DESEMPEÑO COMPUTACIONAL DEL ALGORITMO DE DECODIFICACION HUFFMAN EN UNA GPU

GABRIEL RINCÓN VERGARA

CHRISTIAN DONALDO HERNÁNDEZ DURÁN

Universidad Industrial de Santander

Facultad de Ingenierías Fisicomecánicas

Escuela de Ingeniería Eléctrica, Electrónica y de Telecomunicaciones

Bucaramanga

2015

DESEMPEÑO COMPUTACIONAL DEL ALGORITMO DE DECODIFICACION HUFFMAN EN UNA GPU

GABRIEL RINCÓN VERGARA

CHRISTIAN DONALDO HERNÁNDEZ DURÁN

Trabajo de Grado para optar al título de

Ingeniero Electrónico

Director :

CARLOS AUGUSTO FAJARDO

Ingeniero Electrónico, PhD(c)

Codirector :

CARLOS ARTURO BOADA QUIJANO

Ingeniero Electrónico

Universidad Industrial de Santander

Facultad de Ingenierías Fisicomecánicas

Escuela de Ingeniería Eléctrica, Electrónica y de Telecomunicaciones

Bucaramanga

2015

AGRADECIMIENTOS

Le agradezco a Dios por haberme acompañado y guiado a lo largo de mi carrera, por ser mi fortaleza en los momentos de debilidad y por brindarme una vida llena de aprendizajes, experiencias y sobre todo felicidad.

Le doy gracias a mis padres Jairo e Isabel por apoyarme en todo momento, por los valores que me han inculcado y por haberme dado la oportunidad de tener una excelente educación en el transcurso de mi vida. Sobre todo por ser un excelente ejemplo de vida a seguir. Un agradecimiento muy especial merece la compresión, paciencia y el ánimo recibidos de mi familia y amigos.

A Colciencias, ICP y CPS por la confianza depositada para la elaboración de esta investigación.

Especial reconocimiento merece el interés mostrado por mi trabajo y las sugerencias recibidas de los profesores Carlos Fajardo, Carlos Boada y demás profesores del grupo CPS, con los que me encuentro en deuda por el ánimo infundido, la confianza en mí depositada y sobre todo su amistad. Quisiera hacer extensiva mi gratitud a los profesores de la especialidad de electrónica del Dámaso Zapata, en especial al profesor Hernán Plata, los cuales me formaron con las bases necesarias para iniciarme en el área de la electrónica.

A todos ellos, muchas gracias.

Christian D.H.D.

AGRADECIMIENTOS

Le agradezco a Magola, mi madre, por su constante e incansable apoyo. A Henry, mi hermano por su valiosa ayuda y sus continuos consejos. A Gabriel, mi padre y a Milena y Adriana, mis hermanas, por estar siempre presentes en todos los momentos. A todos ellos gracias por su esmero y dedicación al brindarme las herramientas necesarias para poder tener la mejor educación posible. Mención especial para Daniela, la mujer que siempre me ha apoyado en todas mis decisiones.

Agradezco a los amigos del colegio, a los amigos de la universidad, a los amigos de la vida y a la familia del rugby. A todos ellos por siempre animarme y acompañarme en las etapas transcurridas de mi vida.

A Colciencias, ICP y CPS por la confianza depositada para la elaboración de esta investigación.

Quiero agradecer también a los docentes. Al profesor Carlos Fajardo, por sus enseñanzas y consejos, al profesor Carlos Boada por sus sugerencias y el interés mostrado en el proyecto. A los docentes que tuve en el colegio y a los docentes con los que vi clases en la universidad, los que siempre pacientes pusieron a disposición todo su conocimiento. A todo el personal de CPS, que siempre estuvieron presentes apoyando en el proceso, brindando toda su confianza y su disposición conmigo.

A los que me apoyaron, gracias totales.

Gabriel R.V.

TABLA DE CONTENIDO

LISTA DE FIGURAS

LISTA TABLAS

RESUMEN

TÍTULO: DESEMPEÑO COMPUTACIONAL DEL ALGORITMO DE DECODIFICACIÓN HUFFMAN EN UNA GPU[1]

Autores: GABRIEL RINCÓN VERGARA[2], CHRISTIAN DONALDO HERNÁNDEZ DURAN[2]

Palabras Claves: Huffman, GPU, Coprocesamiento.

Esta investigación está motivada por la creciente necesidad de procesar grandes volúmenes de información y por las oportunidades que actualmente están ofreciendo las GPUs (Graphics Processing Unit) para realizar dicho procesamiento. Esta arquitectura computacional de altas prestaciones, ofrece la posibilidad de aplicar técnicas de coprocesamiento entre CPU y GPU para distribuir la carga computacional entre la capacidad de cómputo predominantemente secuencial de la CPU y la capacidad paralela de la GPU.

El algoritmo de codificación-decodificación Huffman es uno de los algoritmos más utilizados para reducir la redundancia de información presente en grandes volúmenes de datos [1], con esto, se busca reducir los tiempos de transmisión y procesamiento de la información, sin embargo, es un algoritmo estrictamente secuencial debido a la longitud variable de los símbolos de codificación que genera.

El algoritmo de Huffman concatena los símbolos prefijos de codificación sin identificador y para poder implementar la decodificación mediante técnicas de co-procesamiento CPU-GPU se requiere diseñar e implementar una estrategia que permita paralelar el algoritmo de decodificación.

La investigación permitió implementar el algoritmo de codificación-decodificación Huffman en una CPU y el algoritmo de decodificación Huffman en una GPU. Como fruto del trabajo investigativo, se presentan técnicas de programación que permiten procesar bits en lenguaje C para el proceso de codificación y técnicas que permiten paralelar el algoritmo de decodificación. De este modo, se puede utilizar el diccionario de símbolos prefijos de longitud variable y además, disminuir el tiempo de procesamiento del algoritmo de decodificación.

La implementación que llevamos a cabo, muestra una reducción en el tiempo de procesamiento del algoritmo de decodificación en la GPU. Los resultados experimentales del trabajo realizado muestran que la estrategia paralela propuesta es hasta *16.72x* más rápida que el algoritmo de decodificación ejecutado en una CPU.

[1] Trabajo de Grado modalidad en investigación.

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

ABSTRACT

TÍTLE: COMPUTATIONAL PERFORMANCE HUFFMAN DECODING ALGORITHM ON AN GPU.[1]

AUTORS: GABRIEL RINCÓN VERGARA[2], CHRISTIAN DONALDO HERNÁNDEZ DURAN[2].

KEYWORDS: Huffman, GPU, Seismic Data, Decompression.

This research is motivated by the increasing importance of processing large volumes of information in both science and engineering and, the opportunity that GPU technology offers. This high performance computing architecture offers the possibility to carry out co-design techniques (CPU-GPU), in order to distribute the computational load between the CPU and the GPU.

Handling large volumes of information requires the use of compression algorithms to reduce both the transmission and the processing time. The Huffman algorithm is one of the algorithms more used [1] to reduce the redundancy in the information, however, it is strictly sequential due to it concatenates the symbols prefix-free coding identifier. The implementation of this algorithm, using techniques of co-design at the GPU, requires to design (and implement) a strategy to create a parallel version of the decoding process.

This research seeks to implement the decoding process into a GPU, in order to take advantage of this parallel architecture. As a result of our work, programming techniques were used to process the seismic data at low level (using bits). Both the coding and the decoding process were implemented using C and CUDA-C respectively.

The experimental results show the proposed parallel version improves to *16.72x* the decoding process (in terms of time) when it is compared to the decoding process runs on a CPU.

[1] Research work.

[2] Faculty of Physical-Mechanic Engineerinng. School of Electrical, Electronic and Telecommunications Engineering. Advisor: PhD(c) Carlos Augusto Fajardo Ariza.

Co-advisor: M.Sc. (c) Carlos Arturo Boada Quijano.

INTRODUCCIÓN

En el área de exploración sísmica se generan grandes volúmenes de datos, que se obtienen al modelar el subsuelo mediante técnicas matemáticas y de esta forma identificar posibles reservas de hidrocarburos y reducir los riesgos en una posible perforación y extracción. Actualmente, existen sistemas computacionales potentes, que podrían mejorar el procesamiento de los grandes volúmenes de datos sísmicos obtenidos en la exploración [2].

