EVALUACIÓN DEL RENDIMIENTO COMPUTACIONAL EN UN ESQUEMA DE COPROCESAMIENTO CPU-GPU DE TRES ALGORITMOS MATCHING PURSUIT PARA LA COMPRESIÓN DE DATOS SÍSMICOS

MAYRA ALEJANDRA CÁRDENAS ARENAS

REYNALDO FABIAN NORIEGA ZAMBRANO

UNIVERSIDAD INDUSTRIAL DE SANTANDER

FACULTAD DE INGENIERÍAS FISICOMECÁNICAS

ESCUELA DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y DE TELECOMUNICACIONES

BUCARAMANGA

2015

EVALUACIÓN DEL RENDIMIENTO COMPUTACIONAL EN UN ESQUEMA DE COPROCESAMIENTO CPU-GPU DE TRES ALGORITMOS MATCHING PURSUIT PARA LA COMPRESIÓN DE DATOS SÍSMICOS

MAYRA ALEJANDRA CÁRDENAS ARENAS

REYNALDO FABIAN NORIEGA ZAMBRANO

Trabajo de Grado para optar al título de

Ingeniero Electrónico

Director :

CARLOS AUGUSTO FAJARDO

Ingeniero Electrónico, PhD(c)

Codirector :

CARLOS ARTURO BOADA QUIJANO

Ingeniero Electrónico

UNIVERSIDAD INDUSTRIAL DE SANTANDER

FACULTAD DE INGENIERÍAS FISICOMECÁNICAS

ESCUELA DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y DE TELECOMUNICACIONES

BUCARAMANGA

2015

AGRADECIMIENTOS

En primer lugar agradezco a Dios por haberme brindado todas las herramientas necesarias para cumplir con el gran compromiso de terminar esta etapa tan importante de mi vida.

En segundo lugar a mi familia quienes me permiten cada día crecer y ser mejor persona, con su gran apoyo incondicional y sus grandes consejos. También agradezco a los profesores que durante toda la carrera me orientaron y me enamoraron de la ingeniería electrónica, en especial al ingeniero Carlos Fajardo por orientarme al final de esta etapa para la toma de sabias decisiones y llevar a feliz término este proyecto, al codirector quien nos brindó apoyo durante toda la realización de esta etapa. Es meritorio el acompañamiento tan importante de los calificadores quienes por medio de consejos nos brindaron la mejor ayuda para cerrar con broche de oro todos estos años de esfuerzos y enseñanzas.

A todos los integrantes del grupo de investigación CPS que por su apoyo incondicional hicieron posible la consecución de esta gran meta.

A todos mis amigos que siempre estuvieron apoyándome y brindándome toda la ayuda necesaria, quienes me enseñaron lo valioso que puede ser una amistad. Y a toda la gente que de una u otra forma fueron parte de esta formación que ya termina.

A Colciencias e ICP por la confianza depositada para la elaboración de este proyecto.

Mayra A. Cárdenas Arenas.

DEDICATORIA

A mi madre, Rosmery, y mi hermanita, Maria Fernanda, que son los pilares fundamentales en mi proceso de formación personal y académico y que además han sido mis amigas todos estos años de vida.

Reynaldo F. N.

AGRADECIMIENTOS

Agradezco a mi madre, por estar siempre presente y brindarme con esfuerzo las herramientas necesarias para obtener mi grado de educación, a mi hermana por su apoyo incondicional. A ambas por estar siempre presente brindando sus consejos y palabras de ánimo cuando es necesario.

A todos mis amigos por haberme apoyado en las diferentes etapas de mi vida.

A los integrantes del grupo de investigación CPS porque de alguna u otra manera aportaron en la realización de este trabajo. En especial a los profesores, Carlos Fajardo por sus enseñanzas, consejos y su guía y Carlos Boada por su interés y disponibilidad en la solución de problemas.

A ICP por la confianza depositada en la realización de este proyecto de investigación.

A todos los demás que me apoyaron, muchas gracias.

Reynaldo F. N.

TABLA DE CONTENIDO

LISTA DE FIGURAS

Pág

LISTA TABLAS

Pág

RESUMEN

TÍTULO: EVALUACIÓN DEL RENDIMIENTO COMPUTACIONAL EN UN ESQUEMA DE COPROCESAMIENTO CPU-GPU DE TRES ALGORITMOS MATCHING PURSUIT PARA LA COMPRESIÓN DE DATOS SÍSMICOS*

Autores: MAYRA ALEJANDRA CÁRDENAS ARNAS
REYNALDO FABIAN NORIEGA ZAMBRANO**

Palabras Claves: Matching Pursuit, GPU, Trazas Sísmicas, Compresión.

Los algoritmos *Matching Pursuit* son cada vez más utilizados por su eficiencia para hacer representaciones *sparse* de señales digitales. En este trabajo se implementan tres versiones del algoritmo *Matching Pursuit* en un esquema de coprocesamiento CPU - GPU con el propósito de comprimir trazas sísmicas. Los tres algoritmos evaluados fueron *Matching Pursuit*, *Weak Matching Pursuit* y *Least - Squares Orthogonal Matching Pursuit*. Adicionalmente, el trabajo busca hacer una selección entre los tres algoritmos teniendo en cuenta el factor de compresión y el rendimiento computacional. El diccionario utilizado en el algoritmo está basado en ondiculas *Morlet* y se diseñó teniendo en cuenta los contenidos en frecuencia de las señales a comprimir. Nuestros resultados sugieren que el algoritmo *Least - Squares Orthogonal Matching Pursuit* presenta un mejor factor de compresión y rendimiento computacional en comparación con los algoritmos evaluados. Como trabajo adicional se calculó el tiempo que necesita cada una de las funciones presentes en los algoritmos para determinar los cuellos de botella de cada uno de estos, nuestros resultados muestran que la función transpuesta es la que mayor tiempo ocupa en los algoritmos *Matching Pursuit* y *Weak Matching Pursuit.* En el algoritmo *Least - Squares Orthogonal Matching Pursuit* la función con mayor tiempo fue el producto matriz matriz.

* Trabajo de grado

** Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. Director: PhD(c) Carlos Augusto Fajardo Ariza. Codirector: M.Sc. (c) Carlos Arturo Boada Quijano

ABSTRACT

TITLE: TESTING THE COMPUTATIONAL PERFORMANCE OF THREE MATCHING PURSUIT ALGORITHMS FOR SEISMIC DATA COMPRESSION ON A PROCESSING SCHEME OF CPU-GPU*

AUTORS: MAYRA ALEJANDRA CÁRDENAS ARNAS
REYNALDO FABIAN NORIEGA ZAMBRANO**

KEYWORDS: Matching Pursuit, GPU, Seismic Traces, Compression.

Matching Pursuit algorithms are increasingly used by their efficiency to make digital signal sparse representations. In this work are implemented three versions of the algorithm Matching Pursuit in a scheme of co-processing CPU - GPU for the purpose of compressing seismic trace. The three evaluated algorithms were Matching Pursuit, Weak Matching Pursuit and Least - Squares Orthogonal Matching Pursuit. Additionally, the work seeks to make a selection among the three algorithms taking into account the factor of compression and computational performance. The dictionary used in the algorithm is based on Morlet's wavelet and was designed taking into account the contents in frequency of signals to compress. Our results suggest that the Least - Squares Orthogonal Matching Pursuit algorithm presents a best factor of compression and computational performance compared with the tested algorithms. As additional work, it was calculated the time needed by each of the functions present in the algorithms to determine the bottlenecks of each of these ones, the results show that transposed function is that longer deals with Matching Pursuit and Weak Matching Pursuit algorithms. In the algorithm Least - Squares Orthogonal Matching Pursuit with longer function was the matrix matrix product.

* Degree Work

** Faculty of Physical-Mechanic Engineerinng. School of Electrical, Electronic and Telecommunications Engineering. Advisor: PhD(c) Carlos Augusto Fajardo Ariza. Co-advisor: M.Sc. (c) Carlos Arturo Boada Quijano.

INTRODUCCIÓN

El procesamiento de datos sísmicos representa un reto para cualquier sistema de cómputo debido a la cantidad de datos a procesar, los cuales se encuentran en el orden de los terabytes. Se han utilizado algoritmos de compresión para hacer más eficiente su almacenamiento y su transmisión. Uno de estos ha sido el algoritmo *Matching Pursuit*, el cual permite hacer una compresión de los datos sísmicos a través de una representación *sparse* [1].

El algoritmo *Matching Pursuit* [2] es un *greedy algorithm* que busca descomponer una señal como una combinación lineal de elementos llamados *atoms* pertenecientes a un diccionario. En este trabajo se comprimen datos sísmicos utilizando tres versiones del algoritmo *Matching Pursuit*. Las versiones consideradas son: (i) *Matching Pursuit* (MP), que es el algoritmo básico. (ii) *Weak Matching Pursuit* (WMP), que presenta una menor complejidad computacional que el algoritmo base, pero requiere un mayor número de iteraciones. (iii) *Least Squares - Orthogonal Matching Pursuit* (LSOMP), que tiene una mayor complejidad computacional que el algoritmo base, pero requiere un menor número de iteraciones [1].

Nuestro propósito es hacer una implementación para GPU (*Graphics Processing Unit*) de cómputo en paralelo de los tres algoritmos con el fin de establecer cuál versión brinda los mejores resultados en términos del factor de compresión y el rendimiento computacional.

Ya se han implementado versiones de este algoritmo en GPU. En [3], se plantea una versión rápida del algoritmo MP, basada en la librería CUBLAS. Los resultados muestran que esta implementación es hasta 31 veces más rápida que la versión en CPU. Sin embargo, esta implementación es optimizada gracias al

uso de librerías *BLAS*. En [4], se describe la implementación del algoritmo LSOMP, donde se analiza la complejidad de cada módulo y se señalan los cuellos de botella. En este trabajo se plantean dos formas de abordar los módulos de proyección y de mínimos cuadrados, alcanzando una mejora de hasta 70 veces la versión en CPU.

Nosotros, en este trabajo al evaluar el rendimiento computacional tuvimos en cuenta el tiempo de ejecución. Para el proceso de compresión tuvimos en cuenta dos variables: (i) El factor de compresión, el cual se calcula mediante la ecuación 1 y (ii) la SNR, la cual permite evaluar la calidad de la señal reconstruida. La SNR también se utilizó como criterio de parada de los algoritmos.

$$FC = \frac{Tamaño\ vector\ original}{Cantidad\ de\ coeficinetes\ (iteraciones)} \quad (1)$$

Nuestros resultados sugieren que la versión LSOMP ofrece los mejores resultados globales del rendimiento computacional y en el proceso de compresión.

