EVALUACIÓN DEL RENDIMIENTO COMPUTACIONAL Y LA GANANCIA EN LA COMPRESIÓN AL EMPLEAR TÉCNICAS DE CODIFICACIÓN APLICADAS A TRAZAS SÍSMICAS

CARLOS ARTURO BOADA QUIJANO

Escuela de Ingenierías
Eléctrica, Electrónica
y de Telecomunicaciones

Universidad Industrial de Santander
Facultad de Ingenierías Físico-Mecánicas
Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones
Bucaramanga
2017

EVALUACIÓN DEL RENDIMIENTO COMPUTACIONAL Y LA GANANCIA EN LA COMPRESIÓN AL EMPLEAR TÉCNICAS DE CODIFICACIÓN APLICADAS A TRAZAS SÍSMICAS

CARLOS ARTURO BOADA QUIJANO

Trabajo de Investigación para optar al título de
Magister en Ingeniería Electrónica

Director
PhD. CARLOS AUGUSTO FAJARDO ARIZA

Co-Director
PhD. ÓSCAR MAURICIO REYES TORRES

Universidad Industrial de Santander
Facultad de Ingenierías Físico-Mecánicas
Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones
Bucaramanga
2017

Agradecimientos

Este trabajo fue desarrollado con el apoyo de ECOPETROL y COLCIENCIAS como parte del proyecto de investigación No. 0266-2013.

Agradezco a Dios, padre todo poderoso por darnos la vida, la fortaleza y sabiduría para poder llevar nuestras metas a feliz término.

A mi madre Gloria María Quijano (q.e.p.d) por ser una persona entregada a su familia como a su trabajo y a mi padre Luis Martin Bohada por apoyarme siempre.

A mi esposa Silvia Patricia Arenas por ser la persona que siempre me apoyo en todo este proceso con sus consejos, cariño y paciencia.

A mi director Carlos Augusto Fajardo Ariza y codirector Oscar Mauricio Reyes Torres por sus orientaciones y consejos profesionales como personales.

A todos mis maestros y compañero quienes dejaron una huella imborrable en mi formación y en especial a Christian Hernández, Gabriel Rincón, Carlos Angulo y Efrén Acevedo por sus conocimientos y su calidad profesional como personal.

Índice

Pág.

Introducción **13**

1 Compresión de Datos **15**
1.1 Transformación . . . 16
1.1.1 Transformada Wavelet Discreta . . . 16
1.2 Cuantificación . . . 18
1.2.1 Cuantificación escalar . . . 18
1.3 Codificación . . . 19
1.3.1 Codificación Run-Length . . . 20
1.3.2 Codificación Shannon-Fano . . . 20
1.3.3 Codificación Huffman . . . 20
1.3.4 Codificación Aritmética . . . 21
1.3.5 Codificación Tunstall . . . 21
1.4 Trabajos Relacionados . . . 21

2 Estrategia de Codificación **24**
2.1 Conjunto de Datos . . . 24
2.2 Transformación . . . 25
2.2.1 Esquema *Lifting* . . . 26
2.3 Cuantificación . . . 28
2.4 Codificación . . . 31
2.4.1 Algoritmo Huffman . . . 31
2.4.2 Algoritmo Aritmética . . . 32
2.4.3 Algoritmo Tunstall . . . 36
2.4.4 Estimación de la relación de compresión . . . 39
2.5 Ganancia en la compresión . . . 40
2.6 Rendimiento computacional . . . 46
2.7 Resultados generales en la etapa de codificación . . . 48

3 Descompresión en GPU **50**
3.1 Fundamentos de la GPU . . . 50
3.2 Implementación de la Codificación Huffman . . . 51
3.2.1 Estrategia uno . . . 53
3.2.2 Estrategia dos . . . 54
3.2.3 Estrategia tres . . . 55
3.2.4 Estrategia cuatro . . . 56
3.2.5 Estrategia cinco . . . 56
3.2.6 Relación de compresión para las estrategias . . . 56
3.3 Implementación de la Decodificación Huffman . . . 58
3.3.1 Decodificador versión uno . . . 59
3.3.2 Decodificador versión dos . . . 60
3.3.3 Decodificador versión tres . . . 62
3.3.4 Decodificador versión cuatro . . . 64
3.3.5 *Throughput* y aceleración . . . 65
3.3.6 Selección de la versión del decodificador . . . 68
3.4 Implementación de la Cuantificación inversa . . . 68
3.5 Implementación de la Transformación inversa . . . 69
3.5.1 Transformación Inversa 1D . . . 70
3.5.2 Transformación Inversa 2D . . . 71
3.6 Descompresión de datos sísmicos . . . 73

4 Conclusiones **78**

Referencias **81**

Bibliografía **85**

Lista de Figuras

1.1 Esquema de Compresión. 15
1.2 Transformación Wavelet Discreta [15]. 16
1.3 Transformación Wavelet Discreta 1D. 17
1.4 Transformación Wavelet Discreta 2D. 18
1.5 Cuantificación escalar [5]. 19

2.1 Conjunto de datos utilizados. 25
2.2 *DWT 1D* Esquema *Lifting* [12]. 26
2.3 Histograma del *Dataset 1*. 27
2.4 Histograma del *Dataset 1* con Transformación. 28
2.5 Cuantificación de una traza del *Dataset 1* con 6 bits. 29
2.6 Histograma del *Dataset 1* Cuantificado a 12 bits. 30
2.7 Histograma del *Dataset 1* con Transformación mas Cuantificación de 12 bits. 30
2.8 Árbol Huffman del ejemplo. 32
2.9 Intervalos iniciales. 33
2.10 Observación de A. 33
2.11 Observación de B. 34
2.12 Observación de C. 34
2.13 Observación de A. 34
2.14 Observación de B. 35
2.15 Observación final. 35
2.16 Árbol Tunstall del ejemplo $k = 0$. 36
2.17 Árbol Tunstall del ejemplo $k = 1$. 37
2.18 Árbol Tunstall del ejemplo $k = 2$. 37
2.19 *SNR* vs. *Bits de Cuantificación* para los datos sísmicos con y sin transformación. 41
2.20 Compresión de datos sísmicos sin transformación. 42
2.21 Compresión de datos sísmicos con codificación *Huffman*. 43
2.22 Compresión de datos sísmicos con codificación *Aritmética*. 44
2.23 Compresión de datos sísmicos con codificación *Tunstall*. 45
2.24 Tiempo de compresión y descompresión mediante codificación Huffman. 46

2.25 Tiempo de compresión y descompresión mediante codificación Aritmética. 47
2.26 Tiempo de compresión y descompresión mediante codificación Tunstall. 47
2.27 Comparación en términos de *Throughput* para la Descompresión. 48

3.1 Datos de ejemplo y su tabla de frecuencias. 52
3.2 Árbol Huffman del ejemplo. 52
3.3 Datos codificados utilizando codificación tradicional con variables de 32 bits. 53
3.4 Datos codificados usando paquetes de 32-bits. 54
3.5 Datos codificados usando paquetes de 64-bits. 55
3.6 Datos codificados usando paquetes de 32-bits con ordenamiento. 55
3.7 Estructura del paquete de 64-bits para la estrategia cuatro. 56
3.8 Relación de Compresión para los *Datasets* sin transformación. 57
3.9 Relación de Compresión para los *Datasets* con transformación. 57
3.10 Proceso de decodificación aplicando máscaras a los datos codificados 59
3.11 Tiempo de decodificación, versión uno, datos codificados con paquetes de 32-bits. 60
3.12 Tiempo de decodificación, versión uno, datos codificados con paquetes de 64-bits. 60
3.13 Tiempos de decodificación para la Versión Dos sobre la GTX 660. 60
3.14 Tiempo de decodificación, paquetes organizados vs sin organizar (Estrategias 3 vs 1). 61
3.15 Tiempo de decodificación para la versión dos. 62
3.16 Tiempo de decodificación para la versión tres. 63
3.17 Ejemplo de ejecución de los *kernels* para la creación del vector *índice*. 64
3.18 Tiempo de decodificación para la versión cuatro. 65
3.19 *Throughput* vs relación de compresión 66
3.20 Aceleración del decodificador en GPU, con respecto al CPU i7 a 3.2 *GHz*. 67
3.21 Tiempo de ejecución del *kernel* de cuantificación inversa. 69
3.22 Esquema *lifting* para la implementación de la *IDWT 1D*. 70
3.23 Esquema *lifting* para la implementación de la *IDWT 1D* con dos niveles. 70
3.24 Tiempo de ejecución de la *IDWT 1D* con dos niveles. 71
3.25 Esquema *lifting* para la implementación de la *IDWT 2D* 72
3.26 Tiempo de ejecución de la *IDWT 2D*. 72
3.27 *Throughput* descompresión vs relación de compresión. 76
3.28 Tiempo de descompresión para diferentes *Datasets*. 77

Lista de Tablas

2.1 Resumen *Datasets*. 24
2.2 Probabilidad de Símbolo para el ejemplo. 31
2.3 Diccionario Huffman del ejemplo. 32
2.4 Diccionario Tunstall para $n = 2$. 37
2.5 Diccionario Tunstall para $n = 3$. 38
2.6 Diccionario Tunstall para $n = 4$. 38
2.7 *RC* Tunstall *Dataset 1 + DWT 1D Nivel 2 + Cuantificación a 11 bits*. 45
2.8 Relación de compresión y *SNR* para los *Datasets* con 12 bits de cuantificación. 48
2.9 Tiempo de compresión y descompresión para los *Datasets* con 12 bits de cuantificación. 49
2.10 *Throughput* de descompresión para los *Datasets* con 12 bits de cuantificación. 49

3.1 Especificaciones de la GPU. 51
3.2 Porcentaje de uso de la GPU [48]. 51
3.3 Diccionario Huffman del ejemplo. 52
3.4 Diccionario en dos vectores. 53
3.5 Vector de *índice* para la Estrategia uno 54
3.6 Vector de *índice* para la Estrategia dos 55
3.7 Resultados de predecesores. 66
3.8 Tiempo de decodificación para todas las plataformas de computo. 67
3.9 Distribución de hilos por bloque para la descompresión 73
3.10 Tiempo de ejecución *kernels* decodificación, compresión sin etapa de transformación. 73
3.11 Tiempo de ejecución *kernels* decodificación, compresión con *DWT 1D Nivel 2*. 73
3.12 Tiempo de ejecución *kernels* decodificación, compresión con *DWT 2D*. 74
3.13 Tiempo de ejecución *kernel* cuantificación inversa. 74
3.14 Tiempo de ejecución *kernels IDWT 1D* con dos niveles. 74
3.15 Tiempo de ejecución *kernels IDWT 2D*. 74
3.16 Tiempo de descompresión, con esquema de compresión sin etapa de transformación. 75
3.17 Tiempo de descompresión, con esquema de compresión con *DWT 1D Nivel 2*. 75
3.18 Tiempo de descompresión, con esquema de compresión con *DWT 2D*. 75
3.19 Relación de compresión para la compresión con y sin transformación. 75

Resumen

TITULO: EVALUACIÓN DEL RENDIMIENTO COMPUTACIONAL Y LA GANANCIA EN LA COMPRESIÓN AL EMPLEAR TÉCNICAS DE CODIFICACIÓN APLICADAS A TRAZAS SÍSMICAS*

AUTOR: CARLOS ARTURO BOADA QUIJANO**

PALABRAS CLAVE: Datos sísmicos, Compresión, Transformación Wavelet, Cuantificación, Codificación, GPU.

El presente proyecto se encuentra enmarcado dentro del programa estratégico de investigación titulado *"Migración sísmica pre-apilado en profundidad por extrapolación de campos de onda utilizando computación de alto desempeño para datos masivos en zonas complejas"*, con el cual se busca aumentar la resolución de las imágenes del subsuelo y reducir el tiempo de cómputo de las aplicaciones sísmicas mediante plataformas alternativas diferentes a las CPU como las GPU y las FPGA.

Aumentar la resolución de las imágenes sísmicas implica aumentar la cantidad de datos sísmicos que requieren ser adquiridos y procesados. El uso de algoritmos de compresión de datos sísmicos es cada vez más necesario, pues el volumen de los datos que la industria petrolera requiere procesar actualmente se encuentran en el orden de los cientos de Terabytes.

En este proyecto se evaluaron tres algoritmos de codificación de entropía (Huffman, Aritmética y Tunstall), para la compresión de datos sísmicos en un esquema de compresión con pérdidas compuesto por tres etapas: Transformación, Cuantificación y Codificación. Para las dos primeras etapas se usaron transformación *DWT* (1D, 2D) y cuantificación uniforme. Bajo este esquema se determinó el rendimiento computacional y la relación de compresión para cada uno de los algoritmos.

Nuestros resultados evidenciaron que la codificación Huffman ofrece los mejores resultados globales, es decir, este esquema de codificación ofrece una alta relación de compresión y el mejor *throughput* de descompresión.

Se presentaron varias estrategias para la implementación de la decodificación Huffman en una arquitectura GPU. En estas estrategias se hace uso de paquetes con cabeceros. De esta forma, las estrategias usadas permitieron aprovechar la capacidad de cómputo en paralelo de la arquitectura GPU. La estrategia propuesta permite que cada uno de los paquetes pueda ser decodificado de forma independiente por diferentes hilos de la GPU.

* Trabajo de Investigación

** Facultad de Físico-Mecánicas. Escuela Ingenierías Eléctrica, Electrónica y Telecomunicaciones. Maestría en Ingeniería Electrónica. Director: PhD. Carlos Augusto Fajardo Ariza. Codirector: PhD Óscar Mauricio Reyes Torres.

Abstract

TITLE: COMPUTATIONAL PERFORMANCE EVALUATION AND COMPRESSION GAIN BY USING CODIFICATION TECHNIQUES APPLIED TO SEISMIC TRACES*

AUTHOR: CARLOS ARTURO BOADA QUIJANO**

KEYWORDS: Seismic data, Compression, Wavelet Transform, Quantization, Coding, GPU.

The current project is framed within the strategic research program titled *"Seismic migration pre-stacked in depth by extrapolation of wave fields using high performance computing for massive data in complex zones"*, which it is used to increase the resolution of subsoil's images and reduce the time of computation of the seismic applications by handling alternative platforms different to the CPUs like the GPUs and the FPGAs.

Expanding resolution of seismic images involves increasing the amount of seismic data that needs to be gathered and processed. Use of seismic data compression algorithms is completely necessary since data volume that oil industry requires to process is currently around hundreds of Terabytes.

In this project, three entropy coding algorithms (Huffman, Arithmetic and Tunstall) were evaluated for compression of seismic data in a lossy compression scheme composed of three stages: Transformation, Quantization and Coding. Coding stage offers the greatest computational cost. For the first two stages DWT (1D and 2D, with one and two levels of decomposition) and uniform quantification were used. Under this scheme computational performance and compression ratio for each algorithms were determined.

The results showed that Huffman coding offers the best overall results, which means that this coding scheme presents a high compression ratio and the best decompression *throughput*.

Several strategies were presented for Huffman decoding implementation in a GPU architecture. These strategies use packets with headers, in such a way that codes at the boundary of the packets are forced to align. In this way, the strategies used let to take advantage of parallel computing capacity of GPU architecture. The proposed strategy allows each packet to be decoded independently by different threads of the GPU.

* Trabajo de Investigación

** Facultad de Físico-Mecánicas. Escuela Ingenierías Eléctrica, Electrónica y Telecomunicaciones. Maestría en Ingeniería Electrónica. Director: PhD. Carlos Augusto Fajardo Ariza. Codirector: PhD Óscar Mauricio Reyes Torres.

Introducción

Actualmente existe la necesidad de aumentar las reservas de gas y petróleo, para ello es necesario realizar exploraciones sísmicas en escenarios geológicos cada vez más complejos, las cuales requieren el procesamiento y análisis de una mayor cantidad de datos sísmicos [1].

El incremento en la cantidad de datos a procesar conlleva a un aumento del costo computacional, tanto en la transmisión, como en el almacenamiento; en este sentido, es deseable comprimir la información sísmica para reducir el tiempo de transmisión y el espacio de almacenamiento, así como los costos asociados [2, 3].

Para la búsqueda de petróleo es necesario realizar un experimento sísmico, el cual permite recolectar la información correspondiente a la estructura del terreno por medio de trazas sísmicas. Luego estos datos se procesan en un sistema de cómputo para producir una imagen sísmica que posteriormente será interpretada para disminuir la incertidumbre referente a la presencia de hidrocarburos atrapados en las capas de la tierra [1].

En este momento el volumen de los datos generados en un estudio sísmico se encuentra alrededor de los cientos de Terabytes, los cuales deben ser transmitidos y almacenados para su posterior procesamiento. Generalmente estos datos son comprimidos en el lugar donde se realiza el estudio sísmico, antes de ser enviados al centro de procesamiento, con el fin de ahorrar costos de transmisión[1] y de almacenamiento.

Para la industria es deseable comprimir este tipo de datos de forma eficiente, tanto en términos de relación de compresión (para ahorrar costos de almacenamiento y transmisión) como en términos de procesamiento (reducir el tiempo de compresión-descompresión). Lo anterior evidencia la necesidad de encontrar algoritmos compresión-descompresión que ofrezcan los mejores resultados tanto en términos de relación de compresión como en rendimiento computacional.

Dentro de los algoritmos de compresión que ofrecen mejores resultados en cuanto a relación de compresión, se encuentran aquellos que utilizan codificación variable [3, 4, 5] (entre ellos Huffman y Aritmética), sin embargo, debido a la longitud variable de los datos codificados, se requiere de varios ciclos de reloj para decodificar un dato, lo cual incrementa los tiempos de descompresión. Por otro lado, los algoritmos de

[1] Algunas veces la transmisión es vía satélite y los costos de transmisión pueden ascender a cientos de miles de dólares.

codificación de códigos de longitud fija, ofrecen una menor relación de compresión si se compara con los aquellos de longitud variable.

Esta investigación se centra en el análisis y evaluación de los algoritmos de codificación Huffman y Aritmética (códigos de longitud variable) y el algoritmo de codificación Tunstall (códigos de longitud fija). Se busca determinar cuál sería la mejor alternativa para la codificación de los datos sísmicos, teniendo en cuenta la relación de compresión y el rendimiento computacional. El proyecto busca seleccionar una alternativa de decodificación para ser implementada en una plataforma de altas prestaciones.

Varios trabajos han abordado el tema de la compresión de los datos sísmicos [6, 2, 3, 7, 8], en los cuales se evidencia el alto costo computacional que demanda el proceso de decodificación. Esta etapa es altamente secuencial, debido a la longitud variable de los códigos, este proceso secuencial se convierte rápidamente en un cuello de botella en las arquitecturas paralelas, por esta razón la implementación de la decodificación ha resultado ser más eficiente en arquitecturas CPU que en arquitecturas paralelas.

En la presente investigación nosotros analizamos las tres alternativas de decodificación mencionadas anterioremente con el fin de identificar cuál ofrece mejores resultados en términos de relación de compresión y costo computacional. El estudio permitió identificar que la codificación Huffman es la que presenta el mejor balance en términos de relación de compresión y costo computacional. Para su implementación en paralelo se propusieron estrategias que utiliza paquetes con encabezados para guardar los datos codificados. Estas estrategias buscan paralelar el proceso de decodificación Huffman. Los algoritmos fueron implementados en GPU y probados usando diferentes conjuntos de datos suministrados por ECOPETROL. Los resultados experimentales permitieron establecer cómo la relación de compresión se ve afectada por las técnicas propuestas y cómo el rendimiento de los algoritmos se ve afectado por el tamaño del bloque. Finalmente, también se mostró cómo la relación de compresión influye en el tiempo de descompresión.

El documento está organizado de la siguiente forma: el capítulo 2 presenta una descripción teórica del proceso de compresión-descompresión y describe algunos trabajos investigativos previos relacionados con la presente investigación. El capítulo 3 muestra el estudio realizado sobre las tres etapas que comprenden el proceso de compresión, el capítulo se centra principalmente en la etapa de codificación. El capítulo 4 muestra las estrategias implementadas para realizar la descompresión en paralelo sobre una arquitectura GPU. Finalmente, en el capítulo 5 se describen nuestras conclusiones y algunas sugerencias para un trabajo futuro.

Capítulo 1

Compresión de Datos

La compresión de datos es ampliamente usada para reducir los costos de transferencia y de almacenamiento de información. En general, éstos algoritmos utilizan tres etapas: transformación, cuantificación y codificación. La compresión es el proceso mediante el cual una cadena de datos de entrada es convertida en otra cadena salida con una representación en bits menor a la original [4].

Figura 1.1: Esquema de Compresión.

Un esquema de compresión (Figura 1.1) se entiende como un algoritmo de compresión el cual toma una entrada X y genera una representación Xc con menor número de bits. Un algoritmo de descompresión toma Xc y genera una reconstrucción Y. Existen dos clases de algoritmos de compresión:

- **Compresión sin pérdida:** la salida Y resulta ser idéntica a la entrada X. Mantiene toda la integridad de la información, pero su factor de compresión es bajo, es implementado en las transacciones bancarias, imágenes y en la compresión de texto. Posee dos etapas (Transformación y Codificación) o en algunos casos de solo codificación.

- **Compresión con pérdida:** lleva asociada cierta pérdida de información pero favorece el factor de compresión. Consiste en aplicar una transformación a los datos de entrada, con el fin de reducir la entropía de la señal, posteriormente se aplica un umbral de decisión (Cuantificación) y finalmente se comprime mediante un algoritmo de codificación.

Según el teorema Shannon, una cadena de longitud L con N símbolos se puede comprimir hasta $L \times H$ bits, donde H representa la entropía de la cadena y expresa el número promedio de bits necesarios para representar un símbolo presente en la cadena de datos [9]. La entropía está dada por la siguiente ecuación

$$H = -\sum_{i=1}^{n} P_i * log_2(P_i) \tag{1.1}$$

donde P_i es la probabilidad de aparición de cada uno de los N símbolos presentes en la cadena.

Con respecto a la compresión de datos sísmicos, éstos están compuestos por tres tipos de componentes: información geofísica relevante, redundancia y un ruido de amplio espectro no correlacionado [10].

La redundancia R se puede expresar como la cantidad de bits adicionales para representar un dato y se define como la diferencia entre el número de bits que se empleó para representar un símbolo Hs y la entropía H de la cadena, es decir:

$$R = Hs - H = Hs + \sum_{i=1}^{n} P_i * log_2(P_i) \tag{1.2}$$

Los diferentes métodos de compresión, en esencia, buscan reducir la redundancia [4]. Como métrica para determinar la compresión obtenida, está la relación o radio de compresión RC, definido por la siguiente Ecuación 1.3.

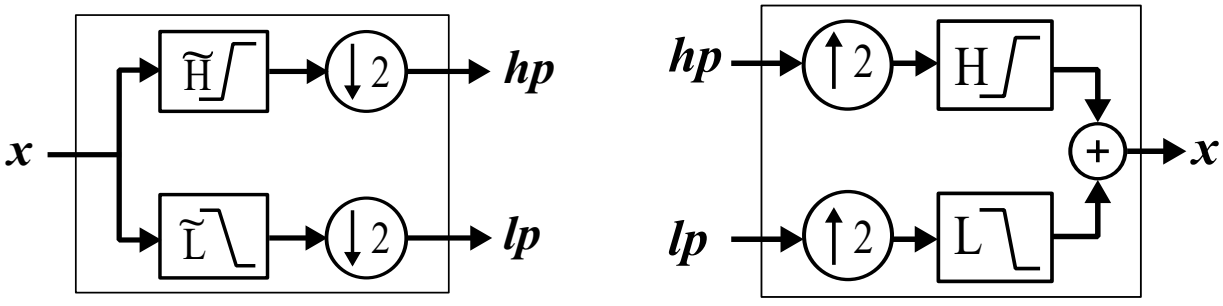
$$RC = \frac{Tamaño\ de\ los\ datos\ sin\ compresión}{Tamaño\ de\ los\ datos\ comprimidos} \tag{1.3}$$

1.1 Transformación

Con la trasformación se busca reorganizar la información mediante un cambio de dominio con el fin de facilitar la compresión en las etapas subsecuentes. La aplicación de la transformación a un conjunto de datos permite concentrar su energía en ciertos coeficientes y reducir la entropía de la información, con ello se favorece la relación de compresión. Dentro de las diferentes transformaciones existentes (Fourier, Coseno, Wavelet, Curvelet, etc) se tendrá en cuenta la transformación Wavelet, la cual es ampliamente usada en la compresión de datos sísmicos [11, 12, 3, 7, 13, 1, 14, 10].

1.1.1 Transformada Wavelet Discreta

La *DWT* (*Discrete Wavelet Transform*) descompone una señal de entrada x mediante la convolución de dos filtros $\widetilde{H_p}$ (Pasa-altas), $\widetilde{L_p}$ (Pasa-bajas) en dos señales h_p y l_p llamadas detalles y aproximaciones, que corresponden a las componentes de alta frecuencia y baja frecuencia de la señal x [15, 16].



(a) Descomposición. (b) Reconstrucción.

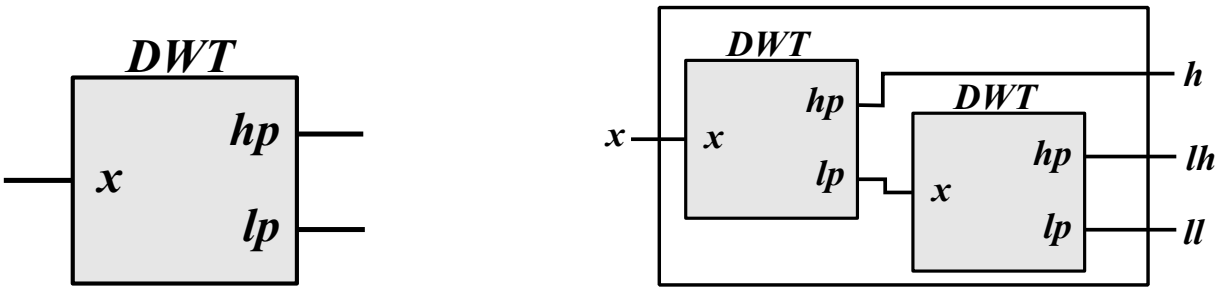
Figura 1.2: Transformación Wavelet Discreta [15].

Para la reconstrucción de la señal se aplica la transformada inversa *IDWT* (*Inverse Discrete Wavelet Transform*), la cual consiste en pasar por los filtros H_p y L_p los coeficientes h_p y l_p, posteriormente unir los resultados para obtener nuevamente la señal x. El esquema completo de la transformada y su inversa se muestra en la Figura 1.2.

La Figura 1.3a muestra la transformada *DWT* en un nivel, la cual consiste en aplicar el esquema presente en la Figura 1.2a, para una entrada x_n de longitud n, los vectores de coeficientes h_p, l_p tendrán una longitud de $n/2$.

La Figura 1.3b muestra la transformada *DWT* en dos niveles, en este caso la componente l_p se pasa por la transformada *DWT*, como resultado se obtienen los coeficientes de segundo nivel lh y ll de tamaños $n/4$ y los coeficientes de detalles de primer nivel h de tamaño $n/2$.

De manera general, la *DWT* se puede aplicar en el número de niveles deseado, lo cual depende del tipo de aplicación.



(a) DWT Nivel 1.

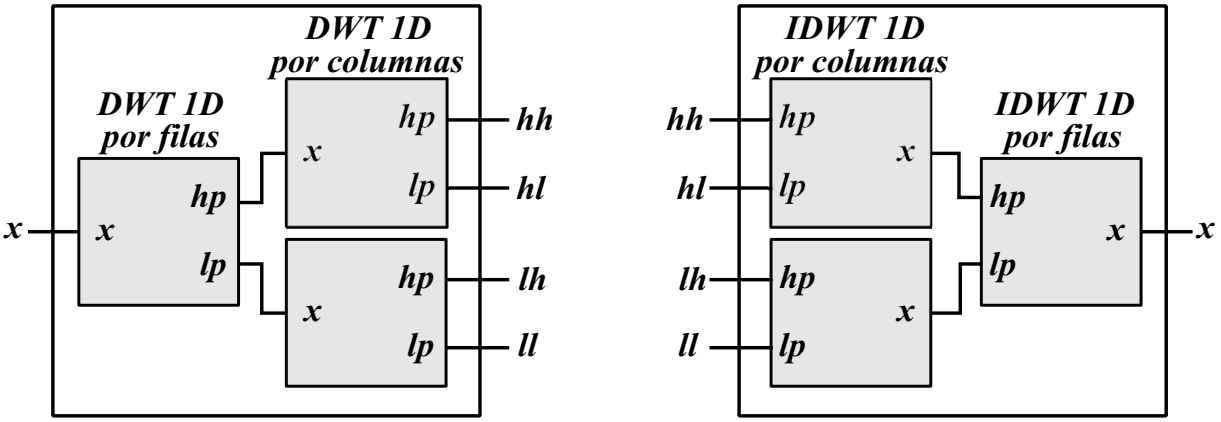
(b) DWT Nivel 2.

Figura 1.3: Transformación Wavelet Discreta 1D.

Cuando la entrada es una señal bidimensional como una imagen o en general una matriz, se aplica la *DWT 2D*, la cual descompone la señal $\boldsymbol{x}$ en cuatro sub-matrices, las cuales contienen los coeficientes de detalles: diagonales $\boldsymbol{hh}$, verticales $\boldsymbol{hl}$, horizontales $\boldsymbol{lh}$ y los coeficientes de aproximación $\boldsymbol{ll}$.

Para realizar la *DWT 2D* de una señal $\boldsymbol{x}_{n\times m}$ de tamaño n filas y m columnas se procede de la siguiente forma:

- Se aplica la *DWT 1D* a cada fila de la señal x, es decir, se aplican n transformadas 1D, como resultado se obtienen dos matrices de tamaño n filas por $m/2$ columnas.
- A las dos matrices resultantes se les aplica la *DWT 1D* a cada columna, por ende se aplican $m/2$ transformadas 1D a cada matriz. En total se tienen m transformadas 1D, el resultado de este paso son cuatro matrices (*hh, hl, lh, ll*) de tamaños $n/2$ filas por $m/2$ columnas. En la Figura 1.4 se muestra el esquema de la transformación 2D y su inversa.



(a) Descomposición. (b) Reconstrucción.

Figura 1.4: Transformación Wavelet Discreta 2D.

Para el primer nivel de la *DWT 2D* se aplica el esquema de la Figura 1.4, para el segundo nivel se procesa la matriz $\boldsymbol{ll}$ nuevamente mediante la *DWT 2D*, obteniendo los coeficientes de primer nivel hh_1, hl_1, lh_1 y de segundo nivel hh_2, hl_2, lh_2, ll_2 [15, 16, 17].

1.2 Cuantificación

La cuantificación es un proceso mediante el cual un conjunto de datos cuyo rango es posiblemente infinito se expresa mediante otro conjunto de salida donde su rango es menor al original.