Sin embargo, la cantidad de datos a procesar es del orden de los terabytes, los costos relacionados con la red y la transmisión por satélite, resultan siendo muy elevados [3]. Por lo tanto, es necesario utilizar técnicas de codificación, las cuales almacenan mayor cantidad de datos en el mismo espacio de memoria y por ende, aumentan la velocidad de transferencia.

El algoritmo de Huffman (codificación-decodificación) es ampliamente utilizado en diversas aplicaciones de procesamiento de datos [4] (texto, imagen o multimedia). A pesar de ser ampliamente utilizado, la decodificación Huffman es un proceso computacionalmente costoso, debido a la longitud variable de los códigos, lo cual hace que el proceso de decodificación sea predominantemente secuencial. En este orden de ideas, se evidencia la necesidad de diseñar e implementar una versión en paralelo del proceso de decodificación Huffman.

El objetivo de este trabajo de investigación consiste en diseñar e implementar una versión paralela del algoritmo de decodificación Huffman; en consecuencia de lo anterior, se busca diseñar e implementar una estrategia de codificación-

decodificación que permita, paralelar un proceso que es estrictamente secuencial. La estrategia implementada consistió en utilizar co-procesamiento entre CPU-GPU, a través de la ejecución del algoritmo de codificación en una CPU y el algoritmo de decodificación en una GPU, por medio de codificación en paquetes con cabeceras.

El proyecto hace parte de una investigación doctoral que se está realizando actualmente en la Universidad Industrial de Santander en convenio con ICP-Colciencias, en la cual se está buscando encontrar nuevas alternativas de software y hardware para optimizar el rendimiento de algoritmos utilizados en el proceso de exploración petrolera, los cuales presentan un alto costo computacional.

Este documento está organizado de la siguiente manera: El capítulo 1 explica brevemente la codificación-decodificación Huffman. Capítulo 2 describe el trabajo realizado. El capítulo 3 contiene los resultados y el análisis de los mismos y el capítulo 4, presenta las conclusiones y las posibilidades de trabajo futuro.

1. TRATAMIENTO DE DATOS SÍSMICOS: ALGORITMO DE CODIFICACIÓN Y DECODIFICACIÓN HUFFMAN

Actualmente en cualquier actividad científica, técnica o profesional, se plantea la necesidad de almacenar y procesar grandes volúmenes de información. La codificación de datos, es un proceso utilizado para reducir el número de bits necesarios para representar la información y de esta forma, optimizar el almacenamiento de los datos permitiendo el procesamiento de un mayor volumen de información [3] [4].

En los datos sísmicos existen tres componentes de información: información geofísica, redundancia de información y ruido no correlacionado [5]. El ruido no correlacionado es inherente al proceso de medición y va a existir en presencia o ausencia de la información [6] y la información geofísica contiene la descripción del subsuelo analizado; teniendo en cuenta esto, las estrategias de compresión de datos sísmicos se enfocan en reducir la redundancia presente en la información analizada [3].

De acuerdo al teorema de Shannon [7], un conjunto de datos $(d_1, d_2, \dots\dots, d_n)$ puede comprimirse hasta nE bits en promedio, donde E es la entropía, una cantidad que determina el número promedio de bits necesarios para representar un grupo de datos y n, el número total de datos. La ecuación 1 representa el cálculo de la entropía, donde P_i es la probabilidad de ocurrencia de d_i.

$$E = -\sum_{i=1}^{n}(P_i log_2 P_i) \quad (1)$$

La entropía es mayor, cuando todas las N probabilidades de aparición de los datos son similares y es menor, a medida que las probabilidades de todos los datos difieren unas a otras. Esta consideración es utilizada para definir la redundancia (R) en los datos. La redundancia (R) se define como la diferencia entre el número de bits utilizados para representar la información (M_b) y el número mínimo necesario para representar todo el conjunto de datos [7] (ver ecuación 2).

$$R = M_b + \sum_{i=1}^{n}(P_i log_2 P_i) \quad (2)$$

Es decir, la redundancia es la cantidad de bits innecesarios utilizados en la representación de los datos. Una forma cuantitativa de medir el proceso de compresión comúnmente utilizada, es el Factor de Compresión (FC). El FC se define como la razón entre el número de bits sin comprimir ($D_{sincomp}$) sobre el número de bits de los datos comprimidos (D_{comp}) (ver ecuación 3).

$$FC = \frac{D_{sincomp}}{D_{comp}} \quad (3)$$

Una de las técnicas más utilizadas para reducir la redundancia de información es el algoritmo de codificación-decodificación Huffman [8]. Dicho algoritmo crea un diccionario de codificación de longitud variable a partir de un árbol binario que reduce el número de bits que representan la información en función de la probabilidad de aparición de cada dato.

La construcción del algoritmo de Huffman, se inicia con n datos agrupados, junto con sus frecuencias (o probabilidad) de aparición. Estos datos son ordenados en función de su frecuencia (o probabilidad) de aparición, de menor a mayor; luego

de esto, se inicia la construcción de un árbol binario, formando parejas de datos con base a su frecuencia. Al agrupar los datos de esta forma, se asignan símbolos cortos a los datos más frecuentes y símbolos largos a los datos menos frecuentes.

Pasos para obtener el diccionario de codificación a partir del algoritmo de Huffman [9] son:

1. Agrupar los datos con respecto a su frecuencia (o probabilidad) de aparición.
2. Para empezar a crear el árbol binario, se ubican en la raíz los datos con sus frecuencias originales, ordenadas de menor a mayor.
3. Se agrupan de a pares las frecuencias de cada dato, subiendo de nivel desde la raíz (base) hasta la copa (tope) del árbol. El valor que tendrá cada hoja será la suma de las dos frecuencias del nivel inferior.
4. Al llegar a la copa, luego de sumar todas las frecuencias de cada dato y obtener el valor de frecuencia igual al número de datos total, se procede a construir el diccionario de codificación.
5. Para crear el diccionario de codificación, se debe asignar un 1 o un 0 a cada lado de las hojas del árbol, desde la copa hasta la raíz. Este dígito binario, se debe asignar de forma simétrica, es decir, si se asigna un 1 a la derecha de la copa, se debe asignar un 1 al lado derecho de todas las hojas, lo mismo sucede con el 0, el cual se asignará al lado izquierdo de la copa. Elegir la posición del 0 o el 1 no afecta la creación del diccionario, siempre y cuando se mantenga la simetría de la elección.

6. La lectura del diccionario de codificación se hace recorriendo el árbol, desde la copa hasta llegar a la raíz del dato deseado, obteniendo el símbolo de codificación con los unos y ceros correspondientes al recorrido.

La codificación se realiza al reemplazar cada dato por el símbolo obtenido tras crear el diccionario de codificación Huffman. Esta codificación es de códigos prefijos, esto significa que cada símbolo es único para cada dato, lo cual permite codificar la información de forma concatenada sin ningún mensaje adicional para distinguir las palabras del mensaje codificado.

El proceso de decodificación se hace utilizando el diccionario creado en la codificación y consiste en obtener los datos originales a partir de los símbolos codificados, teniendo en cuenta la longitud variable de cada símbolo prefijo para recuperar la información. Este proceso de decodificación se logra al comparar cada símbolo prefijo del diccionario con la información codificada, reemplazando cada dato codificado por el dato original representado por el símbolo de codificación.

Para evaluar la eficiencia del algoritmo de Huffman se debe comparar la longitud en bits tanto del símbolo de codificación asociada al dato y el espacio de memoria ocupado por el dato; si el tamaño del vector de bits codificado es menor que el vector de bits original, el proceso de compresión fue adecuado [9].De otra forma, el Factor de Compresión debe ser igual o mayor a 1, para verificar que el procesamiento de la información fue exitoso y eficiente.

2. TRABAJO REALIZADO

A continuación se relatarán los pasos seguidos durante el desarrollo de la presente investigación. El desarrollo está dividido en 4 etapas diferenciadas y por ello se detallarán por separado.