La estructura de este libro consta de las siguientes secciones: Una breve descripción de la GPU, de la SNR y de las tres versiones del algoritmo *Matching Pursuit* en la sección marco teórico. En la sección de trabajo realizado se muestra el proceso utilizado para la creación del diccionario y la implementación en GPU de los algoritmos. En la cuarta sección se presentan los resultados experimentales. Finalmente, las conclusiones y trabajo futuro se exponen en la última sección.

1. MARCO TEORICO

1.1 ALGORITMO MATCHING PURSUIT

La representación de una señal mediante un algoritmo *Matching Pursuit* busca expresar dicha señal como una combinación lineal de *atoms* del diccionario. El proceso consiste en encontrar un vector de coeficientes $x(i)$, tal que al multiplicarlo por algunos elementos del diccionario, acumulados en la matriz soporte $As(j)$, se obtiene una versión aproximada de la señal $Va(j)$(ecuación 2).

$$Va(j) = \sum_{i=1}^{N} As(j) * x(i) \qquad (2)$$

En este trabajo se evaluarán tres versiones del algoritmo: *Matching Pursuit*, *Weak Matching Pursuit* y *Least Square - Orthogonal Matching Pursuit* [1]. A continuación se muestran los pseudocódigos de los algoritmos implementados en el presente trabajo.

Matching Pursuit

Variables de entrada: Señal original: b

Diccionario: A

Inicialización de variables: -Cantidad de iteraciones: $k = 1$

-Vector de coeficientes: $x[k] = 0$

-Vector residual: $r = b$

-Vector de subíndices: $S[k] = [\,]$

-Matriz soporte: $As = [0]$

-Relación señal a ruido: $SNR = 0$

-Vector aproximado: $Va = 0$

-$z = 0$

Mientras SNR sea menor o igual que 40 dB, hacer:

-Escoger elemento máximo del producto punto: $z = A * r^T$

-Actualizar vector de coeficientes con este elemento máximo: $x[k] = \max(z)$

-Actualizar vector de subíndices con el asociado al elemento máximo: $S[k] = j$

-Asignar a la matriz soporte la columna del diccionario asociado a dicho subíndice: $As[k] = A[j]$

-Calcular vector aproximado: $Va = As * x[k]^T$

-Actualizar vector residual: $r = b - Va$

-Actualizar valor de relación señal a ruido:

$$SNR = 20 * \log\left(\frac{\sqrt{\sum_{i=1}^{N} b(i)^2}}{\sqrt{\sum_{i=1}^{N} r(i)^2}}\right)$$

-Incrementar la cantidad de iteraciones en una unidad: $k = k + 1$

Fin ciclo mientras

Fin algoritmo

Weak Matching Pursuit

Variables de entrada: Señal original: b

Diccionario: A

Inicialización de variables: -Cantidad de iteraciones: $k = 1$

-Vector de coeficientes: $x[k] = 0$

-Vector residual: $r = b$

-Vector de subíndices: $S[k] = [\,]$

-Matriz soporte: $As = [0]$

-Relación señal a ruido: $SNR = 0$

-Vector aproximado: $Va = 0$

-$z = 0$

-t = cualquier valor entre 0 y 1

Si t=1 entonces WMP = MP

Mientras SNR sea menor o igual que 40 dB, hacer:

-Se calcula el producto punto: $z = A * r^T$

Si $|z(j)|$ menor que $t * |r|$ entonces:

-Actualizar vector de coeficientes: $x[k] = z(j)$

-Actualizar vector de subíndices con el asociado a este elemento: $S[k] = j$

Si no:

-Escoger elemento máximo del producto punto: $z = A * r^T$

-Actualizar vector de coeficientes con este elemento máximo: $x[k] = \max(z)$

-Actualizar vector de subíndices con el asociado al elemento máximo: $S[k] = j$

Fin si

-Asignar a la matriz soporte la columna del diccionario asociado a dicho subíndice: $As[k] = A[j]$

-Calcular vector aproximado: $Va = As * x[k]^T$

-Actualizar vector residual: $r = b - Va$

-Actualizar valor de relación señal a ruido:

$$SNR = 20 * \log\left(\frac{\sqrt{\sum_{i=1}^{N} b(i)^2}}{\sqrt{\sum_{i=1}^{N} r(i)^2}}\right)$$

-Incrementar la cantidad de iteraciones en una unidad: $k = k + 1$

Fin ciclo mientras

Fin algoritmo

Least-Squares Orthogonal Matching Pursuit

Variables de entrada: Señal original: b

Diccionario: A

Inicialización de variables: -Cantidad de iteraciones: $k = 1$

-Vector de coeficientes: $x[k] = 0$

-Vector residual: $r = b$

-Vector de subíndices: $S[k] = [\,]$

-Matriz soporte: $As = [0]$

-Relación señal a ruido: $SNR = 0$

-Vector aproximado: $Va = 0$

-$z = 0$

Mientras SNR sea menor o igual que 40 dB, hacer:

-Escoger elemento máximo del producto punto: $z = A * r^T$

-Actualizar vector de coeficientes con este elemento máximo: $x[k] = \max(z)$

-Actualizar vector de subíndices con el asociado al elemento máximo: $S[k] = j$

-Asignar a la matriz soporte la columna del diccionario asociado a dicho subíndice: $As[k] = A[j]$

-Calcular x como un problema de mínimos cuadrados:

$$x[k] = [As^T * As]^{-1} * (As^T * b)$$

-Calcular vector aproximado: $Va = As * x[k]$

-Actualizar vector residual: $r = b - Va$

-Actualizar valor de relación señal a ruido:

$$SNR = 20 * \log\left(\frac{\sqrt{\sum_{i=1}^{N} b(i)^2}}{\sqrt{\sum_{i=1}^{N} r(i)^2}}\right)$$

-Incrementar la cantidad de iteraciones en una unidad: $k = k + 1$

Fin ciclo mientras

Fin algoritmo

1.2 GPU *(GRAPHICS PROCESING UNIT)*

La GPU (unidad de procesamiento gráfico, en español), es el dispositivo encargado del procesamiento de píxeles, datos sobre texturas y elementos geométricos (polígonos) en videojuegos y aplicaciones 3D, para liberar de carga a la unidad de procesamiento central (CPU) [5][6].
Su característica principal es la capacidad de poder ejecutar procesos en paralelo, y su bajo costo y la relación al consumo de energía.

Esta capacidad ha de ser muy útil para el tratamiento masivo de información en el campo científico, puesto que al trabajar varios procesos de manera simultánea se puede reducir el tiempo de procesado de señales, que van a ser mejores que los que pueda alcanzar una CPU.

1.3 SNR

La definición de ruido se podría interpretar como interferencia a fenómenos que, en gran medida son controlables mediante un diseño adecuado del sistema y los circuitos que lo conforman. La relación señal a ruido (SNR) es una buena medida de la calidad de una señal en un sistema. Es la suma del ruido de fuentes externas y el ruido propio del sistema. Al obtener altos valores de SNR conlleva un aumento en el costo de implementación del sistema. Un valor adecuado de esta relación es aquél en el que la señal recibida puede considerarse sin defectos o con un mínimo de ellos. Por ejemplo en el caso de transmisión de voz, se desea que la señal recibida sea una reproducción fiel de la transmitida, pero puede tolerarse un cierto nivel de ruido y distorsión que depende de aspectos subjetivos relacionados con la percepción auditiva humana. Lo mismo ocurre en el caso de transmisión de imágenes. En los sistemas digitales de comunicaciones suele utilizarse el concepto de tasa de errores (BER7), equivalente, en cierta medida a la relación señal a ruido, más empleado en los sistemas analógicos. [7]

2. TRABAJO REALIZADO

Inicialmente, los tres algoritmos fueron implementados en CPU en lenguaje de programación C. Después de esto se implementaron los tres algoritmos en un esquema de coprocesamiento CPU-GPU, en lenguaje CUDA-C.

A continuación se describe el proceso de creación del diccionario y el paralelado de las funciones que componen los algoritmos.

2.1 CREACIÓN DEL DICCIONARIO

El diseño del diccionario se realizó en la herramienta de programación *Matlab* usando ondículas *morlet*. El algoritmo que crea el diccionario trabaja de la manera siguiente:

(i) Se calcula la transformada de Fourier.

(ii) Se seleccionan valores de frecuencia que coincidan con valores pico de la amplitud de este espectro.

(iii) A partir de esta cantidad de frecuencias escogidas se crea una matriz de N ondiculas por frecuencia. Las ondiculas deben tener la misma cantidad de muestras que la señal que se quiere procesar.

La tabla 1 muestra los parámetros del diccionario que se creó inicialmente.

Tabla 1. Parámetros del diccionario 1.

Parámetro	
Cantidad de filas	3584
Cantidad de columnas	6144
Cantidad de frecuencias	6
Ondiculas por frecuencia	1024

Posteriormente se mejoró el diseño de dicho diccionario para obtener una reconstrucción con mejor calidad.

La tabla 2 muestra las características del segundo diccionario creado.

Tabla 2.. Parámetros del diccionario 2.

Parámetro	
Cantidad de filas	3584
Cantidad de columnas	30464
Cantidad de frecuencias	17
Ondiculas por frecuencia	1792

La figuras 1 y 2 muestran los espectros en frecuencia de una de las trazas y los diccionarios creados. Nótese que el segundo diccionario cubre de mejor manera el rango de frecuencias de la señal original.

Figura 1. Comparación dominio de la frecuencia traza y diccionario 1.

