. Aunque la función *correlate* de la librería *Obspy* ya está implementada en C, nos aventuramos a realizar nuestra propia implementación con la esperanza de conseguir aún mejores resultados. Esto se justifica porque no sólo desarrollaremos nuestra implementación de la correlación cruzada (línea 11 en la Figura 3) sino que ya de paso implementamos en C++ todo el bucle externo (línea 9 en la Figura 3) que se encarga de hacer todas las comparaciones dos a dos de las Time-Series.

Con este objetivo se ha empleado la biblioteca FFTW [9], cuyas siglas provienen de *Fastest Fourier Transform in the West*. Esta biblioteca, escrita en C, proporciona funciones para que el cómputo de las FFT e IFFT sea muy eficiente. El uso básico de esta biblioteca consta de dos pasos: primero se define el plan y posteriormente se ejecuta. En la creación del plan se indican tanto las posiciones de memoria donde comienzan los datos de entrada y los datos de salida de la operación, como el número de elementos de la operación a realizar. Además, en la creación del plan se pueden añadir flags que permiten indicar si se optimiza la ejecución de la FFT (o IFFT) o si se realiza una estimación de los parámetros, obteniendo una ejecución subóptima. Los flags, respectivamente, son *FFTW_MEASURE* y *FFTW_ESTIMATE*. Como cabe esperar, la obtención de parámetros óptimos no es gratis, sino que conlleva un mayor overhead inicial en la operación.

En esta versión básica se va a crear un plan para cada FFT e IFFT ejecutadas, por ello, el flag *FFTW_ESTIMATE* es el recomendado. En versiones posteriores, como se explicará, se creará un plan constante y únicamente variarán los datos sobre los que se aplica. En este caso será beneficioso usar el flag *FFTW_MEASURE*.

Como se ha mencionado previamente, se hará uso de una interfaz Python que mantenga la productividad del programador al tiempo que internamente se explote una implementación más eficiente. En la Figura 4 se pueden observar las distintas capas presentes al emplear el código desarrollado. El bloque `test_c.py` (Algorithm 1) representa el código que el usuario desarrollaría, donde se han de cargar los datos de entrada e importar el *Interfaz Python*. Este interfaz, a su vez, hace uso de las funciones implementadas en C++.

A continuación, se explicarán las peculiaridades de los dos componentes inferiores:

- Interfaz Python (`correlation_lib.py`, Algorithm 2): en primer lugar realiza la carga, con *ctypes*, de la librería dinámica producto de compilar el código en C++ (línea 1). Ctypes [10], es una librería que permite hacer llamadas a código C/C++ desde código Python de forma productiva y sin pérdida de rendimiento debida a movimiento de datos (siempre que se usen determinadas estructuras de datos linearizadas). Posteriormente, extrae los datos del objeto *obspy.core.trace.Trace*, en el que se encuentran las Time-Series en el problema original, y los escribe en un array bidimensional de forma que todas las Time-Series tengan la misma longitud, que será potencia de dos (líneas 2-4). Finalmente, se realiza la llamada a la función del código C++ (línea 5) que devuelve las matrices con los valores deseados de máximo, mínimo y desplazamientos.
- Código C++ con FFTW (`correlation_c.cpp`, Algorithm 3): en esta implementación se ha modificado la forma de realizar el algoritmo. Inicialmente, por cada elemento de las matrices de salida se realizaban dos FFTs, una multiplicación elemento a elemento y una IFFT, junto con la búsqueda de las características. En nuestro caso, hemos optado por calcular todas las FFTs al comienzo de la ejecución (línea 1), almacenarlas en memoria y realizar las lecturas necesarias para las multiplicaciones posteriores. Así, se ha reducido el número de FFTs a realizar, de $\mathcal{O}(n^2)$ a $\mathcal{O}(n)$. El siguiente paso es calcular las normas de todas las Time-Series (línea 2), pues queremos que los resultados de la correlación cruzada estén normalizados entre -1 y 1. Además, se reserva la memoria necesaria para guardar el resultado de la multiplicación elemento a elemento de dos señales en frecuencia. Finalmente, se da comienzo al doble bucle que itera sobre los distintos elementos de las matrices de salida, realiza los productos elemento a elemento pertinentes (línea 5) y, tras la IFFT (línea 6), extrae las características (línea 7).