2.1 CREACIÓN DEL DICCIONARIO DE CODIFICACIÓN HUFFMAN

Como se mencionó en el capítulo 1, la creación del diccionario de codificación Huffman, se inicia obteniendo la frecuencia (o probabilidad) de aparición de cada dato dentro de la información a analizar. Los datos resultantes son ordenados en una lista según su frecuencia, desde el menos frecuente hasta el más frecuente (Ver Tabla 1). Luego de crear la lista, se comienza a construir un árbol binario, el cual agrupa en pares los datos menos frecuentes de la lista mencionada, creando así, un dato que tendrá como valor de frecuencia la suma de las 2 frecuencias precedentes; los datos utilizados en el anterior proceso son reemplazados por el dato creado y la lista se reorganiza según lo mencionado anteriormente. Este proceso se hace repetitivo hasta llegar a la copa del árbol, en donde la frecuencia de aparición es el número total de datos a analizar [9].

El proceso descrito anteriormente, se segmentó en funciones que al ser ejecutadas de forma secuencial, construirían el árbol binario. Antes de iniciar la ejecución de estas funciones, se debe definir el tamaño total de los datos sobre los cuales se va a operar, por lo cual, se creó una constante con este valor.

Durante el proceso de creación del árbol binario, es necesario agrupar la información, debido a esto, se hizo uso de estructuras, con el fin de aprovechar los recursos ofrecidos por el lenguaje C.

El árbol binario se compone esencialmente de tres elementos: la raíz, las hojas (nodos) y la copa. Cada posición del árbol contiene una probabilidad asociada, la cual deberá ser constante una vez sea asignada. Como resultado de lo anterior, se hace necesario hacer distinción entre los datos de la lista, puesto que unos son los originales y otros son el resultado de la unión de estos, por lo tanto, el dato producto de la unión de dos datos tendrá el carácter "*$*".

Cuando el algoritmo indica que ya se ha llegado a la copa del árbol, se inicia el proceso de creación del símbolo para cada dato. Dado que cada parte del árbol tiene dos posiciones (derecha e izquierda), a cada posición se le deberá asignar un valor, es decir, a todas las posiciones derechas se les asignará el mismo valor binario, de igual forma sucede con las posiciones izquierdas; siendo la codificación Huffman una codificación binaria, el valor a asignar únicamente será uno o cero, por lo que se decidió asignar a las posiciones izquierdas el valor de 1 y a las posiciones derechas el valor de 0. El diccionario se crea al recorrer las estructuras mencionadas, almacenando la etiqueta de posición binaria designada, desde la copa hasta la raíz del árbol. Este proceso se realiza hasta que todos los datos a codificar presentes en la raíz, tengan la codificación correspondiente (ver figura 1).

Figura 1. Ejemplo de árbol binario según Huffman.

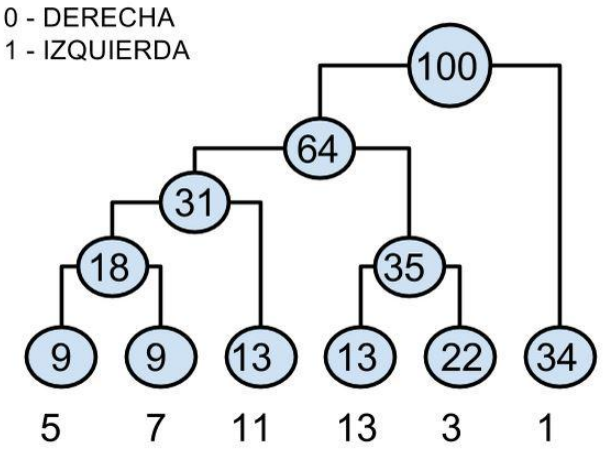


Tabla 1. Ejemplo de diccionario para la figura 1.

Dato	**1**	**3**	**11**	**13**	**5**	**7**
Símbolo de codificación	**0**	**100**	**110**	**101**	**1111**	**1110**
Número de bits del símbolo	**1**	**3**	**3**	**3**	**4**	**4**

Como resultado del proceso descrito anteriormente, el diccionario de codificación Huffman está conformado por: datos, símbolo Huffman asociado a cada dato y el número de bits de cada símbolo (Ver Tabla 1). Teniendo en cuenta que los arreglos mencionados anteriormente son de 32 bits cada uno, la información de un dato dentro del diccionario, ocupará 96 bits (32*3), por lo que se creó una estrategia con el fin de almacenar esta información con una menor cantidad de bits.

Se debe recalcar, que al utilizar el lenguaje de programación C, la memoria reservada en la ejecución de un algoritmo es designada por defecto, según el tipo de variable que se esté utilizando. En este caso, como los datos de estudio tienen formato tipo INT (32 bits) y los símbolos en la codificación Huffman son de longitud variable, fue necesario diseñar una estrategia para almacenar bit a bit cada símbolo de codificación; por consiguiente, se implementó un bucle for para almacenar de forma secuencial bit a bit de cada símbolo, ingresando estos desde el bit de menor peso hacia el bit de mayor peso, hasta almacenar completamente el símbolo en la variable de 32 bits. Este proceso se repite las veces que sea necesario, almacenando así todos los símbolos en sus respectivas variables de 32 bits. De no haber implementado el procedimiento anterior, el bit asociado a cada nivel del árbol binario, hubiese sido una variable tipo INT (32 bits), lo cual no sería eficiente.

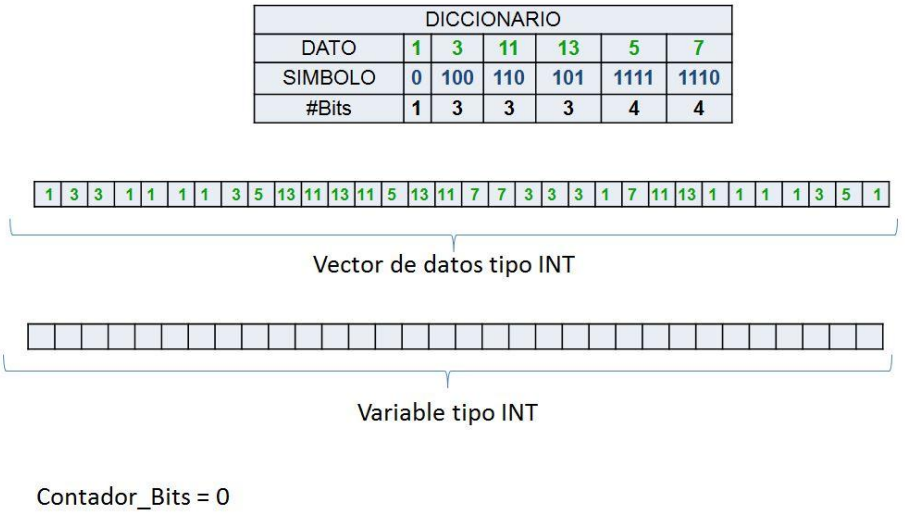
Los resultados de la estrategia implementa muestran que ningún símbolo de codificación será mayor a 27 bits, por lo cual, los 32 bits disponibles se utilizaran de la siguiente forma: los 27 bits de mayor peso almacenaran el símbolo y los 5 bits restantes serán utilizados para guardar la cantidad de bits utilizados, de los 27 bits ya mencionados. Gracias a la estrategia implementada, la información de un dato dentro del diccionario se almacenará en 64 bits (2X32), reduciendo así, aproximadamente un 33% la memoria necesaria para el diccionario.

2.2 ALGORITMO DE CODIFICACIÓN HUFFMAN

2.2.1 Acumulación secuencial Luego de crear el diccionario de codificación Huffman, se procede a reemplazar cada dato estudiado por su respectivo símbolo. Para almacenar los datos codificados, se utiliza una variable de 32 bits de

longitud, la cual se designó acumulador y se usa una variable auxiliar para contar el número de bits ocupados en la variable acumulador, la cual se denominó contador (ver figura 2).

Figura 2. Espacio de memoria reservada y variables utilizadas.



El proceso se inicia comparando cada uno de los datos a codificar de forma secuencial, con cada dato contenido en el diccionario, con el fin de identificar la posición del dato en el diccionario y comenzar el proceso de codificación. Una vez encontrada la posición del dato, se toma el símbolo asociado a ese dato y se guarda en una variable de 32 bits, en la cual, cada símbolo se va acumulando bit a bit. Esta acumulación bit a bit, se realiza ingresando el primer bit del símbolo, por el bit de menor peso de la variable de 32 bits (Ver Fig. 3). Una vez el primer bit se encuentre almacenado, este se desplaza hacia la izquierda dejando lugar al segundo bit del símbolo. Este proceso se realiza cuantas veces sea necesario, hasta guardar todos los bits del símbolo y con cada uno de los datos a codificar.