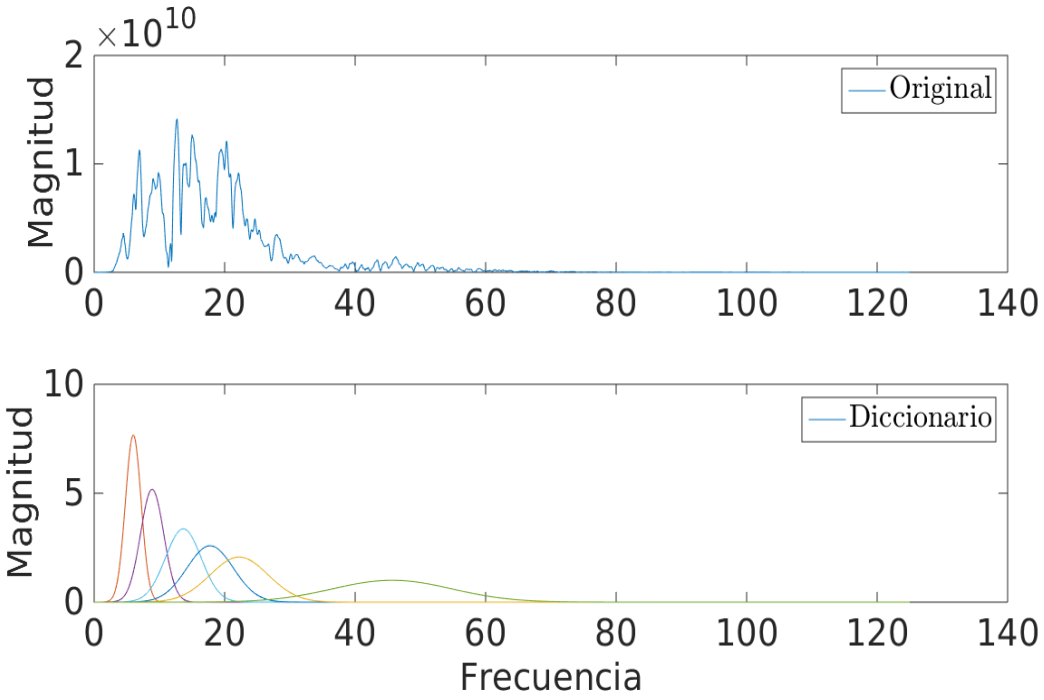
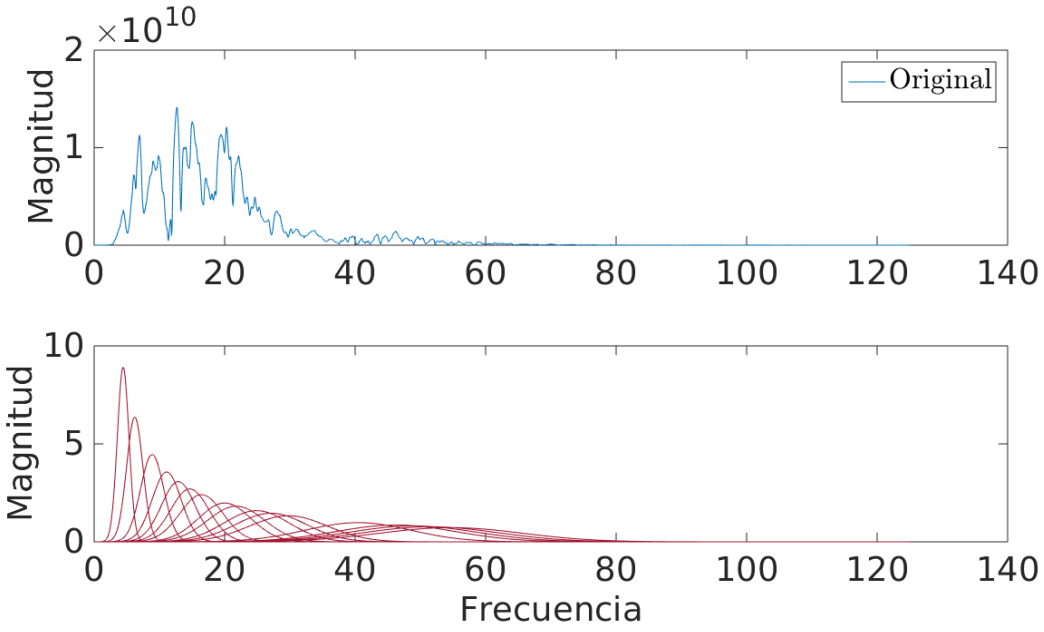


Figura 2. Comparación dominio de la frecuencia traza y diccionario 2.



2.2 PARALELACIÓN DE LAS FUNCIONES

Después de la creación del diccionario se realizó la implementación en CUDA-C para la GPU. A continuación se explica cómo se aplicó el paralelismo a las funciones que componen los algoritmos.

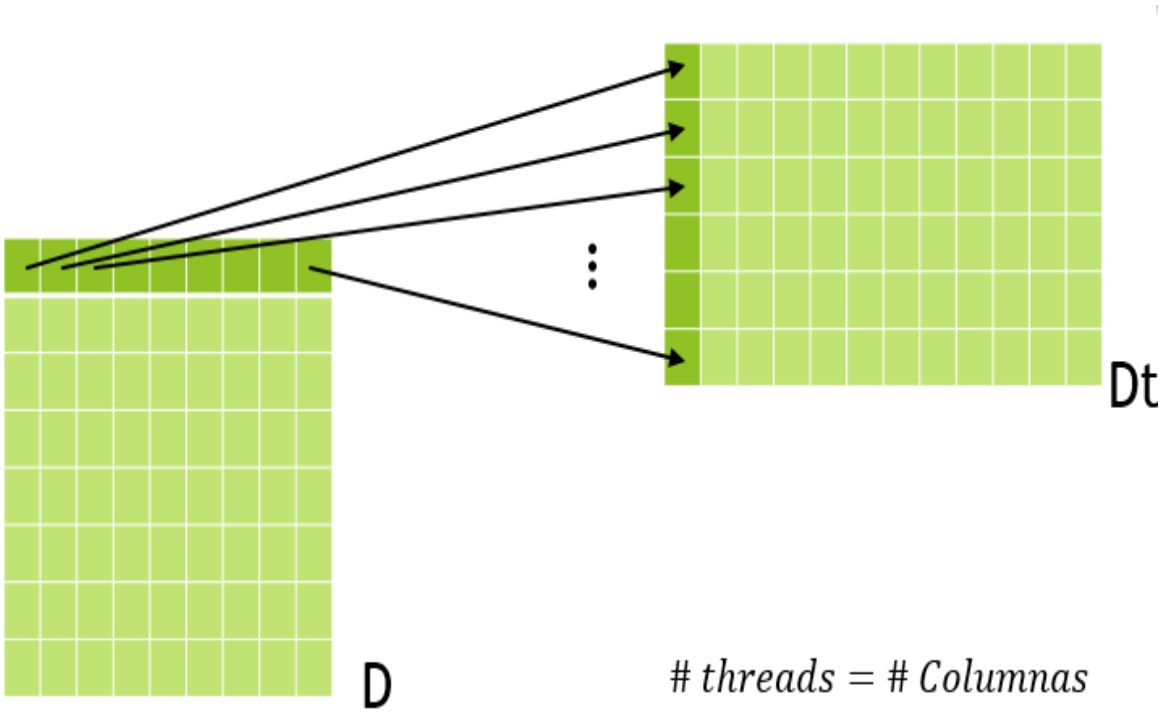
2.2.1 **Función Máximo.** Para la obtención del máximo valor de un vector y su respectivo subíndice, se agrupó por subconjuntos todo el vector, y cada *thread* se encargó de comparar cada uno de los valores del subconjunto hasta encontrar el máximo. Una vez encontrado los máximos de cada subconjunto, fueron almacenados en otro vector de menor tamaño para que finalmente un *thread* calculara el máximo de este subconjunto final.

El tamaño de cada subconjunto fue calculado como se muestra en la ecuación 3:

$$Tamaño\ subconjunto = \frac{Tamaño\ vector}{Número\ de\ threads} \quad (3)$$

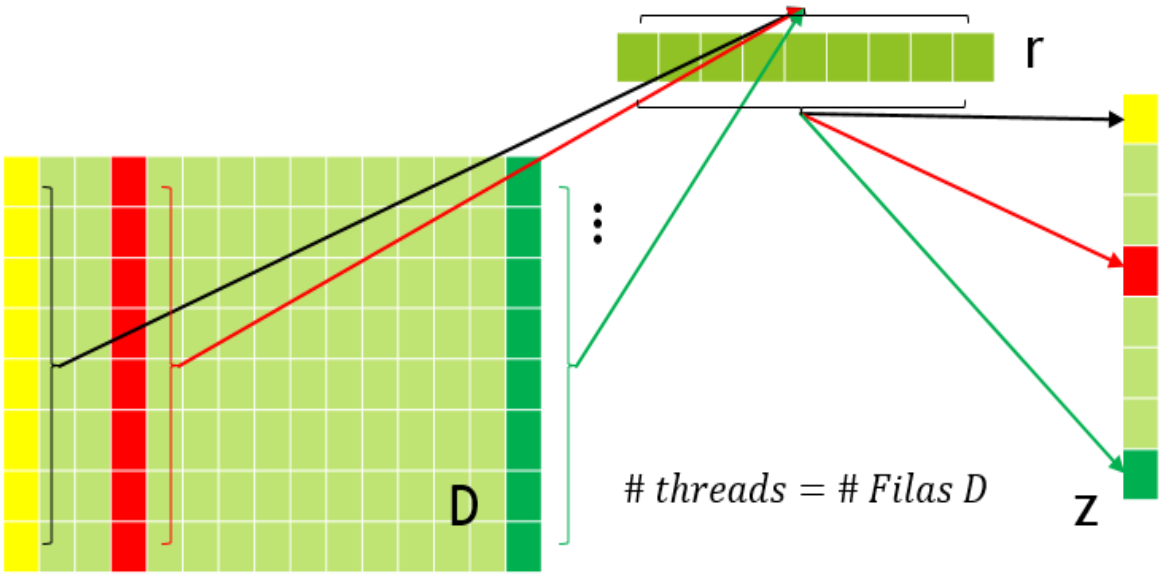
2.2.2 **Función Transpuesta.** Para generar la matriz transpuesta, cada thread se encargó de tomar un elemento de la fila y ubicarlo en un elemento de la cantidad columnas de la matriz a la cual se le realiza la transpuesta. Así, cada thread se encargó de convertir los elementos columna en elementos fila.

Figura 3. Función transpuesta.