Es en esta etapa donde se presenta una pérdida de información, debido al proceso de aproximación, por lo tanto, es importante medir la calidad de la información. Una forma de hacer esto es mediante la relación señal a ruido *SNR*, la cual se calcula como sigue:

$$SNR_{dB} = 10 * log_{10}\left(\frac{\sum_{i=1}^{L}(S_i)^2}{\sum_{i=1}^{L}(\Delta S_i)^2}\right) \tag{1.4}$$

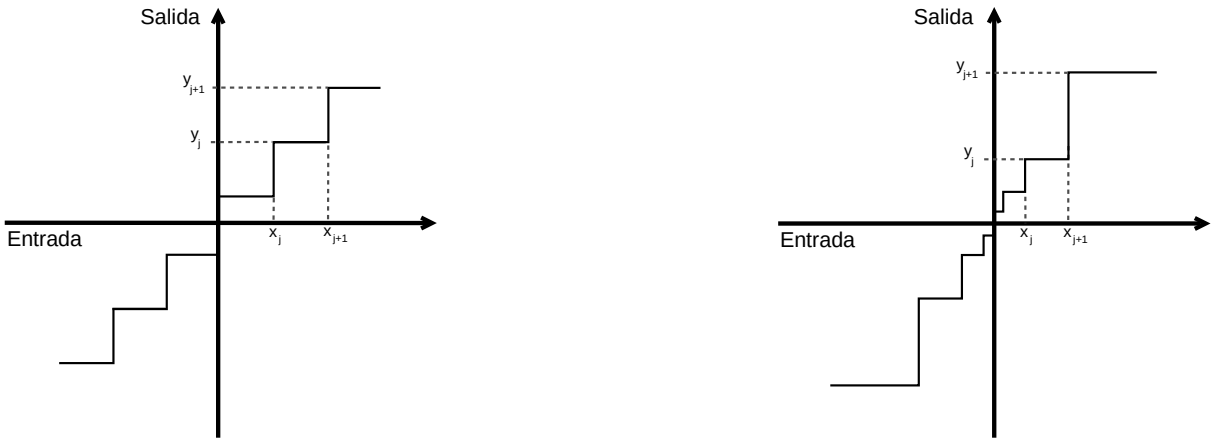
donde S_i son las componentes de la cadena original y ΔS_i es la diferencia entre las componentes de la cadena original y la cadena reconstruida [4].

1.2.1 Cuantificación escalar

En este proceso, cada valor de entrada es representado mediante un valor aproximado entre un conjunto finito de posibles valores a tomar.

La cuantificación escalar la podemos ver como una función Q que mapea un conjunto real $\mathbb{R}$ a un subconjunto discreto, $Q = \mathbb{R} \longrightarrow Y$, siendo Y el conjunto de salida expresado por $Y = \{y_1, ...y_N\}$, donde N representa el tamaño del cuantificador. En palabras más sencillas, el proceso de cuantificación puede verse como un cambio de dominio $(x \in \mathbb{R}) \Rightarrow Q(x) = y_i$ [5].

Un punto importante de la cuantificación escalar es la resolución dada por $r = log_2(N)$, la cual indica la exactitud con la que se representa la señal original y es la que afecta de forma directa a la relación señal a ruido *SNR*.



(a) Cuantificación uniforme.

(b) Cuantificación no uniforme.

Figura 1.5: Cuantificación escalar [5].

En la Figura 1.5 se muestran las dos clases mas importantes de cuantificación escalar.

Cuantificación uniforme: es el tipo de cuantificación más simple, pero no por ello la menos eficaz, es conocido también como cuantificador lineal, debido a que los intervalos de decisión forman una línea recta. El tamaño de todos sus intervalos se representa mediante la Ecuación 1.5.

$$\begin{aligned} \Delta x &= x_{j+1} - x_j \\ \Delta y &= y_{j+1} - y_j \end{aligned} \tag{1.5}$$

donde los x_j, x_{j+1} son los límites de decisión para la entrada y los y_j, y_{j+1} son los niveles de reconstrucción para la salida [5].

Cuantificación no uniforme: como se observa en la Figura 1.5b, en este cuantificador los intervalos de decisión no son uniformes. Este tipo de cuantificación generalmente se aplica en aquellas aplicaciones en las que se conoce la distribución de probabilidad de la entrada. Es posible diseñar un cuantificador de tal manera que tenga un menor espaciado en los intervalos donde la probabilidad de ocurrencia de la entrada es grande y un paso mayor para los intervalos en los cuales la entrada tiene probabilidad baja. Esto con el fin de mejorar la relación señal a ruido al ser comparado con un cuantificador uniforme con el mismo número de niveles [5].

1.3 Codificación

En esta etapa es donde realmente ocurre la compresión, ya que, es la encargada de representar una cadena de símbolos, mediante un número menor de bits. La codificación para la compresión de datos consiste

en aplicarle un cambio de dominio a un conjunto de palabras o símbolos (X) con la idea de generar un conjunto de salida (Y) con nuevos símbolos, donde estos nuevos símbolos son representados en un número menor de bits a los originales [9, 18].

Existen varios esquemas de codificación para la compresión de datos. A continuación, se nombran algunos de los más importantes.

1.3.1 Codificación Run-Length

Consiste en que si un símbolo $\boldsymbol{x}_i$ aparece N veces de forma consecutiva en el conjunto de entrada, estas N apariciones de $\boldsymbol{x}_i$ son remplazadas por el par único Nx_i. Por lo general el código asignado a este par único es representado por 16 bits (2 Bytes), donde el primer Byte expresa el símbolo y el segundo Byte en número de repeticiones. Si el número de repeticiones supera el valor de 255, o en su defecto el símbolo representado sea mayor a este valor, es necesario usar un código de 32 bits [4].

1.3.2 Codificación Shannon-Fano

Desarrollada por Claude Shannon y Rober Fano en 1948 [9], fue el primer esquema de codificación de longitud variable (VLC) en presentar el mejor conjunto de códigos variables. Su algoritmo se basa en un conjunto de n símbolos con frecuencia de repetición conocida, estos se disponen de manera descendente según su frecuencia de aparición. Posteriormente, el conjunto de símbolos es dividido en dos subconjuntos, los cuales tiene similares probabilidades. A todas las componentes del primer subconjunto de símbolos se les asignan códigos que inician en 0, mientras que al segundo se les asignan códigos que inician en 1. El algoritmo repite los pasos de dividir cada subconjunto en dos y asignar un código diferente a cada subdivisión, así de manera sucesiva, hasta que no se puede subdividir más ninguno de los subconjuntos.

1.3.3 Codificación Huffman

Propuesto por David Huffman en 1952 [19] y es uno de los esquemas de codificación más importante en la compresión de datos, se encuentra en muchas aplicaciones que requieren compresión de datos. La codificación Huffman es óptima cuando la probabilidad de ocurrencia es potencia de dos, lo cual es poco probable en la mayoría de las aplicaciones, sin embargo, es el mejor dentro de los algoritmos de su misma clase. Tiene una gran semejanza a la codificación Shannon-Fano, pero Huffman produce un mejor código de salida.

El algoritmo parte de la creación de una lista que contenga los símbolos en una disposición descendente, según su probabilidad de aparición. Seguidamente se crea el árbol de abajo a arriba ubicando un símbolo en cada rama. Se realiza este proceso de manera secuencial, cada paso agrega dos símbolos con las probabilidades más bajas. Ésta secuencia se realiza de forma iterativa hasta que la lista de símbolos se termina. Para determinar los códigos de los símbolos se realiza el recorrido por el árbol.

1.3.4 Codificación Aritmética

La codificación Aritmética puede verse como una generalización de Huffman, por lo cual produce mejores resultados. Al igual que la codificación Huffman, ambas logran compresión reduciendo el número de bits requerido para representar un símbolo. La codificación Huffman asigna un código de un numero de bits entero a cada símbolo, mientras que en la codificación Aritmética no se tiene dicha restricción, por lo cual se mejora la relación de compresión en la mayoría de las aplicaciones.

La idea de la codificación aritmética es representar una secuencia de símbolos mediante un intervalo al que se le asigna un código [20]. Para lograr encontrar el intervalo es necesario conocer de antemano la frecuencia de ocurrencia de los símbolos, luego el intervalo es subdividido en subintervalos según la probabilidad de ocurrencia de los símbolos, el subintervalo que corresponde al primer símbolo se considera como en nuevo intervalo y a éste se le realiza el mismo proceso hasta que se llegue a un número determinado de símbolos [5].

1.3.5 Codificación Tunstall

El algoritmo fue desarrollado por Brain Parker Tunstall en 1967 [21], es un algoritmo de códigos de longitud fija, lo cual beneficia el proceso de decodificación, al poderse paralelar. Por otro lado, al usar códigos de longitud fija se evita la propagación de errores en la decodificación. Usa los principios de la codificación variable para generar secuencias de símbolos variables en lugar de códigos, a estas secuencias variables se les asigna códigos de longitud fija.

La técnica consiste en crear un árbol con $\boldsymbol{N}$ ramas correspondientes a los $\boldsymbol{S}$ símbolos presentes en la cadena, a la rama con mayor probabilidad se remplaza por $\boldsymbol{N}$ subcadenas ($\boldsymbol{Sx}$), donde $\boldsymbol{x}$ corresponde al conjunto de $\boldsymbol{N}$ símbolos, este proceso se repite $\boldsymbol{k}$ veces, donde el número ($\boldsymbol{n}$) de bits necesarios para los códigos, se calcula mediante la siguiente Ecuación: $N + k(N-1) \leq 2^n$ [22, 23].

1.4 Trabajos Relacionados

Dentro de los artículos, tesis y trabajos realizados con respecto al tema de investigación, se resalta el procesos de compresión y descompresión de datos compuesto por tres fases (compresión con perdida): la primera fase consta de una transformación compuesta principalmente por la Transformación Wavelet y en otros por la *DCT*; para la segunda fase conformada por la cuantificación, se menciona en gran media una cuantificación escalar uniforme y en algunos, se presenta una cuantificación no uniforme y finalmente para el tema principal de esta investigación la fase de codificación, se resalta el uso de algoritmos *VLC* (*Variable Length Code*) como lo son Huffman y Aritmética, siendo Huffman el más mencionado. A continuación, se citan algunas de las más importantes investigaciones referentes a la compresión de datos.

A finales de la década de los 70s y entre las décadas de los 80s y los 90s gran parte de los trabajos en compresión con perdida se centraron en la compresión de imágenes y vídeo [24, 25, 26, 27, 28, 29, 30, 31]. A mediados de los 80s se desarrolló el método de compresión de imágenes JPEG, desarrollado por *Joint Photographic Experts Group*, más tarde en 1992, se convirtió en un estándar en la compresión de

imágenes con pérdida. El estándar se basa en la Transformación Discreta del Coseno (DCT), seguido de cuantificación uniforme y un esquema de codificación compuesto por RLE (*Run Length Encoding*) y codificación Huffman [32].

En el contexto de la compresión de datos sísmicos uno de los primeros trabajos reportados al respecto fue desarrollado por Wood en 1974 [33], en el cual, presenta un esquema de compresión con perdida, para la transmisión de los datos comprimidos a larga distancia. Empleo códigos de longitud media de 3 bits, para describir los datos de tal manera que se minimice la pérdida de información, el enfoque de este trabajo fue la reducción de la redundancia en los datos, para ello, uso un método en el domino del tiempo y la Transformación Walsh.

En 1990 Spanias [34], presento una investigación que se concentró en la etapa de transformación, para la compresión de datos sísmicos, en ella se evaluaron la Transformación Discreta de Fourier (DFT), la Transformación Discreta del Coseno (DCT), la Transformación Walsh-Hadamard (WHT) y la Transformación Karhunen-Loeve (KLT); sus mejores resultados en cuanto a compresión y calidad de la información se obtuvieron por medio de la KLT, sin embargo esta transformación es de alto costo computacional.

En la década de los 90s, existió un gran auge en el estudio de los algoritmos compuestos por Transformación Wavelet [35]. En el año 2000, esta transformación entro a ser parte del esquema de compresión de imágenes en el estándar JPEG 2000 [36]. Con respecto a la aplicación de esta transformación en la compresión de datos sísmicos, existen varios estudios que muestran su utilidad [37, 38, 39, 2, 3, 7].

En la etapa de cuantificación, el tipo más usado es una cuantificación uniforme, al ser de menor costo computacional y preservar mejor la información geofísica relevante [34, 39, 2, 3, 7, 40]. Para la etapa de codificación, los esquemas de codificación más utilizados se basan en códigos de longitud variable (VLC) como lo son Huffman [38, 39, 6, 2] y Aritmética [41, 3, 7, 8]. En algunos trabajos se emplea una etapa previa a los algoritmos VLC compuesta por codificación RLE [38, 6].

Referente a la aceleración de la compresión-descompresión de datos, existen varios estudios que abordan este tema. Haugen en el 2009 [42], trabajo con algoritmos de compresión con perdida para incrementar en el ancho de banda entre la CPU y la GPU, su estrategia comprimía los datos sísmicos antes de ser transferidos, para luego ser descomprimidos en la GPU.

Haugen empleo un esquema compuesto por bancos de filtros, seguido de cuantificación uniforme y codificación Huffman y RLE. Sus resultados no lograron acelerar el proceso de transferencia, debido a que, el tiempo empleado en la descompresión era mayor, al de transferir los datos sin compresión; sin embargo, siguiere que la estrategia sería viable si se implementa en otro tipo de bus más lento.

Analizando los resultados obtenidos por Haugen, la principal perdida de rendimiento en esta investigación se debe a que el nivel de paralelado del algoritmo de decodificación fue prácticamente nulo, gran parte del

algoritmo es ejecuto de forma secuencial.

En el 2010 Aqrawi [43], presento una investigación para la aceleración de la transferencia de los datos entre el disco y la memoria principal, su esquema se basa en la compresión de datos sísmicos con y sin perdida compuesto por la *DCT*, cuantificación uniforme, codificación optimizada *RLE* y codificación Huffman. Su estudio uso discos duros tradicionales (*HDDs*) y de estado sólido (*SSDs*), sus resultados muestran una aceleración de 1.4x para los *HDDs*, pero no logró aceleración para los *SSDs*. Su esquema, se ejecutó en una cooperación entre CPU y GPU, en la primera se ejecutaron los algoritmos de codificación, debido a que, los tiempos en la GPU no mejoraban, por lo cual, la GPU quedo encargada de la etapa de transformación y cuantificación.

Capítulo 2

Estrategia de Codificación

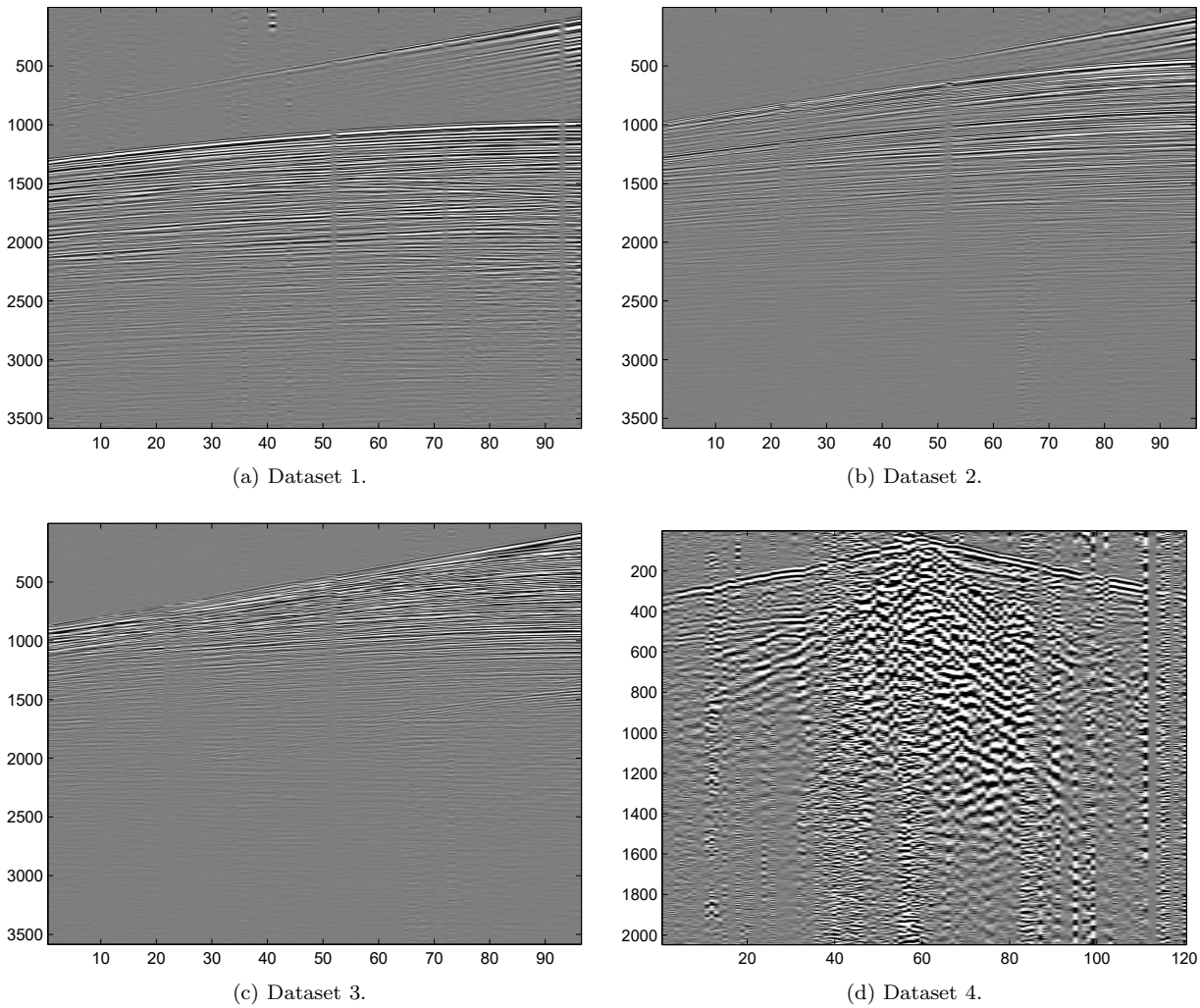
En el presente capitulo se discuten los criterios tenidos en cuenta para determinar la estrategia de codificación más idónea para la compresión de datos sísmicos en un esquema de compresión con pérdida (Transformación, Cuantificación y Codificación) pensando en una implementación para arquitecturas de altas prestaciones. También se muestran los conjuntos de datos usados para la evaluación de las estrategias de codificación. La plataforma de cómputo empleada para el estudio de las estrategias de codificación incluye una CPU *Intel Core I7 - 4700 HQ* a 3.2 GHz, con 8 GB de RAM *DDR3 - 1600*.

2.1 Conjunto de Datos

En la Figuras (2.1a, 2.1b, 2.1c, 2.1d) se muestran los conjuntos de datos empleados para el estudio de las estrategias de codificación. Cada *Dataset* corresponde a un disparo de una adquisición sísmica diferente. En la Tabla 2.1 se presenta un resumen de los datos utilizados en cuanto a número de muestras por traza, cantidad de trazas, total de muestras y tamaño ocupado en disco. Estos conjuntos de datos sísmicos fueron suministrados por Ecopetrol, empresa que financió esta investigación.

Tabla 2.1: Resumen *Datasets*.

Dataset	Muestras	Trazas	Total	Tamaño $[MB]$
1	3584	96	344064	1.3763
2	3584	96	344064	1.3763
3	3584	96	344064	1.3763
4	2048	120	245760	0.9830



(a) Dataset 1.

(b) Dataset 2.

(c) Dataset 3.

(d) Dataset 4.

Figura 2.1: Conjunto de datos utilizados.

2.2 Transformación

Al analizar el esquema de la Transformada Wavelet Discreta Figura 1.2, la entrada $\boldsymbol{x}$ es filtrada mediante los filtros H_P, L_p. Una vez se ha filtrado la señal, los dos resultados son diezmados por un factor de dos, para obtener las salidas h_p y l_p cuyas dimensiones son la mitad de la entrada.

Desde el punto de vista computacional, la etapa de transformación es costosa e ineficiente. En primer lugar, el proceso de filtrado se realiza por medio de convoluciones, las cuales representan un alto costo computacional. De otro lado, se procesa toda la señal para después descartar la mitad de los resultados obtenidos.

Con fin de mejorar el rendimiento computacional de la etapa de transformación, W. Swelden en 1998 [44], propuso un esquema, que mejora el rendimiento computacional. Este esquema reduce los requisitos

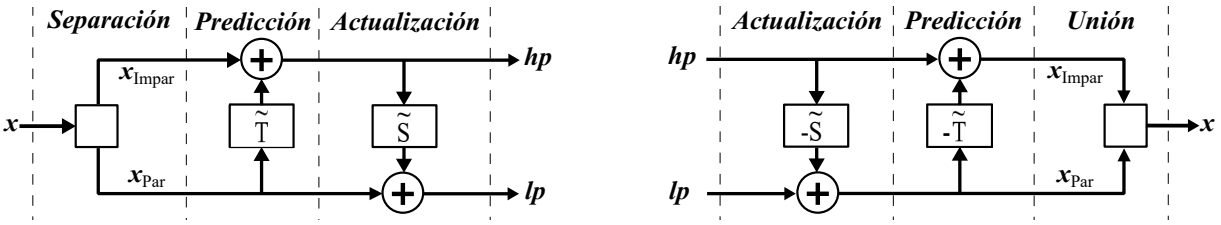
computacionales y de almacenamiento al remplazar las convoluciones por multiplicaciones de polinomios de *Laurent*, el esquema se conoce como esquema *Lifting*. Diferentes estudios al respecto han demostrado su efectividad en la reducción de la complejidad computacional de la transformada wavelet [11, 12, 3, 7, 13, 1, 14, 10].

2.2.1 Esquema *Lifting*

El esquema *lifting* es una forma de implementar la *DWT*, en el cual se remplazan las convoluciones por multiplicaciones de polinomios de *Laurent*. Para la obtención de h_p y l_p (Figura 2.2a) se aplican tres etapas:

- **Separación:** se toma la señal $\boldsymbol{x}_n$ de longitud n y se descompone en dos señales que contengan las muestras pares y las impares.
- **Predicción:** se calculan nuevas componentes impares entre las componentes impares y la combinación lineal del predictor $\widetilde{T}$ con las componentes pares.
- **Actualización:** se actualizan las componentes pares mediante combinación lineal del actualizador $\widetilde{S}$ con las nuevas componentes impares.

En la Figura 2.2 se muestra la descomposición y reconstrucción de la Transformada Wavelet Discreta 1D mediante el esquema *lifting*



(a) Descomposición. (b) Reconstrucción.

Figura 2.2: *DWT 1D* Esquema *Lifting* [12].

Para la reconstrucción (*IDWT*) se aplica primero la etapa de actualización, luego la de predicción, intercambiando los signos de los polinomios $\widetilde{S}$ y $\widetilde{T}$, finalmente se unen los resultados para obtener nuevamente el vector $\boldsymbol{x}$.

Para la aplicación de la *DWT 1D* por niveles se procede de igual forma como en el esquema presentado en la Figura 1.3. Para la *DWT 2D* se sigue el mismo esquema de la Figura 1.4.

Revisando la literatura [12, 1, 14, 10] y por medio de pruebas realizadas durante la presente investigación, se escogió la familia de filtros *CDF* (*Cohen-Daubechies-Feauveau*), debido a que con ellos se obtiene una mejor representación de los datos sísmicos por su mejor aproximación en tiempo y frecuencia [11, 3, 7, 13].

Para la etapa de transformación se trabajó con los filtros $CDF_{2.2}$ representados en la Ecuación 2.1 de los cuales se derivan los polinomios de predicción y actualización del esquema *lifting* mostrados en la Ecuación 2.2

Filtros de Análisis

$$\boldsymbol{CDF_{2.2}}: \begin{cases} \widetilde{H_p} = \frac{\sqrt{2}}{4}\Big(z^{-1} - 2 + z\Big) \\ \widetilde{L_p} = \frac{\sqrt{2}}{8}\Big(-z^{-2} + 2z^{-1} + 6 + 2z - z^2\Big) \end{cases} \tag{2.1}$$

Polinomios *Lifting*

$$\boldsymbol{CDF_{2.2}}: \begin{cases} \widetilde{T} = -\frac{1}{2}\Big(1 + z\Big) \\ \widetilde{S} = \frac{1}{4}\Big(z^{-1} + 1\Big) \end{cases} \tag{2.2}$$

Se trabajó con los polinomios *lifting* $CDF_{2,2}$ en vista que son computacionalmente más sencillos de implementar en arquitecturas de cómputo alternativas, además se ha demostrado su eficacia en mejorar la relación de compresión de los datos sísmicos. Con respecto a los niveles de la Transformada se evaluaron el primero y segundo de esta, tanto para la 1D como la 2D dado que niveles posteriores no presentan una mejora importante en la relación de compresión [10].

En la Figura 2.3, se muestra el histograma para el *Dataset 1*, de igual forma en las Figuras (2.4a, 2.4b), se aprecia el histograma del mismo conjunto de datos al cual se le aplicó la transformación correspondiente. En ellos se puede ver cómo se concentra la energía, lo cual mejora las etapas subsecuentes del proceso de compresión.

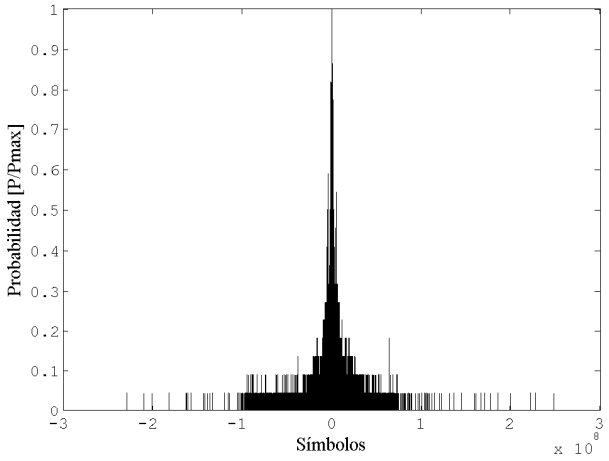
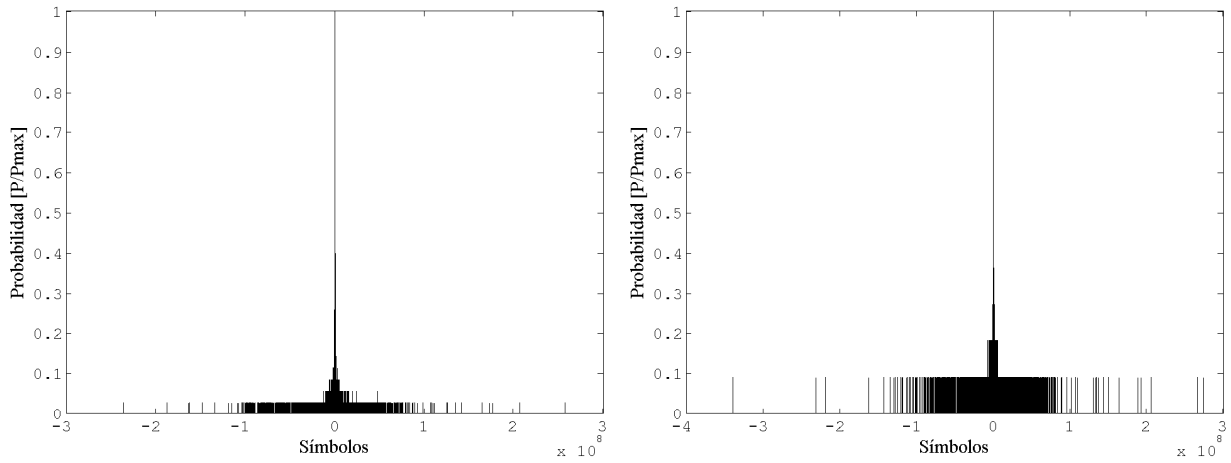


Figura 2.3: Histograma del *Dataset 1*.



(a) DWT 1D. (b) DWT 2D.

Figura 2.4: Histograma del *Dataset 1* con Transformación.

2.3 Cuantificación

Con la aplicación de la etapa de transformación, se reduce la entropía en los datos, pero no ha ocurrido compresión porque el número de coeficientes producto de la transformación es el mismo que el número de muestras presentes en los datos originales, la compresión se logra al reducir la precisión en los datos o quitando coeficientes producto de la transformación, esto se obtiene mediante la cuantificación, la cual aproxima los coeficientes en formato flotante a formato entero. Con esta etapa se reducen el rango dinámico de los datos sísmicos.

Dentro de los diferentes tipos de cuantificación se resalta la cuantificación uniforme, para la compresión de datos sísmicos, debido a que ofrece mejores resultados en cuanto a *SNR* [3, 7, 40, 10].

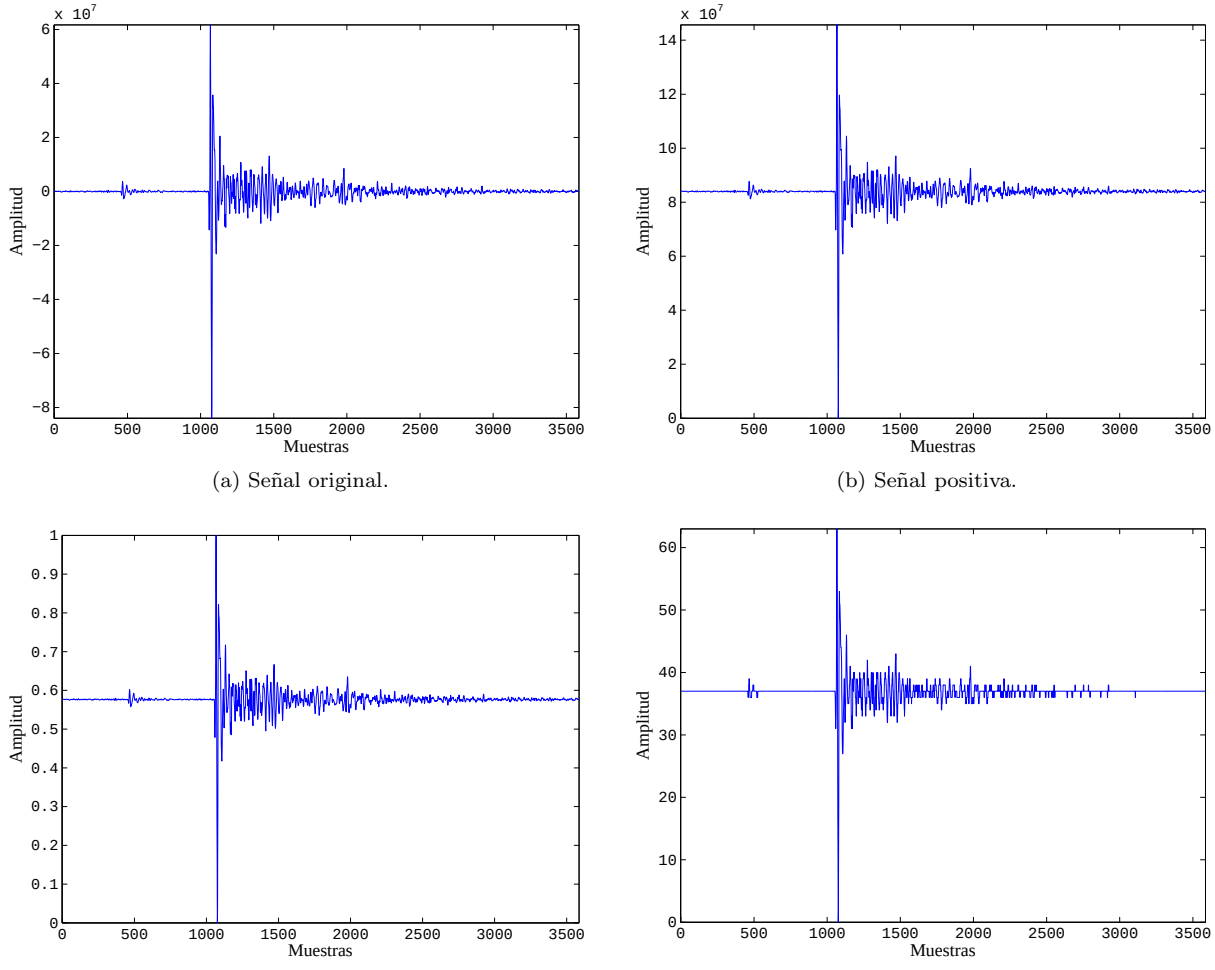
Por lo tanto para la etapa de cuantificación se empleó la cuantificación escalar uniforme, su implementación se realizó de la siguiente manera:

- Se aplica un valor de *dc* a la señal de entrada (Figura 2.5b).
- Luego se normaliza la amplitud a un rango de 0 a 1 (Figura 2.5c).
- Finalmente se lleva a valores cuantificados (Figura 2.5d).