Figura 3. Ejemplo para el primer dato a codificar y símbolo encontrado en el diccionario.

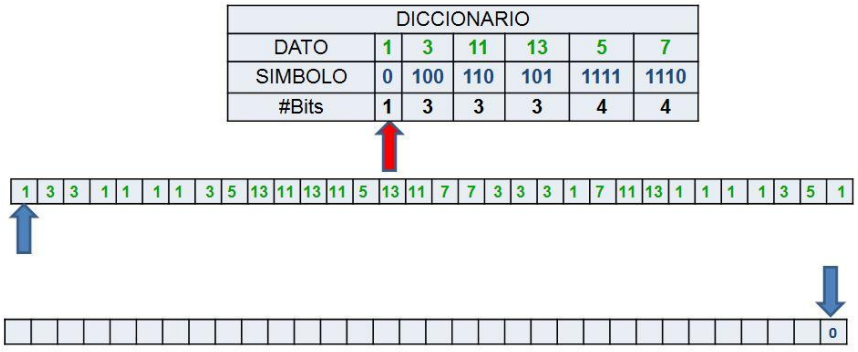
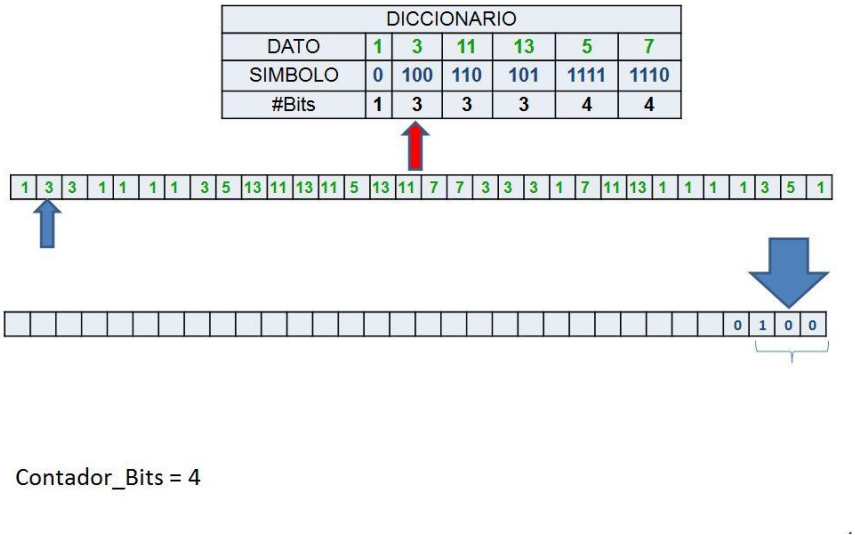


Figura 4. Ejemplo de codificación de un símbolo de 3 bits.

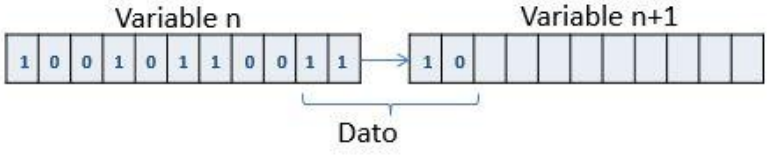


El conteo de los bits utilizados en la acumulación, se hace necesario para que al estar los 27 bits de la primera variable acumulador ocupados, se siga codificando

en una variable sucesiva, esto con el fin de tener el espacio de memoria suficiente para la cantidad de datos a codificar.

Como se puede apreciar, el algoritmo de codificación Huffman es estrictamente secuencial, lo cual implícitamente es un problema para intentar paralelar un algoritmo de decodificación. Es posible que un símbolo tenga más bits que el espacio disponible para almacenarlo, por lo que los bits resultantes de este símbolo serán guardados en los siguientes 32 bits (ver figura 5).

Figura 5. Dato codificado compartido entre dos variables acumuladoras.

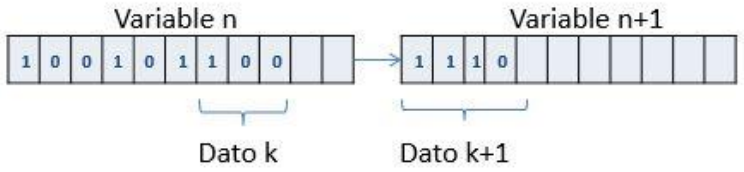


Por lo tanto, al pensar en un algoritmo de decodificación en paralelo, se hace necesario fragmentar los datos codificados en M paquetes, para así utilizar M decodificadores en paralelo. El empaquetamiento restringido se utiliza para evitar errores por perdida de información en la decodificación, debidos a los símbolos compartidos entre dos variables de 32 bits.

2.2.2 Acumulación en paquetes de 32 bits Como se mencionó anteriormente, al querer implementar un algoritmo de decodificación en paralelo, fue necesario fragmentar los datos codificados, por lo cual, se diseñaron e implementaron las siguientes estrategias.

De forma semejante a la codificación secuencial, esta estrategia reemplaza cada dato de estudio por su correspondiente símbolo Huffman y utiliza variables de 32 bits para almacenar los datos codificados. El proceso para guardar bit a bit el símbolo del dato, dentro de la variable de 32 bits, es semejante a la acumulación secuencial, la diferencia radica, en la condición de almacenar N símbolos siempre y cuando estos no superen en conjunto más de 32 bits, evitando así que un símbolo se divida en dos partes y de esta forma implementar un algoritmo de decodificación en paralelo.

Figura 6. Acumulación de símbolos completos en la variable de 32 bits.



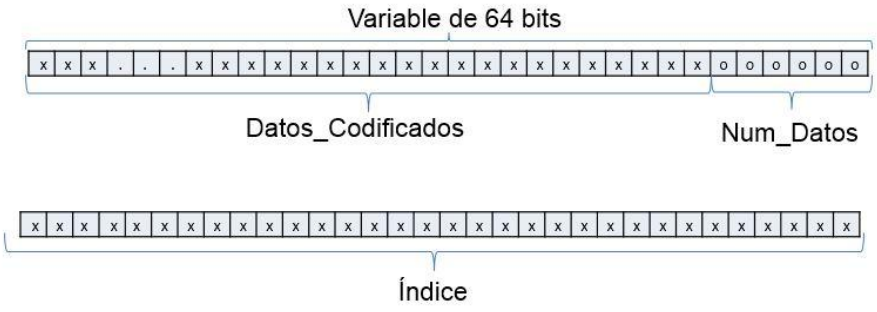
Como se observa en la Figura 6, en la -Variable n- se almacenan bits del símbolo de codificación hasta el bit 30, lo cual ocurre, porque solamente existen 2 bits disponibles, de los 4 bits necesarios para guardar el siguiente símbolo de codificación, por lo cual, este símbolo se amacena en los siguientes 32 bits. Este proceso se repite hasta codificar todos los datos de estudio, utilizando la cantidad necesaria de paquetes para su almacenamiento.

Para implementar un algoritmo de decodificación en paralelo se requiere información adicional, con el fin de direccionar los datos decodificados de cada paquete a sus respectivas posiciones en el puntero de salida. Teniendo en cuenta lo anterior, fue necesario crear un fichero extra denominado índice, en el cual, la posición cero tendrá un valor de 0 y las posiciones siguientes tendrán un valor

igual a la acumulación de datos presentes en cada paquete, de las posiciones anteriores a su posición.