Con todo lo mencionado anteriormente, empleando el *dataset* de referencia se obtiene un tiempo mediano de 25,09 minutos. Esto supone un speedup de 74,25x sobre el código original y de 5,57x sobre la versión de Cython. Se aprecia que se obtiene un importante speedup respecto a la versión de Cython, pero esto requiere más esfuerzo por parte del desarrollador.

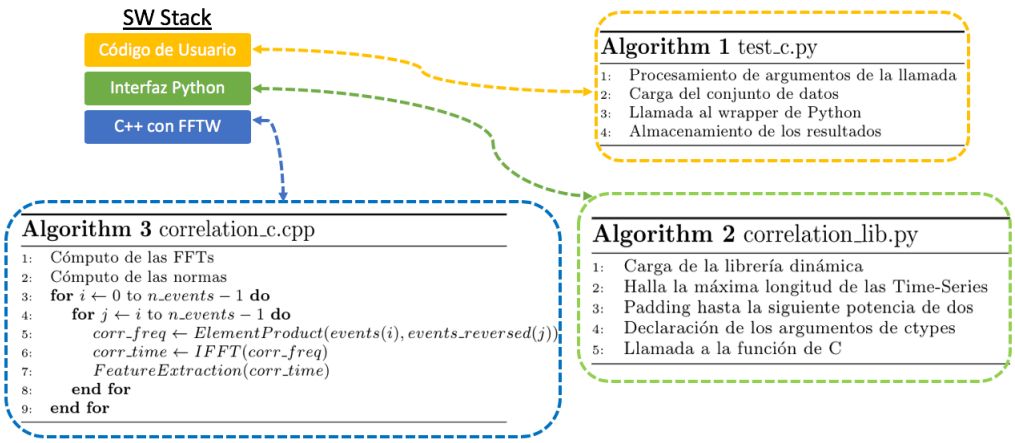

Fig. 4. Jerarquía de llamadas de la implementación en C++.

C. Fase 3: Optimizaciones conscientes de la arquitectura

C.1 Paralelización con OpenMP

Sabiendo que nos enfrentamos a un problema con un elevado grado de paralelismo, donde los elementos a computar son independientes entre sí, es imperioso emplear todos los núcleos del procesador para intentar acelerar la ejecución de la aplicación. Para ello, sobre la versión básica de C++, se van a integrar las llamadas necesarias para convertir nuestra versión serie en una versión paralela.

Como cabe esperar, la sección paralela comprende el doble bucle. Por tanto, como cada hilo va a computar multiplicaciones elemento a elemento e IFFTs independientemente, todos necesitan memoria donde almacenar estos resultados. Esta memoria será privada para cada thread.

En primer lugar, se empleará un *scheduler* estático, que repartirá el conjunto de iteraciones del bucle exterior de forma equitativa entre los hilos que especifique el usuario de la aplicación, desde Python. Para este problema en concreto, el *scheduler* estático no es particularmente beneficioso por el carácter triangular que presenta. Así, al primer hilo se le asignará un mayor número de iteraciones paralelas, que disminuirá progresivamente hasta el último hilo, para de esta manera garantizar que la carga de trabajo en cada hilo esta equidistribuída. Por tanto, podemos esperar que el tiempo total de la ejecución sea igual que el tiempo que tarde el primer hilo en terminar de computar la carga de trabajo que se le ha asignado.

Si se consideran los elementos de las matrices a calcular como una superficie, podemos determinar qué carga de trabajo paralelo (N_i) se asigna a cada hilo con la ecuación mostrada a continuación (empleando el planificador estático):

$$N_i = (n_{events}/n_{threads})^2 * (n_{threads} - (i + 1/2))$$

con i entre [0, $n_{threads}$). En esta ecuación n_{events} representa el número total de Time-Series de entrada y $n_{threads}$, el número de hilos totales. Por su parte, i representa el índice de cada hilo. Tal y como se ha mencionado previamente, puede observarse que número de iteraciones paralelas a computar por cada hilo disminuye al incrementar i.