Los pasos se resumen en las siguientes Ecuaciones 2.3, donde x es la señal de entrada, V_{pp} representa el valor pico a pico de la señal. A_+ es la señal de entrada desplazada a valores solo positivos, A_1 señal con amplitud normalizada, n es el número de bits utilizados para la cuantificación, finalmente $x_{\text{cuantificado}}$ es la señal de salida.

$$
\begin{aligned}
V_{pp} &= max(x) - min(x) \\
A_{+} &= x - min(x) \\
A_1 &= \frac{A_{+}}{V_{pp}} \\
x_{\text{cuantificado}} &= \text{round}\left[A_1 * (2^n - 1)\right]
\end{aligned}
\tag{2.3}
$$

En la Figura 2.5 se muestra el proceso de cuantificar una señal con la estrategia descrita anteriormente.



(a) Señal original.

(b) Señal positiva.

(c) Señal normalizada.

(d) Señal cuantificada.

Figura 2.5: Cuantificación de una traza del *Dataset 1* con 6 bits.

El proceso inverso de la cuantificación se realiza aplicando la siguiente fórmula:

$$
\tilde{x} = \left[x_{\text{cuantificado}} * \left(\frac{V_{pp}}{2^n - 1}\right)\right] + min(x) \tag{2.4}
$$

Con la etapa de transformación y cuantificación es más perceptible los efectos en la concentración de la energía de los datos a comprimir, para ello veamos el siguiente ejemplo, el cual compara un conjunto de datos cuantificados contra el mismo transformado y cuantificado.

La Figura 2.6 muestra los símbolos que representan el 80% de la probabilidad del conjunto de datos para el *Dataset 1* cuantificado a 12 bits.

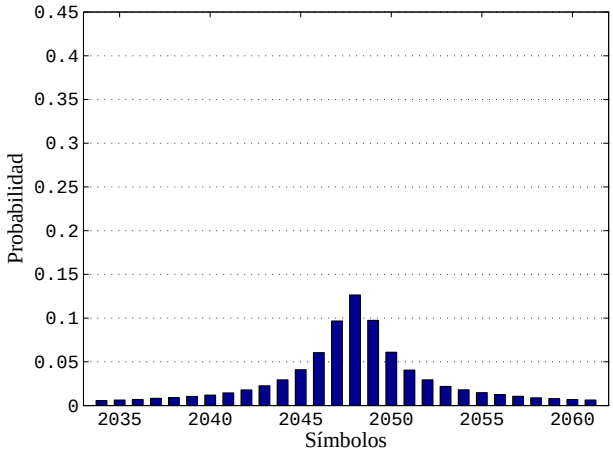
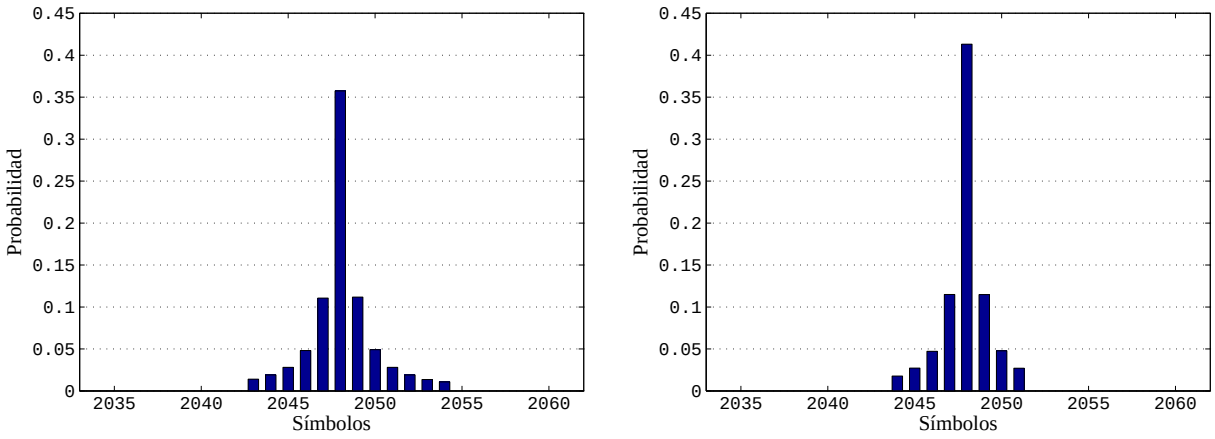


Figura 2.6: Histograma del *Dataset 1* Cuantificado a 12 bits.

Las Figuras (2.7a, 2.7b) muestran el efecto de Transformar y luego Cuantificar los datos, para ello se gráfico los símbolos, donde se encuentra el 80% de la probabilidad para cada una de las transformadas.



(a) DWT 1D.

(b) DWT 2D.

Figura 2.7: Histograma del *Dataset 1* con Transformación mas Cuantificación de 12 bits.

2.4 Codificación

Para el estudio del rendimiento computacional y la ganancia en compresión se tuvieron en cuenta tres algoritmos de codificación (Huffman, Aritmética, Tunstall): los dos primeros métodos de codificación se basan en la asignación de códigos de longitud variable, es decir, los códigos más cortos son asociados a los símbolos más frecuentes y los códigos largos a los menos frecuentes. Estas técnicas son conocidas como VLC por sus siglas en inglés, las cuales ofrecen una alta relación de compresión, pero presentan problemas para paralelar la decodificación, debido a la misma longitud variable de los códigos generados. Por ello, se exploró también el algoritmo codificación Tunstall, el cual genera códigos de longitud fija, a partir de secuencias de símbolos de longitud variables.

A continuación, se explica en mayor detalle cada uno de los algoritmos mediante un ejemplo sencillo, con el fin de entender las implementaciones realizadas en el presente proyecto.

Para todos los casos usaremos una secuencia $\boldsymbol{X}$, mostrada en la Ecuación 2.5, la cual proviene de un alfabeto $\Sigma = \{A, B, C\}$ de cardinalidad $\varsigma = 3$.

$$\boldsymbol{X} = \{A, B, C, A, B, A\} \tag{2.5}$$

La Tabla 2.2 muestra los símbolos, su frecuencia y probabilidad para la cadena $\boldsymbol{X}$ (Ecuación 2.5)

Tabla 2.2: Probabilidad de Símbolo para el ejemplo.

Símbolo	Frecuencia	Probabilidad
A	3	1/2
B	2	1/3
C	1	1/6

2.4.1 Algoritmo Huffman

La codificación Huffman es un algoritmo de compresión de datos sin pérdidas, que asigna palabras de código más cortas, para los símbolos más frecuentes y palabras de código más largas, para los símbolos de menor frecuencia. El algoritmo utiliza códigos prefijo [5], es decir el código que representa un símbolo en particular no es prefijo de cualquier otro código.

El algoritmo se desarrolla mediante la creación de un árbol binario (Figura 2.8), con el propósito de obtener el diccionario con el cual se codificarán los datos de la cadena analizada. Para ello, se siguen los siguientes pasos:

- Se calculan las frecuencias de aparición de cada símbolo y su probabilidad como se muestra en la Tabla 2.2
- Los símbolos de los datos se disponen en orden descendente, de acuerdo con su frecuencia (o probabilidad) de aparición. Estos símbolos se escriben de manera ordenada en los nodos inferiores, es decir, las hojas del árbol.

- Los nodos correspondientes a los dos símbolos de menor frecuencia, se agrupan con el fin de crear un nuevo nodo. La frecuencia del nuevo nodo es la suma de las frecuencias de los nodos agrupados.
- El paso anterior se repite para todos los nodos, hasta que queda un sólo nodo, es decir, la raíz del árbol. En este nodo, la frecuencia de ocurrencia debe ser el total de datos presentes en la cadena.

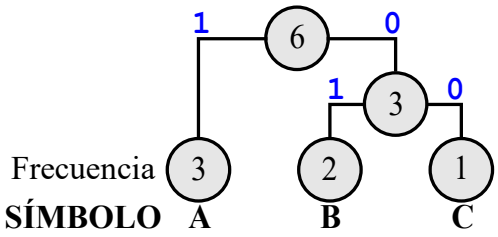


Figura 2.8: Árbol Huffman del ejemplo.

Una vez construido el árbol binario, los bits '1' y '0' se asigna a cada rama (en azul como se muestra en la Figura 2.8). Las palabras de código para cada símbolo se obtienen agrupando estos bits desde la raíz hasta cada hoja. El diccionario consta de los símbolos y su respectiva palabra de código. En la Tabla 2.3 se muestra el diccionario para el árbol binario de la Figura 2.8.

Tabla 2.3: Diccionario Huffman del ejemplo.

Símbolo	Código
A	1
B	01
C	00

La compresión se consigue mediante la sustitución de cada símbolo presente en la cadena de datos con la palabra de código Huffman correspondiente. Como resultado, se obtiene un flujo de bits que contiene los datos codificados. Como se aprecia a continuación:

$$X_H = \{1\ 01\ 00\ 1\ 01\ 1\} \tag{2.6}$$

La longitud de la cadena $\boldsymbol{X}$(Ecuación 2.5) representada mediante la codificación Huffman dada por X_H es de 9 bits.

Para el proceso de decodificación es necesario contar con el diccionario, el cual está compuesto por los símbolos con su código correspondiente. La decodificación se realiza mediante la comparación de la cadena de bits codificada con las palabras de códigos de longitud variable presentes en el diccionario, al encontrar una coincidencia se agrega en la cadena de salida el símbolo correspondiente a esta coincidencia, se descartan los bits encontrados en la trama de bits, se procede de este modo hasta el final de la cadena de bits.

2.4.2 Algoritmo Aritmética

La idea de la codificación aritmética es representar una secuencia de símbolos mediante un intervalo al que se le asigna un código [20]. Para la codificación Aritmética se siguen los siguientes pasos:

- Se calculan las frecuencias de aparición de cada símbolo y su probabilidad como se muestra en la Tabla 2.2

- Se divide el intervalo unidad $[0, 1)$ en los intervalos para los símbolos mediante la función de distribución de probabilidad (Ecuación 2.7)[5], para ello el alfabeto $A = \{a1, ..., aN\}$, lo representamos mediante $A_i \longleftrightarrow i$, por lo tanto $P(a1) = P(i)$

$$F_x(i) = \sum_{k=1}^{i} P(X = k) \tag{2.7}$$

donde X represente la variable aleatoria de la probabilidad del alfabeto fuente, quedando dichos intervalos de la siguiente manera: $I(A) = [0, 1/2)$, $I(B) = [1/2, 5/6)$, $I(C) = [5/6, 1)$. En la Figura 2.9 se muestran los intervalos iniciales.

Figura 2.9: Intervalos iniciales.

- Para identificar cada observación de un símbolo en la secuencia se procede a subdividir el intervalo inmediatamente anterior. Para ello, se calcula nuevamente los límites de los intervalos mediante las siguientes ecuaciones, donde $[l_i, h_i)$ representan el límite inferior y superior del intervalo para los símbolos donde $i = \{1, ..., m\}$ representa la secuencia a codificar $Sx = \{x_1, ..., x_m\}$

$$l_0 = 0 \qquad h_0 = 1$$
$$l_i = l_{i-1} + (h_{i-1} - l_{i-1})F_x(x_i) \qquad h_i = l_{i-1} + (h_{i-1} - l_{i-1})F_x(x_i - 1)$$

Con estas ecuaciones y las observaciones de la cadena a codificar se busca la etiqueta final, para poder codificar la secuencia siguiendo el proceso de subdividir los intervalos en las mismas proporciones dadas por la probabilidad de los símbolos como se muestran siguiendo la secuencia en las figuras:

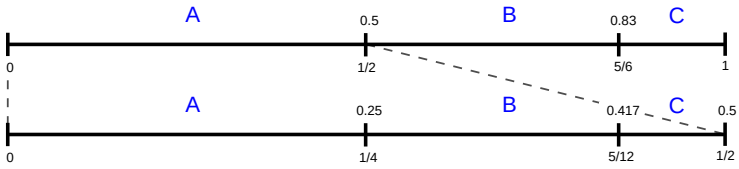


Figura 2.10: Observación de A.

Al encontrar en la secuencia el símbolo A se subdivide el intervalo de este [0, 1/2) en proporción con las probabilidades del alfabeto como se observa en la Figura 2.10.

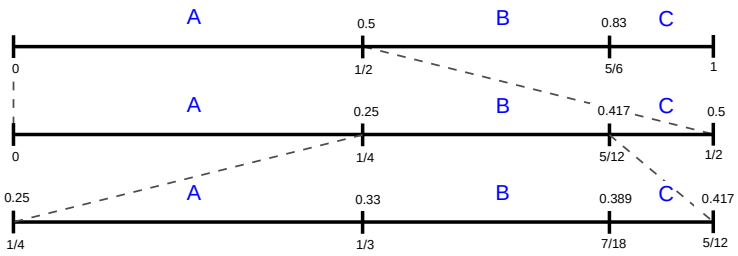


Figura 2.11: Observación de B.

El segundo símbolo en la secuencia corresponde a B, presente en el intervalo [1/4, 5/12), el cual es subdividido nuevamente como se aprecia en la Figura 2.11.

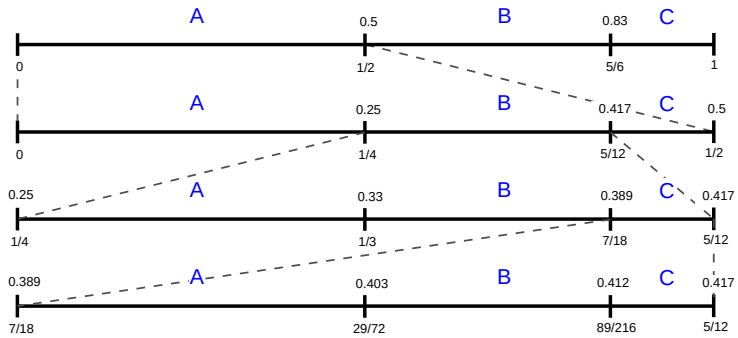


Figura 2.12: Observación de C.

El tercer símbolo es C correspondiente al intervalo [7/18, 5/12) mostrado en la Figura 2.12.

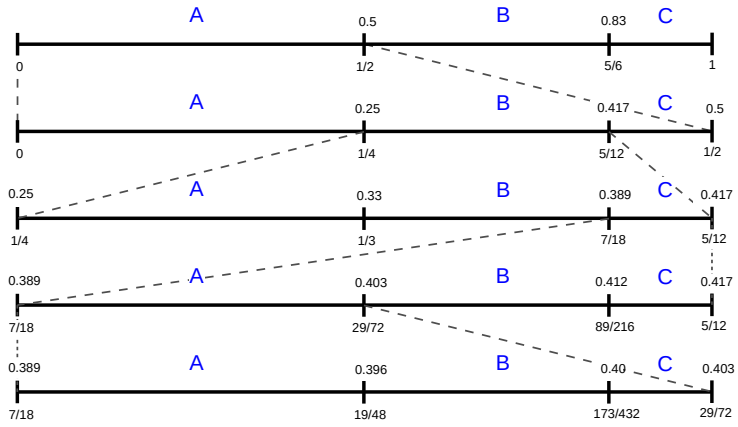


Figura 2.13: Observación de A.

Para el cuarto símbolo A el intervalo es [7/18, 29/72), Figura 2.13.

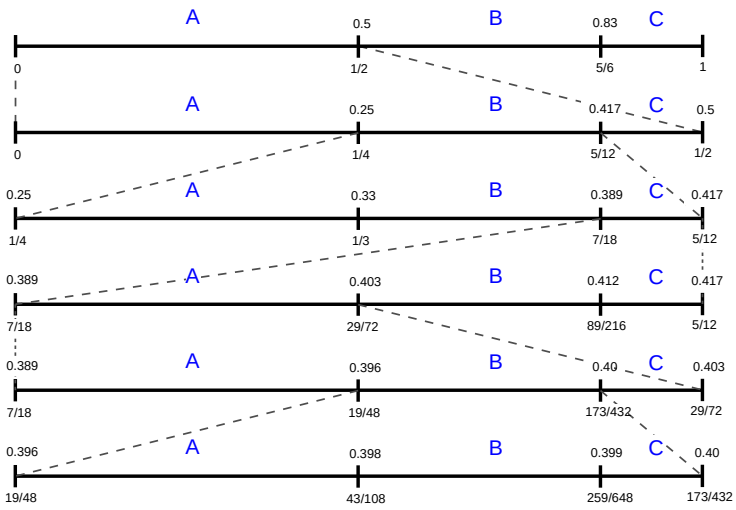


Figura 2.14: Observación de B.

Seguimos este proceso para los símbolos *B, A* de la secuencia Figuras(2.14, 2.15).

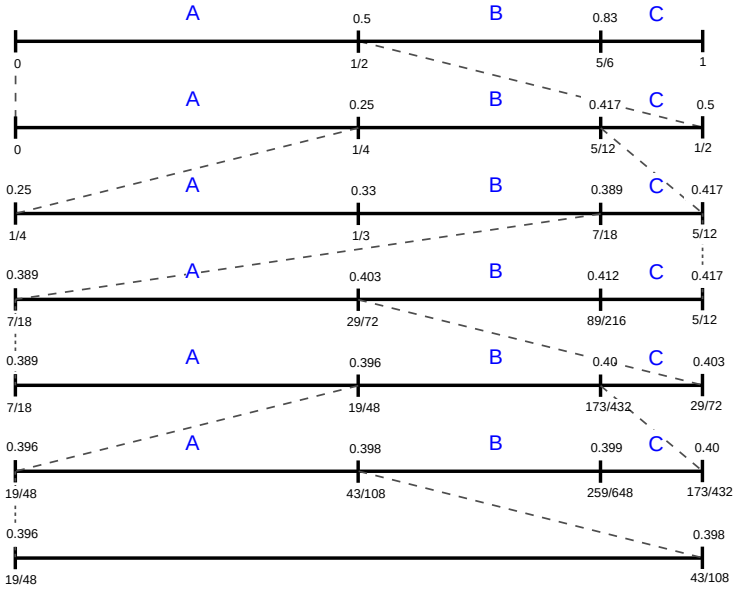


Figura 2.15: Observación final.

- Finalmente después de realizar las observaciones de todos los símbolos presentes en la secuencia se llega a un intervalo $[l_m = 19/48, h_m = 43/108)$, sobre el cual se elige un valor en éste, para ser la etiqueta, con la cual codificaremos la secuencia, para ello se toma el valor medio de este $\frac{l_m + h_m}{2}$, la etiqueta es $T = 343/864 \approx 0.397$.

Una vez obtenida la etiqueta de codificación se debe trasmitir para su posterior decodificación. Debido a que el valor de la etiqueta es una fracción, se requiere trabajar en formato de coma flotante, lo cual limita

la relación de compresión.

Para resolver este problema se busca la menor representación binaria de la etiqueta, que permita distinguir el intervalo origen. Para ello, se recurre a la siguiente ecuación $n = \left\lceil log_2 \left(\frac{1}{h_m - l_m} \right) \right\rceil$, para el ejemplo siguiendo la formula $n = \lceil log_2 (432) \rceil = \lceil 8.755 \rceil$ se aproxima hacia arriba tenemos que $n = 9$.

La etiqueta nueva será dada por $T_n = round(T * 2^n)$, entonces $T_n = 203$ expresado en binario $T_n = 011001011$. Por lo tanto, la cadena $\boldsymbol{X}$(Ecuación 2.5) codificada mediante Aritmética queda expresada por:

$$X_A = \{011001011\} \tag{2.8}$$

Para la codificación Aritmética la cadena de salida X_A tiene una longitud de 9 bits.

Para la decodificación es necesario tener la etiqueta, el alfabeto y su probabilidad. Los pasos son similares a la codificación, se ubica la etiqueta en los intervalos iniciales (Figura 2.9), se determina en cuál de estos intervalos se encuentra la etiqueta, para lo cual éste sería el primer dato decodificado. Luego éste intervalo se subdivide según la probabilidad del alfabeto. Nuevamente se analiza en qué intervalo se encuentra la etiqueta y éste será el siguiente dato decodificado. Se subdivide su intervalo tal como en la codificación, hasta llegar a la longitud de la cadena [4].

2.4.3 Algoritmo Tunstall

El algoritmo sigue los siguientes pasos para codificar la cadena del ejemplo $\boldsymbol{X}$(Ecuación 2.5), recordemos la ecuación que rige el árbol de Tunstall viene dada por:

$$N + k(N - 1) \leq 2^n \tag{2.9}$$

donde $\boldsymbol{N}$ son las ramas principales del árbol correspondiente a la cardinaliadad del alfabeto ($\boldsymbol{N} = 3$), $\boldsymbol{k}$ representa el número de repeticiones de dichas ramas en el árbol y $\boldsymbol{n}$ es el número de bits necesario para generar los códigos.

- El primer paso es obtener las probabilidades de ocurrencia para cada símbolo representadas en la Tabla 2.2
- Con los datos de probabilidad se construye el árbol con la raíz y las ramas principales como se aprecia en la Figura 2.16

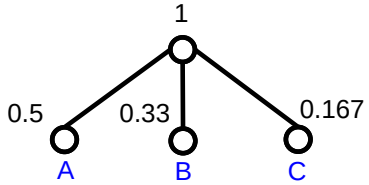


Figura 2.16: Árbol Tunstall del ejemplo $k = 0$.

- Una vez obtenido las ramas principales, se procede a remplazar el nodo con mayor probabilidad con N subcadenas de Sx como se aprecia en la Figura 2.17

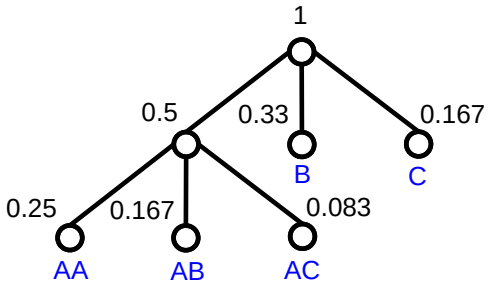


Figura 2.17: Árbol Tunstall del ejemplo $k = 1$.

- El paso anterior se repite cuanto niveles k del árbol se deseen Figura 2.18

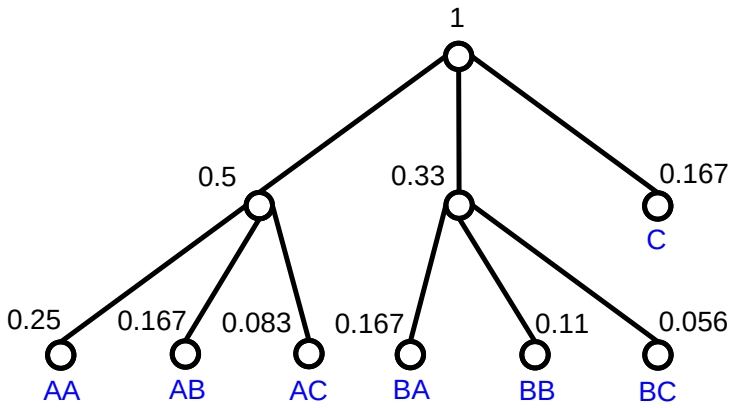


Figura 2.18: Árbol Tunstall del ejemplo $k = 2$.

- Con el árbol construido y el número de niveles k, se determinan la cantidad de bits necesarios para la codificación, luego, se procede a crear el diccionario leyendo las secuencias presentes en cada hoja del árbol y asignándole un código. A continuación se muestra el diccionario para $n = 2$, $n = 3$ y $n = 4$ en las Tablas (2.4, 2.5, 2.6).

Tabla 2.4: Diccionario Tunstall para $n = 2$.

Secuencia	Código	Probabilidad	Nivel k
A	00	0.5000	0
B	01	0.3333	0
C	10	0.1667	0

Tabla 2.5: Diccionario Tunstall para $n = 3$.

Secuencia	Código	Probabilidad	Nivel k
BA	000	0.1667	2
BB	001	0.1111	2
BC	010	0.0556	2
AC	011	0.0833	1
AB	100	0.1667	1
AA	101	0.2500	1
C	110	0.1667	0

Tabla 2.6: Diccionario Tunstall para $n = 4$.

Secuencia	Código	Probabilidad	Nivel k
ABC	0000	0.0278	5
ABB	0001	0.0556	5
ABA	0010	0.0833	5
BAC	0011	0.0278	4
BAB	0100	0.0556	4
BAA	0101	0.0833	4
AAC	0110	0.0417	3
AAB	0111	0.0833	3
AAA	1000	0.1250	3
CA	1001	0.0833	6
CB	1010	0.0556	6
CC	1011	0.0278	6
BC	1100	0.0556	2
BB	1101	0.1111	2
AC	1110	0.0833	1

Para la compresión se analiza la cadena de entrada y se reemplaza cada secuencia de símbolos que coincidan con las secuencias presentes en el diccionario, por su correspondiente código de ésta forma se crea la cadena de bits de codificación.

Una consideración importante, el código de mayor representación decimal se reserva, para cuando el símbolo presente en la cadena a codificar, no se encuentre en el diccionario; en tal caso, se envía este código seguido del valor binario del símbolo.

Como vemos en el ejemplo se generaron tres diccionarios, para el primero $n = 2$ (Tabla 2.4) la cadena codificada se representaría de forma tradicional como se muestra a continuación:

$$X_{T_{n=2}} = \{00\ 01\ 10\ 00\ 01\ 00\} \tag{2.10}$$

La cantidad de bits para la representación $X_{T_{n=2}}$ es de 12 bits.

Con el diccionario dos para $n = 3$ (Tabla 2.5) la codificación se expresa como:

$$X_{T_{n=3}} = \{100\ 110\ 100\ \mathbf{111}\ 1000001\} \tag{2.11}$$

En este caso, vemos cómo se introduce el código (**111**), el cual representa, que no se encontró el símbolo (A) en el diccionario; por ende, seguido al código (**111**), se introduce el código ASCII de la letra A, que corresponde a 65 en decimal, en binario (1000001). El número de bits de la representación $X_{T_{n=3}}$ es de 19 bits.

Ahora para el diccionario tres con $n = 4$ (Tabla 2.6) la cadena codificada se representa como:

$$X_{T_{n=4}} = \{0000\ 0010\} \tag{2.12}$$

En este caso, la longitud de bits de la cadena de salida es de 8 bits.

Para la decodificación; es necesario contar con el diccionario o con los símbolos y su probabilidad para poderlo crear, además, se necesita conocer el número de bits empleado para la codificación, con ésta información, se procede a comparar cada n bits presentes en la cadena codificada con el diccionario, y si existe alguna coincidencia con esté se remplaza el código con la secuencia correspondiente.

2.4.4 Estimación de la relación de compresión

A continuación, se puede observar cómo sería la codificación de la cadena $\boldsymbol{X}$(Ecuación 2.5) de forma normal es decir mediante código ASCII la cual queda expresada por:

$$X_N = \{1000001\ \ 1000010\ \ 1000011\ \ 1000001\ \ 1000010\ \ 1000001\} \tag{2.13}$$

Para esta codificación la longitud sería de 42 bits. Para Huffman y Aritmética la longitud alcanzada fue de 9 bits, mientras que con codificación tradicional fue de 12 bits, pero la mejor obtenida con Tunstall es de 8 bits.

Si sólo se tuviera éste criterio para la relación de compresión, se estaría hablando que Tunstall es superior a todos los demás porque logra codificar la cadena $\boldsymbol{X}$(Ecuación 2.5) con menor número de bits.

He aquí la importancia de la estimación de la relación de compresión, ya que, estos datos comprimidos se deben transferir a otro dispositivo de procesamiento. Se necesita contar con toda la información para poderlos descomprimir, en esencia se necesita de la cadena codificada seguido del diccionario, para el método empleado. Por ende, para estimar la relación de compresión, se tiene en cuenta el tamaño del diccionario, incluido en la cantidad de los datos comprimidos de la Ecuación 1.3.