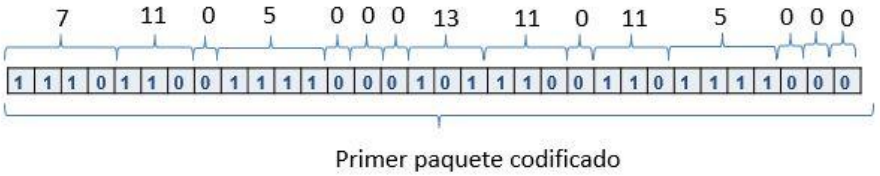
2.2.3 Acumulación en paquetes de 64 bits Del mismo modo que la acumulación en paquetes de 32 bits, esta estrategia reemplaza cada dato de estudio por su correspondiente símbolo Huffman y utiliza variables de 64 bits por las de 32 bits. A su vez, almacena estos símbolos bit a bit de forma secuencial; la condición presente en esta estrategia, consiste en guardar el número total de datos dentro de cada paquete, en los 6 bits de menor peso de los 64 bits disponibles y almacenar P símbolos siempre y cuando estos no superen en conjunto más de 58 bits; Además se crea el fichero (índice) necesario para direccionar los datos decodificados, de igual forma que en la acumulación en paquetes de 32 bits (ver figura 7).

Figura 7. Variables utilizadas en la acumulación en paquetes de 64 bits.



2.3 ALGORITMO DE DECODIFICACIÓN HUFFMAN

Figura 8. Primer paquete de datos codificados.

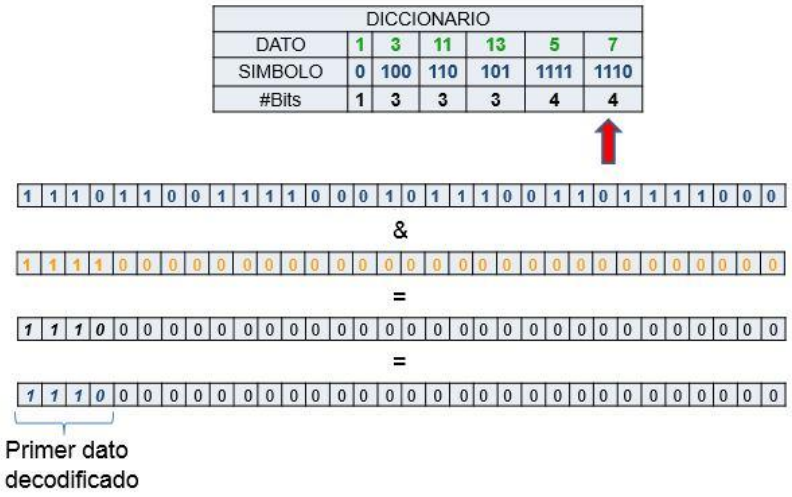


El algoritmo de decodificación Huffman diseñado realiza el proceso inverso a la codificación partiendo de los símbolos de codificación presentes en cada paquete codificado, realiza el proceso de búsqueda en el diccionario y al encontrar coincidencia con el símbolo, decodifica el dato original.

El proceso de comparar cada dato codificado con el símbolo del diccionario, se realiza utilizando una máscara, la cual compara el número de bits del dato codificado y del símbolo del diccionario, si estos tienen la misma cantidad de bits, se procede a comparar el dato codificado y el símbolo, en caso de que estos datos concuerden, se guarda el dato original como representación del dato codificado.

La utilidad de la máscara permite aumentar la eficiencia del algoritmo de decodificación, ya que no se gasta tiempo decodificando bit a bit cada dato, permitiendo así, decodificar un dato completo sin perder información (Ver Fig 9). La latencia de este proceso, está en la ejecución de la búsqueda en el diccionario, para encontrar el símbolo de codificación y su dato correspondiente.

Figura 9. Proceso de decodificación utilizando la estrategia de la máscara.



Luego de decodificar el símbolo correspondiente en el vector de datos codificados, se desplazan bit a bit el número de bits del símbolo del dato decodificado de derecha a izquierda y dichos bits desplazados se reemplazan por los bits del dato codificado siguiente (ver figura 10).

Figura 10. N bits codificados del siguiente dato de codificación.

Este proceso se realiza de forma secuencial, con cada uno de los paquetes de datos codificados. Como se puede observar, es un proceso costoso

computacionalmente, ya que solo se puede decodificar un dato a la vez, aumentando así, la latencia del proceso de decodificación debida a la cantidad de paquetes que contienen los datos que serán decodificados.

2.4 ALGORITMO DE DECODIFICACIÓN HUFFMAN EN PARALELO

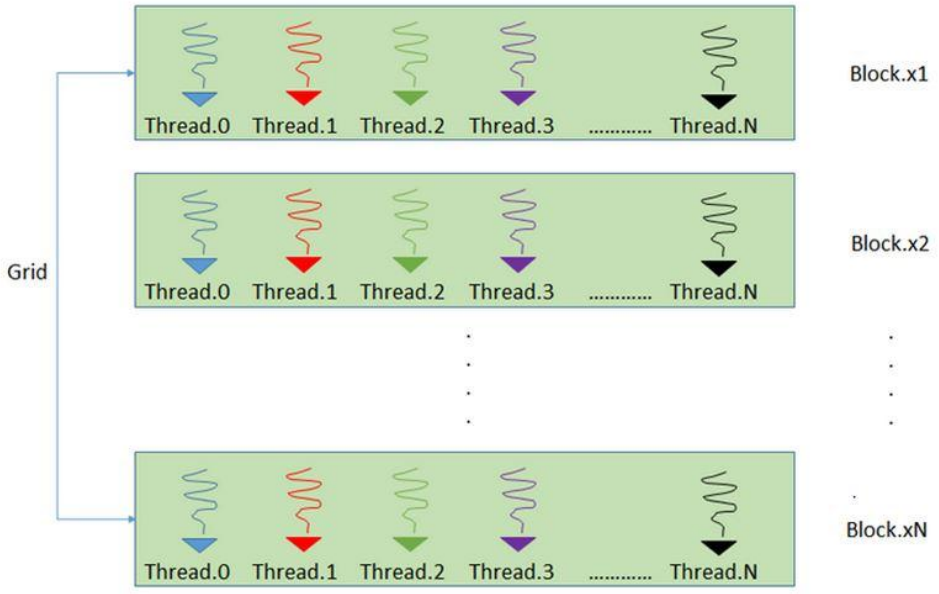
Antes de definir la estrategia utilizada para paralelar el algoritmo de decodificación Huffman, es importante mencionar algunos conceptos importantes sobre la herramienta de procesamiento utilizada y su posterior utilización, teniendo en cuenta que para obtener un mejor rendimiento en la GPU es importante desarrollar la aplicación partiendo de la arquitectura disponible en el dispositivo.

La capacidad de cómputo de una GPU, se define con base a la versión del core de la arquitectura. La capacidad de cómputo 3.X cubre los dispositivos de la familia Kepler [10], la versión 2.X los dispositivos de la familia Fermi [11] y la versión 1.X es para los dispositivos de arquitectura Tesla [12].

La GPU a utilizar tiene arquitectura Kepler y permite realizar operaciones indexando los hilos (threads) en vectores lineales, matriciales o tridimensionales, utilizando las funciones proporcionadas por el lenguaje CUDA-C llamadas threadIdx y blockIdx, generando de esta forma bloques de hilos unidimensionales, bidimensionales o tridimensionales y por lo tanto, se puede procesar información en la GPU organizando bloques de hilos en 1D, 2D y 3D [13].