En la Tabla I se muestra la estimación analítica del speedup máximo que puede esperarse al incrementar el número de hilos, empleando el planificador estático previamente mencionado. Estos valores han sido verificados experimentalmente, siendo los valores medios similares a los estimados analíticamente. Por ejemplo, para 8 hilos, se obtiene un tiempo de 360.3 segundos, que equivalen a poco más de 6 minutos (speedup de 310,23x sobre el baseline).

TABLA I
Speedup máximo con el planificador estático.

N. Proc.	1	2	4	8
Speedup	1	1,33	2,29	4,27

Claramente, con este planificador no se explota todo el paralelismo que ofrece el problema original y, por ello, se emplearán otros como el *Dynamic* o el *Guided*. Estos dos planificadores presentan un parámetro, *chunk size*, que permite fijar, en el primer caso, el tamaño máximo del número de iteraciones que se va asignando a cada hilo, mientras que en el segundo caso determina el tamaño mínimo del bloque asignado. Esto se debe a que el *Dynamic* va asignando bloques de iteraciones a los hilos según vayan pidiendo más carga de trabajo, mientras que el *Guided* realiza un reparto de forma proporcional entre el número de iteraciones que quedan por distribuir y el número total de hilos en ejecución. Este parámetro puede tener un impacto perjudicial en el tiempo de ejecución si el valor indicado es relativamente elevado respecto al número total de iteraciones del bucle exterior. Varios factores influyen en una correcta elección de este parámetro, tales como el tamaño de las Time-Series (para el uso de cachés), el número total de iteraciones y el número total de hilos. Para nuestro *dataset* de referencia, se recomienda un valor entre 10 y 200 para el *chunk size*.

En la Figura 5 se muestran los speedup obteni-

dos empleando tanto el planificador *Dynamic* como el *Guided* y el *Static*. Como cabría esperar, el *Dynamic* devuelve mejores resultados, pues reparte bloques del mismo tamaño durante toda la ejecución, a excepción de la parte final del bucle; mientras que el *Guided* distribuye bloques cuyo tamaño depende del número de iteraciones restantes. Además, podemos comprobar que los valores de speedup obtenidos para el planificador *Static* se aproximan mucho a los valores teóricos calculados previamente.

Además, a fin de comprobar todas las posibilidades que ofrece el problema, se ha estudiado emplear el planificador *Guided* recorriendo el conjunto de iteraciones de final al principio (en sentido inverso del bucle "i"). Así, al distribuir de forma conjunta más iteraciones inicialmente y menos al final, habrá un mayor balance en la carga entregada a los hilos.

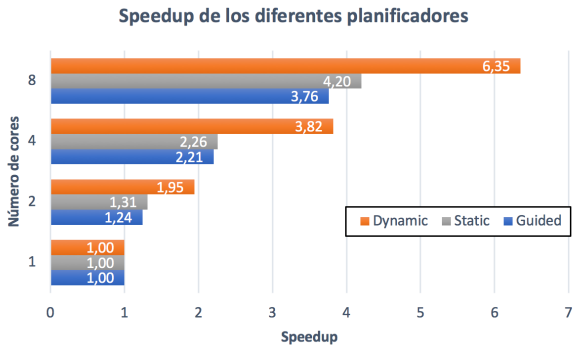

Fig. 5. Speedup obtenido con los distintos planificadores

En la Tabla II pueden observarse los tiempos de ejecución, para diferente número de hilos empleados. Se observa que, para el planificador *Guided* que recorre el conjunto de iteraciones de final a principio ("Guided rev.") se obtienen los mismos tiempos que para el planificador *Dynamic*.

TABLA II
Tiempos de ejecución para distintos planificadores y cores.