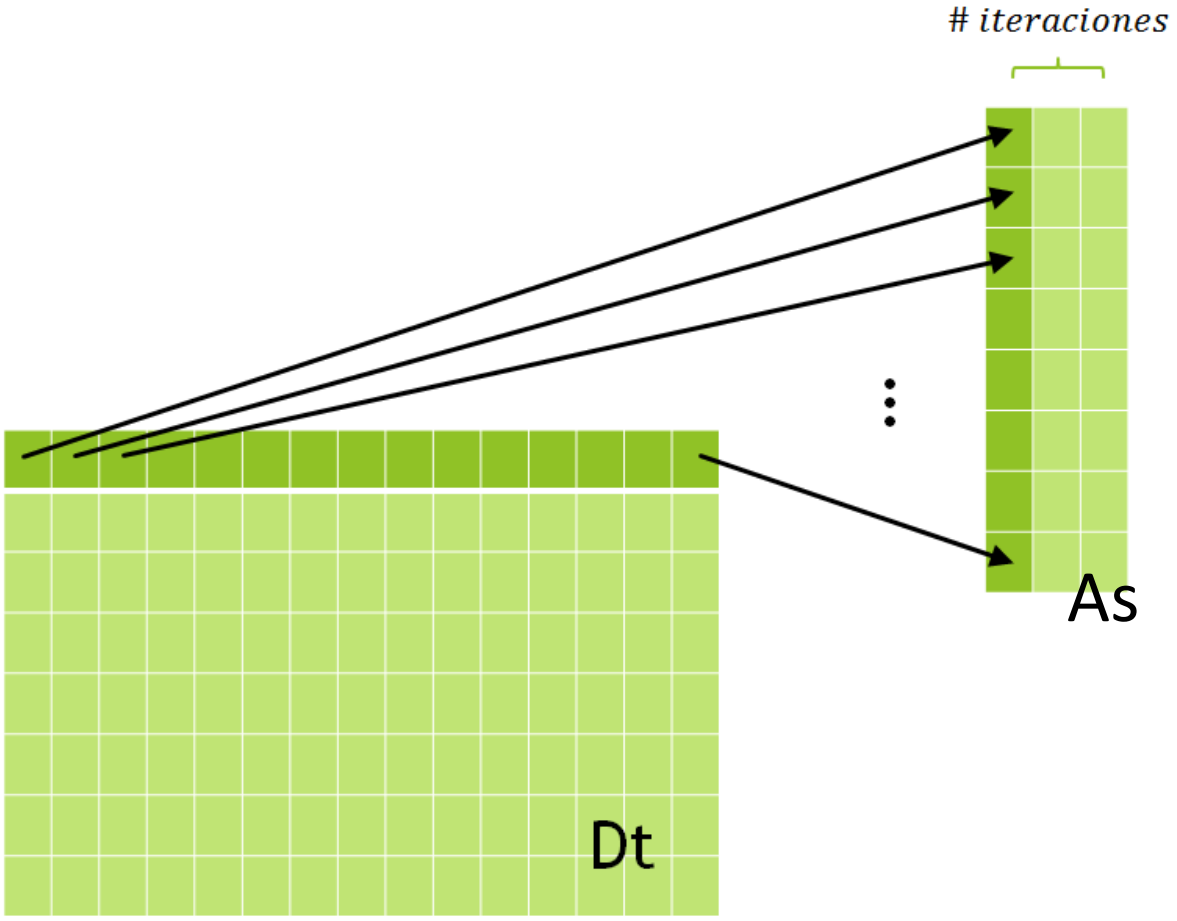
2.2.3 **Función producto matriz – vector**. En esta operación cada *thread* se encarga de acumular la multiplicación de cada elemento de una columna de la matriz con cada elemento del vector a multiplicar, generando así, cada elemento del vector resultante (ver figura 4).

Figura 4. Función producto entre una matriz y un vector.



2.2.4 **Función que genera la matriz soporte.** Para generar la matriz soporte se usó la misma estrategia de la matriz transpuesta, ya que cada *thread* se encargó de convertir los elementos de la fila del diccionario traspuesto, en elementos de la columna de la matriz soporte, como se muestra en la figura 5. Para obtener un mayor rendimiento se utilizó el diccionario transpuesto, pues se manipulan elementos de las filas y no elementos de las columnas.

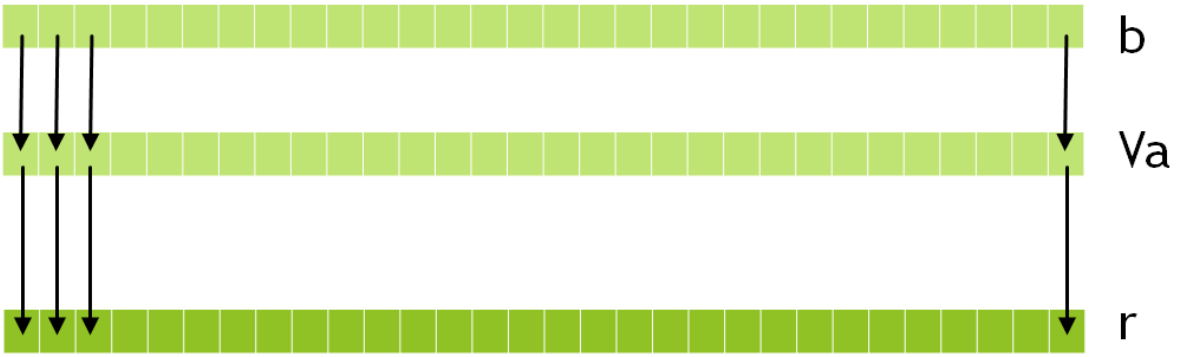
Figura 5. Función que genera la matriz soporte.



Se debe tener en cuenta que el tamaño de esta matriz va aumentando conforme se incrementan el número de iteraciones.

2.2.5 **Función resta entre Vectores.** Para paralelar la resta entre dos vectores, cada *thread* se encarga de restar cada elemento de los vectores y almacenarlo en otro vector. (Ver figura 6).

Figura 6. Función resta entre vectores.

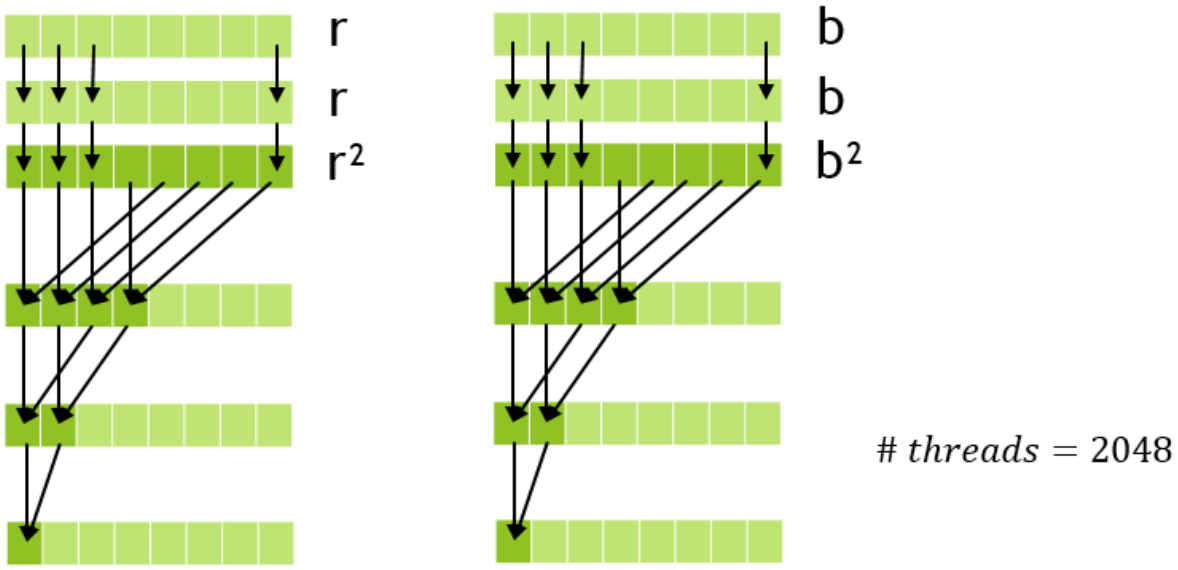


2.2.6 **Función SNR.** Para el cálculo de la relación señal a ruido se aplicó la ecuación 4 en CPU. Los valores de la sumatoria del cuadrado de los elementos de la traza original (el numerador) y del denominador de la ecuación (el residuo), fueron calculados en GPU basados en técnica de reducción desarrollada por NVIDIA [8], esta consiste en agrupar de a dos los elementos e ir reduciendo a la mitad la cantidad de estos por iteración.

$$SNR = 20 * \log\left(\frac{\sqrt{\sum_{i=1}^{N} b(i)^2}}{\sqrt{\sum_{i=1}^{N} r(i)^2}}\right) \quad (4)$$

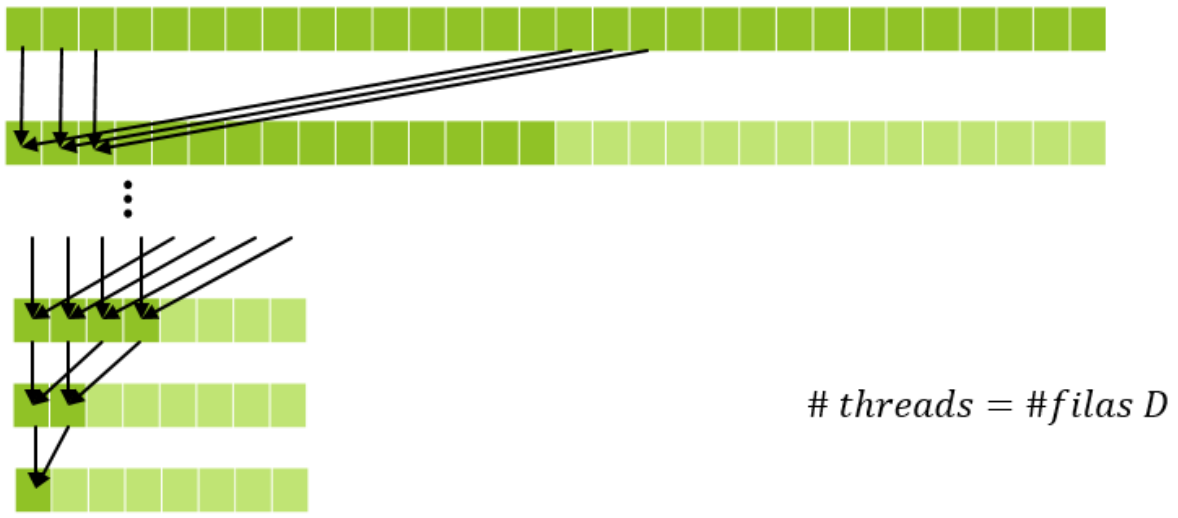
Inicialmente se asegura que la cantidad de elementos del vector sea potencia de dos, donde cada *thread* se encarga de multiplicar cada elemento de los vectores con el mismo, calculando los elementos al cuadrado. Después se van sumando el primer elemento con un elemento distanciado un "*offset*" como se ve en la figura 7, reduciendo a la mitad la cantidad de elementos del vector en cada iteración, para que finalmente el total de la sumatoria se encuentre en el primer valor del vector.