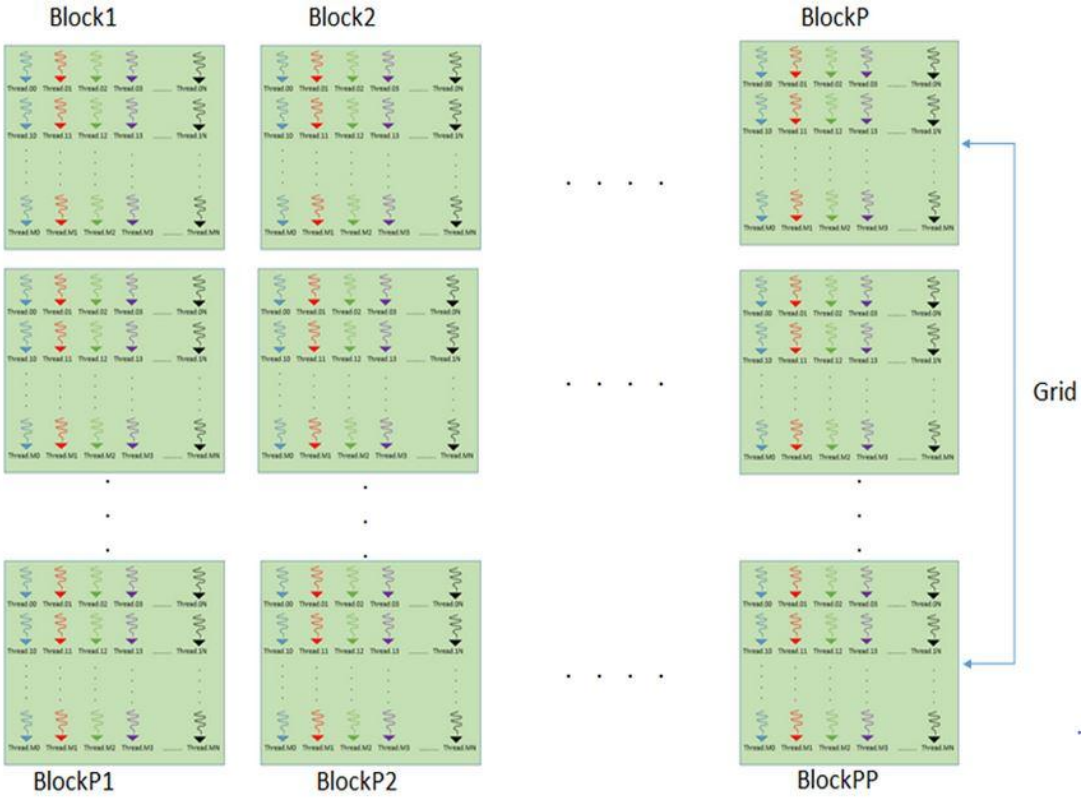
En el proceso de decodificación Huffman en paralelo cada hilo dentro del mismo bloque se comporta como un decodificador, por lo tanto, cada hilo se encarga de decodificar un paquete de datos codificados utilizando el diccionario y recupera los datos originales realizando el proceso de decodificación Huffman. Según la declaración del kernel que se realice, se pueden tener en total 128, 256, 512 o 1024 decodificadores en paralelo por cada bloque de hilos en la GPU [14].

Figura 11. Kernel ejecutado en 1D



Para la invocación del kernel en forma unidimensional se declaran 128, 256, 512 y 1024 hilos por bloque (ver figura 11), para la ejecución del algoritmo de forma bidimensional se declara la invocación del kernel en bloques de hilos de 8x8, 16x16 y 32x32 (ver figura 12) y la declaración tridimensional no se utiliza, dado que el algoritmo de Huffman es un algoritmo secuencial y no tiene el volumen de datos suficientes para realizarlo en 3D.

Figura 12. Kernel ejecutado en 2D.



3. ANÁLISIS Y DISCUSIÓN DE RESULTADOS

3.1 SISTEMAS DE CÓMPUTO UTILIZADOS

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

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

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

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

3.1.3 Puerto PCI-EXPRESS La comunicación entre la CPU y GPU se realiza a través del puerto PCI Express. Este bus de datos es desarrollado por Intel y permite intercambiar información entre dispositivos como tarjetas gráficas, de sonido o de red con el procesador del equipo.

El PCI Express tiene estructura de carriles punto a punto *full dúplex* trabajando en serie. La tabla 2, presenta las versiones 1.1, 2.0 y 3.0 de este puerto. Allí se puede apreciar, que el ancho de banda entre cada versión del bus PCIe, es el doble de la anterior versión, por lo cual al estar la GPU conectada a través del puerto PCI Express 3.0, se tendrá un ancho de banda disponible de 8 GB/s.

Tabla 2. Generaciones de puerto PCI-express [15].

Versión	**Velocidad de transferencia**	**Ancho de banda de interconexión**	**Ancho de banda por línea**	**Ancho de banda máximo**
1.1	2.5 GT/s	2 GB/s	250 MB/s	8 GB/s
2.0	5 GT/s	4 GB/s	500 MB/s	16 GB/s
3.0	8 GT/s	8 GB/s	1 GB/s	32 GB/s

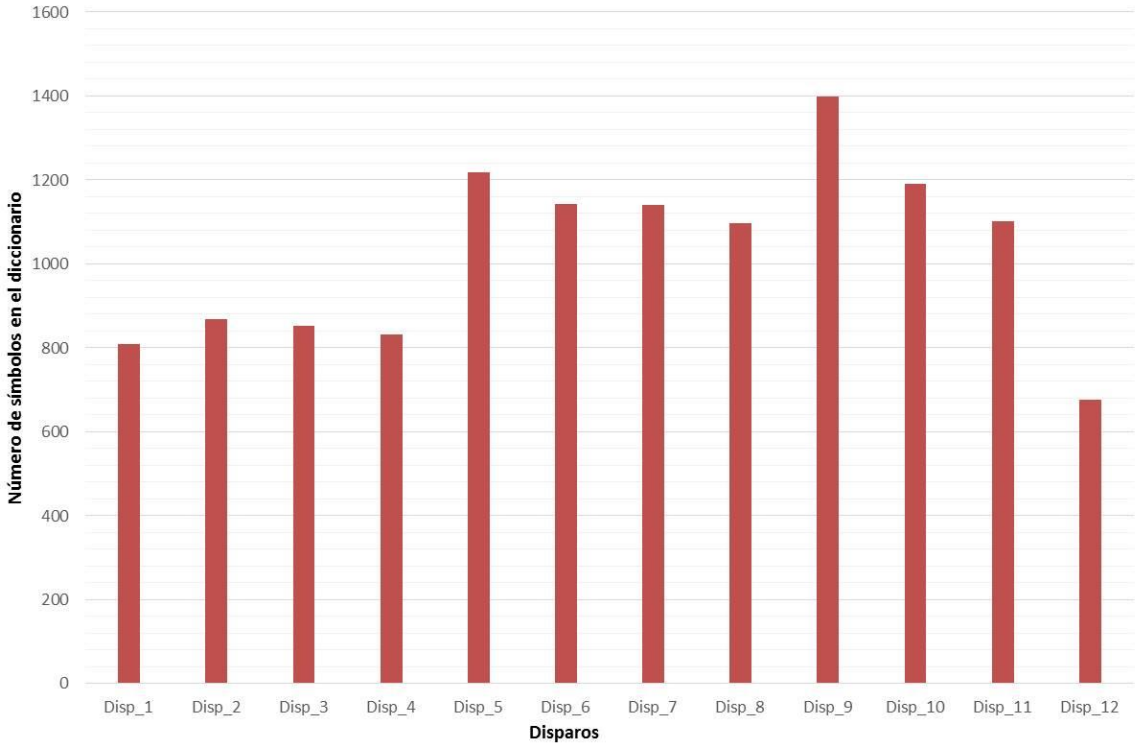
3.2 ALGORITMO DE CODIFICACIÓN

Los datos sísmicos utilizados en la presente investigación corresponden 12 disparos de trazas sísmicas conformados por 96 trazas con 3.584 datos cada una, para un total de 344.064 datos por disparo, guardados en formato tipo INT en archivos binarios.

El algoritmo de codificación Huffman, es un algoritmo de compresión de datos sin perdidas, el cual toma un alfabeto de *n* datos, junto con sus frecuencias de aparición asociadas y produce un símbolo (ver capítulo 1); esta codificación tiene como característica, asignar símbolos cortos a los datos más frecuentes y símbolos largos a los datos menos frecuentes. El diccionario de codificación Huffman, está conformado por: datos, símbolo asociado a cada dato y número de bits de cada símbolo.

La figura 13, muestra la cantidad de datos que contiene el diccionario de codificación generados con el algoritmo de *Creación del Diccionario de Codificación Huffman* (ver sección 2.1). Se puede apreciar que aunque los disparos tengan igual tamaño, la cantidad de datos presentes en el diccionario, varía para cada uno de ellos, dado que la cantidad de datos del diccionario depende de la frecuencia de aparición de cada uno de los datos en cada disparo.

Figura 13. Número de datos en el diccionario.



3.2.1 Estrategias de codificación implementadas En el presente proyecto se diseñaron e implementaron las estrategias de codificación secuencial, codificación en paquetes de 32 bits y codificación en paquetes de 64 bits, cuyos resultados se presentan a continuación:

Algoritmo de Codificación Secuencial: Esta estrategia consiste en reemplazar cada dato del disparo por su correspondiente símbolo Huffman y almacenar bit a bit este símbolo utilizando variables de 32 bits (ver sección 2.2). La característica principal de este proceso, es la utilización completa de estos 32 bits, sin embargo, se puede presentar el caso en el cual un símbolo presente mayor cantidad de bits

que espacio disponible para almacenar, por lo cual, los bits restantes de este símbolo serán guardados en los siguientes 32 bits.