N. Proc.	1	2	4	8
Guided	1512,69	1216,60	685,89	402,25
Dynamic	1509,49	777,38	396,11	238,18
Guided rev.	1509,97	783,87	394,32	240,75

C.2 Optimizaciones avanzadas sobre OpenMP

En este punto, discutiendo con el personal de la *School of GeoSciences*, de la *University of Edinburgh*, llegamos a la conclusión de que no era necesario emplear los datos en formato de doble precisión, pues sólo los 3 primeros decimales son necesarios para la aplicación. Por lo tanto, se pueden declarar los arrays para que sean de tipo *floats*. Al cambiar el tipo de dato, además, cambian las estructuras de datos de la FFTW que se han de invocar (así como las llamadas a otras funciones auxiliares), que ahora operarán sobre datos de simple precisión. En la Tabla III se pueden observar los distintos valores medianos obtenidos al ir variando el número de hilos en ejecución cuando se aplica la optimizacion de simple precisión sobre la implementación OpenMP con planificador Dynamic. Claramente al pasar de de 4 a 8 núcleos no se está obteniendo una mejora, siquiera, considerable.

TABLA III
Tiempos de ejecución empleando datos con simple precisión para el planificacor Dynamic.

N. Proc.	1	2	4	8
Tiempo	823,62	429,52	235,75	229,33

Realizando un *profiling* sobre la aplicación desarrollada hasta ahora, se descubre que la mayor parte del tiempo de ejecución se debe a la creación y destrucción de los planes de las IFFTs. Por ello, se reescribió todo el código correspondiente a esta sección, declarando un único plan por cada hilo, que se reutilizará continuamente. Por tanto, en la computación de la IFFT ya no habrá presente una sección crítica, que previamente se empleaba para realizar la declaración de los planes, pues la única rutina *thread-safe* es la *fftwf_execute*, que lanza la computación de la FFT o IFFT en cuestión.

En la Tabla IV pueden observarse los valores obtenidos para esta nueva versión, que demuestran la importancia de declarar cuantos menos planes como sea posible. Además, con esta nueva versión, se optimiza la computación de la IFFT para estas longitudes de Time-Series, con el flag *FFTW_MEASURE*, del que ya se ha hablado previamente. Asimismo, la escalabilidad con el número de hilos es prácticamente ideal, excepto alguna pequeña desviación numérica.

TABLA IV
Tiempos de ejecución reescribiendo el proceso de IFFT.

N. Proc.	1	2	4	8
Tiempo	482,02	241,40	121,39	61,07

De momento, se habría obtenido con 8 núcleos un speedup muy similar al de ArrayFire (ver Sección III-C.3). Sin embargo, la versión de CPU seguía empleando datos complejos como entrada y, estudiando el manual de FFTW concienzudamente, pudimos encontrar las interfaces avanzadas que permiten realizar operaciones FFTs e IFFTs entre reales y complejos directamente. Esto permitió reducir, además, el tamaño que ocupaban las Time-Series, pues antes realizábamos un *padding* intercalado, para tomar la parte real proveniente de los datos y fijar las partes imaginarias a cero. Ahora, únicamente hay que realizar un *padding* hasta la siguiente potencia de dos.

Con todo esto, las señales en dominio de la frecuencia ocupan $N/2+1$ elementos complejos (donde N es la longitud de la Time-Series en dominio temporal), donde únicamente están presentes la mitad de los coeficientes, pues la otra mitad no es más que la conjugada de la primera. Finalmente, en la Tabla V puede observarse el efecto de saber explotar toda la capacidad que la biblioteca FFTW ofrece. Sigue ob-

servándose la cuasi-perfecta escalabilidad que presentan los resultados obtenidos. Con todos los cambios realizados, para 8 cores se obtiene un speedup de 5990x sobre el código original.

TABLA V
Tiempos de ejecución con uso de funciones de real a complejos y viceversa.

N. Proc.	1	2	4	8
Tiempo	139,42	69,72	35,65	18,66

Adicionalmente, algunas mejoras añadidas han sido aplicadas:

- Se ha paralelizado el cálculo inicial de las FFTs.
- Se ha modificado la estructura de datos que componen las matrices de salida para que únicamente almacenen los valores necesarios, sin tener que reservar memoria para la triangular inferior de cada una.
- Se han implementado funcionalidades que originalmente el personal de la *University of Edinburgh* realizaba en una segunda fase, como son el cálculo de un histograma de las magnitudes y la aplicación de un filtrado, que elimina valores por debajo de un cierto umbral. Tanto el número de *bins* del histograma como el valor para el filtrado son introducidos por el usuario desde Python.
- Los resultados se almacenarán como matrices dispersas comprimidas, permitiendo ahorrar gran cantidad de espacio (especialmente tras el filtrado).