Figura 7. Sumatoria de valores al cuadrado.



2.2.7 **Función norma.** Aplicar paralelismo para el cálculo de la norma implica la reducción mencionada en la realización de la SNR.

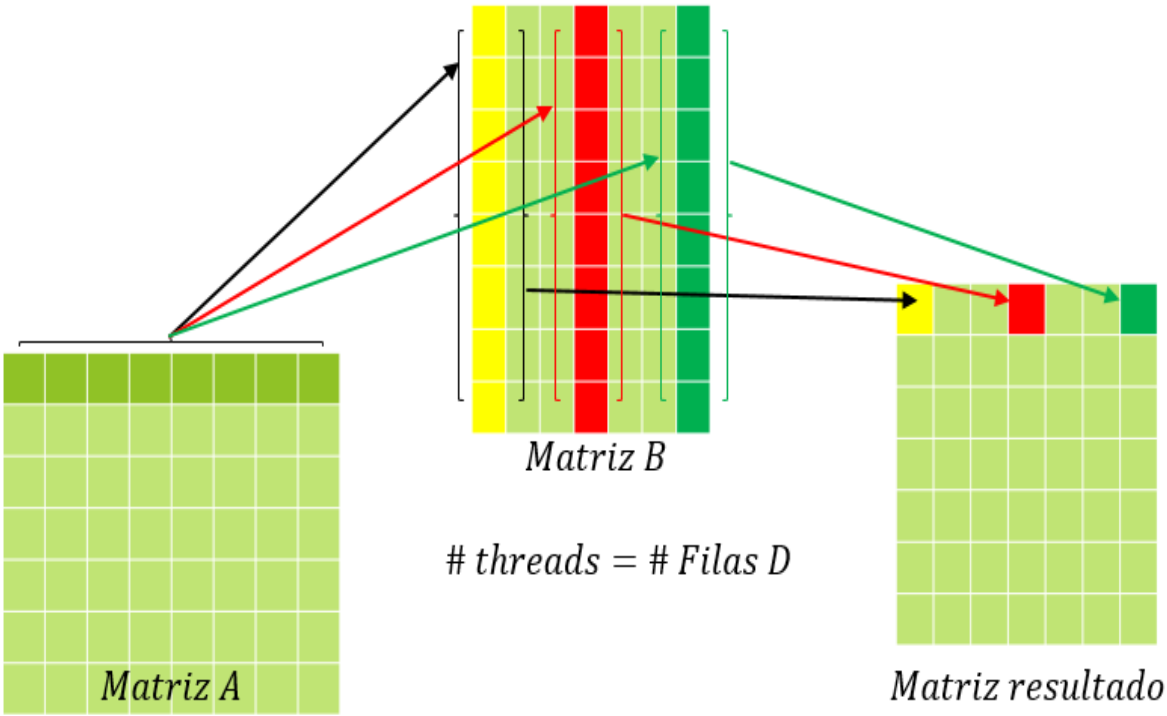
Figura 8. Función norma.



Al final un único *thread*, como se aprecia en la figura 8, contiene la información de la norma, que se usa en la implementación del algoritmo WMP.

2.2.8 **Función producto matriz – matriz.** En el producto matriz-matriz, cada *thread* se encarga de hacer una multiplicación matriz - vector y así calcular columnas de la matriz resultante como lo muestra la figura 9.

Figura 9. Función producto matriz - matriz.



2.2.9 **Función inversa.** El cálculo de la matriz inversa se basó en [9]. Se comienza por la creación de una matriz aumentada, como lo muestra la figura 10, en donde cada *thread* se encarga de ubicar columna a columna los elementos de la matriz. Luego se comprueba, de manera secuencial, que los elementos de la diagonal principal sean diferentes de cero, figura 11, si no es así se realiza un

intercambio de filas para quitar estos ceros, donde cada *thread* se encarga de un elemento.

La ecuación 5 se aplica para convertir, columna a columna, los elementos debajo de la diagonal principal en ceros.

Paso seguido, los elementos por encima de esta diagonal también se convierten en ceros, como se expone en la figura 12.

$$Ri = Ri - Rjx\frac{aij}{ajj} \qquad (5)$$

Figura 10. Creación matriz aumentada.

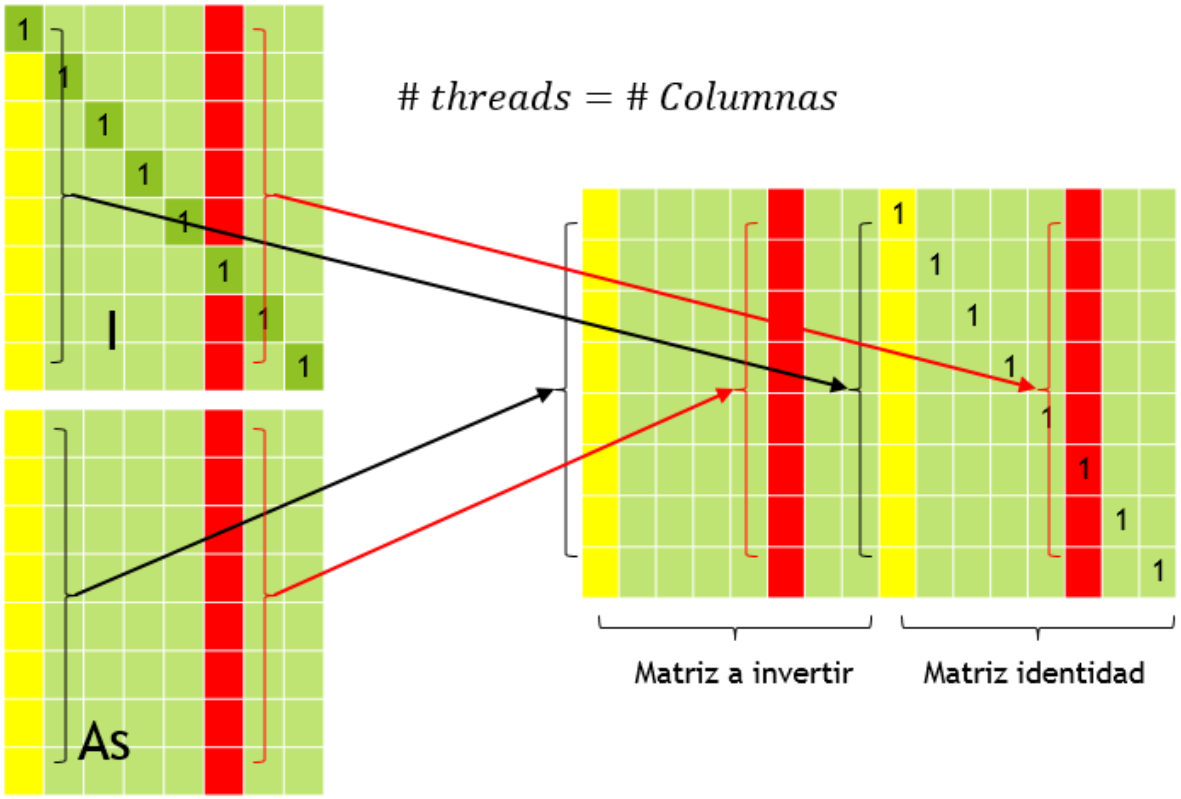


Figura 11. Diagonal principal de la matriz aumentada.

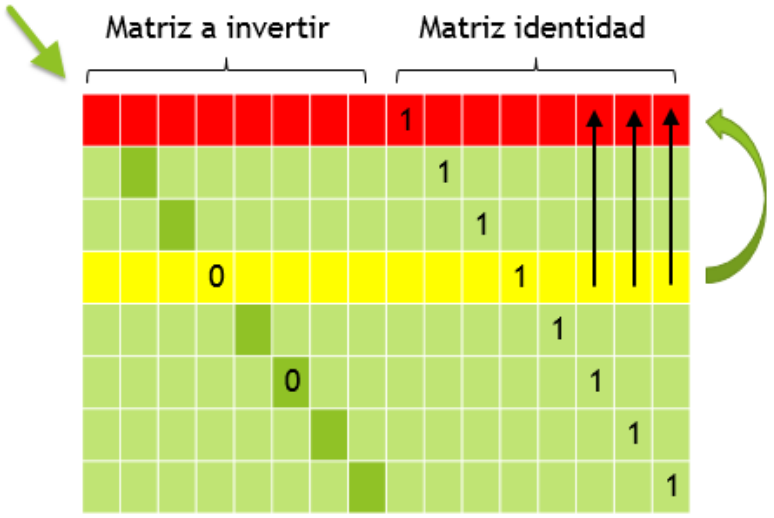
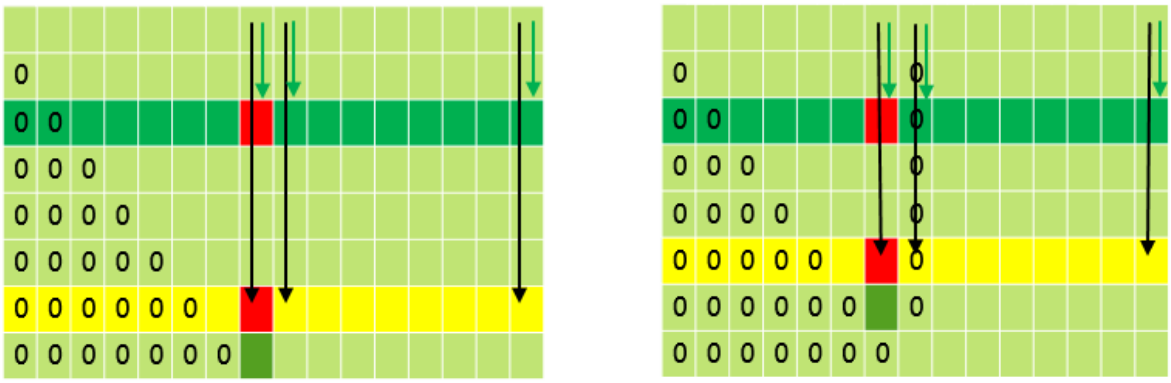


Figura 12. Matriz inversa.