Algoritmo de Codificación en Paquetes de 32 bits: Partiendo de la restricción de que un símbolo de codificación debe tener longitud máxima de 27 bits para que el proceso de codificación sea eficiente, reservando 5 bits para almacenar la cantidad de datos codificados en los 27 bits iniciales. De forma semejante al *Algoritmo de Codificación Secuencial*, esta estrategia reemplaza cada dato del disparo por su correspondiente símbolo Huffman y utiliza variables de 32 bits para almacenar los datos codificados, sin embargo, este proceso debe satisfacer la condición de almacenar *N* símbolos siempre y cuando estos no superen en conjunto más de 27 bits, evitando así que un símbolo se divida en dos partes y de esta forma se podría implementar un proceso de decodificación en paralelo.

Como se mencionó en la *sección 2.3*, la estrategia de codificación en paquetes de 32 bits, se caracteriza por la restricción de almacenar bit a bit, grupos de símbolos siempre y cuando estos no superen en conjunto más de 32 bits. Además de cada paquete de codificación, se requiere conocer el número de símbolos almacenados en cada paquete, por lo cual, fue necesaria la creación de un fichero con esta información.

Siendo los símbolos Huffman de longitud variable, cada paquete de 32 bits no almacena igual cantidad de símbolos, por lo cual, al querer implementar un algoritmo de decodificación en paralelo, se requiere contar con información adicional para direccionar los datos decodificados de cada paquete a sus respectivas posiciones en el puntero de salida. Teniendo en cuenta lo anterior, fue necesario crear un fichero extra en el cual, la posición 0 tendrá un valor de cero y

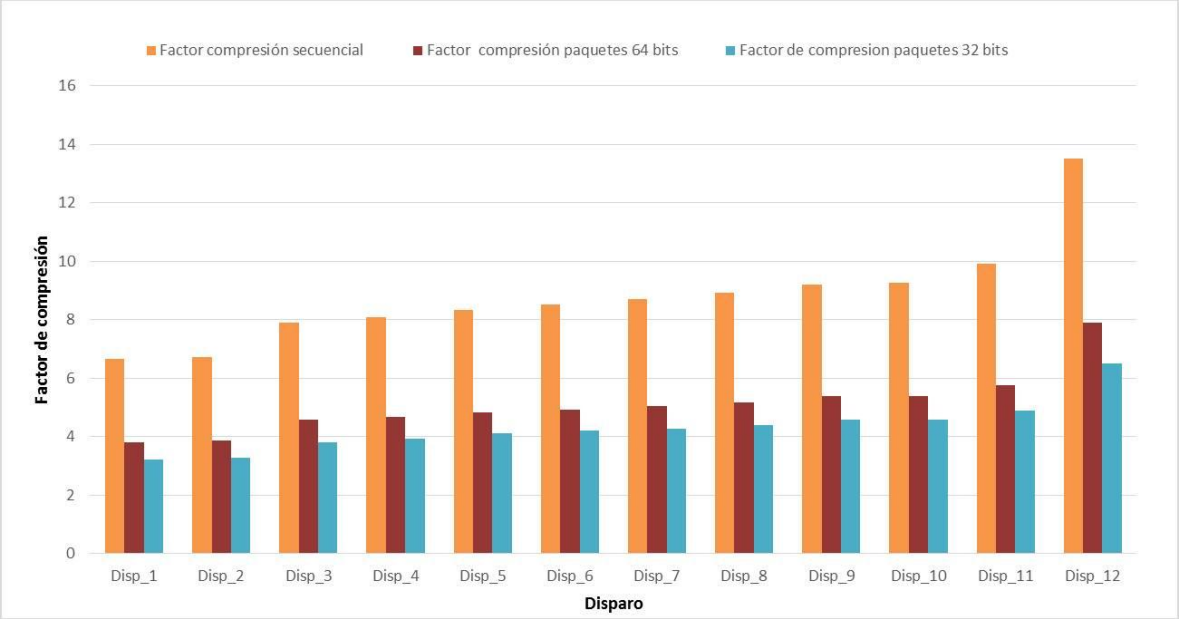
las posiciones siguientes tendrán un valor igual a la acumulación de datos presentes en cada paquete, de las posiciones anteriores a su posición.

Algoritmo de Codificación en Paquetes de 64 bits: Del mismo modo que el *Algoritmo de Codificación en Paquetes de 32 bits*, esta estrategia presenta una condición, la cual consiste en almacenar *P* símbolos en variables de 64 bits, siempre y cuando estos no superen en conjunto más de 58 bits. Adicionalmente, se designaron los 6 bits faltantes para guardar el número de datos dentro de cada paquete (ver sección 2.3).

.

La figura 14 muestra el factor de compresión obtenido al utilizar las tres estrategias mencionadas anteriormente.

Figura 14. Factor de compresión



Comparando los 3 tipos de codificación implementados, la codificación secuencial fue la que obtuvo mejor factor de compresión, puesto que su factor de compresión promedio fue de 8.82, con un valor pico de 13.53, alcanzado por el disparo número 12.

Al no utilizar completamente los 32 bits y emplear ficheros adicionales, el factor de compresión en la codificación en paquetes de 32 bits se redujo con relación al proceso secuencial obteniendo un valor promedio de 4.315, alcanzando un valor máximo de 6.5 en el disparo 12.

Para mejorar el factor de compresión obtenido con la codificación en paquetes de 32 bits, se planteó la hipótesis que al almacenar más datos por paquete, el factor de compresión aumentaría, por lo cual, se decidió utilizar variables de 64 bits. De igual forma que la codificación en paquetes de 32 bits, es necesaria la creación del fichero *índice* mencionado anteriormente; sin embargo, para evitar el fichero con la información del número de datos almacenados por cada paquete, los 64 bits disponibles se dividieron de la siguiente forma: los 58 bits de mayor peso, serán destinados para almacenar los símbolos y los 6 bits faltantes se reservaran para guardar el número de datos dentro los 58 bits mencionados. Por consiguiente, esta estrategia solo requirió la creación del fichero extra necesario, para direccionar los datos decodificados de cada paquete, a sus respectivas posiciones en el puntero de salida.

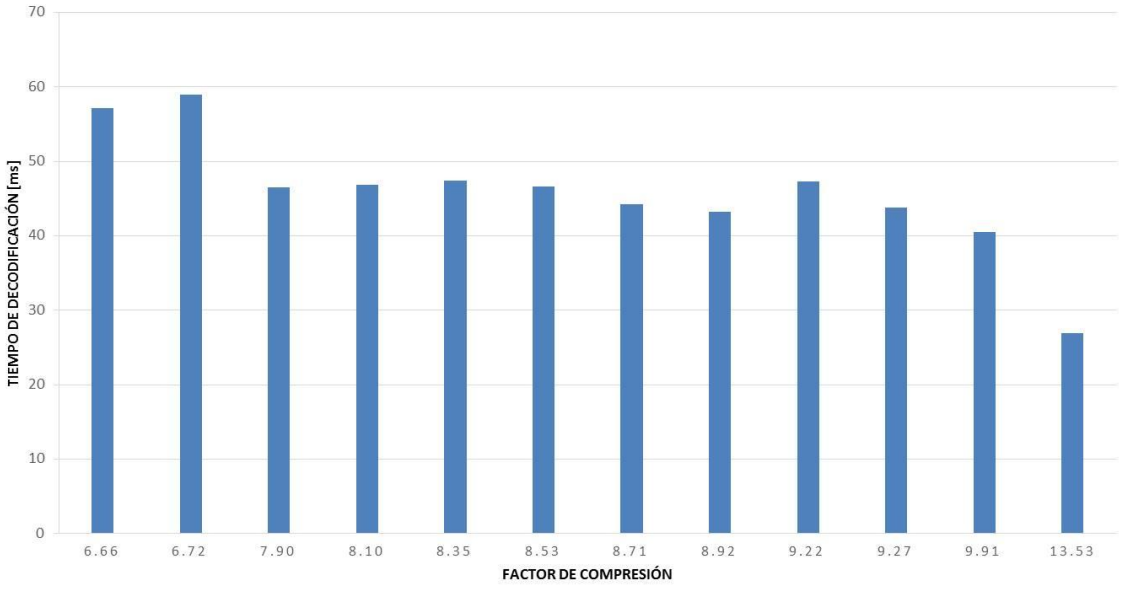
Como resultado de la codificación en paquetes de 64 bits, según la figura 14, se obtuvo un factor de compresión promedio de 5.11, con un máximo de 7.91 en el disparo 12. Gracias a las mejoras mencionadas anteriormente, el factor de compresión de la técnica en paquetes de 64 bits se acerca al factor de compresión

obtenido en la codificación secuencial, superando el resultado obtenido con la codificación en paquetes de 32 bits.