Con todo lo comentado previamente, la versión final del código presenta unos tiempos de ejecución que podemos ver en la Tabla VI. Estos tiempos sólo mejoran ligeramente los tiempos para 4 y 8 núcleos cuando los comparamos con los obtenidos en la Tabla V.

TABLA VI
Tiempos de ejecución con funcionalidades añadidas.

N. Proc.	1	2	4	8
Tiempo	141,83	70,19	35,56	18,26

Además, se ha realizado la implementación de una versión que incluye *caché blocking* y de una versión que realiza múltiples IFFTs en paralelo. Sin embargo, ninguna de las versiones consigue mejorar el tiempo de ejecución obtenido anteriormente. La versión que explota *caché blocking* no consigue mejorar los resultados porque la forma de acceder los datos ya explota las localidades espacial (el *pre-fetching* se encarga de proporcionar lo datos necesarios satisfactoriamente) y temporal.

La principal motivación de hacer múltiples IFFTs en paralelo vino dada por el resultado obtenido al realizar un *profiling* sobre las iteraciones del bucle. Se obtuvo un tiempo de 6,4 microsegundos por elemento de salida, cuyo porcentaje de tiempo invertido

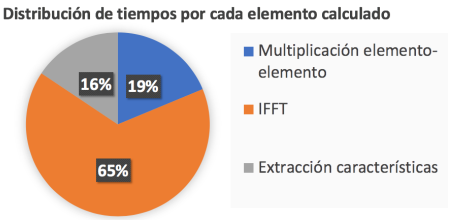

Fig. 6. Análisis detallado del tiempo por Time-Series

en cada operación puede observarse en la Figura 6. Claramente, la mayor parte del tiempo se dedica a realizar IFFTs. Sin embargo, el cómputo de múltiples IFFTs a la vez no devolvió mejores resultados.

C.3 Implementación con ArrayFire

Hasta ahora únicamente hemos explotado la potencia computacional de los cores incluidos en la CPU. Sin embargo, dado que nuestro problema en cuestión exhibe un elevado grado de paralelismo y que la plataforma de ejecución, MinoTauro, también contiene GPUs, cobra sentido usar también estos aceleradores. El objetivo es proporcionar aún menores tiempos de ejecución y explotar toda la potencia computacional disponible. Aunque las 2 tarjetas GPU disponibles por cada nodo son NVIDIA, en lugar de emplear CUDA, únicamente válido para este tipo de procesadores gráficos, se ha decidido realizar una versión portable a cualquier plataforma que soporte CUDA u OpenCL.

Para ello, trabajamos con ArrayFire [11], librería *Open Source* que facilita el desarrollo de aplicaciones tanto para arquitecturas paralelas (CPU) como masivamente paralelas (GPU). ArrayFire proporciona al desarrollador una abstracción de alto nivel, que permite a éste emplear los formatos de datos y funciones definidas por la librería. El código desarrollado se traduce en *kernels* que se ejecutan en diferentes plataformas, como CPUs de Intel, AMD, Arm y GPUs de NVIDIA, AMD y Qualcomm. Un aspecto relevante de ArrayFire es que cada llamada a una función del API de ArrayFire no se traduce en la invocación de un kernel a la GPU. En su lugar, un procesado Just In Time, JIT, en tiempo de ejecución, combina un número configurable de llamadas consecutivas al API en un único kernel. Esto resulta en menos llamadas al driver de GPU y kernels de mayor granularidad que consiguen reducir los *overheads* debidos a sincronizaciones entre el *host* y el dispositivo.