2.2.10 **Diferencia entre Algoritmos.** Estas son las funciones utilizadas en cada uno de los algoritmos, para detallar la diferencia entre los algoritmos WMP y LSOMP con el algoritmo base MP, se presentan las siguientes tablas:

Tabla 3. Funciones de los algoritmos MP y WMP.

MP	***WMP***
Transpuesta	Transpuesta
Producto matriz vector	Producto matriz vector
Max	**Norma**
	¿ z[k] > t * norma ?
Matriz soporte	Matriz soporte
Transpuesta	Transpuesta
Producto matriz vector	Producto matriz vector
Resta	Resta
SNR	SNR

Tabla 4. Funciones de los algoritmos MP y LSOMP.

MP	***LSOMP***
Transpuesta	Transpuesta
Producto matriz vector	Producto matriz vector
Max	Max
Matriz soporte	Matriz soporte
Transpuesta	**Transpuesta**
	Producto matriz matriz
	Inversa
Producto matriz vector	Producto matriz vector
Resta	Resta

3. RESULTADOS

3.1 SISTEMAS DE CÓMPUTO UTILIZADOS

El presente proyecto fue implementado con un esquema de coprocesamiento CPU - GPU, utilizando el lenguaje de programación CUDA-C.

3.1.1 **CPU.**

El equipo de cómputo utilizado en este proyecto cuenta con un procesador Intel Xeon E5-2609, que trabaja a una frecuencia de 2.40GHz, posee 4 núcleos físicos y 8 GB de memoria RAM. Adicionalmente, este equipo dispone de una tarjeta Nvidia GeForce GTX 660 conectada a través del puerto PCIe 3.0.

3.1.2 **GPU.**

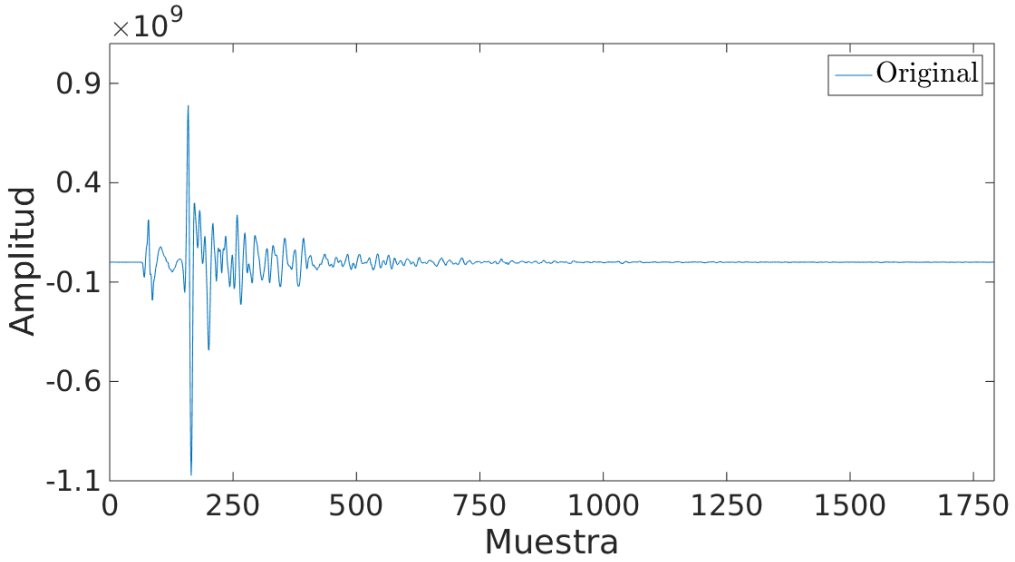
La GPU es una GeForce GTx 660, la cual cuenta con arquitectura Kepler y tiene las siguientes características:

- Capacidad de cómputo: 3.0.
- Reloj: 1.1 GHz.
- Memoria global: 2,147 GB.
- Streaming Multiprocessors: 5.
- Hilos por warp: 32.
- Cantidad máxima de hilos por bloque: 1024.
- Dimensión máxima de bloque *(AxBxC)*: (1024, 1024, 64).
- Dimensión máxima de grillas de hilos: (2147483647, 65535, 65535).

3.2 RESULTADOS PARA DICCIONARIOS DIFERENTES

Los tres algoritmos fueron testeados utilizando la traza a, la traza b y la traza c que hacen parte de un disparo sísmico. La figura 13 muestra una parte de una de estas trazas, la cual contiene 3584 muestras.

Figura 13. Traza a.



La tabla 5 muestra el factor de compresión obtenido al alcanzar una SNR de 20 [dB] con el diccionario 1. Nótese que el algoritmo LSOMP ofrece los mejores resultados con cada traza.

Tabla 5. Factor de compresión con el diccionario 1.

SNR 20 [dB]	Factor de compresión		
	Traza a	**Traza b**	**Traza c**
MP	14,44	4,14	28,9
WMP	13,37	3,89	21,46
LSOMP	55,13	39,38	89,6

La tabla 6 muestra los resultados para el diccionario 2. Los tres algoritmos convergen en este caso a una SNR de 36 [dB] y LSOMP sigue siendo el algoritmo con mejor desempeño.

Tabla 6. Factor de compresión con el diccionario 2.

SNR 36 [dB]	Factor de compresión		
	Traza a	**Traza b**	**Traza c**
MP	2,72	1,78	1,66
WMP	2,46	1,67	1,43
LSOMP	18,19	14,68	11,23

El rendimiento de los tres algoritmos y la calidad de la señal reconstruida mejora considerablemente usando el segundo diccionario. Esto es debido a que dicho diccionario contiene más ondículas, lo que implica más frecuencias que permiten realizar una mejor combinación de los elementos de dicho diccionario para representar la señal que se desea comprimir.

3.3 SEÑAL RECONSTRUIDA

La figura 14 muestra la señal original y la señal reconstruida con una SNR de 40[dB] apreciándose que ambas señales tienden a ser similares. Al acercarse sobre una zona de la imagen donde la amplitud de la señal es alta, se tiene la figura 15, en donde visualmente no se aprecian diferencias. Como expone la figura 16, las zonas donde la amplitud es baja, la reconstrucción de la señal presenta algunas modificaciones.

Figura 14. Traza original versus reconstruida.

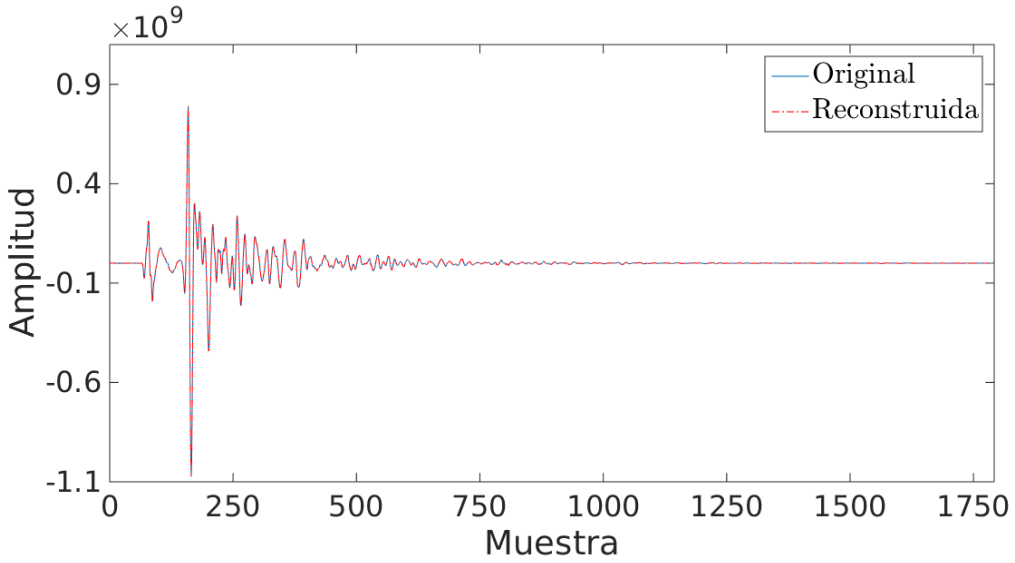


Figura 15. Traza original versus reconstruida en zonas de alta amplitud.