Empecemos por explicar cómo sería la codificación tradicional: para poder decodificar la trama comprimida, se necesita la cadena de bits codificada, es decir, los 12 bits de la codificación $X_{T_{n=2}}$ más el tamaño de su diccionario, compuesto por tres símbolos que se representan en 7 bits y sus tres códigos correspondientes con 2 bits, para un tamaño de diccionario de 27 bits. Entonces la relación de compresión para la codificación tradicional está dada por:

$$RC_T = \frac{42}{12 + 27} = 1.08 \tag{2.14}$$

De igual forma, en la codificación Huffman se tiene que la cadena X_H posee una longitud de 9 bits, el diccionario cuenta con tres símbolos representados en ASCII, más los códigos de longitud variable, como son códigos que varían su longitud según su probabilidad de aparición, se opta por reservar el tamaño de cada uno de ellos usando la longitud del código más largo, en este ejemplo es de dos bits, por lo tanto la relación de compresión con codificación Huffman para el ejemplo estada dada por:

$$RC_H = \frac{42}{9 + 3 * 7 + 3 * 2} = 1.17 \tag{2.15}$$

Para la codificación Aritmética, la cadena codificada X_A tiene una longitud de 9 bits, para poderla decodificar necesitamos transmitir: los símbolos, los cuales son tres, representados por 7 bits, cada uno y sus frecuencias de aparición, para ello se usa el tamaño en bits que pueda representar la frecuencia del símbolo más probable, en este ejemplo la mayor frecuencias es de tres; por lo tanto se necesitan dos bits por cada símbolo. Con estos datos podemos estimar la relación de compresión para la codificación Aritmética del ejemplo así:

$$RC_A = \frac{42}{9 + 3 * 7 + 3 * 2} = 1.17 \tag{2.16}$$

Sin embargo, para Tunstall la cadena codificada con menor longitud ($X_{T_{n=4}}$) es de 8 bits, el diccionario es un factor importante en este algoritmo, ya que, crece a medida que se logra menor longitud de cadena de salida, para éste caso, la estimación de la relación de compresión tiene en cuenta que se necesitan enviar subcadenas de símbolos (Secuencias), donde se cuenta la secuencia más larga en este caso de tres símbolos, dicha secuencia emplea 21 bits, para todas las 15 secuencias (Tabla 2.6), y el código de 4 bits para cada secuencia. Entonces la relación de compresión para Tunstall es de:

$$RC_{T_{n=4}} = \frac{42}{8 + 15 * 21 + 15 * 4} = 0.11 \tag{2.17}$$

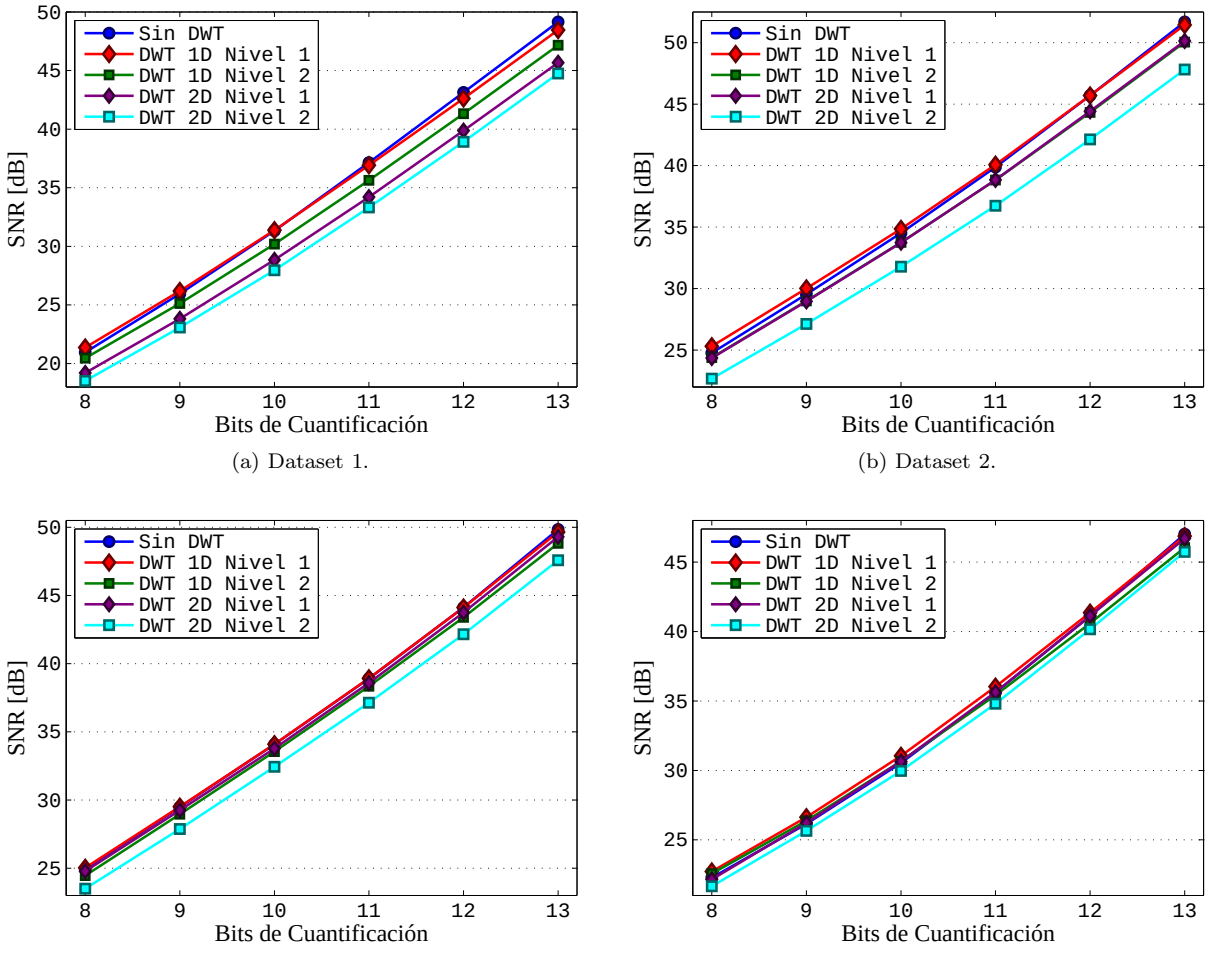
Esto, a manera de ejemplo recordando que la cadena usada para explicar los algoritmos es demasiado corta aun así las codificaciones Tradicional, Huffman y Aritmética logran obtener compresión ($RC > 1$), mientras que Tunstall no logra compresión debido al costo de transmitir el diccionario.

2.5 Ganancia en la compresión

Luego de entender y realizar los algoritmos correspondientes para la codificación mediante Huffman, Aritmética y Tunstall, se procedió a realizar la compresión de los *Datasets* usados en la presente presente investigación (sección 2.1). Primero se determinó la relación señal a ruido (*SNR*), para cada conjunto de datos a medida que se aumenta el número de bits de cuantificación desde 8 bits hasta 13 bits, para los datos con y sin transformación.

En la Figura 2.19, se muestra el comportamiento de la relación señal a ruido, a medida que se aumenta el número de bits de cuantificación desde 8 bits hasta 13 bits. Cada linea representa una forma diferente de la implementación de la transformación. Se probaron implementaciones deferentes: *DWT 1D* con uno y dos niveles de descomposición, *DWT 2D* con uno y dos niveles de descomposición y la linea azul representa los conjuntos de datos sin transformación.

Note que, se alcanza una relación señal a ruido cercana a los 40 dB, cuando los datos son cuantificados con 12 bits. Algunos trabajos sugieren que debe ser este nivel de *SNR* para que el proceso de compresión no se convierta en una fuente de ruido, en los procesos posteriores [10].



(a) Dataset 1.

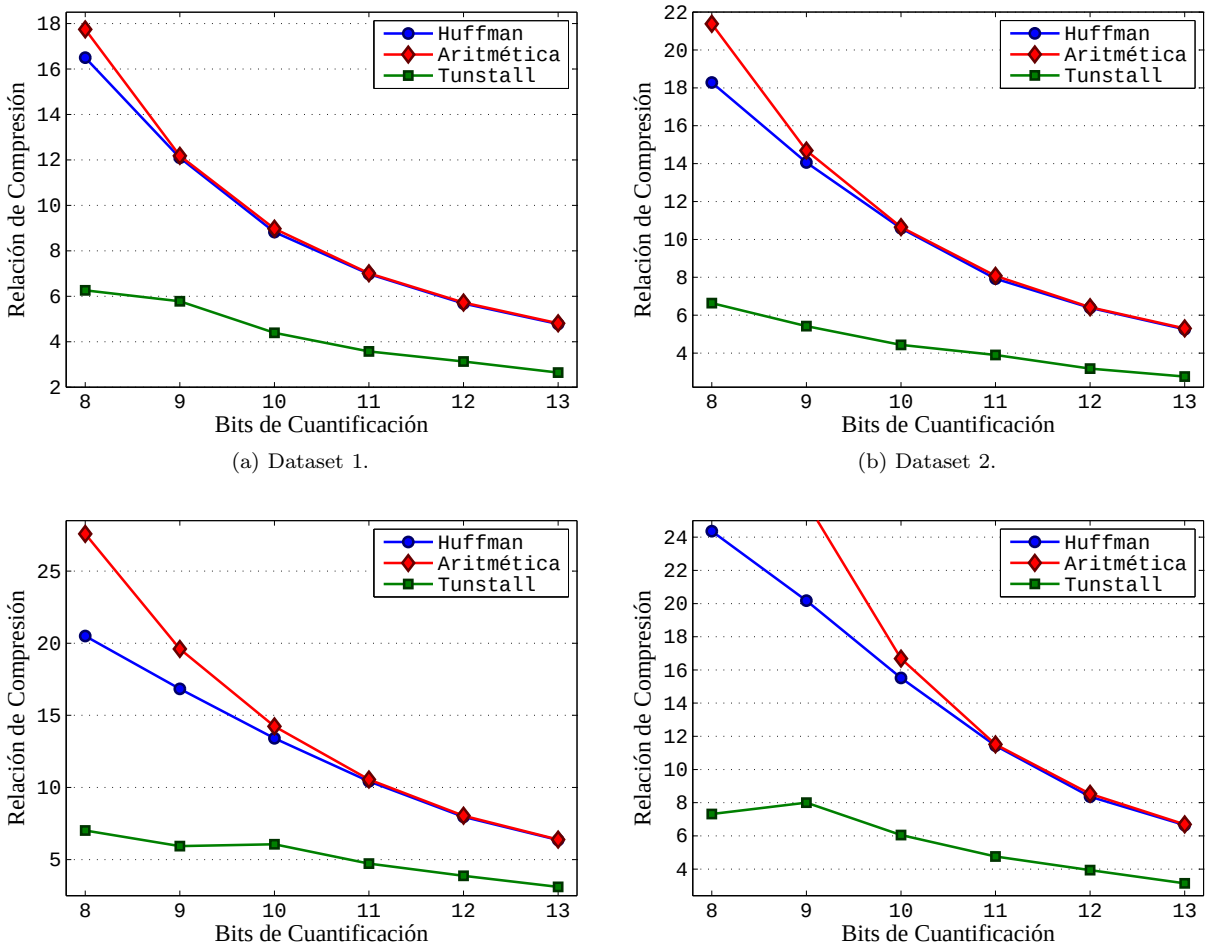
(b) Dataset 2.

(c) Dataset 3.

(d) Dataset 4.

Figura 2.19: *SNR* vs. *Bits de Cuantificación* para los datos sísmicos con y sin transformación.

La Figura 2.20 muestra los resultados en términos de la *RC* vs el número de bits de cuantificación, para los diferentes *Datasets*. Note que la técnica con mejor *RC* es la codificación Aritmética, sin embargo, el rendimiento de Huffman y Aritmética es similar cuando los datos son cuantificados con 10 bits o más.



(a) Dataset 1.

(b) Dataset 2.

(c) Dataset 3.

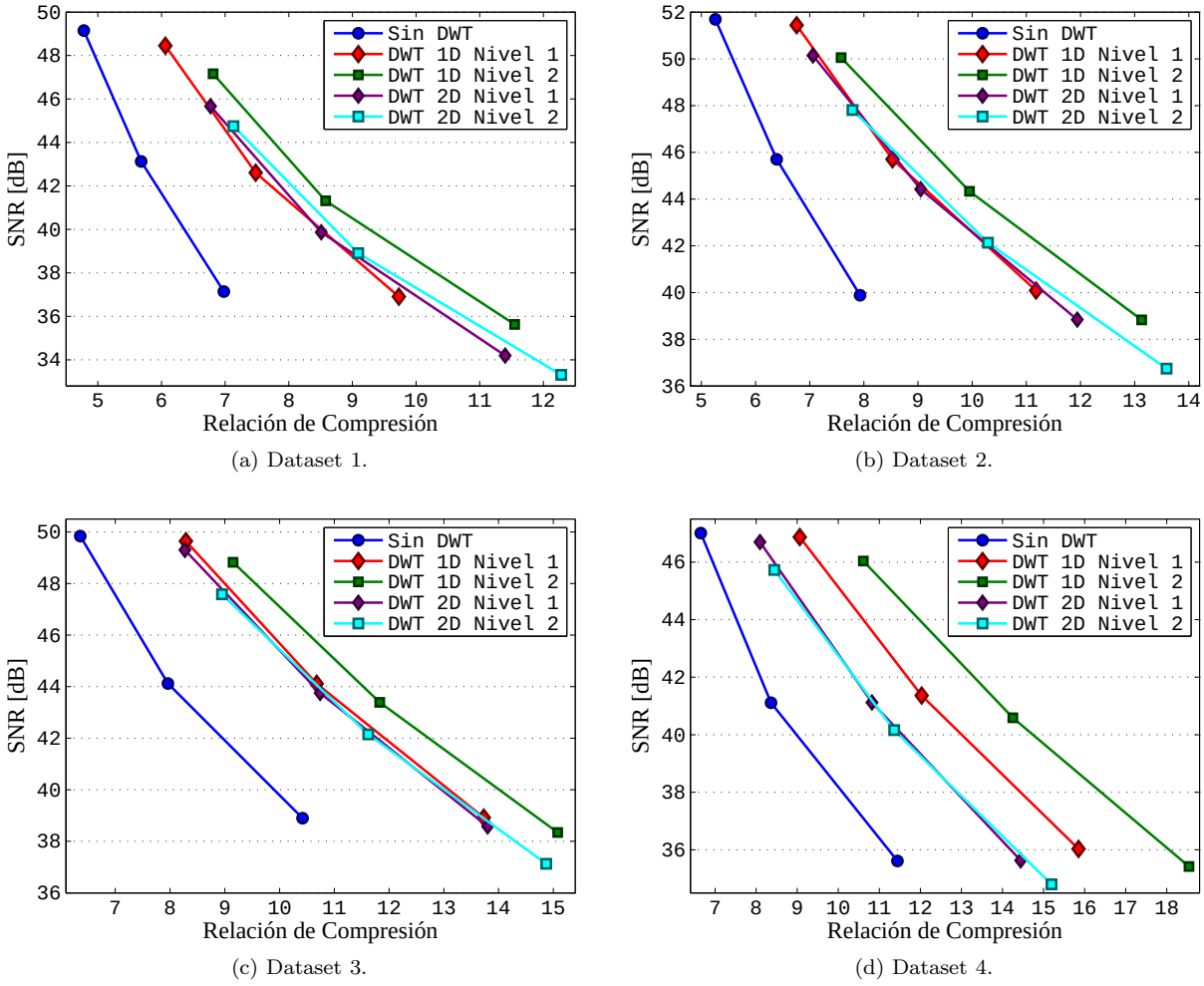
(d) Dataset 4.

Figura 2.20: Compresión de datos sísmicos sin transformación.

Con el fin de tener un mejor criterio de comparación se decidió analizar los datos en cuanto a *SNR* vs *RC* [10]. En el primer caso se uso la codificación Huffman (Figura 2.21), para el segundo la codificación Aritmética (Figura 2.22) y para el tercero la codificación Tunstall (Figura 2.23). Para cada uno de los casos a los *Datasets* se les aplico las diferentes transformaciones antes mencionadas, previo al proceso de codificación.

En el análisis se tomaron los bits de cuantificación del 11 al 13, debido a que con este número de bits se logra una *SNR* cercana a los 40 dB. Como era de esperarse el punto más arriba en las gráficas corresponde a la cuantificación con 13 bits y el punto más abajo a 11 bits de cuantificación, puesto que a medida que se disminuyen los bits de cuantificación la calidad de la reconstrucción se ve afectada.

Note que para todas las transformadas, al emplear 12 bits de cuantificación, se logra una *SNR* sobre los 40 dB, exceptuando a la *DWT 2D* Nivel 2, para el *Dataset* 1 (Figura 2.21a).



(a) Dataset 1.

(b) Dataset 2.

(c) Dataset 3.

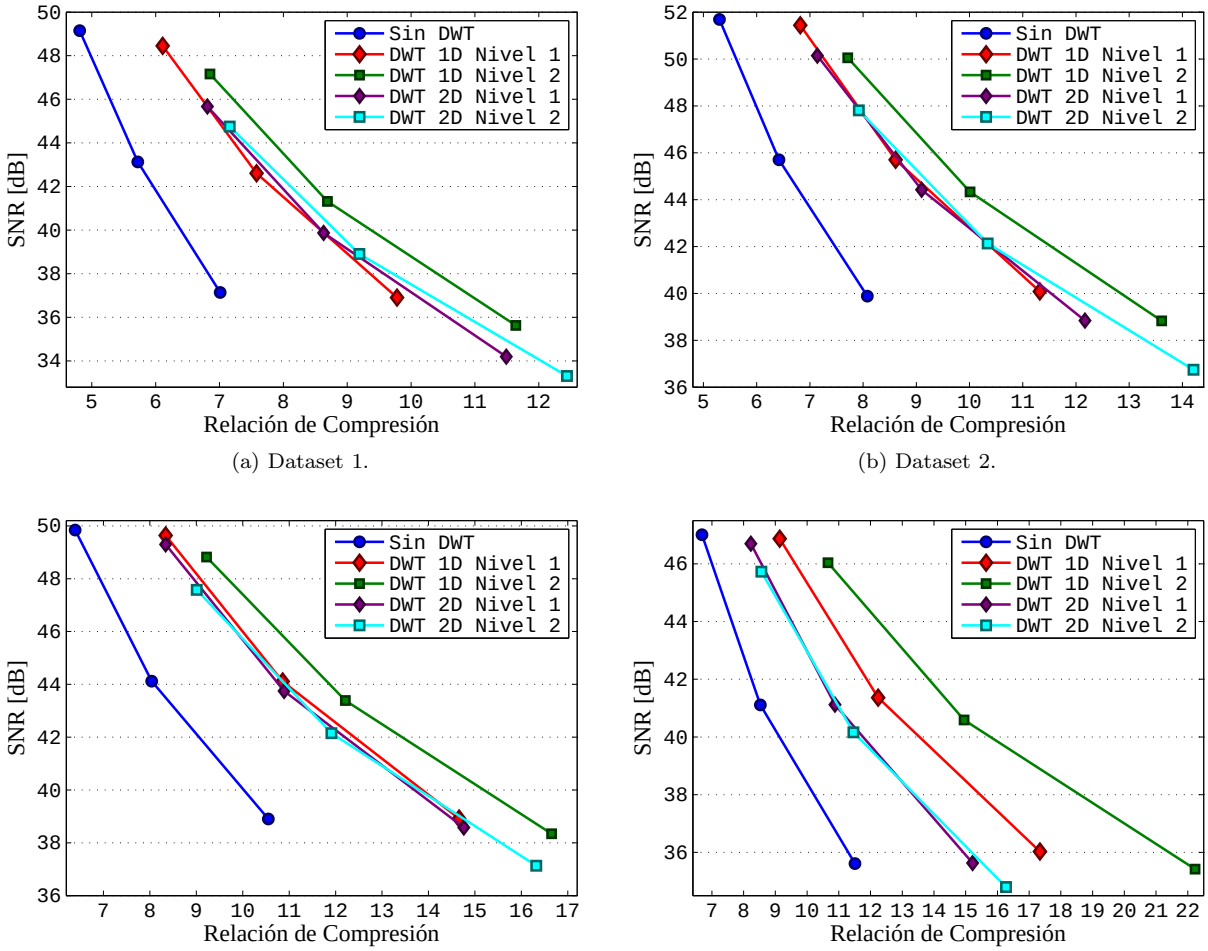
(d) Dataset 4.

Figura 2.21: Compresión de datos sísmicos con codificación *Huffman*.

Debido al uso de la transformada, se distingue un aumento promedio de 1.5 unidades en la RC al comparar la codificación Huffman para los *Datasets* con y sin transformación.

Al igual que Huffman, la codificación Aritmética (Figura 2.22) se ve beneficiada por la aplicación de la transformada Wavelet, sólo que al comparar bajo la misma SNR, la codificación Aritmética alcanza un 0.2 unidades de RC más que Huffman en los valores cercanos a los 40 dB, pero para valores inferiores como los obtenidos para 10 Bits o menos la codificación Aritmética incrementa la RC.

Con respecto a Tunstall, nuevamente la transformación mejora la relación de compresión en promedio en dos unidades al comparar los datos sin transformación. Para las Figuras (2.23a, 2.23b, 2.23c, 2.23d), los bits de cuantificación coinciden con los bits empleados para la codificación Tunstall, ya que, es la mejor configuración para la creación del diccionario Tunstall, recordando que a medida que éste número de bits aumenta, se disminuye la longitud de la cadena codificada, pero al costo de incrementar el tamaño del diccionario.



(a) Dataset 1.

(b) Dataset 2.

(c) Dataset 3.

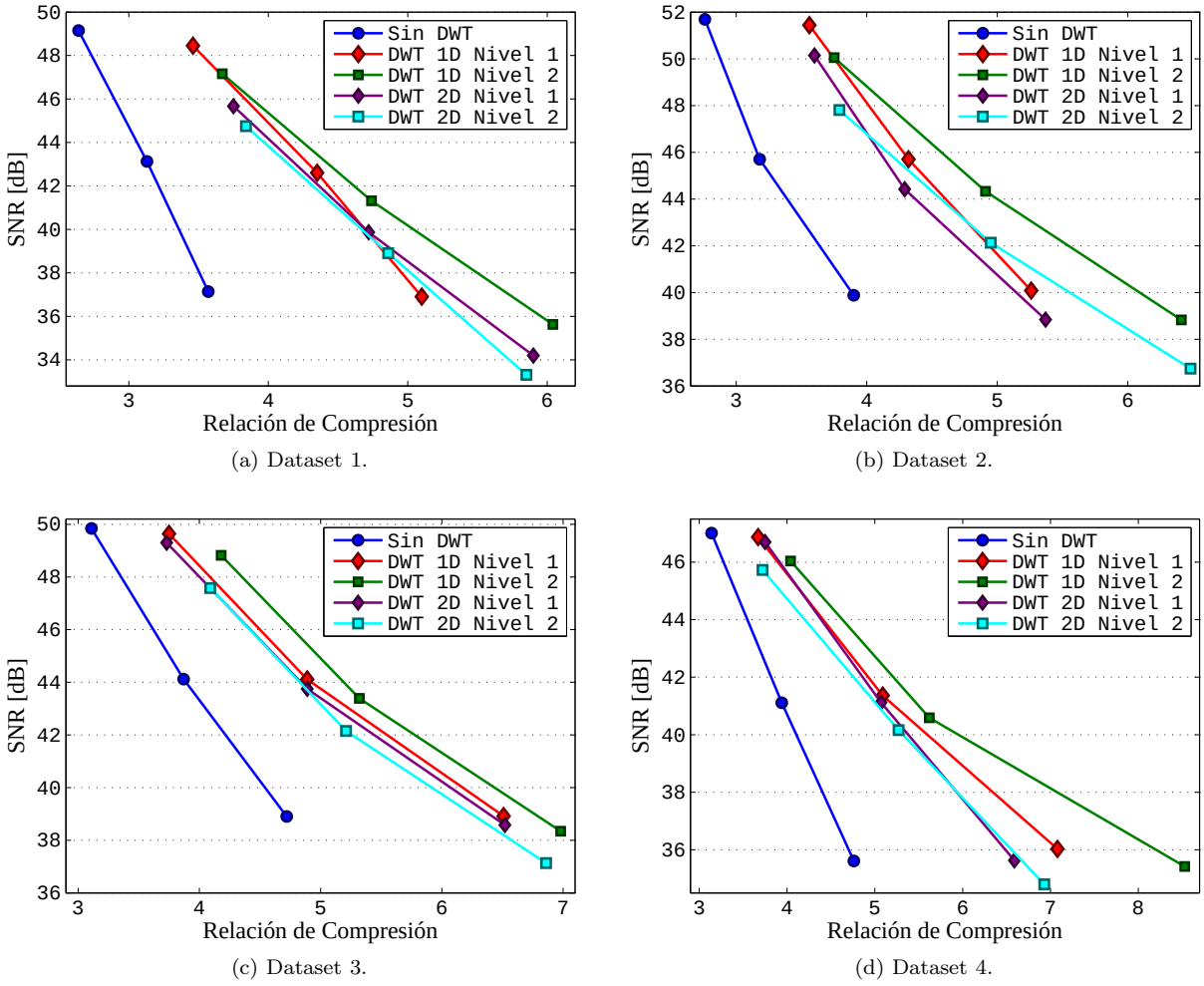
(d) Dataset 4.

Figura 2.22: Compresión de datos sísmicos con codificación *Aritmética*.

En la Tabla 2.7 se muestra los resultados de variar el número de bits de Tunstall, para el *Dataset* 1, donde se aplicó la *DWT 1D* Nivel 2, se cuantificó a 11 bits y se codificó con N bits de Tunstall desde 11 al 16.

En la tabla podemos apreciar cómo impacta el Diccionario en la codificación Tunstall, para ello se muestran en dos columnas la Relación de Compresión: la primera incluyendo el costo de enviar el diccionario y la segunada sin este costo. Los resultados son muy similares para las otros *Datasets*.

La tabla evidencia que para la mejor RC mediante Tunstall, los bits de cuantificación deben coincidir con los bits de codificación.



(a) Dataset 1.

(b) Dataset 2.

(c) Dataset 3.

(d) Dataset 4.

Figura 2.23: Compresión de datos sísmicos con codificación *Tunstall*.

Tabla 2.7: *RC* Tunstall *Dataset 1 + DWT 1D Nivel 2 + Cuantificación a 11 bits.*

Bits	Relación de Compresión	
Tunstall	Con Diccionario	Sin Diccionario
11	6.04	7.05
12	5.32	7.42
13	4.31	8.54
14	2.64	9.18
15	1.29	9.51
16	0.58	9.82

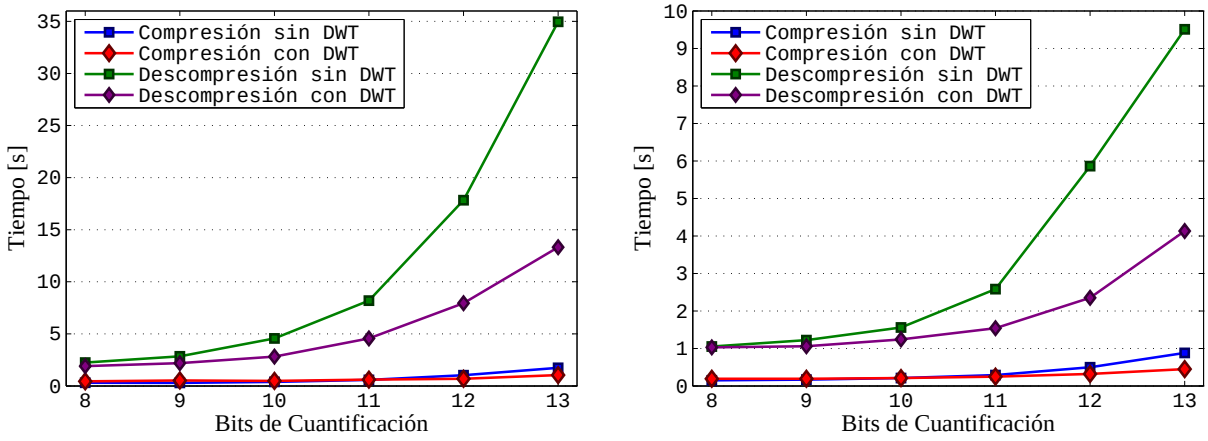
Los resultados obtenidos en cuanto a la ganancia de comprensión, determinan que el uso de la Transformada Wavelet favorece la relación de compresión para todos los casos (*DWT 1D, 2D* Niveles 1 y 2), cuando se combinan con las técnicas de codificación (Huffman, Aritmética y Tunstall). También se aprecia que la *DWT 1D Nivel 2*, presenta la mejor relación entre *SNR vs RC*. De igual forma podemos concluir que las codificaciones Huffman y Aritmética presentan un comportamiento muy similar en cuanto a la *RC*, siendo la codificación Aritmética la que ofrece mayor *RC*.

2.6 Rendimiento computacional

Para la evaluación del rendimiento computacional se tomaron los tiempos de compresión y descompresión, de los esquemas compuesto por las transformaciones Wavelet 1D y 2D con las codifiaciones Huffman, Aritmética y Tunstall pero para hacer más claros.

Para hacer mas claras las comparaciones, los resultados se muestran con los datos transformados mediante la Wavelet 1D nivel 2 y se compararon con los datos no transformados para los *Datasets* 1 y 4, debido a que los *Datasets* 2 y 3 presentan bastante similitud en cuanto a tiempos al *Dataset* 1.

En las Figuras (2.24, 2.25, 2.26), se muestran los tiempos de compresión, descompresión para las diferentes codificaciones al variar el número de bits de cuantificación de 8 a 13. Adicionalmente, estas figuras muestran como afecta a estos tiempos el empleo de la transformación wavelet. Como se puede apreciar en ellas, para todas las codificaciones el uso de la la transformada mejora el tiempo, tanto en la compresión como la descompresión.

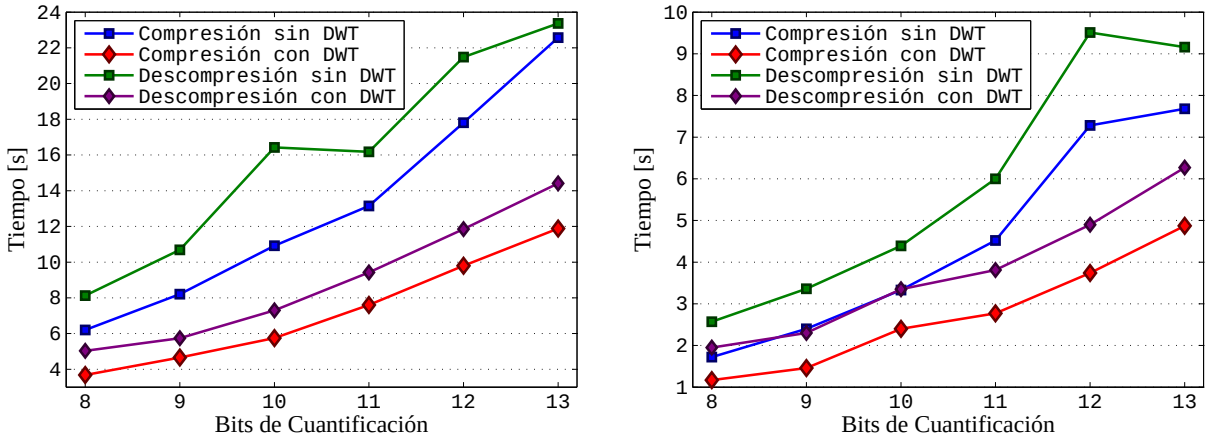


(a) Dataset 1.

(b) Dataset 4.

Figura 2.24: Tiempo de compresión y descompresión mediante codificación Huffman.

Con respecto a la codificación Aritmética, el costo computacional de la compresión y descompresión es bastante similar, siendo mayor el tiempo empleado para la descompresión. También se aprecia que el tiempo en la descompresión mediante la codificación Aritmética es casi el doble al empleado por el algoritmo Huffman, lo cual evidencia un mayor grado de complejidad en la codificación Aritmética.