En la Figura 7 se puede observar el código en C++ que emplea ArrayFire. En primer lugar, entre las líneas 1-13, se observan varias llamadas al constructor de *af::array*, que crea arrays en el dispositivo en que se trabaje. Los arrays creados se corresponden con las Time-Series de entrada y las mismas invertidas, las matrices de salida, las normas de las señales y varios arrays auxiliares para poder almacenar la correlación cruzada y los valores a extraer. Tras estas declaraciones, comienza el doble bucle, en la línea 15 que conforma la parte computacionalmente más pesada de la aplicación. En este doble bucle se reali-

```
1  af::array tss(event_length, n_events, events); //creates array in the device
2  af::array tss_flipped = af::flip(tss, 0);
3
4  af::array af_Corr_max = af::constant(0, tss.dims(1), tss.dims(1), af::dtype::f32);
5  af::array af_Corr_min = af::constant(0, tss.dims(1), tss.dims(1), af::dtype::f32);
6  af::array af_Lag_max = af::constant(0, tss.dims(1), tss.dims(1), af::dtype::s32);
7  af::array af_Lag_min = af::constant(0, tss.dims(1), tss.dims(1), af::dtype::s32);
8
9  af::array norms = af::matmul(matrixNorm(tss, 0).T(), matrixNorm(tss, 0));
10
11 af::array corr = af::constant(0, tss.dims(0), tss.dims(1), tss.type());
12 af::array magnitude;
13 af::array lag;
14
15 for(int row=0; row<n_events; row++){
16       corr = af::constant(0, tss.dims(0), tss.dims(1), tss.type());
17
18       gfor(af::seq column, row, n_events-1){
19         corr(af::span, column) = af::convolve(tss(af::span, row), tss_flipped(af::span, column, 0),
                 AF_CONV_DEFAULT, AF_CONV_FREQ);
20       }
21
22       af::max(magnitude, lag, corr.rows(event_length/2 - shift, event_length/2 + shift), 0);
23       af_Corr_max(row, af::span) = magnitude;
24       af_Lag_max(row, af::span) = lag;
25
26       af::min(magnitude, lag, corr.rows(event_length/2 - shift, event_length/2 + shift), 0);
27       af_Corr_min(row, af::span) = magnitude;
28       af_Lag_min(row, af::span) = lag;
29 }
```

Fig. 7. Código desarrollado en C++ con ArrayFire.

za una llamada a la estructura de control *gfor*, en la línea 18 la cual permite lanzar muchas iteraciones del bucle de forma simultánea, gracias a la replicación de datos (*tiling* según ArrayFire). Así, en la línea 19, se realiza la llamada a la función de ArrayFire que computa la correlación cruzada. Posteriormente, en las líneas 22-28 se observa la extracción de características, primero para el máximo y su desplazamiento y, después para el mínimo y su desplazamiento correspondiente.

Una vez ha terminado la ejecución del doble bucle, realizamos un ajuste de los valores obtenidos, dividiendo las magnitudes entre la norma correspondiente y corrigiendo los desplazamientos. Finalmente, se copian los datos de vuelta al *host*.

Con el código desarrollado, haciendo uso del *backend* de CUDA, se obtiene un tiempo mediano de 131,56 segundos, que supone un speedup de 849,61x sobre el código original. En este punto, tras aplicar la recomendación de que no es necesario emplear los datos en formato de doble precisión, pues sólo los 3 primeros decimales son necesarios para esta aplicaciones, se decide reescribir el código para que todos los arrays fuesen de *floats*, es decir, simple precisión. Esta optimización tiene gran relevancia, especialmente en las arquitecturas de NVIDIA, donde suele haber el doble de unidades de procesamiento de datos de simple precisión. Con ello, obtuvimos un tiempo de ejecución de 64,48 segundos, lo que supone un speedup de 1733,50x respecto al problema original.

D. *Resumen de los resultados de las distintas implementaciones*

En la Figura 8 pueden compararse, en escala logarítmica, los speedups obtenidos (desde 1 a 8 cores) para cada implementación de las optimizaciones que se han aplicado en la estrategia de 3 fases que se ha seguido en este trabajo.