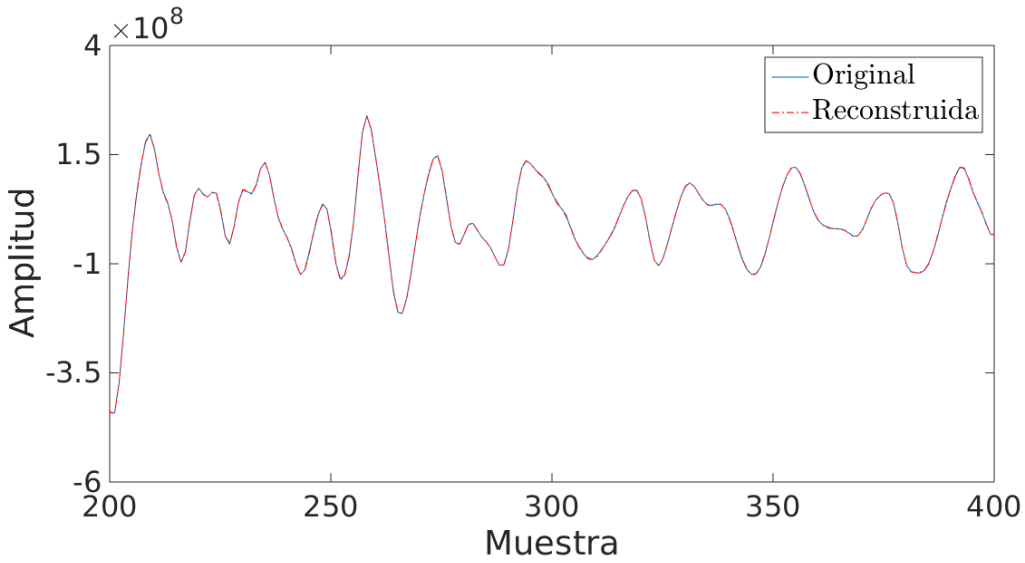
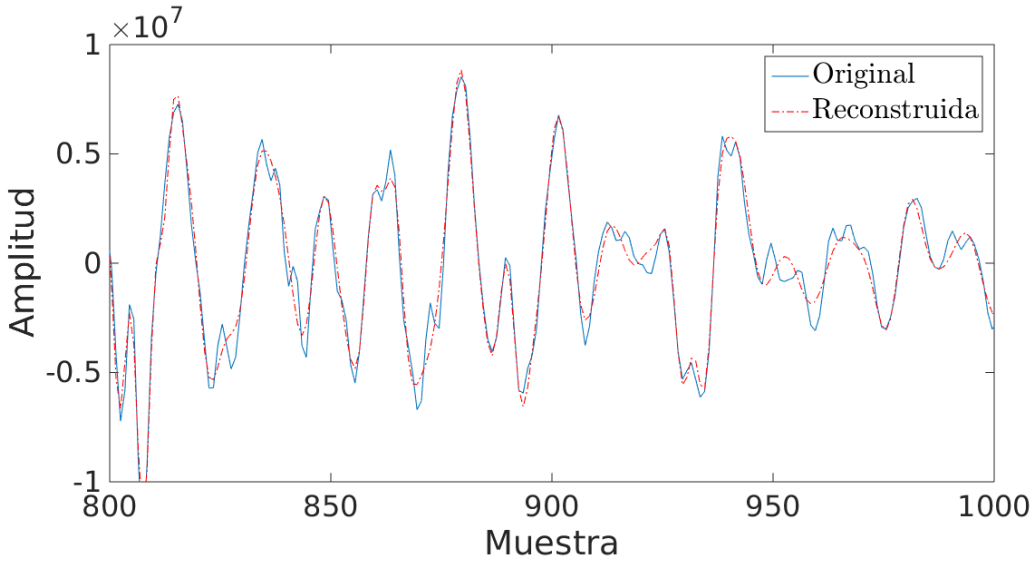


Figura 16. Traza original versus reconstruida en zonas de baja amplitud.



La figura 17 muestra el análisis en frecuencia de la señal original y la reconstruida. Las figuras 18 y 19 muestran que las señales, original y reconstruida, son semejantes en las frecuencias donde la magnitud del espectro es alta. La figura 20 muestra que, la recuperación de la señal en zonas donde decrece dicha magnitud, la calidad de los datos se ve afectada. La figura 21 muestra la fase de la traza y la señal recuperada, donde después de aproximadamente los 50 [Hz] la señal reconstruida no sigue el mismo comportamiento de la original, como lo hacía a bajas frecuencias.

Figura 17. Espectro de la traza original y reconstruida.

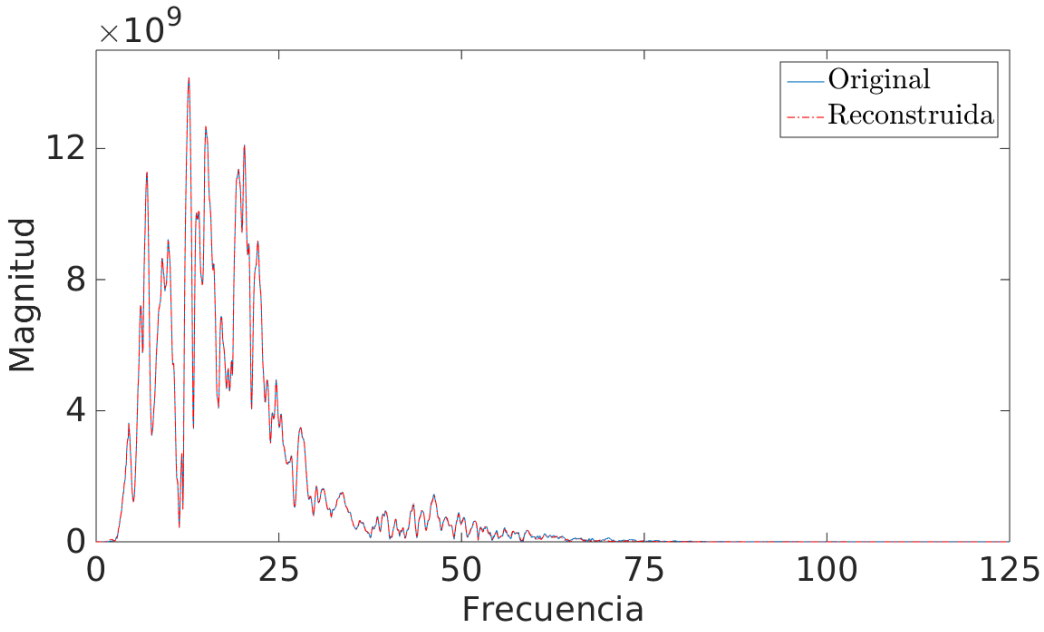


Figura 18. Espectro de la traza original y reconstruida. Baja frecuencia. Alta amplitud

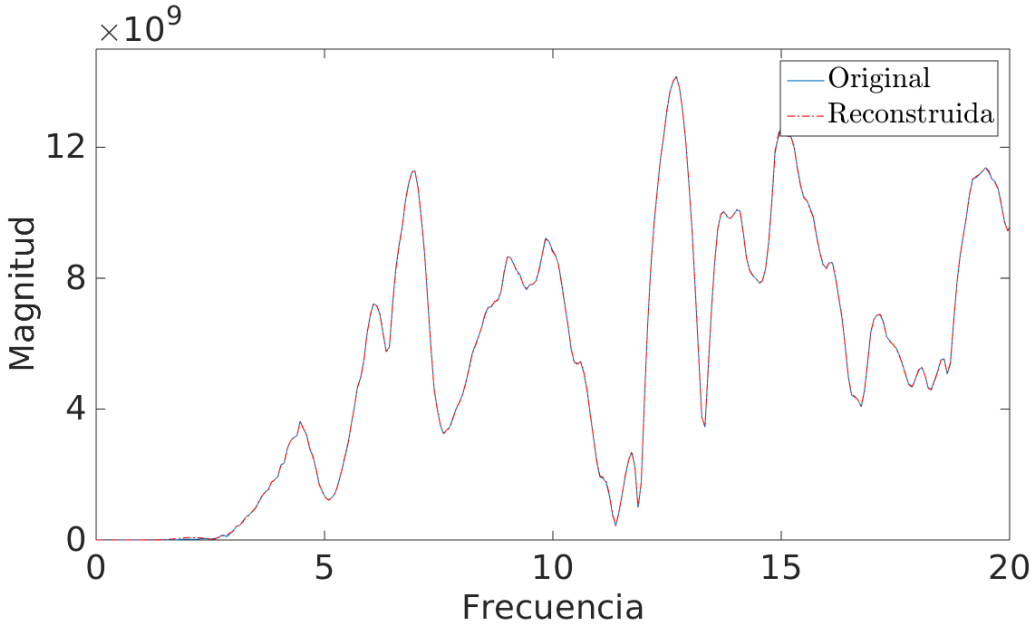


Figura 19. Espectro de la traza original y reconstruida. Alta amplitud

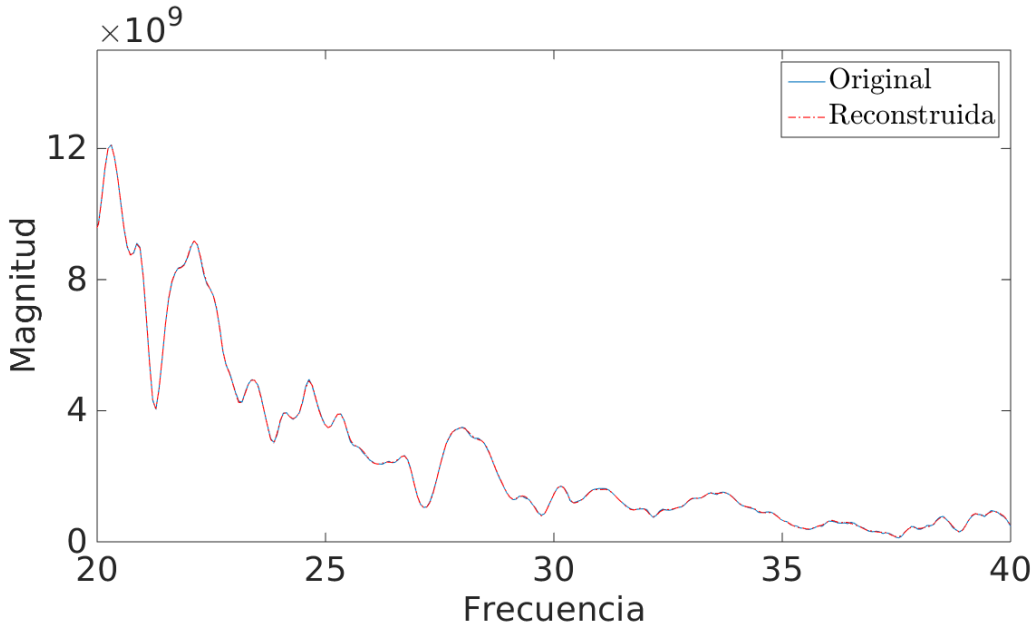


Figura 20. Espectro de la traza original y reconstruida. Alta frecuencia. Baja amplitud.