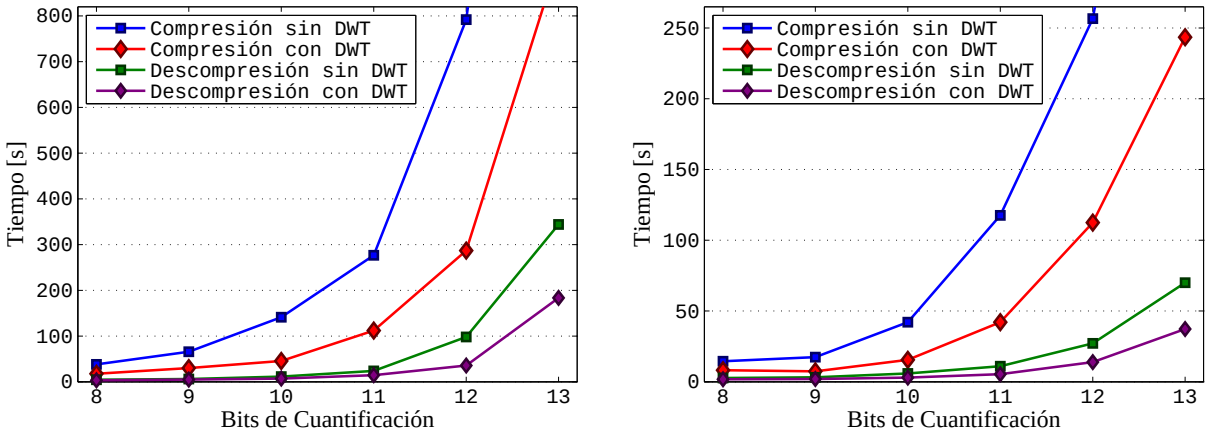
(a) Dataset 1.

(b) Dataset 4.

Figura 2.25: Tiempo de compresión y descompresión mediante codificación Aritmética.

Al comparar los tiempos de la codificación Huffman y la codificación Tunstall, se observa un comportamiento inverso, es decir, para Huffman es mucho más costoso computacionalmente realizar la descompresión que la compresión, caso contrario ocurre con Tunstall.

Los tiempos totales para la codificación Tuntall son mayores que en la codificación Huffman, lo cual evidencia el impacto que tiene el aumento del diccionario (longitud tres veces mayor) y la búsqueda en él, siendo la principal razón por la cual se incrementan sustancialmente los tiempos de compresión-descompresión.



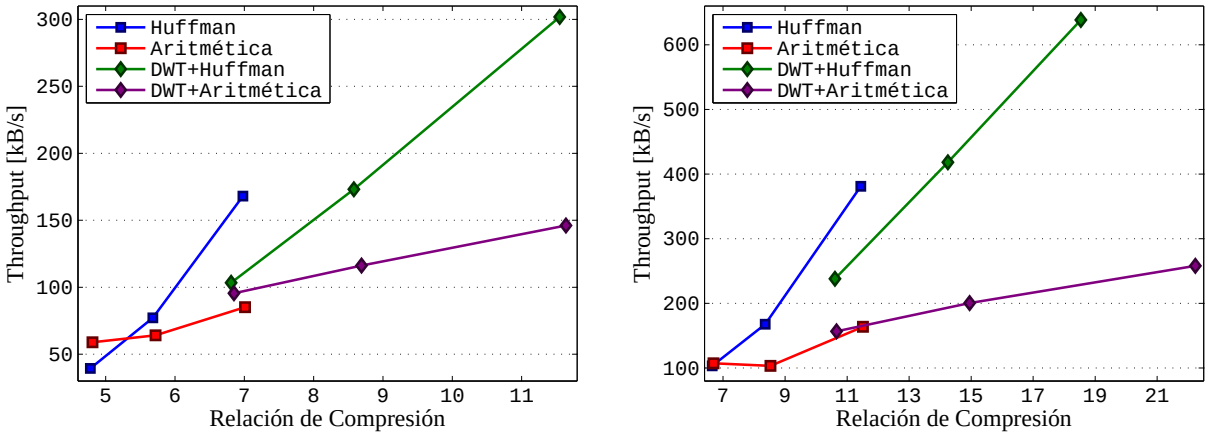
(a) Dataset 1.

(b) Dataset 4.

Figura 2.26: Tiempo de compresión y descompresión mediante codificación Tunstall.

En la Figuras (2.27a, 2.27b) se presenta el *Troughput* de la descompresión vs relación de compresión. En este caso se usó cuantificación de 11 a 13 bits. para Huffman y Aritmética, con y sin transformación, para

la cuantificación de 11 a 13 bits. Los puntos más a la derecha y hacia la parte superior corresponde al bit 11 y los puntos más abajo y a la izquierda son los cuantificados a 13 bits.



(a) Dataset 1.

(b) Dataset 4.

Figura 2.27: Comparación en términos de *Throughput* para la Descompresión.

Como se puede apreciar el algoritmo Huffman decodifica más datos por unidad de tiempo que el algoritmo Aritmética, ambos se ven favorecidos por el uso de la transformada.

2.7 Resultados generales en la etapa de codificación

Las Tablas (2.8, 2.9, 2.10), resumen los resultados, donde los datos se cuantificaron a 12 bits, ya que, con este número de bits los resultados se encuentran en el rango de los 40 dB.

Tabla 2.8: Relación de compresión y *SNR* para los *Datasets* con 12 bits de cuantificación.

Dataset	**Codificación**	**Sin DWT**		**DWT 1D Nivel 1**		**DWT 1D Nivel 2**		**DWT 2D Nivel 1**		**DWT 2D Nivel 2**	
		SNR[*dB*]	**RC**	**SNR[*dB*]**	**RC**	**SNR[*dB*]**	**RC**	**SNR[*dB*]**	**RC**	**SNR[*dB*]**	**RC**
	Huffman		5.68		7.48		8.58		8.51		9.09
1	Aritmetíca	43.13	5.72	42.61	7.58	41.32	8.69	39.87	8.63	38.91	9.19
	Tunstall		3.13		4.35		4.74		4.72		4.86
	Huffman		6.39		8.53		9.95		9.05		10.29
2	Aritmetíca	45.7	6.42	45.7	8.61	44.33	10.01	44.42	9.1	42.13	10.34
	Tunstall		3.18		4.32		4.91		4.29		4.95
	Huffman		7.96		10.68		11.83		10.74		11.62
3	Aritmetíca	44.12	8.04	44.11	10.86	43.39	12.21	43.75	10.89	42.15	11.91
	Tunstall		3.87		4.89		5.32		4.89		5.21
	Huffman		8.36		12.03		14.25		10.82		11.36
4	Aritmetíca	41.11	8.53	41.36	12.24	40.59	14.95	41.12	10.88	40.16	11.46
	Tunstall		3.94		5.09		5.62		5.07		5.27

De la Tabla 2.8 podemos distinguir con mayor claridad la cercanía en cuanto a *RC* de Huffman y Aritmética, donde Huffman está 0.15 unidades de *RC* en promedio por debajo de Aritmética. Otro punto importante es cómo la transformación mejora la *RC* en 2.6 unidades en promedio para cada uno de los *Datasets*.

Se aprecia como cada *Dataset* presenta una *SNR* y *RC* diferente donde las Transformadas *DWT 1D Nivel 2* y *DWT 2D Nivel 1* obtienen un mayor *RC*.

En la Tabla 2.9 se listan los tiempos empleados tanto en la compresión como la descompresión para los cuatro *Datasets* con y sin transformación. Para cada uno de los algoritmos de codificación, es de destacar que la Codificación en la etapa de Compresión representa en promedio para Huffman el 70%, para Aritmética el 95% y para Tunstall el 99%. En la etapa de Descompresión la decodificación Huffman, Aritmetica y Tunstall representan más del 95% de esta etapa.

Tabla 2.9: Tiempo de compresión y descompresión para los *Datasets* con 12 bits de cuantificación.

Dataset	**Codificación**	**Tiempo** $[s]$									
		Sin DWT		**DWT 1D Nivel 1**		**DWT 1D Nivel 2**		**DWT 2D Nivel 1**		**DWT 2D Nivel 2**	
		Compre	**Descom**	**Compre**	**Descom**	**Compre**	**Descom**	**Compre**	**Descom**	**Compre**	**Descom**
1	Huffman	1.03	17.82	0.88	10.59	0.67	7.94	0.6	7.82	0.62	6.89
	Aritmetíca	17.81	21.47	11.19	14.12	9.78	11.84	10.41	13.17	8.96	12.01
	Tunstall	791.83	98.04	348.46	46.16	286.87	35.55	327.51	43.57	281.58	30.39
2	Huffman	1.38	18.38	0.9	10.73	0.78	7.79	0.96	9.75	0.73	7.47
	Aritmetíca	16.49	18.99	10.31	12.85	8.41	10.85	10.17	12.79	8.63	11.46
	Tunstall	743.68	112.68	338.19	46.89	268.81	34.30	356.63	44.06	254.9	33.32
3	Huffman	0.77	10.43	0.66	6.63	0.56	5.32	0.64	6.68	0.6	5.7
	Aritmetíca	8.78	11.21	6.88	8.81	6.15	8.13	6.78	8.89	6.38	8.38
	Tunstall	267.71	37.28	192.25	27.81	178.76	25	184.90	27.62	185.15	30.61
4	Huffman	0.5	5.86	0.37	3.05	0.32	2.35	0.4	3.67	0.38	3.46
	Aritmetíca	7.28	9.51	4.38	5.7	3.74	4.9	4.84	6.26	4.59	5.94
	Tunstall	256.63	27.11	130.19	18.27	112.49	13.94	122.32	17.92	112.75	17.04

Para poder discernir mejor qué estrategia ofrece mejores resultandos en términos de tiempo de decodificacios, se usó la cantidad de datos decodificados por segundo (*Throughput*) (ver Tabla 2.10). Igualmente se muestra otra forma de medida de *Throughput* en la cual se relaciona la cantidad de símbolos decodificados por unidad de tiempo.

Tabla 2.10: *Throughput* de descompresión para los *Datasets* con 12 bits de cuantificación.

Dataset	**Codificación**	**Sin DWT**		**DWT 1D Nivel 1**		**DWT 1D Nivel 2**		**DWT 2D Nivel 1**		**DWT 2D Nivel 2**	
		Throughput		**Throughput**		**Throughput**		**Throughput**		**Throughput**	
		$[kB/s]$	$[S*10^3/s]$	$[kB/s]$	$[S*10^3/s]$	$[kB/s]$	$[S*10^3/s]$	$[kB/s]$	$[S*10^3/s]$	$[kB/s]$	$[S*10^3/s]$
1	Huffman	77.23	19.31	129.96	32.49	173.34	43.33	176	44	199.75	49.94
	Aritmetíca	64.1	16.03	97.47	24.37	116.24	29.06	104.5	26.12	114.6	28.65
	Tunstall	14.04	3.51	29.82	7.45	38.71	9.68	31.59	7.9	45.29	11.32
2	Huffman	74.88	18.72	128.27	32.07	176.68	44.17	141.16	35.29	184.24	46.06
	Aritmetíca	72.47	18.12	107.11	26.78	126.85	31.71	107.61	26.90	120.1	30.02
	Tunstall	12.21	3.05	29.35	7.34	40.13	10.03	31.24	7.81	41.31	10.33
3	Huffman	131.96	32.99	207.59	51.9	258.7	64.67	206.03	51.51	241.46	60.36
	Aritmetíca	122.77	30.69	156.22	39.05	169.29	42.32	154.81	38.7	164.24	41.06
	Tunstall	36.92	9.23	49.49	12.37	55.05	13.76	49.83	12.46	44.96	11.24
4	Huffman	167.75	41.94	322.3	80.58	418.3	104.58	267.85	66.96	284.1	71.03
	Aritmetíca	103.36	25.84	172.46	43.12	200.61	50.16	157.03	39.26	165.49	41.37
	Tunstall	36.26	9.07	53.8	13.45	70.52	17.63	54.85	13.71	57.69	14.42

Capítulo 3

Descompresión en GPU

En el capítulo se discuten las estrategias usadas para la implementación de la descompresión sobre GPU, se discuten algunas características de ésta arquitectura, se presentan las técnicas usadas para la codificación y cómo se adecuó para una implementación paralela, también se muestran los resultados obtenidos. Las implementaciones realizadas se compilaron mediante CUDA 6.5 [45].

3.1 Fundamentos de la GPU

Las GPU son procesadores de múltiples núcleos de alto rendimiento, diseñados inicialmente para el procesamiento en paralelo de gráficos para computadora. Actualmente las GPUs se están usando para diferentes aplicaciones aparte del manejo visual para la cual fueron diseñadas.

Hoy en día las industrias creadoras de las GPUs han mejorado estos dispositivos permitiendo la compatibilidad con entornos de programación similares a C y para tal fin se rediseñó la arquitectura interna de la GPU dotándola con capacidades para que los programadores puedan usar gran parte de los recursos, en especial, el procesamiento de operaciones en punto flotante.

Cuentan con Arquitecturas *Many-Core* con una gran cantidad de núcleos que realizan operaciones sobre múltiples datos presentando un alto nivel de paralelado, mediante la estrategia *SIMD* (*Single Instruction, Multiple Data*) ésta estrategia se basa un ejecutar una instrucción sobre muchos datos, principal característica de las GPU, que ejecuta la misma instrucción mediante una gran cantidad de procesadores a los datos de entrada.

Dentro de las GPUs a disposición en el momento de desarrollar ésta investigación se contó con cuatro dispositivos (GTX 760M, GTX 660, Quadro K4200 y Tesla K40) todas con arquitectura Kepler [46, 47]. Algunas de las características principales de estas GPUs se muestran en la Tabla 3.1.

La parte de la GPU que se encarga de la ejecución en paralelo de los algoritmos o *kernels* se llama *Streaming Multiprocessor* (SM). Dentro de cada SM están los registros, memorias, funciones especiales, entre otros. Su unidad básica de ejecución se llama *Warp*, el cual consta de 32 hilos consecutivos ejecutando en paralelo la misma instrucción.

Tabla 3.1: Especificaciones de la GPU.

GPU	Núcleos CUDA	Capacidad CUDA	Streaming Multiprocessor	Reloj Base $[MHz]$	Bus de Memoria $[bits]$	Reloj de Memoria $[MHz]$	Memoria $[GB]$
GTX 760M	768	3.0	4	719	128	2004	2
GTX 660	960	3.0	5	1110	192	3004	2
Quadro K4200	1344	3.0	7	784	256	2700	4
Tesla K40	2880	3.5	15	745	384	3004	12

Es importante tener en cuenta la capacidad de cómputo de la GPU y el máximo número de hilos permitidos en el SM cuando se desea ejecutar un *kernel*. Esto es debido a que las GPUs tienen mejor rendimiento cuando varios bloques de hilos se ejecutan en un mismo SM. Para generar varios bloques dentro de un mismo SM, todos los hilos disponibles en el bloque no deben ser asignados [48].

Tabla 3.2: Porcentaje de uso de la GPU [48].

Hilos por	Capacidad de Cómputo						
Bloque	1.0	1.1	1.2	1.3	2.0	2.1	3.0
64	67	67	50	50	33	33	50
96	100	100	75	75	50	50	75
128	100	100	100	100	67	67	100
192	100	100	94	94	100	100	94
256	100	100	100	100	100	100	100
384	100	100	75	75	100	100	94
512	67	67	100	100	100	100	100
768	N/A	N/A	N/A	N/A	100	100	75
1024	N/A	N/A	N/A	N/A	67	67	100

En la Tabla 3.2, se presenta el porcentaje de utilización de una GPU teniendo en cuenta su capacidad de cómputo y el número de hilos por bloque. Como se puede observar, el porcentaje de utilización depende de la capacidad de cómputo de la GPU y éste va incrementando de tecnología a tecnología. Sin embargo, utilizando 256 hilos por bloque es la única configuración que logra el 100% de utilización del SM para todas las tecnologías.

Otro punto que afecta el rendimiento de la GPU es la divergencia, la cual se presenta al ejecutar algoritmos que contenga bucles condicionales, de tal forma que algunos hilos dentro del *warp* estarán inactivos cuando la condición del bucle no se cumpla, limitando la velocidad a la que se ejecuta este *warp* [47].

3.2 Implementación de la Codificación Huffman

Para comprender mejor las estrategias usadas para la Codificación Huffman cuya decodificación será implementada sobre una GPU, veremos el siguiente ejemplo (Figura 3.1).

Con los datos de frecuencia se construye el árbol Huffman mostrado en la Figura 3.2.

DBAEEBAEAAEADECDBCEACDABEBAEDEAABABECEAD

Símbolo	A	E	B	D	C
Frecuencia	12	11	7	6	4

Figura 3.1: Datos de ejemplo y su tabla de frecuencias.

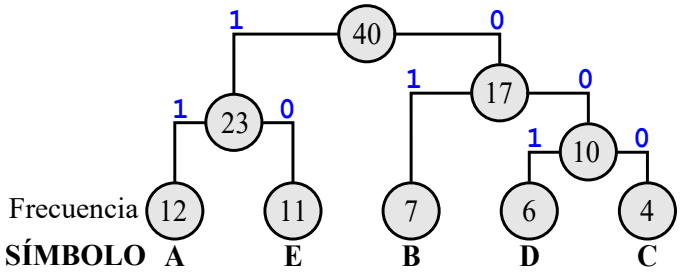


Figura 3.2: Árbol Huffman del ejemplo.

Además del algoritmo de Huffman en sí, la implementación de la codificación implica cómo el diccionario y los datos codificados se guardan. El primer paso fue agregar una tercera columna al diccionario en la cual se especifica el número de bits empleado por cada código recordando que éstos son de longitud variable (Tabla 3.3), con el propósito de facilitar la decodificación, al favorecer la búsqueda de los códigos en la trama codificada.

Tabla 3.3: Diccionario Huffman del ejemplo.

Símbolo	Código	Longitud
A	11	2
E	10	2
B	01	2
D	001	3
C	000	3

Como vemos en la Tabla 3.3 el diccionario se puede guardar con tres vectores. El primero se utiliza para los símbolos, el segundo para los códigos y el tercer vector se utiliza para la longitud de cada código.

Se propuso una técnica para reducir los bits necesarios para guardar el diccionario, teniendo en cuenta la longitud de las palabras de código para los conjuntos de datos sísmicos, que en los experimentos realizados el código con mayor longitud no superó los 19 bits.

La técnica consiste en guardar el diccionario en dos vectores usando variables de 32-bits. En el primer vector se almacenan los símbolos. En el segundo, se guardan los códigos usando los primeros 27 bits y su longitud utilizando los últimos 5 bits. La Tabla 3.4 muestra el diccionario guardado para el ejemplo utilizando este procedimiento.

Tabla 3.4: Diccionario en dos vectores.

Vector 1:	*Vector 2:*	
Símbolo	Código & Longitud	
A	`11xx ... x`	`00010`
E	`10xx ... x`	`00010`
B	`01xx ... x`	`00010`
D	`001x ... x`	`00011`
C	`000x ... x`	`00011`

Por otro lado, los datos codificados se deben guardar en otro vector cuyos elementos son variables de 32 o 64 bits. En la codificación tradicional, los datos codificados se guardan en el vector como un flujo de bits, no importa si un código se guarda en dos elementos diferentes. En la Figura 3.3 se resaltan en negrilla los códigos que quedan truncados en dos variables para la codificación tradicional usando variables de 32 bits.

DBAEEBAEAAEADE**C**DBCEACDABEBAE**D**EAABABECEAD

Datos codificados

001 01 11 10 10 01 11 10 11 11 10 11 001 10 **00**

0 001 01 000 10 11 000 001 11 01 10 01 11 10 **0**

01 10 11 11 01 11 01 10 000 10 11 001 ------

Figura 3.3: Datos codificados utilizando codificación tradicional con variables de 32 bits.

Debido a la longitud variable de los códigos, el algoritmo de decodificación tiene que empezar desde el primer bit del flujo de bits. Esto hace que la decodificación sea un proceso altamente secuencial. De tal forma, que la codificación tradicional no es efectiva para la implementación de la decodificación en GPU. Por ello se propone varias estrategias para guardar los datos codificados que permitan paralelar el proceso de decodificación en la GPU.

Estas estrategias implican el uso de paquetes con cabeceros, de tal manera, que los códigos en el límite de los paquetes se fuerzan a alinearse, dicho de otro modo, no se truncan los códigos en la frontera de cada variable. La primera estrategia utiliza paquetes de 32-bits, mientras que el segundo utiliza paquetes de 64-bits. Ambas estrategias hacen uso del mismo diccionario.

3.2.1 Estrategia uno

La primera estrategia utiliza un vector de variables de 32-bits para guardar los datos codificados, siendo cada elemento de éste vector, un paquete de 32-bits. En cada paquete, los 27 bits más significativos se utilizan para guardar los datos codificados, mientras que los 5 bits menos significativos se utilizan para guardar la cantidad de símbolos presentes en el paquete (Figura 3.4). Esta disposición tiene en cuenta que se puede dar el caso de guardar 27 símbolos codificados con 1 bit de código. Por lo tanto, se requieren 5 bits para indicar que hay un máximo de 27 símbolos en el paquete.

DBAEEBAEAAEADECDBCEACDABEBAEDEAABABECEAD

Datos codificados	símbolos por paquete	
001 01 11 10 10 01 11 10 11 11 10 11 --	01100	(12)
001 10 000 001 01 000 10 11 000 001 -	01010	(10)
11 01 10 01 11 10 001 10 11 11 01 11 01	01101	(13)
10 000 10 11 001 ----- ----- -----	00101	(5)

Figura 3.4: Datos codificados usando paquetes de 32-bits.

La estrategia obliga que ningún código se divida en dos paquetes diferentes; por lo tanto, algunos paquetes tendrán algunos bits sin utilizar, indicados como '–' en la Figura 3.4. De ésta manera, cada paquete puede ser tratado como un pequeño conjunto de datos que podrían ser decodificado individualmente. Sin embargo, se requiere crear un vector adicional llamado *índice*, para conocer la posición de los símbolos codificados en el conjunto de datos original.

Cada elemento del *índice* almacena la suma de los símbolos codificados en los paquetes anteriores. En la Tabla 3.5 se muestra el vector de *índice*, para los datos codificados de la Figura 3.4. Debido al tamaño de los datos sísmicos usados, el vector de *índice*, se compone de variables de 32 bits, para no perder la posición de cada paquete en el conjunto de datos de salida de la decodificación, a causa del desborde de la variable.

Tabla 3.5: Vector de *índice* para la Estrategia uno

Paquete	Símbolos por paquete	***índice***
1	12	**0**
2	10	**12**
3	13	**22**
4	5	**35**

3.2.2 Estrategia dos

Esta estrategia es similar a la estrategia de codificación con paquetes de 32-bits, pero utiliza variables de 64 bits para guardar los datos codificados. Aquí, los 58 bits más significativos de cada paquete se utilizan para guardar los datos codificados, mientras que los 6 bits menos significativos se usan para guardar la cantidad de símbolos que se encuentran en su correspondiente paquete (Figura 3.5). Además, un vector de índice también es necesario en esta estrategia.

Para esta estrategia el vector de *índice* tiene menos componentes que para la estrategia uno, sin embargo se siguen necesitando que las variables de éste vector sean de 32 bits, para el ejemplo el vector de *índice* tiene dos componentes (Tabla 3.6).

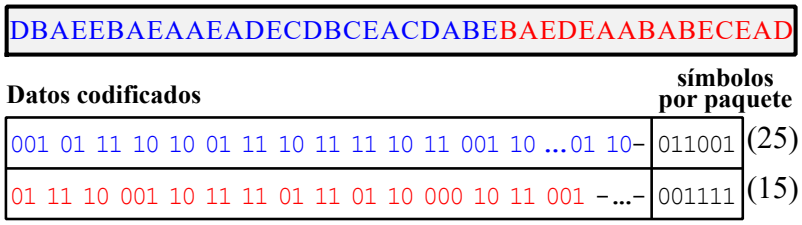


Figura 3.5: Datos codificados usando paquetes de 64-bits.

Tabla 3.6: Vector de *índice* para la Estrategia dos

Paquete	Símbolos por paquete	***índice***
1	25	**0**
2	15	**25**

3.2.3 Estrategia tres

Luego de tener los resultados del tiempo de decodificación para las estrategias anteriormente planteadas, me di cuenta que se podría mejora estos tiempos reduciendo la *divergencia* que se presenta por el desbalance en los paquetes de datos codificados.

Se planteó la estrategia tres, la cual parte de la estrategia uno (paquetes de 32-bits) a la cual mediante un ordenamiento con respecto al número de símbolos presentes en cada paquete se llega a una nueva trama de paquetes, dicho ordenamiento debe ser simultaneo para los paquetes de 32-bits como para el vector de *índice*. En la Figura 3.6, se aprecia el resultado del ordenamiento tanto para los paquetes de datos como para el vector *índice*. Esta estrategia tiene igual relación de compresión que la estrategia uno, ya que, el enfoque de esta es mejorar el tiempo de decodificación.

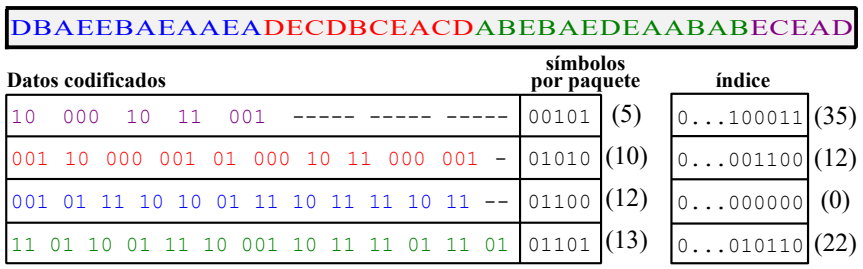


Figura 3.6: Datos codificados usando paquetes de 32-bits con ordenamiento.

Las tres estrategias de codificación anteriormente expuestas permitieron realizar en paralelo el proceso de decodificación, pero tienen un inconveniente. Debido a los bits no utilizados y el vector de *índice* adicional, la relación de compresión es menor en comparación a la estrategia de codificación tradicional (Figura 3.8).

Por ello se buscaron nuevas estrategias para mejorar la relación de compresión, el primer enfoque se concentro en mejorar la forma como se guardan los datos del diccionario, para ello se analizó mejor el

conjunto de datos utilizados, encontrando que con 16 de bits de cuantificación, el código más largo para la codificación Huffman para los datos analizados era de 19 bits. Por lo tanto, se planteó diferentes formas de guardar el diccionario:

- Enviar en un solo vector toda la información del diccionario, para lo cual, se necesitan 40 bits, como no es posible tener este tamaño de variable en C como en CUDA se requiere de 64 bits para la variable, con este tamaño no se consigue ninguna mejor y el rendimiento del decodificador decrece.
- Enviar dos vectores el primero de 32 bits con el símbolo concatenado con el código y en el segundo la longitud del código en variables tipo *char*(8 bits). Primera limitante los símbolos no pueden superar los 13 bits, segunda al ejecutar la decodificación se pierde rendimiento por los accesos no contiguos entre el código y su longitud.
- Usar dos vectores el primero para los símbolos en variables tipo *short*(16 bits) y el segundo de 32 bits que contiene los códigos concatenados con su longitud. De esta forma pasamos de usar 64 bits a 48 bits. Fue la mejor opción para almacenar el diccionario.

Para las siguientes dos estrategias para los datos codificados se usó el diccionario con la mejora planteada anteriormente.

3.2.4 Estrategia cuatro

Esta estrategia se basa en la codificación en paquetes de 32-bits. Se tomó el vector de datos codificados en paquetes de 32-bits y se concateno con el vector de *índice*, es decir, mediante una nueva variable de 64 bits en la cual los 32 bits más significativos se usaron para almacenar los datos codificados en paquetes de 32-bits y los 32 bits menos significativos se emplearon para alojar el contenido del vector *índice* (Figura 3.7). Esta estrategia se enfoca en mejorar aún más el proceso de decodificación.

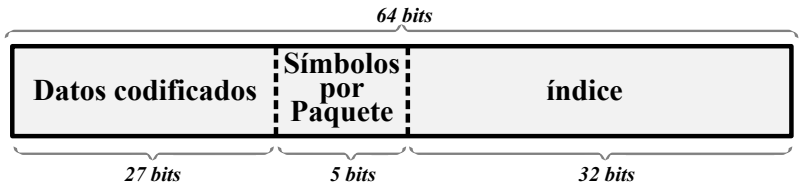


Figura 3.7: Estructura del paquete de 64-bits para la estrategia cuatro.

3.2.5 Estrategia cinco

Básicamente es la misma estrategia uno que hace uso de vectores de variables de 32-bits para guardar los datos codificados, pero con un cambio importante, se le quito el vector de *índice*, con lo cual se mejora la relación de compresión al costo de incrementar el tiempo de la decodificación.

3.2.6 Relación de compresión para las estrategias

Como se mencionó anteriormente las estrategias planteadas reducen la relación de compresión, pero permiten realizar una implementación de la decodificación en paralelo, para observar esta reducción en la

relación de compresión, se compararon las diferentes estrategias planteadas contra la forma tradicional de enviar los datos codificados mediante Huffman.

En la Figura 3.8, se muestra la *RC* obtenida para los datos sin transformacion y cuantificados a 12 bits, mediante las estrategias: uno (Paquetes de 32-bits), dos (Paquetes de 64-bits), cinco (Paquetes de 32-bits sin el vector de *índice*). No se muestran los resultados para la estrategia tres, ya que, son idénticos a la estrategia uno, la estrategia cuatro tampoco se muestra, debido a que, en la gráfica no es apreciable la mejora con respecto a la estrategia uno, siendo esta mejora de 0.02 unidades de *RC* en promedio.

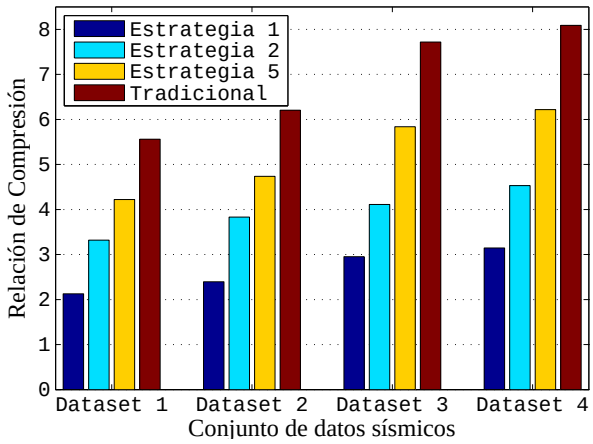
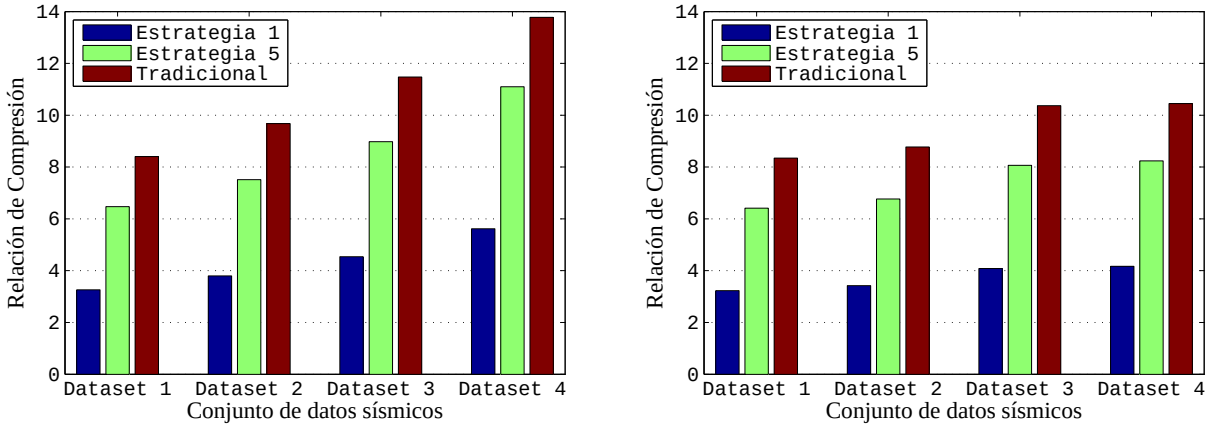


Figura 3.8: Relación de Compresión para los *Datasets* sin transformación.