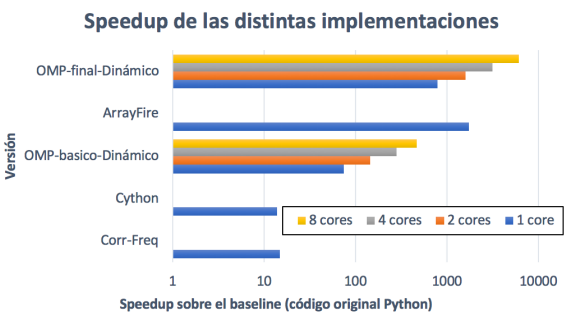

Fig. 8. Comparación de los speedups obtenidos por versión

Durante la primera fase, con la aplicación del Teorema de la Convolución para realizar correlaciones de forma más eficiente en el dominio de la frecuencia, conseguimos reducir la complejidad computacional. Para ello se invocan las funciones *fftconvolve* de la librería *SciPy* de Python, obteniendo la primera versión del código que denominamos `Cor-Freq`, la cual consigue una aceleración de 14,83x con respecto a la versión original. Durante la segunda fase, en la que apostamos por un cambio en el modelo de programación, primero tratamos, usando Cython que incorpora un compilador a C/C++, eliminar gran parte del overhead de ejecución de Python debido a la ejecución interpretada y la resolución de los tipos de datos en tiempo de ejecución. Esta nueva implementación es la que denominamos `Cython` en la figura. A continuación realizamos una implementación nativa en C++ que invoca funciones de la librería FFTW, y donde se puede observar el gran beneficio al desarrollar un código en un lenguaje con tipado estático y emplear librerías muy optimizadas para el cálculo de FFTs. Para esta versión ya tenemos una

speedup de 74,25x (ver OMP-basico-Dinámico para 1 core en la figura). El cambio de modelo de programación nos permite, en una tercera fase, añadir más optimizaciones conscientes de la arquitectura. Así realizamos una primera implementación paralela del bucle externo del kernel empleando OpenMP, y tras el estudio de distintas estrategias de planificación encontramos que *Dynamic* es la que mayor rendimiento proporciona, consiguiendo una speedup de hasta 469,29x (ver OMP-basico-Dinámico para 8 cores en la figura). Tras analizar la importancia de hacer un uso optimizado de las funciones de la librería FFTW y de declarar cuantos menos planes posibles para optimizar del uso de memoria, lo que conseguimos explotando localidad mediante privatización de buffers compartidos, a la vez que eliminamos puntos de sincronización debidos a secciones críticas, conseguimos reducir aún más los tiempos de computación, alcanzando ahora una speedup de 6121,3x con 8 cores (ver OMP-final-Dinámico para 8 cores en la figura). También estudiamos una nueva implementación con la librería ArrayFire lo que nos permite explotar la GPU de nuestro sistema, conseguiendo en este caso una speedup de 1733,5x (ver ArrayFire en la figura).

IV. Trabajos relacionados

Actualmente, muchos esfuerzos y proyectos están enfocados a mejorar las aplicaciones de tratamiento de Time-Series. Sin ir más lejos, la aplicación para *smartphones*, *Shazam* [12] está basada en el tratamiento de pequeños fragmentos de canciones capturadas con el terminal móvil, mediante la realización de transformadas de Fourier que permitan identificar los picos de frecuencias con los de canciones previamente almacenadas en una base de datos.

También encontramos empresas españolas dentro de este ámbito, como *Shapelets*, que ha desarrollado una librería que incluye técnicas novedosas de tratamiento de Time-Series, *Khiva* [13], empleando ArrayFire.

Por otra parte, también son dignas de mención las empresas como Ericsson o Huawei, que forman parte del 3GPP (*3rd Generation Partnership Project*), y actualmente están desarrollando algoritmos de extracción de información a partir de registros (Time-Series) para 5G. Estos desarrollos están enfocados no sólo a extraer la información, sino también a hacerlo lo más eficientemente posible.