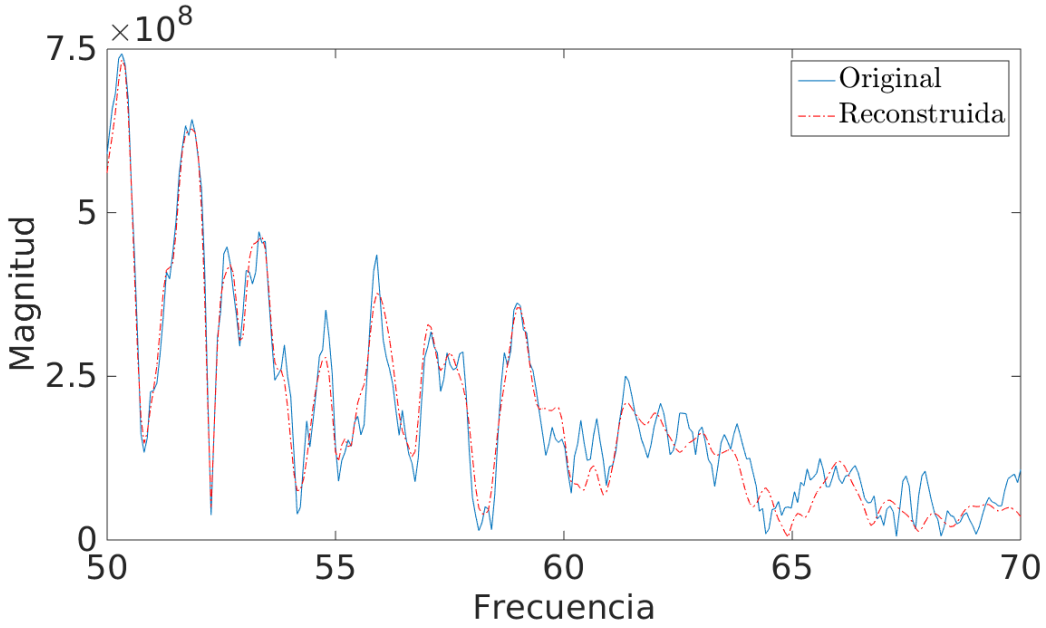
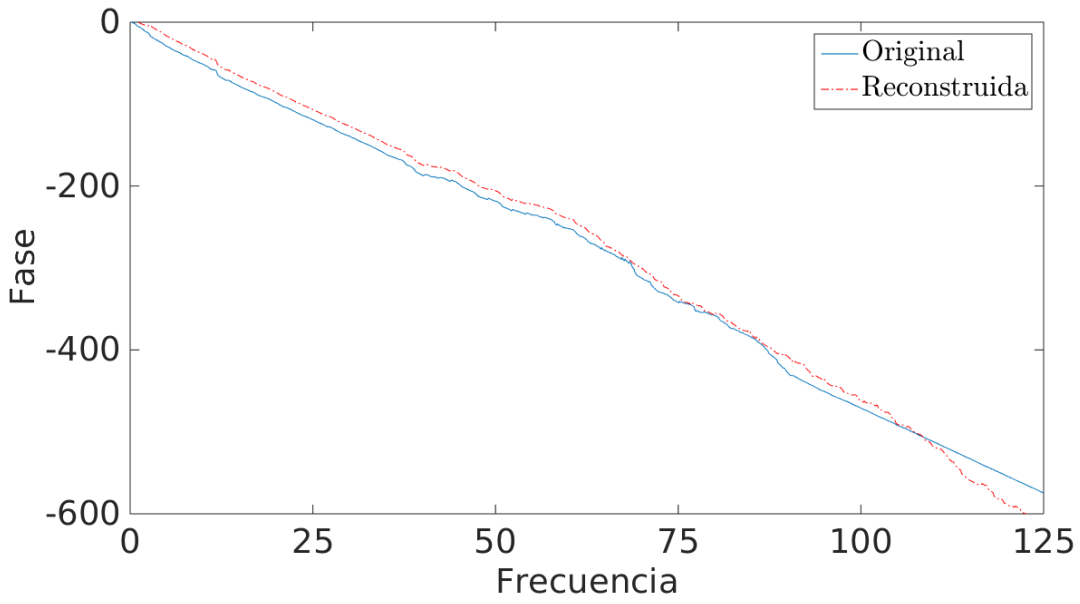


Figura 21. Análisis de fase de los espectros de la traza original y la reconstruida.



3.4 TIEMPOS DE EJECUCIÓN

Para la evaluación de los tiempos de ejecución, se usaron las trazas a, b y c con cada uno de los algoritmos. Las tablas 7, 8 y 9 muestran los tiempos que las trazas a, b y c demoran en alcanzar una determinada SNR, respectivamente. Se observa que el algoritmo LSOMP es el que brinda menores tiempos para las trazas a y b.

Tabla 7. Tiempos de ejecución traza a.

	CPU [s]	**GPU-CPU [s]**
MP	1463,19	64,52
WMP	1596,99	69,89
LSOMP	468,9	57,5

Para la traza a, con la que fue posible alcanzar una SNR de 40 [dB], se observa que en la CPU:

(i) El algoritmo LSOMP es 3,12 veces más rápido que el MP.

(ii) El algoritmo LSOMP es 3,4 veces más rápido que el algoritmo WMP.

Por otro lado, se aprecia que en la implementación en la implementación CPU-GPU:

(i) El LSOMP es 1,12 veces más rápido que el MP.

(ii) El LSOMP es 1,21 veces más rápido que el WMP.

Tabla 8. Tiempos de ejecución traza b.

	CPU [s]	**GPU-CPU [s]**
MP	1693,49	73,37
WMP	1817	78,09
LS OMP	522,3	59,06

Tabla 9. Tiempos de ejecución traza c.

	CPU [s]	**GPU-CPU [s]**
MP	1677,94	72,28
WMP	2024,75	86,1
LS OMP	860,63	97,6

3.5 ANÁLISIS DE EJECUCIÓN

Finalmente, se realizó un análisis de cada una de las funciones usadas en los tres algoritmos para encontrar los cuellos de botella que limitaban a cada uno de estos. Los tiempos registrados para los algoritmos MP y WMP son registrados en las tablas 10 y 11, donde se observa que las funciones de mayor tiempo fueron la traspuesta y el producto matriz vector.

Tabla 10. Funciones MP.

MP	**PORCENTAJE DE TIEMPO**	**LLAMADOS POR ITERACIÓN**
Transpuesta	61,23%	1
Producto matriz vector	36,65%	2
Max	1,73%	1
SNR	0,05%	1
Soporte	0,01%	1
Resta	0,01%	1
Transferencia, llamados a función, etc..	0,32%	-

Tabla 11. Funciones WMP.

WMP	**PORCENTAJE DE TIEMPO**	**LLAMADOS POR ITERACIÓN**
Transpuesta	61,2%	1
Producto matriz vector	36,68%	2
Max	1,53%	1
SNR	0,05%	1
Norma	0,03%	1
Soporte	0,01%	1
Resta	0,01%	1
Transferencia, llamados a función, etc..	0,49%	-

Por otro lado, en la tabla 12 se muestran los tiempos requeridos por cada una de las funciones del algoritmo LSOMP. Nótese que las funciones que requieren de mayor tiempo son, la función producto matriz-matriz y la función inversa.

Tabla 12. Funciones LSOMP.

LSOMP	**Porcentaje de tiempo**	**Llamados por iteración**
Producto matriz matriz	78,38%	1
Inversa	13,19%	1
Producto matriz vector	6,4%	4
Transpuesta	0,71%	1
Max	0,31%	1
Aumentada	0,06%	1
SNR	0,01%	1
Soporte	<0,01%	1
Resta	<0,01%	1
Transferencia, llamados a función, etc..	0,92%	-

4. CONCLUSIONES

Se evaluaron tres versiones del algoritmo *Matching Pursuit* en un esquema de coprocesamiento CPU-GPU, para la compresión de datos sísmicos. Se estableció cuál de las tres versiones ofrece mejores resultados en términos de factor de compresión y rendimiento computacional.

Nuestros resultados sugieren que el algoritmo *Least Squares Matching Pursuit* ofrece el mejor rendimiento respecto a los otros algoritmos evaluados. La señal reconstruida presenta algunas diferencias en aquellas secciones dónde la amplitud es baja, sin embargo estas modificaciones no son tan relevantes, pues la mayor parte de la información geofísica se encuentra en las amplitudes grandes, las cuales se reconstruyeron sin modificaciones. De igual manera la información en frecuencia para la señal reconstruida se mantiene sin modificaciones en el rango de interés para los geofísicos.

El trabajo futuro se enfocará en la creación de nuevos diccionarios, pues este es un factor importante para la convergencia de los algoritmos y la mejora de las funciones implementadas en el algoritmo *Least Squares Matching Pursuit*, especialmente en la función que efectúa la inversa de las matrices.

CITAS

[1] Elad, M. (2010). Sparse and Redundant Representations From Theory to Applications in Signal and Image Processing (1st ed. Sp).

[2] S. Mallat and Z. Zhang, Matching pursuits with time-frequency dictionaries, IEEE Trans. on Signal Processing, 41(12):3397–3415, 1993.

[3] Andrecut, M. (2009). Fast GPU implementation of sparse signal recovery from random projections. *Engineering Letters*, *17*(3), 8. Retrieved from http://arxiv.org/abs/0809.1833.

[4] Fang, Y., Chen, L., Wu, J., & Huang, B. (2011). GPU Implementation of Orthogonal Matching Pursuit for Compressive Sensing. *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, 1044–1047. doi:10.1109/ICPADS.2011.158

[5] S. Cook, *CUDA Programming*. 2013.

[6] Nvidia, *CUDA C programming guide*, no. July. 2013.

[7] Constantino Pérez Vega, José María Zamanillo Sainz de la Maza, Alicia Casanueva López, “Sistemas de telecomunicación”, Ed. Universidad de Cantabria, 2007.

[8] Harris, M., Blelloch, G. E., Maggs, B. M., Govindaraju, N. K., Lloyd, B., Wang, W., … Margolin, L. G. (2007). Optimizing parallel reduction in CUDA. Proc. of ACM SIGMOD, 21, 13, 104–110.

[9] Sharma, G., Agarwala, A., & Bhattacharya, B. (2013). A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA. Computers and Structures, 128(NOVEMBER 2013), 31–37. http://doi.org/10.1016/j.compstruc.2013.06.015

BIBLIOGRAFIA

FAJARDO ARIZA, Carlos Augusto; REYES TORRES, Oscar Mauricio and RAMIREZ SILVA, Ana Beatriz. Seismic Data Compression Using 2D Lifting Wavelet Algorithms. Revista Ingeniería y Ciencia. Universidad Eafit. [online] 2015. Vol 11, N° 21. [cited: 5 jun. 2015] Available from Internet : http://publicaciones.eafit.edu.co/index.php/ingciencia/article/view/2561/html_11

LARSON, Roland E. and FALVO, David C. Elementary Linear Algebra. 6ed. México: Cengage Learning Editores, 2010.

NVIDIA. CUDA C Programming Guide: Design Guide. [online] Nvidia. [Santa Clara, California, E.U] Nvidia Corporation, March 2015. Vol. 7. PG-02829-001. [cited 30 may 2015] . Available from Internet: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

SALOMON, David. Data Compression The Complete Reference. 4ed. London: Springer, 2001.