También se realizaron las implementaciones con la Transfomada Wavelet (1D Nivel 2 y 2D Nivel 1), recordando que con ellas se obtuvo los más altos valores de *RC*, para 12 bits de Cuantificación. En la Figura 3.9, se aprecia la relación de compresión obtenida para las estrategias de Paquetes de 32-bits con y sin vector de *índice*.



(a) DWT 1D Nivel 2.

(b) DWT 2D Nivel 1.

Figura 3.9: Relación de Compresión para los *Datasets* con transformación.

Nuevamente se mantiene la tendencia presentada en la sección 2.7, en la cual la transformada wavelet 1D Nivel 2, permite una mayor relación de compresión para las estrategias mostradas.

Los resultados muestran como la estrategia de Paquetes de 32-bits reduce la relación de compresión en un factor aproximado de 2.5, al comprarse con la técnica tradicional. También se observa que la estrategia de Paquetes de 64-bits mejora la relación de compresión con respecto a la estrategia de Paquetes de 32-bits, esto se justifica debido a que se cuenta con un mayor tamaño para los paquetes, con lo cual, existe menor probabilidad de dejar bits sin usar en éste.

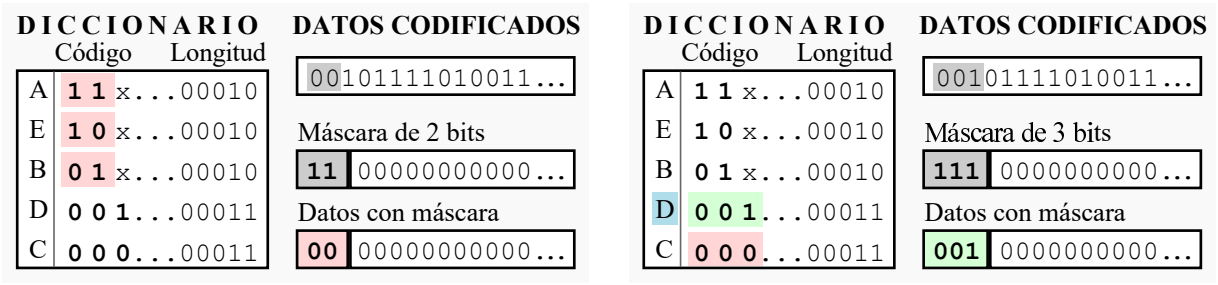
Para todos los casos la estrategia de Paquetes de 32-bits sin el vector de índice (Estrategia cinco), es la que se encuentra más cercana a la relación de compresión obtenida mediante la técnica tradicional de almacenamiento de los datos codificados, en promedio ésta estrategia se encuentra dos unidades de RC por debajo a la Tradicional.

3.3 Implementación de la Decodificación Huffman

El proceso de decodificación se desarrolla comparando los datos codificados con los códigos Huffman guardados en el vector del diccionario. La comparación se realiza en segmentos de bits, teniendo en cuenta la longitud de los códigos. Los pasos para desarrollar el algoritmo de decodificación son los siguiente:

1. Leer los cinco bits menos significativos de la primera posición del vector del diccionario para conocer la longitud del primer código.
2. Aplicar una máscara a los datos codificados utilizando el operador lógico AND. La máscara tiene tantos '1' como bits tiene el código. Los otros bits son '0'.
3. Comparar los datos codificados al aplicarle la máscara con los códigos del diccionario que tienen la misma longitud. Si hay una coincidencia, guardar el símbolo decodificado y pasar al paso 5. De lo contrario, continúe con el paso 4.
4. Buscar la longitud del código siguiente y volver al paso 2.
5. Si todos los símbolos de los datos codificados han sido descodificados, pasar al paso 7. De lo contrario, continuar con el paso 6.
6. Desplazar los bits de los datos codificados a la izquierda tantos bits tenga el código previo decodificado. Luego, volver al paso 1 para decodificar el siguiente símbolo.
7. El proceso de decodificación ha terminado.

La Figura 3.10, muestra cómo se utilizan diferentes máscaras para decodificar los datos. En la Figura 3.10a, no se decodifica ningún símbolo porque los bits de los datos codificados al aplicarle la máscara no coinciden con ningún código de 2 bits del diccionario. La Figura 3.10b muestra cómo se puede decodificar el símbolo ***D***, debido a que su código coincide con los datos codificados aplicando la máscara correspondiente.



(a) Coincidencia fallida.

(b) Coincidencia exitosa.

Figura 3.10: Proceso de decodificación aplicando máscaras a los datos codificados

Dentro de las diferentes implementaciones realizadas en la GPU para la decodificación se resaltan tres con las cueles se obtuvieron buenos resultados. Todas las implementaciones parten de la premisa que el proceso de decodificación puede ser paralelado suponiendo que cada paquete corresponde a un conjunto de datos individuales. De esta manera, cada paquete puede ser ejecutado por un hilo GPU, teniendo en cuenta que los símbolos decodificados por cada hilo deben organizarse de acuerdo con el vector de índice.

En primera instancia si intento trabajar con los datos codificados guardados mediante la estrategia tradicional, para ello se realizó un *kernel*, en el cual se le asigno todos los hilos a cada una de las posiciones del diccionario con el objetivo de poder decodificar al instante un símbolo independiente de la longitud de su código. Esta estrategia fue infructuosa, ya que, el tiempo empleado en la decodificación no mejoraba de forma apreciable a la decodificacion en CPU. Se descartó entonces trabajar con los datos codificados guardados mediante estrategia tradicional.

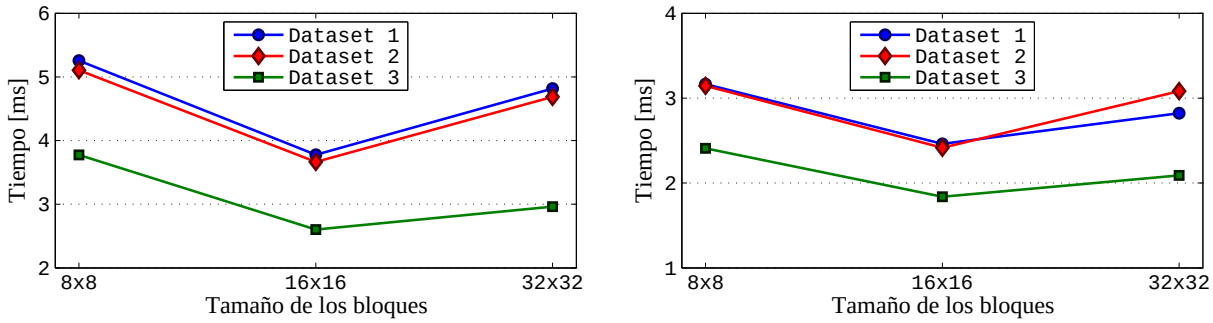
A continuación, se discuten cuatro versiones para la decodificación Huffman con las cuales se obtuvo aceleración para el proceso de descompresión de datos sísmicos.

3.3.1 Decodificador versión uno

Esta implementación hace uso de las estrategias de Paquetes de 32 y 64 bits, el *kernel* para cada una de ellas solo varia en el tamaño de las máscaras usadas y el tamaño de las variables empleadas, el *kernel* se lanzó en 2D, los resultados de tiempos de decodificación para cada una de las estrategias se muestran en la Figuras (3.11, 3.12).

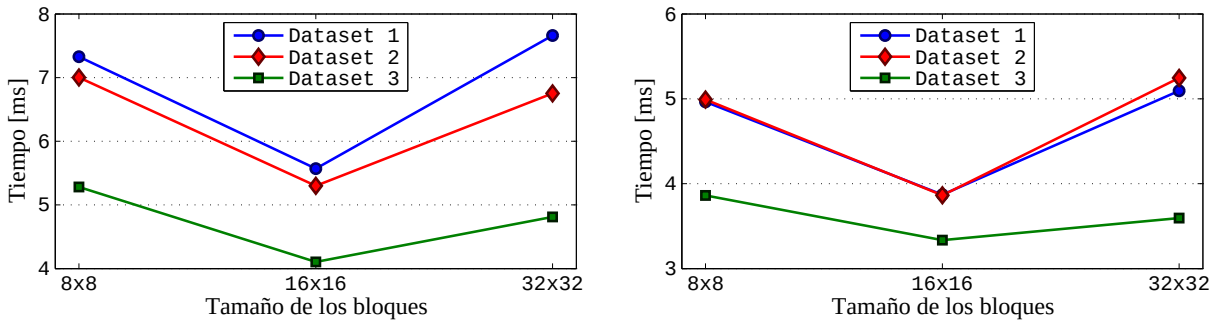
Se aprecia que la mejor configuración de bloque para ambas GPUs es de 16×16 hilos por bloque. También note como era de esperarse la GPU Tesla K40 exhibe los mejores tiempos de decodificación para ambas estrategias (Figuras (3.11b, 3.12b)).

Para ambas GPUs bajo la configuración de bloques bidimensional de 16×16 hilos, la mejor estrategia en cuanto a tiempos de decodificacion fue la estrategia de paquetes de 32-btis (Figuras (3.11a, 3.11b)).



(a) GTX 660. (b) Tesla K40.

Figura 3.11: Tiempo de decodificación, versión uno, datos codificados con paquetes de 32-bits.

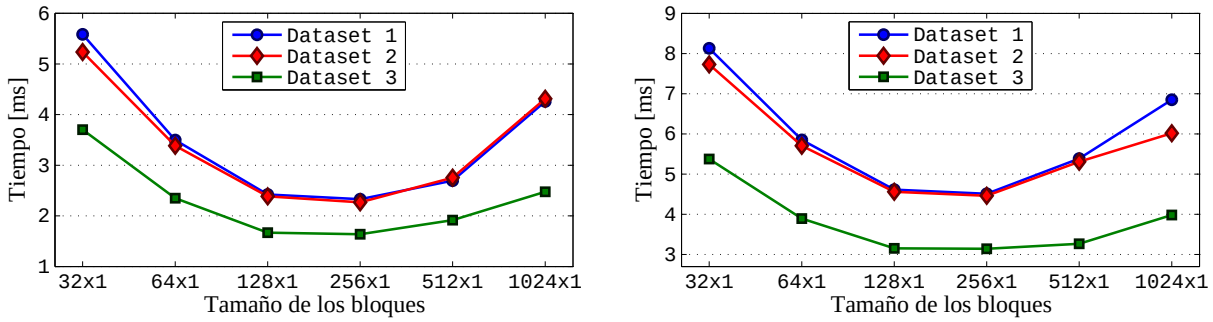


(a) GTX 660. (b) Tesla K40.

Figura 3.12: Tiempo de decodificación, versión uno, datos codificados con paquetes de 64-bits.

3.3.2 Decodificador versión dos

Para esta versión se trabajó con las estrategias (1, 2, 3). El *kernel* desarrollado se ejecutó en bloques unidimensionales, en el cual cada hilo se comporta como un decodificador independiente, el diccionario se accede de manera secuencial por cada uno de los hilos de ejecución. Los resultados obtenidos se muestran en la Figura 3.13



(a) Datos codificados con paquetes de 32-bits. (b) Datos codificados con paquetes de 64-bits.

Figura 3.13: Tiempos de decodificación para la Versión Dos sobre la GTX 660.

Se observa que, para tamaños de bloques unidimensionales, la distribución de 256×1 presenta un mejor rendimiento, en ambas estrategias de codificación con paquetes de 32 y 64 bits. Nuevamente la estrategia de paquetes de 32-bits exhibe los mejores tiempos de decodificación.

Al comparar las versiones uno y dos, la versión dos es superior a la versión uno, incluso a la implementación realizada para la Tesla K40 de la versión uno, al compararse con la implementación en la GTX 660 de la versión dos (Figuras (3.11, 3.12, 3.13)).

Una apreciación importante con respecto al rendimiento de las diferentes implementaciones para la decodificación Huffman sobre GPU, es que, a medida que cada paquete es decodificado por un hilo, el tiempo de decodificación para cada paquete en el *warp* será diferente. Esta diferencia en el tiempo de decodificación genera divergencia, porque cada paquete tiene un número diferente de símbolos codificados.

Una forma de reducir la divergencia causada por la diferencia en el número de símbolos en cada paquete, es organizándolos de acuerdo al número de símbolos almacenados. De esta forma los hilos que conformarán un *warp* en un instante decodificarán un número similar de símbolos.

Por tal razón, usando la configuración con la cual se obtuvieron los mejores resultados hasta el momento (datos codificados en paquetes de 32 bits y decodificados en bloques de 256×1 hilos); se planteó la hipótesis que la eficiencia incrementaría al organizar los paquetes de acuerdo al número de símbolos que estos contengan (Estrategia 3).

Como se puede observar en la Figura 3.14, se comparan los tiempos de decodificación obtenidos mediante el decodificador versión dos al ejecutarse en la Tesla K40, los resultados confirman la hipótesis.

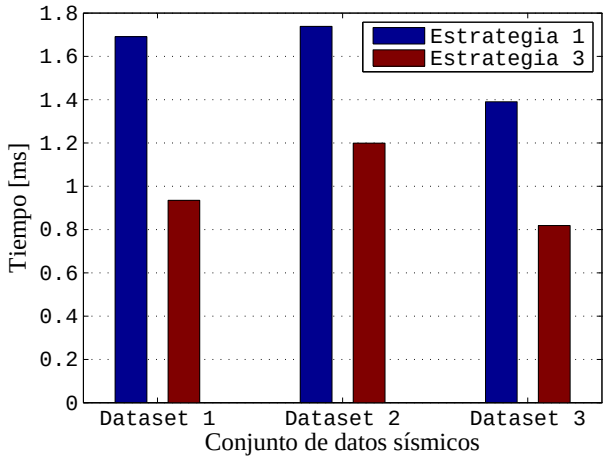


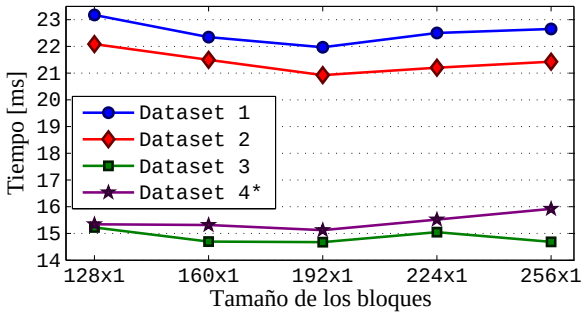
Figura 3.14: Tiempo de decodificación, paquetes organizados vs sin organizar (Estrategias 3 vs 1).

Sin embargo, la estrategia tres resulta ineficiente debido a la naturaleza secuencial de los algoritmos utilizados para organizar datos, en mi caso Burbuja [49]. Lo anterior se debe a que el tiempo de ejecución del algoritmo para ordenar los paquetes, resulta superior al tiempo de decodificación si estos están sin

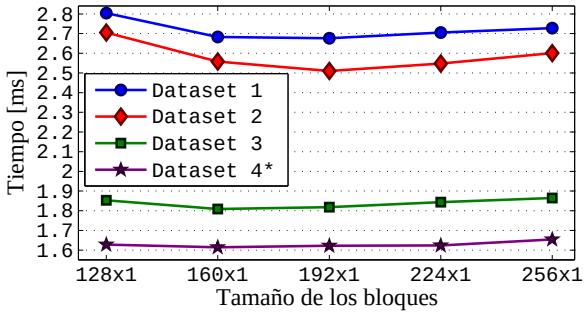
organizar. Por lo anterior, los resultados sugieren que el mejor tiempo de decodificación se obtiene usando la estrategia de paquetes de 32 bits sin organizar (Estrategia 1).

Con los resultados anteriores se optó por buscar si existía una configuración de bloques mejor a la encontrada hasta el momento (256×1), recordando que el rango optimo para la decodificación se encuentra ente 128×1 a 256×1. Para ello se evaluó nuevamente la estrategia uno (Paquetes de 32-bits) sobre esté intervalo.

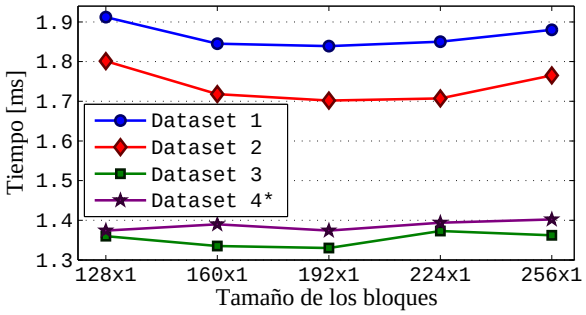
En la Figuras (3.15a, 3.15b, 3.15c), se muestran los resultados para la estrategia de paquetes de 32 bits sobre las diferentes GPUs. El tiempo del *Dataset 4* se encuentra escalado por un factor de 2.5 para la GTX 760M, para la Quadro K4200 y la Tesla K40 se escaló por 2, para poder visualizar todos los *Datasets* en un rango que permita distinguir el comportamiento sobre el intervalo de bloques de 128×1 a 256×1.



(a) GTX 760M.



(b) Quadro K4200.



(c) Tesla K40.

Figura 3.15: Tiempo de decodificación para la versión dos.

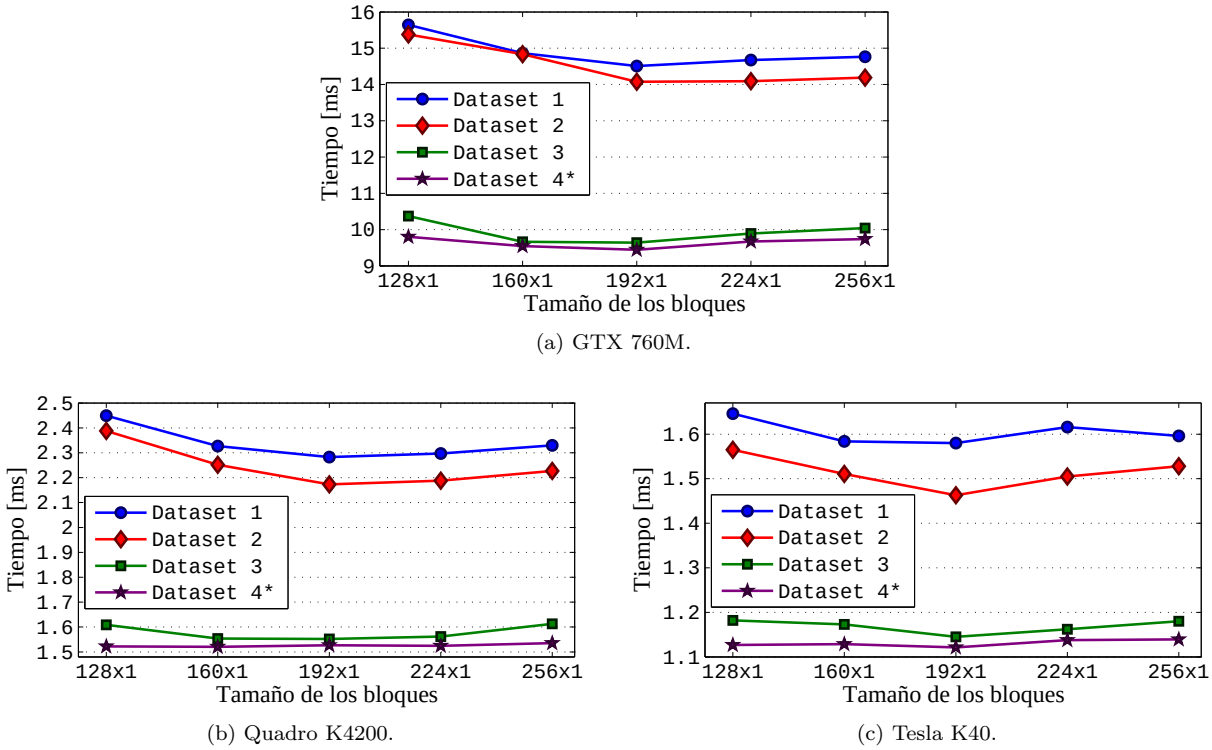
Para la GTX 760M la mejor distribución de hilos por bloque encontrada es de 192×1, para todos los *Datasets*. En la Quadro K4200 para los *Datasets* 1 y 2 la mejor distribución es de 192×1, mientras que para los *Datasets* 3 y 4 es de 160×1. Con respecto a la Tesla K40 el mejor rendimiento se logra con 192×1 hilos, para todos los *Datasets*.

3.3.3 Decodificador versión tres

Para esta implementación, en una primera instancia se intentó usar memoria compartida, para la lectura del diccionario, pero al ver que el rendimiento del decodificador decaía se descartó esta opción y se optó

por mejorar la lectura del diccionario mediante registros.

Por otro lado el mejor aporte a esta implementación se obtuvo al usar la estrategia cuatro para guardar los datos codificados, ya que, esta estrategia optimiza la lectura de la memoria global al realizar un accesos unificado [47], debido a que se lee el paquete y su *índice* en una sola transacción de memoria de 8 bytes por hilo a registros, donde la parte baja de los 8 bytes es el *índice* y la parte alta corresponde a los datos codificados con la cantidad de símbolos presentes en este, ambos almacenados en registros de 32 bits.



(a) GTX 760M.

(b) Quadro K4200.

(c) Tesla K40.

Figura 3.16: Tiempo de decodificación para la versión tres.

En las Figuras (3.16a, 3.16b, 3.16c), el *Dataset* 4 se escaló por un factor de 2.5 para la GTX 760M, 2.2 para la Quadro K4200 y de 1.8 para la Tesla K40. La GTX 760M muestra su mejor rendimiento con una distribución de 192×1 hilos por bloque. Para la Quadro con 192×1 hilos por bloque en los *Dataset* 1 al 3 exhibe el mejor rendimiento, mientras que para el *Dataset* 4 este se logra con 160×1. En la Tesla K40 el mejor tiempo de decodificación para todos los *Datasets* se logró con 192×1 hilos por bloque.

Como se evidencia la versión tres el tiempo de decodificación es menor en todos los casos con respecto a la version dos. Esta versión es la que obtiene el menor tiempo de decodificación de todas las implementaciones realizadas en esta investigación. La aceleración de esta versión con respecto a la dos es de 1.5x para la GTX 760M y de 1.16x tanto para la Quadro como para la Tesla K40.

3.3.4 Decodificador versión cuatro

Para esta implementación se usó la estrategia cinco en la cual se guardan los datos codificados en paquetes de 32 bits, pero no se almacena el vector de *índice*. El principal reto para esta implementación es la creación del vector *índice*, para ello fue necesario crear cuatro *kernels* (Figura 3.17):

- El primer *kernel*, es el encargado de leer todos los paquetes de 32 bits y extraer la cantidad de símbolos presentes en cada paquete para luego guardarlos en el vector $\boldsymbol{x}$.
- El segundo *kernel*, se encarga de realizar la suma acumulada por cada bloque, almacenado el resultado en el vector $\boldsymbol{y}$. La primera posición de cada bloque siempre es cero ($\{y_0,\ y_3,\ y_6\} = 0$), la segunda posición en el bloque es la misma entrada en la posición anterior ($\{y_1,\ y_4,\ y_7\} = \{x_0,\ x_3,\ x_6\}$), el resto de componentes del vector $\boldsymbol{y}$ en el bloque son la suma de las entradas anteriores ($\{y_2,\ y_5,\ y_8\} = \{x_0 + x_1,\ x_3 + x_4,\ x_6 + x_7\}$), realizadas mediante suma reducción [50, 47].
- El tercer *kernel*, realiza la suma de la última componte de cada bloque del *kernel* inmediatamente anterior con la del *kernel* uno ($\{z_0,\ z_1,\ z_2\} = \{y_2 + x_2,\ y_5 + x_5,\ 0\}$).
- El cuarto *kernel*, crea el vector *índice*, al sumar los resultados de la suma total de cada bloque con la suma parcial acumulada de cada bloque producto de los *kernels* dos ($\{r_{(0,1,2)},\ r_{(3,4,5)},\ r_{(6,7,8)}\} = \{w_0 + y_{(0,1,2)},\ w_1 + y_{(3,4,5)},\ w_2 + y_{(6,7,8)}\}$).

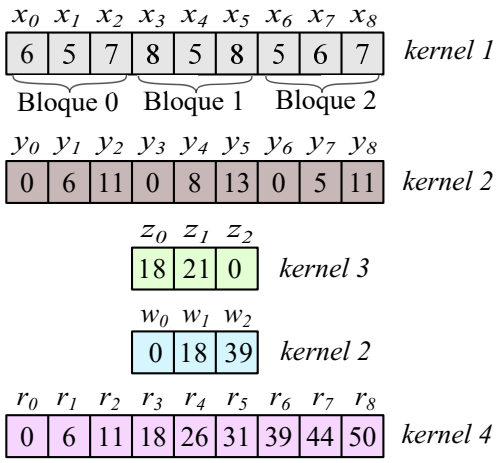
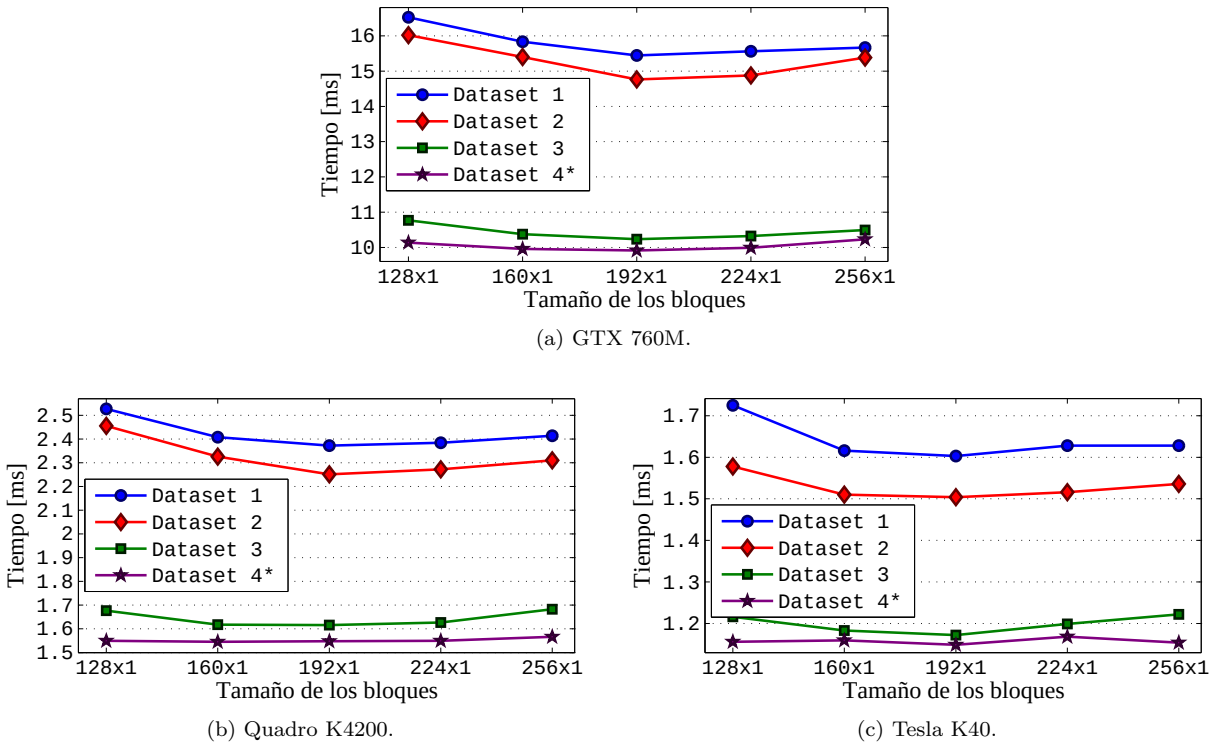


Figura 3.17: Ejemplo de ejecución de los *kernels* para la creación del vector *índice*.

Para el *kernel* del decodificador Huffman, se basó en los mejores resultados de la versión tres, de la cual se extrajo el uso de registros para agilizar la lectura del diccionario, como para las comparaciones con las máscaras y almacenamiento de los paquetes con sus corrimientos a medida que se decodifican los datos.

Los tiempos para la decodificación para esta versión incluyen el tiempo empleado para la creación del vector *índice*. El tiempo del *Dataset 4* se escaló por un factor de 2.4 para la GXT 760M (Figura 3.18a), de 2.1 para la Quadro (Figura 3.18b) y por 1.77 en la Tesla K40 (Figura 3.18c).



(a) GTX 760M.

(b) Quadro K4200.

(c) Tesla K40.

Figura 3.18: Tiempo de decodificación para la versión cuatro.

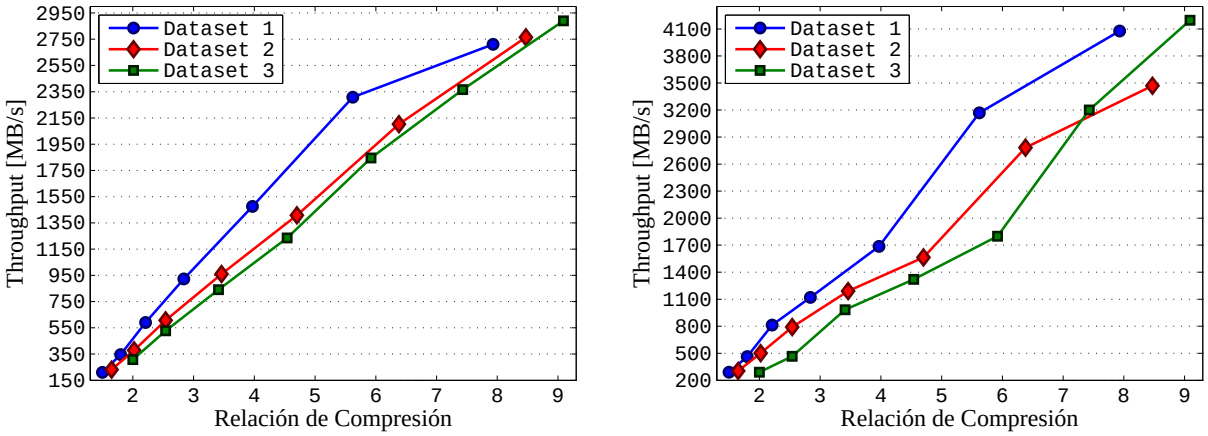
Como era de esperase el tiempo empleado para la decodificación es mayor al empleado por la versión tres pero sigue siendo inferior al de la versión dos (Figuras (3.15, 3.16, 3.18)). En cuanto a la mejor distribucion de bloques esta versión conserva las mismas características comentadas en la versión tres, es decir, para las tres GPUs (760M, Quadro, K40) la mejor distribución es de 192×1 hilos por bloque, para todos los *Datasets* exceptuando el *Dataset* 4 para la Quadro, en el cual dicho rendimiento se logra con 160×1.