V. Conclusiones y líneas de trabajo futuro

En este trabajo hemos ilustrado cómo utilizando una estrategia en tres fases, podemos acelerar en más de tres ódenes de magnitud una aplicación de Python que realiza tratamiento masivo de datos para el procesado de Time-Series sismográficas. Nuestra estrategia ha demostrado que elegir el algoritmo más apropiado para el tratamiento de la información es el primer paso para reducir la complejidad computacional, y por lo tanto el tiempo. En nuestro caso, pasar a trabajar en el dominio de la frecuencia ha supuesto un orden de mangnitud de mejora. A continuación, la selección del modelo de programación con el que se optimiza el núcleo computacional de la aplicación es otro paso crítico. En nuestro caso de estudio, el codificar en Ctypes y C++ nos ha permitido reducir la sobrecarga que supone en Python la ejecución interpretada y la resolución de tipos de datos en tiempo de ejecución, a la vez que nos abre la posibilidad de usar librerías optimizadas para ciertos dominios de aplicación, en particular en nuestra aplicación, la librería FFTW. Todo ello ha supuesto casi dos órdenes de magnitud de mejora. Por último, y gracias al cambio del modelo de programación hemos podido aplicar un conjunto de optimizaciones de bajo nivel conscientes de la arquitectura (paralelización basada en OpenMP, optimización de la localidad mediante privatización y eliminación de secciones críticas, y aceleración en GPU basada en ArrayFire), lo que nos ha permitido obtener finalmente más de tres órdenes de magnitud de mejora con respecto a la versión original.

Como líneas de trabajo futuras se plantean entre otras: i) optimización de las llamadas en CUDA; y ii) implementación heterogénea que haga uso tanto de la CPU como de la GPU, simultáneamente.

Agradecimientos

El presente trabajo ha sido financiado mediante el proyecto TIN2016-80920-R del Ministerio de Economía, Industria y Competitividad y por la Universidad de Málaga (Campus de Excelencia Internacional Andalucía Tech). También agradecemos al BSC que haya puesto a nuestra disposición el servidor de altas prestaciones MinoTauro.

Referencias

[1] Alan V. Oppenheim, "Signals and systems," *Prentice-hall Signal Processing Series, Prentice Hall*, 1996.
[2] "FFTW," `http://www.fftw.org/`, Accessed: 2019-05-23.
[3] Gabriele Jost Barbara Chapman and Ruud van der Pas, "Using openmp – portable shared memory parallel programming," *The MIT Press*, october 2007.
[4] "ArrayFire," `https://arrayfire.com/`, Accesses: 2019-05-23.
[5] A.F. Bell, S. Hernandez, H.E. Gaunt, P. Mothes, M. Ruiz, D. Sierra, and S. Aguaiza, "The rise and fall of periodic 'drumbeat' seismicity at tungurahua volcano, ecuador," *Earth and Planetary Sci. Lett.*, vol. 475, pp. 58–70, 2017.
[6] A.F. Bell, M. Naylor, S. Hernandez, I.G. Main, H.E. Gaunt, P. Mothes, and M. Ruiz, "Volcanic eruption forecasts from accelerating rates of drumbeat long-period earthquakes," *Geophysical Research Letters*, 2018.
[7] "MinoTauro User's Guide," `https://www.bsc.es/support/MinoTauro-ug.pdf`, Accessed: 2019-05-23.
[8] M. Beyreuther, R. Barsch, L. Krischer, T. Megies, Y. Behr, and J. Wassermann, "Volcanic eruption forecasts from accelerating rates of drumbeat long-period earthquakes," *Geophys*, vol. 81, no. 3, pp. 530–533, 2018.
[9] "FFTW 3.3.8 Documentation," `http://www.fftw.org/fftw3.pdf`, Accessed: 2019-05-23.
[10] "Ctypes Documentation," `https://docs.python.org/3/library/ctypes.html`, Accesses: 2019-05-25.
[11] "ArrayFire Documentation," `http://arrayfire.org/docs/index.htm`, Accesses: 2019-05-23.
[12] Avery Li-Chun Wang, "An industrial-strength audio search algorithm," *Shazam Entertainment, Ltd.*, 2003.
[13] J. Ruiz-Ferrer, A. Vilches, O. Torreno, and D. Cuesta, "Khiva: Accelerated time-series anlaytics on GPUs and CPU multicores," 2018, `https://github.com/shapelets/khiva`.
[14] Kurt W. Smith, "Cython, a guide for python programmers," *O'Reilly*, 2015.