3.3.5 *Throughput* y aceleración

Para fines de pruebas, se comprimieron los conjuntos de datos variando el número de bits de cuantificación, con el fin de lograr diferentes relaciones de compresión. La Figura 3.19 muestra la relación entre el *Throughput*(MegaBytes por segundo) y la relación de compresión. En esta prueba, se utilizado la estrategia de codificación de paquetes de 32 bits (Estrategia uno) y una configuración de 256×1 hilos por bloque, sobre el decodificador version dos.

En las Figuras (3.19a, 3.19b) se observa que el *Throughput* mejora a medida que aumenta la relación de compresión. Este comportamiento es más lineal para la GTX 660 que en la Tesla K40.

Es difícil comparar nuestros resultados con otros decodificadores Huffman implementados en GPU, porque la naturaleza de cada dato afecta cómo se codifica. Además, la tecnología de GPU es diferente de un trabajo a otro. Sin embargo, algunas comparaciones se pueden hacer si tenemos en cuenta el *Throughput*.



(a) GTX 660.

(b) Tesla K40.

Figura 3.19: *Throughput* vs relación de compresión

En la Tabla 3.7, se presenta una comparación con un trabajo predecesor que informa el *Throughput* de la decodificación. Para establecer la comparación, fue necesario comprimir otro conjunto de datos que tiene 19200 trazas y 3584 muestras cada una. Los resultados sugieren que nuestro trabajo podría ser superior al trabajo mencionado.

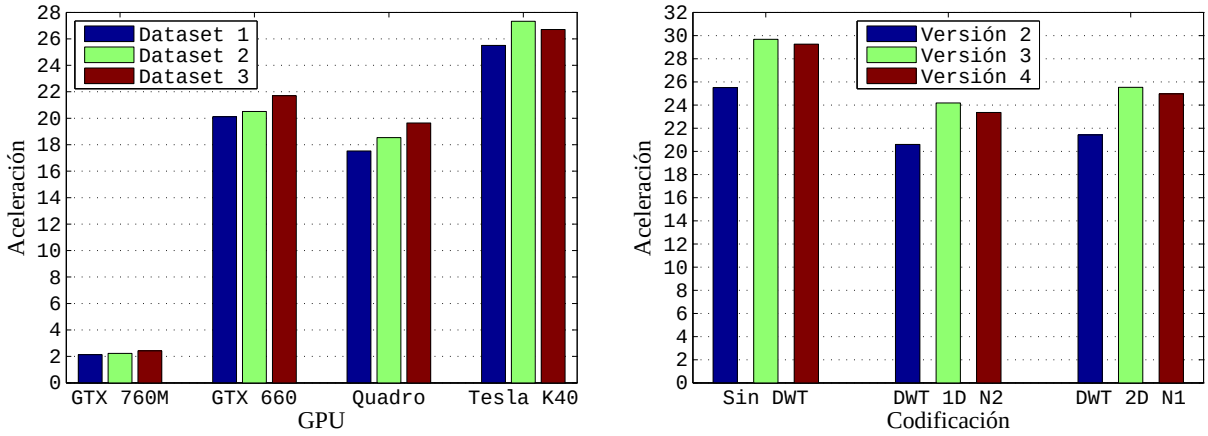
Tabla 3.7: Resultados de predecesores.

Autor	GPU	Reloj Base	*RC*	Tamaño [*MB*]	Throughput [*MB/s*]
Decodificador	Tesla K40	745 *MHz*	1.6	275.30	435.64
Versión dos	GTX 660	1110 *MHz*	1.6	275.30	260.71
Aqrawi	Quad Core i7	2.81 *GHz*	1.4	244.14	301.41
2010 [43]	Tesla C1060	1.30 *GHz*	1.4	244.14	116.26

En la Figura 3.20, muestra la aceleración obtenida al implementar las diferentes versiones sobre cada una de las GPU, al comparase con una implementación en CPU de referencia i7 a 3.2 *GHz*. La Figura 3.20a, muestra el incremento en la aceleración, para cada dispositivo, como es de esperase al contar con más recurso en la GPU, la aceleración incrementa.

Al comparar la aceleración obtenida mediante la GTX 660 contra la Quadro K4200 se observa una particularidad, en la cual, el mayor rendimiento se logra con la GTX 660, teniendo está menos recursos que su contra parte (5 *SM* contra 7 *SM*), este incremento se debe a un mayor reloj de la GTX 660 (1110 *MHz* contra 784 *MHz*), como consecuencia se favorece la parte secuencial del algoritmo (búsqueda en el diccionario), mejorando de esta manera el rendimiento global de la GPU.

También se aprecia que la GTX 760M alcanza en promedio una aceleración de 2x, esta aceleración es baja comparándola con las obtenidas para las otras GPUs, esto se debe a que la GTX 760M está pensada para dispositivos móviles y por ende los relojes de los núcleos como de la memoria son menores que para un dispositivo de mesa.



(a) Versión 2 sin DWT, par las cuatro GPUs.

(b) Versiones (2-4) para el *Dataset* 1, Tesla K40.

Figura 3.20: Aceleración del decodificador en GPU, con respecto al CPU i7 a 3.2 *GHz*.

Con respecto a las diferentes versiones del decodificador Huffman implementado en la GPU (Figura 3.20b), se observa una mejora en promedio de 3.5x para las versiones tres y cuatro contra la versión dos, mientras que la diferencia en aceleración para la versión tres con respecto a la cuatro es de 0.4x.

Se observó que la mayor aceleración se obtiene cuando no se usa la etapa de transformación, puesto que al agregar la etapa de transformación también se beneficia la decodificación en CPU, sin embargo, el uso de esta etapa obtiene buenos resultados en todas la GPUs, este análisis se pude entender mejor en las siguientes secciones donde se exponen las siguientes etapas para la descompresión de los datos.

En la Tabla 3.8, se resumen todos los tiempos empleados en la decodificación de los cuatro *Datasets*, para las diferentes plataformas de computo empleadas en esta investigación, al emplear o no la etapa de Transformación. El tiempo para el *Dataset* 4 y los tiempos con la etapa de transformación de los otros *Datasets* no se muestran para la GTX 660 debido a que no se tenía a disposición para realizar estas pruebas.

Tabla 3.8: Tiempo de decodificación para todas las plataformas de computo.

Dataset	**Decodificación**	**Tiempo [*ms*]**												
		CPU i7	**GTX 760M**			**GTX 660**		**Quadro K4200**			**Tesla K40**			
		3.2 *GHz*	**V2**	**V3**	**V4**	**V1**	**V2**	**V2**	**V3**	**V4**	**V1**	**V2**	**V3**	**V4**
1	Sin DWT	46.895	21.966	14.508	15.445	3.777	2.331	2.676	2.283	2.372	2.461	1.839	1.580	1.603
	DWT 1D Nivel 2	27.514	15.138	9.943	10.584	-	-	1.849	1.581	1.645	-	1.336	1.138	1.178
	DWT 2D	27.032	13.305	8.754	9.219	-	-	1.754	1.445	1.510	-	1.262	1.059	1.083
2	Sin DWT	46.520	20.927	14.077	14.762	3.666	2.268	2.510	2.173	2.251	2.416	1.702	1.463	1.504
	DWT 1D Nivel 2	31.270	14.265	8.440	10.233	-	-	1.726	1.501	1.561	-	1.246	1.074	1.110
	DWT 2D	28.034	16.179	10.988	10.250	-	-	2.104	1.857	1.920	-	1.628	1.400	1.472
3	Sin DWT	35.525	14.676	9.641	10.236	2.602	1.637	1.809	1.552	1.616	1.838	1.330	1.145	1.172
	DWT 1D Nivel 2	21.029	11.992	8.055	8.542	-	-	1.609	1.441	1.491	-	1.349	1.178	1.206
	DWT 2D	24.516	12.727	8.452	9.089	-	-	1.712	1.519	1.571	-	1.505	1.323	1.309
4	Sin DWT	19.514	6.050	3.777	4.130	-	-	0.807	0.691	0.736	-	0.687	0.623	0.649
	DWT 1D Nivel 2	15.625	5.607	3.862	4.095	-	-	0.907	0.835	0.870	-	0.941	0.843	0.871
	DWT 2D	15.011	5.980	3.884	4.204	-	-	0.974	0.835	0.878	-	0.866	0.775	0.802

Al revisar los tiempos de ejecución de la decodificación se observa como era de esperar por los análisis previos una disminución en estos tiempos cuando se aplica la etapa de transformación, en todas las arquitecturas, sin embargo, para el *Dataset* 4 en todas las arquitecturas el tiempo empleado en la decodificación al emplear la transformación es mayor que sin no se usara esta etapa.

Al analizar las posibles causa de este comportamiento aparte de la divergencia, se observó que la eficiencia en la intrusiones asignadas y ejecutadas por la GPU era menor con la etapa de transformación (80% contra 75%), de igual forma la Ocupación de la GPU pasaba del 70% al 60%, esto se debe a que al comprimir este *Dataset* se llega al punto de menor eficiencia al no contar con suficientes paquetes para mantener una mayor ocupación de la GPU, sin embargo se sigue acelerando el proceso de decodificación.

3.3.6 Selección de la versión del decodificador

Para la selección del decodificador que se usó para la descompresión, es necesario evaluar el rendimiento de este en términos de tiempo de ejecución como en la relación de compresión obtenida para su implementación.

Como se aprecia en la Tabla 3.8, la versión uno es más lenta en promedio por 1.27 ms en la GTX 660 y de 61 μs en la Tesla K40, que la versión dos, al tener una relación de compresión igual, se descartó la implementación más lenta es decir la versión uno.

De igual forma al comparar la versión dos con la tres, la versión dos es más lenta en promedio en 4.54 ms sobre la GTX 760M, en 227 μs para la Quadro K4200 y de 174 μs para la Tesla K40, además la relación de compresión es 0.04 inferior en promedio, por lo cual también se descarta la versión dos.

Como se mencionó anteriormente la versión tres es la que obtiene los mejores resultados en cuanto a tiempo de ejecución sin embargo la versión cuatro se encuentra muy cercana a está, en promedio se encuentra a 534 μs para la GTX 760M, en 59 μs sobre la Quadro K4200 y de 29.8 μs en la Tesla K40. En cambio para la relación de compresión (Figuras (3.8, 3.9)), la versión cuatro al ser desarrollada bajo la estrategia cinco logra obtener en promedio 3.5 unidades de RC sobre la versión tres con la estrategia cuatro; dicho de otro modo la versión cuatro dobla la relación de compresión con respeto a la version tres.

En conclusión, la versión cuatro ofrece los mejores resultados globales en términos de tiempo de decodificación y relación de compresión, en consecuencia, esta es la versión del decodificador empleada para la descompresión de datos sísmicos.

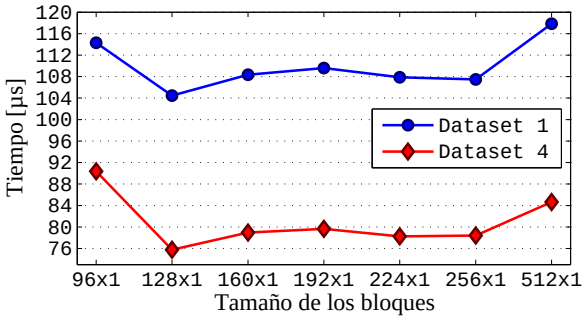
3.4 Implementación de la Cuantificación inversa

Para la implementación de la cuantificación inversa en GPU, se desarrolló un *kernel* encargado de realizar la siguiente ecuación:

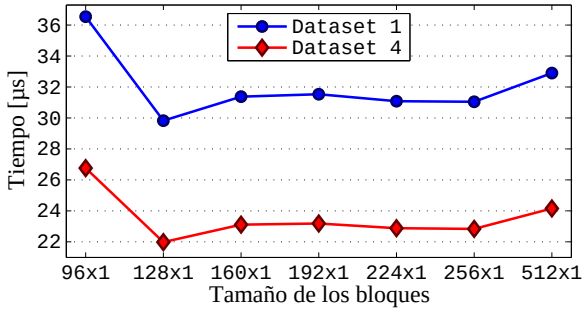
$$x_{iQ} = (x_{iC} * K1) + K2 \tag{3.1}$$

donde *K1* y *K2* son constantes proporcionadas al momento de realizar la cuantificación (sección 2.3). El *kernel* tiene como entrada los datos decodificados (x_{iC}) producto de la etapa anterior (sección 3.3) y las constantes *K1* y *K2*.

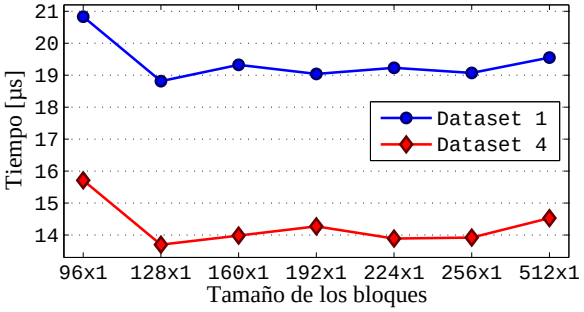
Para el lanzamiento del *kernel* se usaron bloques unidimensionales, los resaltado para el tiempo de ejecución en las diferentes plataformas se muestran en la Figura 3.21. Como es de esperase el tiempo depende del tamaño del conjunto de datos, por lo cual los *Datasets* 2 y 3 presenta el mismo rendimiento que el *Dataset* 1. El Dataset 4 presenta en menor tiempo de ejecución para todas las plataformas, ya que, este *Dataset* es el de menor tamaño.



(a) GTX 760M.



(b) Quadro K4200.



(c) Tesla K40.

Figura 3.21: Tiempo de ejecución del *kernel* de cuantificación inversa.

Con respecto a la mejor distribución hilos por bloque, se encontró que con 128×1 hilos por bloque, para todos los dispositivos (Figuras (3.21a, 3.21b, 3.21c)), se logra el mejor rendimiento.

La escalabilidad para la cuantificación inversa en GPU, tomando como referencia los tiempos en la GTX 760M, es de 3.5x para la Quadro K4200 y de 5.5x para la Tesla K40.

3.5 Implementación de la Transformación inversa

Para la transformación inversa se implementó la *IDWT 1D* y *2D*, mediante esquema *lifting* (sección 2.2.1). La *IDWT 1D* se implementó para dos nivel y la *IDWT 2D* se implementó a un solo nivel, esto debido a que tiene un número similar de operaciones y presentaron una *RC* similar (sección 2.7).

3.5.1 Transformación Inversa 1D

Para la implementación en GPU se utilizó el esquema mostrado en la Figura 3.22. Para el filtro se usó el $CDF_{2,2}$, cuyos polinomios en el esquema *liftin* son $\widetilde{T} = -\frac{1}{2}\left(1+z\right)$ y $\widetilde{S} = \frac{1}{4}\left(z^{-1}+1\right)$. La implementación del algoritmo para la transformada inversa *IDWT 1D* se desarrolla mediante las siguiente ecuaciones:

$$\textbf{\textit{Separación:}} \quad \begin{aligned} h_p &= y_{[2*i+1]} \\ l_p &= y_{[2*i]} \end{aligned} \tag{3.2}$$

$$\textbf{\textit{Actualización:}} \quad l_{f[i]} = \begin{cases} l_{p[i]} - \left\lfloor \frac{1}{4} h_{p[i]} \right\rfloor, & i = 0 \\ l_{p[i]} - \left\lfloor \frac{1}{4}\left(h_{p[i-1]} + h_{p[i]}\right) \right\rfloor, & i > 0 \end{cases} \tag{3.3}$$

$$\textbf{\textit{Predicción:}} \quad h_{f[i]} = \begin{cases} h_{p[i]} + \left\lfloor \frac{1}{2}\left(l_{f[i]} + l_{f[i+1]}\right) \right\rfloor, & i < N-1 \\ h_{p[i]} - \left\lfloor \frac{1}{2} l_{f[i]} \right\rfloor, & i = N-1 \end{cases} \tag{3.4}$$

$$\textbf{\textit{Unión:}} \quad \begin{aligned} x_{[2i+1]} &= h_{f[i]} \\ x_{[2i]} &= l_{f[i]} \end{aligned} \tag{3.5}$$

donde $i = \{0, 1, ..., N-1\}$, N representa la mitad de longitud de la trama de entrada.

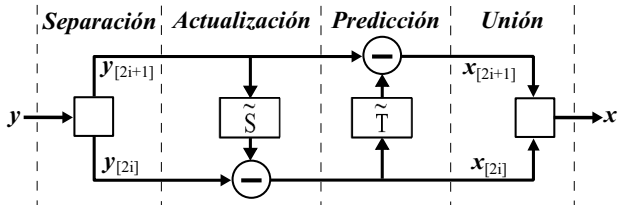


Figura 3.22: Esquema *lifting* para la implementación de la *IDWT 1D*.

La implementación de la transformación inversa con dos nivel, se realiza aplicando la fase de *Separación*, como resultado se obtiene h_p y l_p, luego a l_p se le vuelve a aplicar la fase de *Separación*, a los resultados de esta última separación se les aplican las siguientes fases (*Actualización, Predicción, Unión*), el resultado de la *Unión* formar el nuevo l_p^*. Con h_p de la primera separación y este nuevo l_p^*, se proceden con las fases de *Actualización, Predicción, Unión*, de esta forma se obtiene la transformación inversa 1D con dos niveles (Figura 3.23).

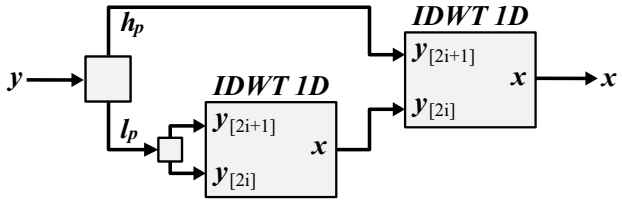
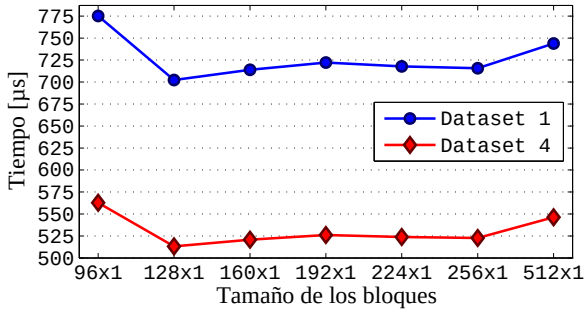
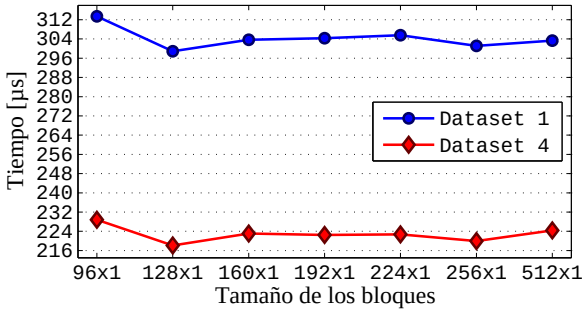


Figura 3.23: Esquema *lifting* para la implementación de la *IDWT 1D* con dos niveles.

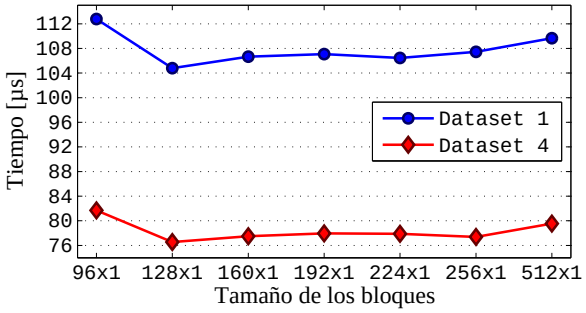
Los resultados de la implementación para la *IDWT 1D* con dos niveles se muestran en la Figura 3.24. Los *kernels* se lanzaron en bloques unidimensionales, el tiempo de ejecución de la *IDWT 1D* con dos niveles depende del tamaño del conjunto de datos, es por esto que el *Dataset* 4 es el más rápido. Con respecto a la mejor distribución de hilos por bloque se encontró que con 128×1 hilos por bloque, el tiempo de ejecución para todos los dispositivos es el menor.



(a) GTX 760M.



(b) Quadro K4200.



(c) Tesla K40.

Figura 3.24: Tiempo de ejecución de la *IDWT 1D* con dos niveles.

La escalabilidad para la *IDWT 1D* con dos niveles en GPU, tomando como referencia los tiempos en la GTX 760M (Figura 3.24a), es de 2.35x para la Quadro K4200 (Figura 3.24b) y de 6.7x para la Tesla K40 (Figura 3.24c).

3.5.2 Transformación Inversa 2D

La implementación de la *IDWT 2D*, se basa en el desarrollo echo para la *IDWT 1D*, con algunas modificaciones para poder trabajar en dos dimensiones. Si observamos la Figura 1.4b (sección 1.1.1), para la reconstrucción de la matriz es necesario aplicar tres bloques compuestos por la *IDWT 1D*, los dos primeros aplican la *IDWT 1D* por columnas y el tercero por filas.

Se planteó una nueva estructura para aplicar la *IDWT 2D*, en la cual mediante un nuevo bloque, se organizan los datos y luego se separaran con el propósito de funcionar los dos bloques de *IDWT 1D* por columnas a uno solo. En la Figura 3.25 se muestra la esquema para la implementación de la transformación inversa 2D.

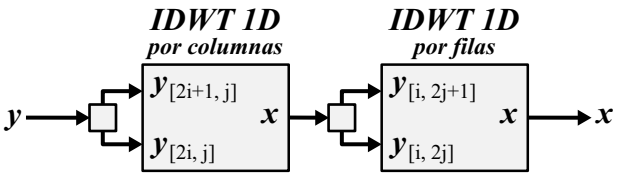
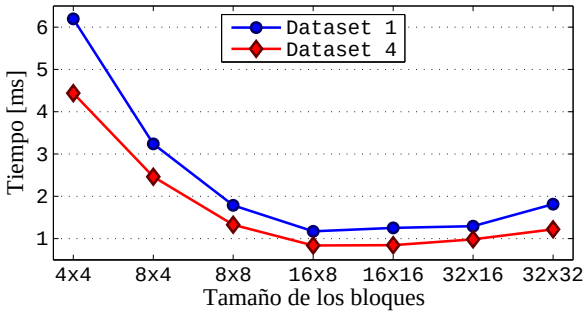
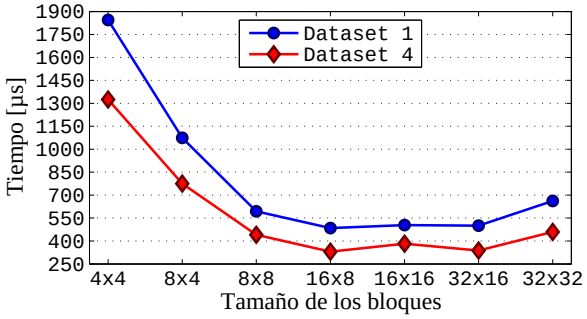


Figura 3.25: Esquema *lifting* para la implementación de la *IDWT 2D*

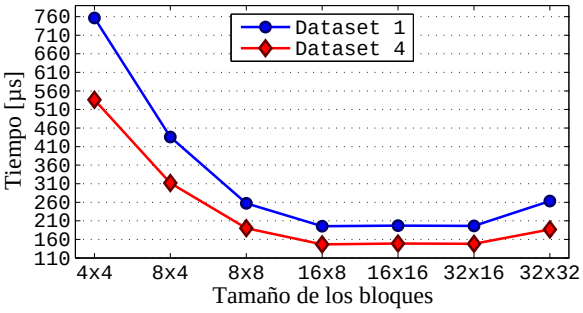
Los resultados de la implementación para la *IDWT 2D* se muestran en la Figura 3.26. Los *kernels* se lanzaron en bloques bidimensionales, el tiempo de ejecución de la *IDWT 2D* depende del tamaño del conjunto de datos, la mejor distribución de hilos por bloque es de 16×8.



(a) GTX 760M.



(b) Quadro K4200.



(c) Tesla K40.

Figura 3.26: Tiempo de ejecución de la *IDWT 2D*.

La escalabilidad para la *IDWT 2D*, tomando como referencia los tiempos en la GTX 760M (Figura 3.26a), es de 2.48x para la Quadro K4200 (Figura 3.24b) y de 5.87x para la Tesla K40 (Figura 3.24c).

los resultados obtenidos en tiempo de ejecución para la *IDWT 1D Nivel 2* siguiere que esta implementación es superior a la realizada en un procesador de propósito especifico (FPGA) [14], la aceleración obtenida por mi implementación es de 14.5x con respecto a dicho trabajo. Al respecto del rendimiento de las dos transformaciones, la *IDWT 1D Nivel 2* es 1.6x más rápida que la *IDWT 2D*, esto se justifica, debió a que la *IDWT 2D* tiene 1.33 veces más operaciones que su contraparte 1D a dos niveles de descomposición.

3.6 Descompresión de datos sísmicos

Luego de ver el rendimiento de cada una de las etapas que conforman la descompresión de datos sísmicos, se procedió a calcular el tiempo total de descompresión, para ello se trabajó con el decodificador versión cuatro bajo la distribución de hilos por bloque que mejor se desempeñó para todos los *Datasets*, seguido del *kernel* de la cuantificación inversa y finalmente los *kernels* que compone la etapa de transformación inversa correspondiente.

En la Tabla 3.9, muestra la mejor distribución de hilos por bloque de los *kernels*, para cada una de las etapas que componen la descompresión de datos sísmicos.

Tabla 3.9: Distribución de hilos por bloque para la descompresión

Etapa	Tamaño bloques
Decodificación	192×1
Cuantificación inversa	128×1
IDWT 1D Nivel 2	128×1
IDWT 2D	16×8

Las Tablas (3.10, 3.11, 3.12), muestran los tiempos empleados por la etapa de decodificación con los datos comprimidos con y sin transformación. La primera muestra el tiempo empleado por el decodificador cuando se comprimen los datos sin la etapa de transformación, la segundo y tercera muestran el tiempo de ejecución de la decodificación para un esquema de compresión con Transformación Wavelet (*DWT 1D Nivel 2* y *DWT 2D*).

Tabla 3.10: Tiempo de ejecución *kernels* decodificación, compresión sin etapa de transformación.

	Tiempo $[ms]$		
Dataset	GTX 760M	Quadro K4200	Tesla K40
1	15.445	2.372	1.603
2	14.762	2.251	1.504
3	10.236	1.616	1.172
4	4.130	0.737	0.649

Tabla 3.11: Tiempo de ejecución *kernels* decodificación, compresión con *DWT 1D Nivel 2*.

	Tiempo $[ms]$		
Dataset	GTX 760M	Quadro K4200	Tesla K40
1	10.584	1.645	1.179
2	10.233	1.561	1.128
3	8.542	1.495	1.248
4	4.095	0.876	0.885

Tabla 3.12: Tiempo de ejecución *kernels* decodificación, compresión con *DWT 2D*.

	Tiempo $[ms]$		
Dataset	GTX 760M	Quadro K4200	Tesla K40
1	9.219	1.515	1.083
2	10.250	1.920	1.472
3	9.089	1.571	1.388
4	4.307	0.889	0.818

La Tabla 3.13, muestra el tiempo empleado por la etapa de cuantificación inversa, recordado que este tiempo depende del tamaño del *Dataset* o número de muestras total.

Tabla 3.13: Tiempo de ejecución *kernel* cuantificación inversa.

Total muestras	Tiempo $[\mu s]$		
del Dataset	GTX 760M	Quadro K4200	Tesla K40
344064	105.120	29.831	18.816
245760	75.776	21.987	13.696

Las Tablas (3.14, 3.15), muestra el tiempo total empleado por los *kernels* al desarrollar la etapa de transformación inversa. La primera muestra el tiempo de ejecución para la *IDWT 1D* con dos niveles y la segundo el tiempo de ejecución para la *IDWT 2D*.

Tabla 3.14: Tiempo de ejecución *kernels IDWT 1D* con dos niveles.

Total muestras	Tiempo $[\mu s]$		
del Dataset	GTX 760M	Quadro K4200	Tesla K40
344064	703.392	298.903	104.799
245760	513.183	220.121	76.544

Tabla 3.15: Tiempo de ejecución *kernels IDWT 2D*.

Total muestras	Tiempo $[\mu s]$		
del Dataset	GTX 760M	Quadro K4200	Tesla K40
344064	1172.032	484.082	195.335
245760	839.501	329.943	146.399

Las Tablas (3.16, 3.17, 3.18), muestran los tiempos de descompresión para los cuatro *Datasets*, en las diferentes GPUs. En la primera se observa los tiempos de descompresión cuando se comprimen los datos sin etapa de transformación, la segunda y tercera muestra los tiempos de descompresión para un esquema de compresión compuesto por: Cuantificación Uniforme, Transformación Wavelet (*DWT 1D Nivel 2* y *DWT 2D*) y Codificación Huffman.

Tabla 3.16: Tiempo de descompresión, con esquema de compresión sin etapa de transformación.

	Tiempo [*ms*]		
Dataset	GTX 760M	Quadro K4200	Tesla K40
1	15.550	2.402	1.622
2	14.867	2.281	1.523
3	10.341	1.646	1.191
4	4.206	0.759	0.663

Tabla 3.17: Tiempo de descompresión, con esquema de compresión con *DWT 1D Nivel 2*.

	Tiempo [*ms*]		
Dataset	GTX 760M	Quadro K4200	Tesla K40
1	11.393	1.974	1.303
2	11.042	1.890	1.252
3	9.351	1.824	1.372
4	4.684	1.118	0.975

Tabla 3.18: Tiempo de descompresión, con esquema de compresión con *DWT 2D*.

	Tiempo [*ms*]		
Dataset	GTX 760M	Quadro K4200	Tesla K40
1	10.496	2.029	1.297
2	11.527	2.434	1.686
3	10.366	2.085	1.602
4	5.222	1.241	0.978

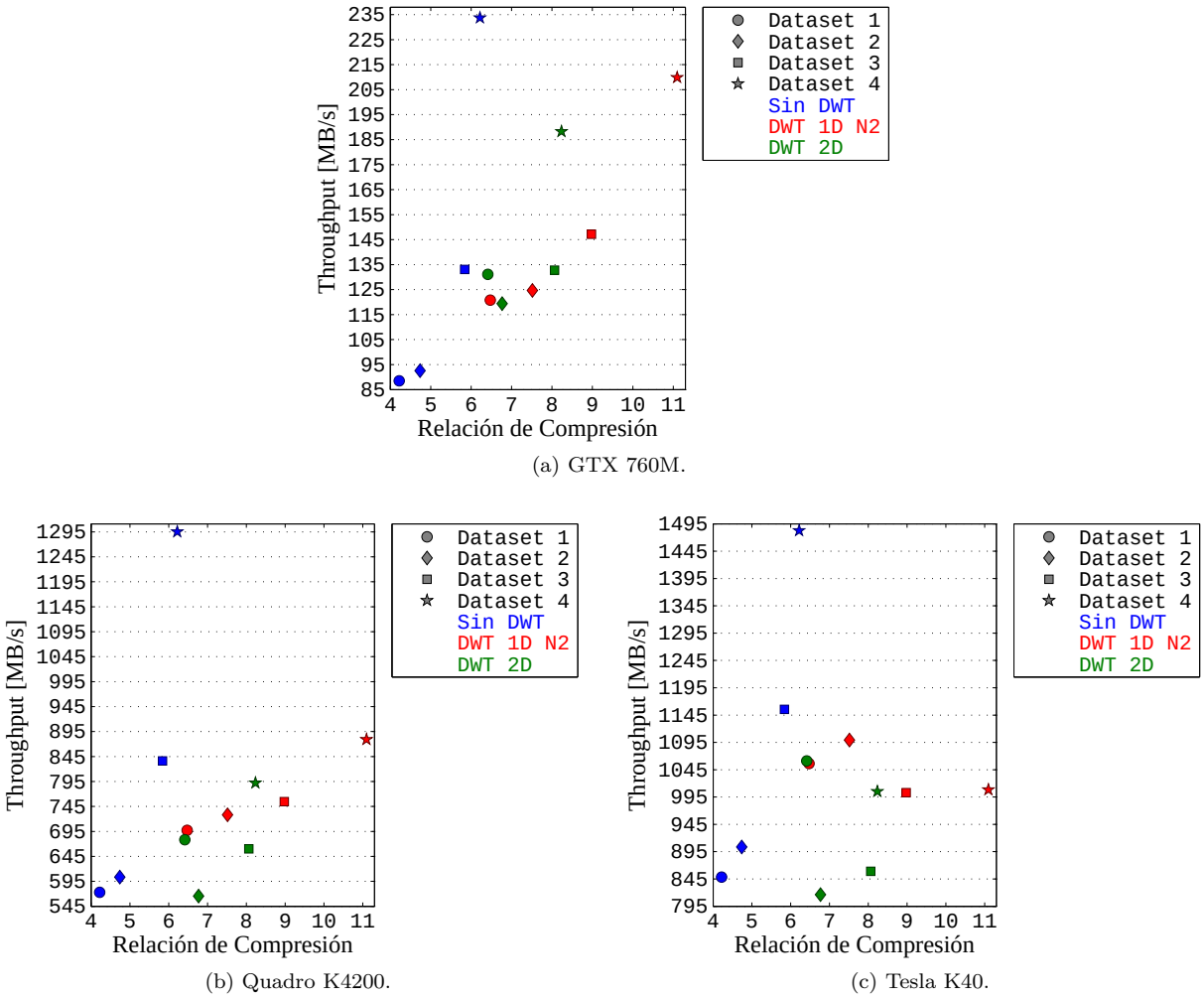
La relación de compresión obtenida por la estrategia de compresión plantea para una descompresión en paralelo se muestra en la Tabla 3.19.

Tabla 3.19: Relación de compresión para la compresión con y sin transformación.

	Relación de Compresión		
Dataset	Sin DWT	DWT 1D Nivel 2	DWT 2D
1	4.220	6.472	6.413
2	4.737	7.514	6.768
3	5.839	8.977	8.063
4	6.219	11.096	8.235

La Figura 3.27, muestra el *Throughput* de descompresión contra la relación compresión. El punto de interés para el esquema de compresión-descompresión se encuentra en la parte derecha superior. Se observó que el punto de interés se logra en la mayoría de los casos cundo el conjunto de datos es comprimido con la etapa de transformación *DWT 1D Nivel 2*.

Observe que el tiempo de decodificación representa el 90% del tiempo de descompresión para el esquema compuesto por la *IDWT 1D* y el 85% para el compuesto por la *IDWT 2D*, por lo cual, los tiempos



(a) GTX 760M.

(b) Quadro K4200.

(c) Tesla K40.

Figura 3.27: *Throughput* descompresión vs relación de compresión.

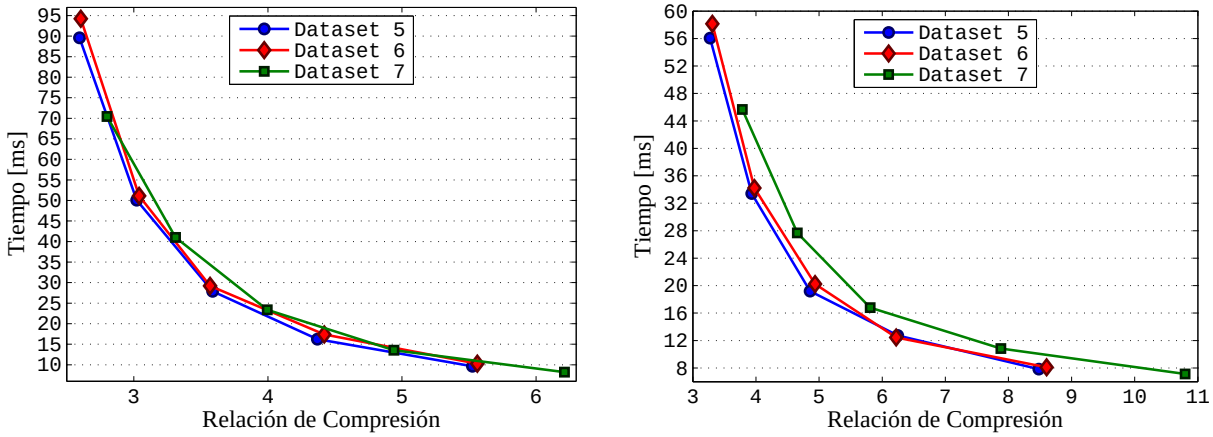
empleados en la etapa de transformación inversa afecta de manera leve al tiempo total de descompresión, por lo tanto, al incluir la etapa de transformación se aumenta la *RC* y se disminuye el tiempo del *kernel* de decodificación, como consecuencia se disminuye el tiempo de descompresión.

Sin embargo, este comportamiento no se observó en el *Dataset 4*, debido a la perdida de ocupación y menor *IPC* de la GPU, presentada por la menor cantidad de paquetes de compresión, sumado a la divergencia del algoritmo de decodificación, el tiempo de decodificación se incrementa al incluir la etapa de transformación sobre todo en la Tesla K40. Análisis posteriores sobre esta observación particular arrojaron que con la etapa de trasformación la mayor ocupación y eficiencia de los SM de esta GPU, se logra con una distribución de 96×1 hilos por bloque, por lo tanto, se mejora el desempeño del *kernel* de decodificación.

Para analizar el desempeño de la descompresión cuando se varia la *RC*, se comprimió otros tres conjuntos de datos diferentes a los trabajados hasta ahora, todos con el mismo número de muestras (344064), para

ello se realizaron dos pruebas, en la primera se comprimieron los datos sin la etapa de transformación y en la segunda se usó la etapa *DWT 2D*; los datos se cuantificador desde 10 bits hasta 14 bits, ambas pruebas se ejecutaron en la GTX 760M.

Los resultados obtenidos se muestran en la Figura 3.28. Se observa que a medida que el tiempo de descompresión disminuye aumenta la relación de compresión, se puede decir que existe una relación inversa entre estos dos parámetros. También se observa que el uso de la etapa de transformación favorece al tiempo de descompresión como a la relación de compresión.



(a) Sin etapa de transformación.

(b) Con etapa *DWT 2D*.

Figura 3.28: Tiempo de descompresión para diferentes *Datasets*.

Al comparar los tiempos de descompresión de los *Datasets* con el mismo número de muestras, se observó que estos tiempos varían dada la naturaleza de cada conjunto de datos lo cual conlleva a un diccionario Huffman diferente y por lo tanto, a tiempos de búsqueda diferentes.

Capítulo 4

Conclusiones

La implementación de algoritmos de compresión-descompresión en arquitecturas de altas prestaciones aún tiene varios retos computacionales, especialmente por el alto costo computacional que presenta la etapa de decodificación. En esta investigación se evaluó y analizó el proceso de compresión-descompresión con el fin de determinar cuál estrategia de codificación ofrece los mejores resultados en términos globales de rendimiento computacional y relación de compresión.

Se evaluaron tres algoritmos de codificación de entropía (Huffman, Aritmética y Tunstall), para la compresión de datos sísmicos en un esquema de compresión con pérdidas compuesto por tres etapas: Transformación, Cuantificación y Codificación. Para las dos primeras etapas se usaron transformación *DWT* (1D y 2D, con uno y dos niveles de descomposición) y cuantificación uniforme. Bajo este esquema se determinó el rendimiento computacional y la relación de compresión para cada uno de los algoritmos.

Nuestros resultados evidenciaron que la codificación Huffman ofrece los mejores resultados globales, es decir, este esquema de codificación ofrece una alta relación de compresión y el mejor *throughput* de descompresión indistinto si se usa o no la etapa de transformación, viéndose favorecida por ésta. Se evidenció que la inclusión de una etapa de transformación (Transformación Wavelet), mejora por lo menos en 1.5 veces la relación de compresión para todos los casos. Asimismo, los resultados permitieron establecer que se obtiene mejor *SNR* al utilizar la *DWT 1D Nivel 1* en el esquema de compresión, sin embargo, la *DWT 1D Nivel 2* presenta la mejor relación *SNR vs RC*.

La principal limitante en el proceso de descompresión se encuentra en la etapa de decodificación. A pesar de los esfuerzos realizados para acelerar la etapa de decodificación (23x), esta etapa representa al menos el 85% del tiempo total empleado para la descompresión. Los principales factores que afectan el rendimiento son: la divergencia, la búsqueda secuencial en el diccionario y la posible sobrecarga en la GPU al procesar los códigos a nivel de bits, ya que, esta arquitectura está optimizada para trabajar en formato de punto flotante.

Se presentaron varias estrategias para la implementación de la decodificación Huffman en una arquitectura GPU. En estas estrategias se hace uso de paquetes con cabeceros, de tal manera que los códigos en el límite de los paquetes son forzados a alinearse. Esto es, si uno de los códigos no puede ser guardado por

completo en un paquete, éste se almacenará en el siguiente paquete, lo cual implica que no se usen algunos bits en algunos paquetes. De esta forma, las estrategias usadas permitieron aprovechar la capacidad de cómputo en paralelo de la arquitectura GPU. La estrategia propuesta permite que cada uno de los paquetes pueda ser decodificado de forma independiente por diferentes hilos de la GPU. Los resultados evidenciaron que al usar la estrategia de almacenamiento en paquetes de 32-bits, concatenados con el vector índice, ofrece los mejores resultados en términos de tiempo de procesamiento.

Una desventaja de la estrategia propuesta es la reducción de la relación de compresión al compararse con la codificación Huffman convencional. Se encontró que la estrategia de paquetes de 32-bits, sin vector de índice ofrece la mayor relación de compresión y un tiempo de decodificación similar a la implementación más rápida.

Se evidenció una relación entre el rendimiento del decodificador Huffman, la longitud del diccionario y la cantidad de símbolos en cada paquete. La decodificación de los paquetes en cada uno de los hilos presenta un tiempo de ejecución diferente, ya que, cada paquete tiene un número diferente de símbolos codificados. Esta diferencia en el tiempo de decodificación genera divergencia. Los resultados muestran que se requiere más tiempo para decodificar los paquetes que tienen almacenados menos símbolos codificados, es decir, aquellos que tienen códigos de mayor longitud. Esto se debe a que estos códigos están al final del diccionario y exigen más tiempo de búsqueda. La longitud media de los códigos disminuye a medida que la relación de compresión aumenta, por esta razón, el rendimiento del algoritmo de descompresión mejora a medida que aumenta la relación de compresión.

Los resultados experimentales evidenciaron que el tiempo de ejecución de los algoritmos para cada una de las etapas de la descompresión depende de la distribución de hilos por bloque con la cual se ejecuta el algoritmo. En este sentido, se logró establecer la distribución que permite el mejor rendimiento computacional para cada una de las etapas.

Trabajo Futuro

Futuras investigaciones podrían girar en torno a encontrar alternativas de compresión de datos sísmicos que no utilicen algoritmos de codificación basados en la entropía. Aunque estas *nuevas* alternativas podrían reducir la relación de compresión, esta disminución se podría compensar con aspectos relacionados con el costo computacional. Por ejemplo, el uso de algoritmos *Matching Pursuit* podrían ofrecer una alternativa de compresión, con el beneficio de no tener que descomprir los datos para poder realizar el procesamiento sísmico.

Nuevas implementaciones del proceso de compresión-descompresión basados en la codificación Huffman, en arquitecturas GPU, podrían tener un cuenta las siguientes sugerencias:

- Almacenar los datos codificados mediante la estrategia de paquetes de 32-bits concatenados con el vector de índice en variables de 64 bits y organizados de acuerdo al número de símbolos en los

paquetes, lo anterior con el fin de reducir el tiempo de procesamiento.

- Usar la estrategia de paquetes de 64-bits sin vector de índice, con el fin de mejorar la relación de compresión.
- Emplear la estrategia de paquetes de 32-bits sin vector de índice si se desea un equilibrio entre tiempo de descompresión y relación de compresión.

Referencias

[1] Fajardo Carlos, Villar Javier Castillo, and Pedraza César, "Reducción de los tiempos de cómputo de la migración sísmica usando FPGAs y GPGPUs: Un artículo de revisión," *Ingeniería y Ciencia*, vol. 9, no. 17, pp. 261–293, 2013.

[2] Wang Xi-zhen, Teng Yun-tian, Gao Meng-tan, and Jiang Hui, "Seismic data compression based on integer wavelet transform," *Acta Seismologica Sinica*, vol. 17, no. 1, pp. 123–128, 2004.

[3] Al-Moohimeed MA, "Towards an efficient compression algorithm for seismic data," en *Radio Science Conference, 2004. Proceedings. 2004 Asia-Pacific*, pp. 550–553, IEEE, 2004.

[4] Salomon David, *Data compression: the complete reference*. Springer London, 2007.

[5] Sayood Khalid, *Introduction to data compression*. Newnes, 2012.

[6] Averbuch Amir Z, Meyer F, Stromberg J-O, Coifman R, and Vassiliou Anthony, "Low bit-rate efficient compression for seismic data," *IEEE transactions on image processing*, vol. 10, no. 12, pp. 1801–1814, 2001.

[7] Wu Wenbo, Yang Zhigao, Qin Qianqing, and Hu Fuxiang, "Adaptive seismic data compression using wavelet packets," en *2006 IEEE International Symposium on Geoscience and Remote Sensing*, pp. 787–789, IEEE, 2006.

[8] Xie Xing and Qin Qianqing, "Fast lossless compression of seismic floating-point data," en *Information Technology and Applications, 2009. IFITA'09. International Forum on*, vol. 1, pp. 235–238, IEEE, 2009.

[9] Shannon Claude Elwood, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423,623–656, 1948.

[10] Fajardo Carlos, Reyes Oscar, and Ramirez Ana, "Seismic data compression using 2d lifting-wavelet algorithms," *Ingeniería y Ciencia*, vol. 11, no. 21, pp. 221–238, 2015.

[11] Averbuch Amir Z, Meyer F, Stromberg J-O, Coifman R, and Vassiliou Anthony, "Low bit-rate efficient compression for seismic data," *IEEE transactions on image processing*, vol. 10, no. 12, pp. 1801–1814, 2001.

[12] Andra Kishore, Chakrabarti Chaitali, and Acharya Tinku, "A vlsi architecture for lifting-based forward and inverse wavelet transform," *IEEE Transactions on Signal Processing*, vol. 50, no. 4, pp. 966–977, 2002.

[13] Zheng Fan and Liu Shufen, "A fast compression algorithm for seismic data from non-cable seismographs," en *Information and Communication Technologies (WICT), 2012 World Congress on*, pp. 1215–1219, IEEE, 2012.

[14] Sánchez Fabián, Fajardo Carlos A, Angulo Carlos A, Reyes Óscar M, and Bouman Charles A, "A computational architecture for discrete wavelet transform using lifting scheme," en *2014 XIX Symposium on Image, Signal Processing and Artificial Vision*, pp. 1–4, IEEE, 2014.

[15] Fugal D. Lee, *Conceptual Wavelets In Digital Signal Processing*. Space & Signals Techonologies LLC, 2009.

[16] Weeks Michael, *Digital Signal Processing Using MATLAB & Wavelets*. Jones & Bartlett Learning, 2010.

[17] Hamdi M., "Parallel architectures for wavelet transforms," en *Computer Architectures for Machine Perception, 1993. Proceedings*, pp. 376–384, Dec 1993.

[18] Abramson Norman, *Teoriá de la Información y Codificación*. Paraninfo, 1981.

[19] Huffman D. A., "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, pp. 1098–1101, Sept 1952.

[20] Howard Paul G and Vitter Jeffrey Scott, "Arithmetic coding for data compression," *Proceedings of the IEEE*, vol. 82, no. 6, pp. 857–865, 1994.

[21] Tunstall B.P., *Synthesis of Noiseless Compression Codes*. Georgia Institute of Technology, 1967.

[22] Fabris Francesco, Sgarro Andrea, and Pauletti Rudy, "Tunstall adaptive coding and miscoding," *IEEE Transactions on Information Theory*, vol. 42, no. 6, pp. 2167–2180, 1996.

[23] Savari Serap A and Gallager Robert G, "Generalized tunstall codes for sources with memory," *IEEE Transactions on Information Theory*, vol. 43, no. 2, pp. 658–668, 1997.

[24] Judd I. D., "Compression of binary images by stroke encoding," *Computers and Digital Techniques, IEE Journal on*, vol. 2, pp. 41–48, February 1979.

[25] Shapiro Stephen D, "Use of the hough transform for image data compression," *Pattern Recognition*, vol. 12, no. 5, pp. 333–337, 1980.

[26] Nill N, "A visual model weighted cosine transform for image compression and quality assessment," *IEEE Transactions on communications*, vol. 33, no. 6, pp. 551–557, 1985.

[27] Zettler William R, Huffman John C, and Linden David CP, "Application of compactly supported wavelets to image compression," en *Electronic Imaging'90, Santa Clara, 11-16 Feb'92*, pp. 150–160, International Society for Optics and Photonics, 1990.

[28] Wallace G. K., "The jpeg still picture compression standard," *IEEE Transactions on Consumer Electronics*, vol. 38, pp. xviii–xxxiv, Feb 1992.

[29] Aravind R., Civanlar M. R., and Reibman A. R., "Packet loss resilience of mpeg-2 scalable video coding algorithms," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, pp. 426–435, Oct 1996.

[30] Ono S. and Suzuki J., "Perspective for super-high definition image systems," *IEEE Communications Magazine*, vol. 34, pp. 114–118, Jun 1996.

[31] Bernas M., Pata P., and Hudec R., "Lossless and lossy compression of images from the omc experiment of integral project," *Astrophysical Letters and Communications*, vol. 39, no. 1-6, pp. 429–432, 1999.

[32] "Overview of JPEG." `https://jpeg.org/jpeg/index.html`, 1992. [Online] Accedido: Febrero de 2015.

[33] Wood Lawrence C., "Seismic data compression methods," *GEOPHYSICS*, vol. 39, no. 4, pp. 499–525, 1974.

[34] Spanias A. S., Jonsson S. B., and Stearns S. D., "Transform coding algorithms for seismic data compression," en *IEEE International Symposium on Circuits and Systems*, pp. 1573–1576 vol.2, May 1990.

[35] Daubechies Ingrid *et al.*, *Ten lectures on wavelets*, vol. 61. SIAM, 1992.

[36] "Overview of JPEG 2000." `http://jpeg.org/jpeg2000`, 2002. [Online] Accedido: Marzo de 2016.

[37] Bosman Cheryl, Reiter Edmund, *et al.*, "Seismic data compression using wavelet transforms," en *1993 SEG Annual Meeting*, Society of Exploration Geophysicists, 1993.

[38] Villasenor John D, Ergas RA, and Donoho PL, "Seismic data compression using high-dimensional wavelet transforms," en *Data Compression Conference, 1996. DCC'96. Proceedings*, pp. 396–405, IEEE, 1996.

[39] Vassiliou Anthony A and Wickerhouser Mladen V, "Comparison of wavelet image coding schemes for seismic data compression," en *Optical Science, Engineering and Instrumentation'97*, pp. 118–126, International Society for Optics and Photonics, 1997.

[40] Aparna P and David Sumam, "Adaptive local cosine transform for seismic image compression," en *2006 International Conference on Advanced Computing and Communications*, pp. 254–257, IEEE, 2006.

[41] Stearns S. D., "Arithmetic coding in lossless waveform compression," *IEEE Transactions on Signal Processing*, vol. 43, pp. 1874–1879, Aug 1995.

[42] Haugen Daniel, "Seismic data compression and gpu memory latency," 2009.

[43] Aqrawi Ahmed Adnan, *Effects of Compression on Data Intensive Algorithms*. Master thesis, Norwegian University of Science and Technology, 2010.

[44] Calderbank AR, Daubechies Ingrid, Sweldens Wim, and Yeo Boon-Lock, "Wavelet transforms that map integers to integers," *Applied and computational harmonic analysis*, vol. 5, no. 3, pp. 332–369, 1998.

[45] NVIDIA, "CUDA C Programming Guide." `http://docs.nvidia.com/cuda`, 2015. [Online] Accedido: Octubre de 2016.

[46] NVIDIA, "Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210." `http://www.nvidia.com/object/nvidia-kepler.html`, 2014. [Online] Accedido: Julio de 2015.

[47] Cheng J., Grossman M., and Mckercher Ty., *Professional CUDA C Programming*. Wrox, 1 ed., 2014.

[48] Cook Shane, *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[49] Min W., "Analysis on bubble sort algorithm optimization," en *Information Technology and Applications (IFITA), 2010 International Forum on*, vol. 1, pp. 208–211, July 2010.

[50] Diéguez A. P., Amor M., and Doallo R., "Efficient scan operator methods on a gpu," en *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pp. 190–197, Oct 2014.

Bibliografía

Abramson Norman, *Teoriá de la Información y Codificación.* Paraninfo, 1981

Al-Moohimeed MA, "Towards an efficient compression algorithm for seismic data," en *Radio Science Conference, 2004. Proceedings. 2004 Asia-Pacific*, pp. 550–553, IEEE, 2004

Andra Kishore, Chakrabarti Chaitali, and Acharya Tinku, "A vlsi architecture for lifting-based forward and inverse wavelet transform," *IEEE Transactions on Signal Processing*, vol. 50, no. 4, pp. 966–977, 2002

Aparna P and David Sumam, "Adaptive local cosine transform for seismic image compression," en *2006 International Conference on Advanced Computing and Communications*, pp. 254–257, IEEE, 2006

Aqrawi Ahmed Adnan, *Effects of Compression on Data Intensive Algorithms.* Master thesis, Norwegian University of Science and Technology, 2010

Aravind R., Civanlar M. R., and Reibman A. R., "Packet loss resilience of mpeg-2 scalable video coding algorithms," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 6, pp. 426–435, Oct 1996

Averbuch Amir Z, Meyer F, Stromberg J-O, Coifman R, and Vassiliou Anthony, "Low bit-rate efficient compression for seismic data," *IEEE transactions on image processing*, vol. 10, no. 12, pp. 1801–1814, 2001

Bernas M., Pata P., and Hudec R., "Lossless and lossy compression of images from the omc experiment of integral project," *Astrophysical Letters and Communications*, vol. 39, no. 1-6, pp. 429–432, 1999

Bosman Cheryl, Reiter Edmund, *et al.*, "Seismic data compression using wavelet transforms," en *1993 SEG Annual Meeting*, Society of Exploration Geophysicists, 1993

Calderbank AR, Daubechies Ingrid, Sweldens Wim, and Yeo Boon-Lock, "Wavelet transforms that map integers to integers," *Applied and computational harmonic analysis*, vol. 5, no. 3, pp. 332–369, 1998

Cheng J., Grossman M., and Mckercher Ty., *Professional CUDA C Programming*. Wrox, 1 ed., 2014

Cook Shane, *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012

Daubechies Ingrid *et al.*, *Ten lectures on wavelets*, vol. 61. SIAM, 1992

Diéguez A. P., Amor M., and Doallo R., "Efficient scan operator methods on a gpu," en *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pp. 190–197, Oct 2014

Fabris Francesco, Sgarro Andrea, and Pauletti Rudy, "Tunstall adaptive coding and miscoding," *IEEE Transactions on Information Theory*, vol. 42, no. 6, pp. 2167–2180, 1996

Fajardo Carlos, Villar Javier Castillo, and Pedraza César, "Reducción de los tiempos de cómputo de la migración sísmica usando FPGAs y GPGPUs: Un artículo de revisión," *Ingeniería y Ciencia*, vol. 9, no. 17, pp. 261–293, 2013

Fajardo Carlos, Reyes Oscar, and Ramirez Ana, "Seismic data compression using 2d lifting-wavelet algorithms," *Ingeniería y Ciencia*, vol. 11, no. 21, pp. 221–238, 2015

Fugal D. Lee, *Conceptual Wavelets In Digital Signal Processing*. Space & Signals Techonologies LLC, 2009

Hamdi M., "Parallel architectures for wavelet transforms," en *Computer Architectures for Machine Perception, 1993. Proceedings*, pp. 376–384, Dec 1993

Haugen Daniel, "Seismic data compression and gpu memory latency," 2009

Howard Paul G and Vitter Jeffrey Scott, "Arithmetic coding for data compression," *Proceedings of the IEEE*, vol. 82, no. 6, pp. 857–865, 1994

Huffman D. A., "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, pp. 1098–1101, Sept 1952

Judd I. D., "Compression of binary images by stroke encoding," *Computers and Digital Techniques, IEE Journal on*, vol. 2, pp. 41–48, February 1979

Min W., "Analysis on bubble sort algorithm optimization," en *Information Technology and Applications (IFITA), 2010 International Forum on*, vol. 1, pp. 208–211, July 2010

Nill N, "A visual model weighted cosine transform for image compression and quality assessment," *IEEE Transactions on communications*, vol. 33, no. 6, pp. 551–557, 1985

NVIDIA, "CUDA C Programming Guide." `http://docs.nvidia.com/cuda`, 2015. [Online] Accedido: Octubre de 2016

NVIDIA, "Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210." `http://www.nvidia.com/object/nvidia-kepler.html`, 2014. [Online] Accedido: Julio de 2015

Ono S. and Suzuki J., "Perspective for super-high definition image systems," *IEEE Communications Magazine*, vol. 34, pp. 114–118, Jun 1996

"Overview of JPEG." `https://jpeg.org/jpeg/index.html`, 1992. [Online] Accedido: Febrero de 2015

"Overview of JPEG 2000." `http://jpeg.org/jpeg2000`, 2002. [Online] Accedido: Marzo de 2016

Salomon David, *Data compression: the complete reference*. Springer London, 2007

Sánchez Fabián, Fajardo Carlos A, Angulo Carlos A, Reyes Óscar M, and Bouman Charles A, "A computational architecture for discrete wavelet transform using lifting scheme," en *2014 XIX Symposium on Image, Signal Processing and Artificial Vision*, pp. 1–4, IEEE, 2014

Savari Serap A and Gallager Robert G, "Generalized tunstall codes for sources with memory," *IEEE Transactions on Information Theory*, vol. 43, no. 2, pp. 658–668, 1997

Sayood Khalid, *Introduction to data compression*. Newnes, 2012

Shannon Claude Elwood, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423,623–656, 1948

Shapiro Stephen D, "Use of the hough transform for image data compression," *Pattern Recognition*, vol. 12, no. 5, pp. 333–337, 1980

Spanias A. S., Jonsson S. B., and Stearns S. D., "Transform coding algorithms for seismic data compression," en *IEEE International Symposium on Circuits and Systems*, pp. 1573–1576 vol.2, May 1990

Stearns S. D., "Arithmetic coding in lossless waveform compression," *IEEE Transactions on Signal Processing*, vol. 43, pp. 1874–1879, Aug 1995

Tunstall B.P., *Synthesis of Noiseless Compression Codes*. Georgia Institute of Technology, 1967

Vassiliou Anthony A and Wickerhouser Mladen V, "Comparison of wavelet image coding schemes for seismic data compression," en *Optical Science, Engineering and Instrumentation'97*, pp. 118–126, International Society for Optics and Photonics, 1997

Villasenor John D, Ergas RA, and Donoho PL, "Seismic data compression using high-dimensional wavelet transforms," en *Data Compression Conference, 1996. DCC'96. Proceedings*, pp. 396–405, IEEE, 1996

Wallace G. K., "The jpeg still picture compression standard," *IEEE Transactions on Consumer Electronics*, vol. 38, pp. xviii–xxxiv, Feb 1992

Wang Xi-zhen, Teng Yun-tian, Gao Meng-tan, and Jiang Hui, "Seismic data compression based on integer wavelet transform," *Acta Seismologica Sinica*, vol. 17, no. 1, pp. 123–128, 2004

Weeks Michael, *Digital Signal Processing Using MATLAB & Wavelets*. Jones & Bartlett Learning, 2010

Wood Lawrence C., "Seismic data compression methods," *GEOPHYSICS*, vol. 39, no. 4, pp. 499–525, 1974

Wu Wenbo, Yang Zhigao, Qin Qianqing, and Hu Fuxiang, "Adaptive seismic data compression using wavelet packets," en *2006 IEEE International Symposium on Geoscience and Remote Sensing*, pp. 787–789, IEEE, 2006

Xie Xing and Qin Qianqing, "Fast lossless compression of seismic floating-point data," en *Information Technology and Applications, 2009. IFITA'09. International Forum on*, vol. 1, pp. 235–238, IEEE, 2009

Zettler William R, Huffman John C, and Linden David CP, "Application of compactly supported wavelets to image compression," en *Electronic Imaging'90, Santa Clara, 11-16 Feb'92*, pp. 150–160, International Society for Optics and Photonics, 1990

Zheng Fan and Liu Shufen, "A fast compression algorithm for seismic data from non-cable seismographs," en *Information and Communication Technologies (WICT), 2012 World Congress on*, pp. 1215–1219, IEEE, 2012