3.3 ALGORITMO DE DECODIFICACIÓN

Los tiempos de decodificación medidos en la ejecución en CPU se muestran en la figura 15.

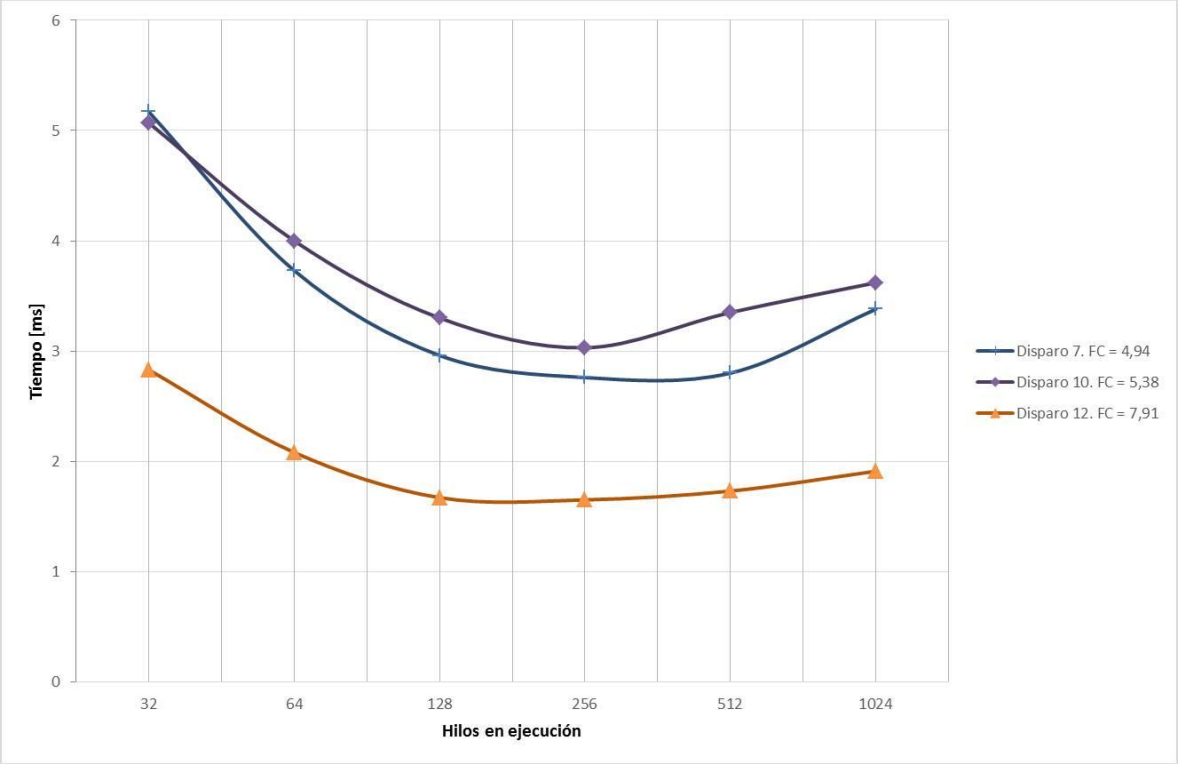
Figura 15. Tiempo de decodificación en CPU



El algoritmo de decodificación Huffman diseñado para su ejecución en CPU (ver sección 2.4) emplea entre 26.89 ms y 58.93 ms, para decodificar los datos de los 12 disparos de trazas sísmicas. Los resultados muestran que a mejor factor de compresión menor será el tiempo de decodificación para recuperar los datos originales.

Para la ejecución del Algoritmo de decodificación en GPU, se desarrollaron 2 alternativas (ver sección 2.5), una ejecutando el algoritmo con un kernel en bloques de hilos organizados en 1D y otra con el kernel en bloques de hilos agrupados en 2D.

Figura 16. Tiempo de decodificación en la GPU. Versión 1D

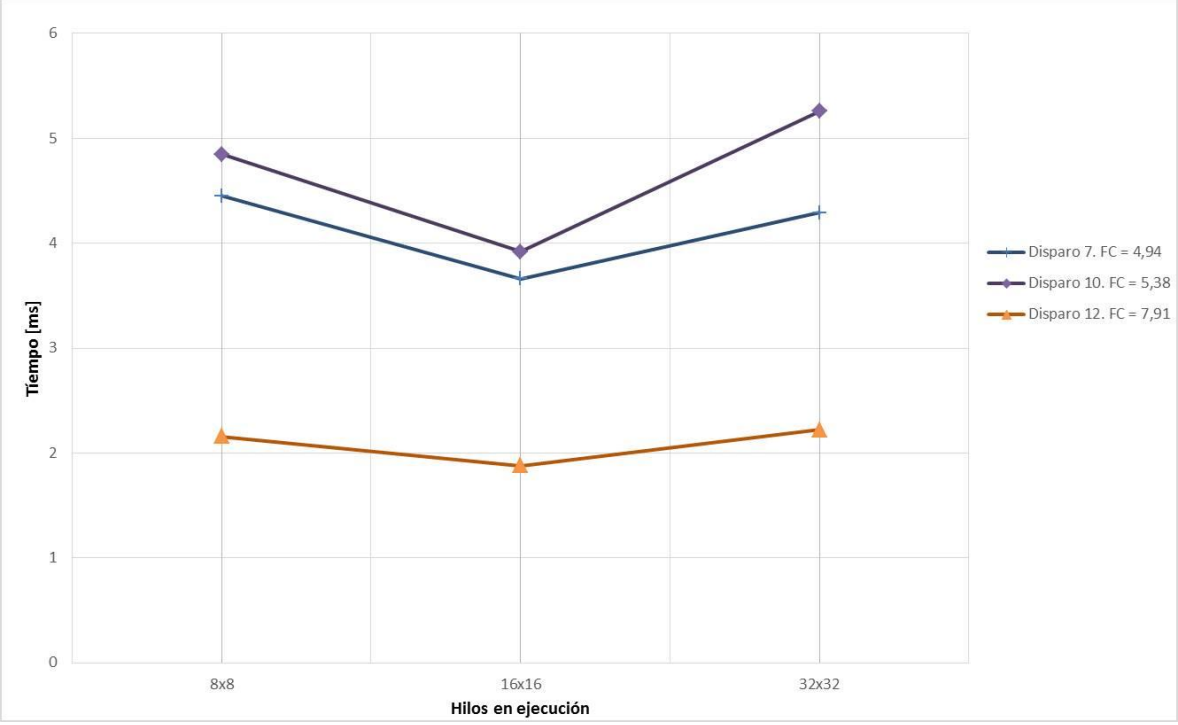


En la figura 16, se observan los resultados obtenidos al ejecutar el *Algoritmo de decodificación Huffman en paralelo* lanzando el kernel en bloques de hilos en 1D. Se implementó la decodificación con 128, 256, 512 y 1024 hilos en la ejecución del kernel obteniendo el mejor resultado con 256 hilos al decodificar el disparo 12, ya que el algoritmo tardo tras la ejecución con 256 *hilos* entre 1.65 ms y 3.24 ms para decodificar la información, lo cual mejora el tiempo utilizado por el

algoritmo de decodificación en la CPU 16.3 veces en el mismo disparo, recordando que el disparo 12 tiene el mejor factor de compresión entre todos los disparos estudiados.

Los resultados al lanzar el kernel en 2D, se muestran en la figura 17. Esta estrategia se implementó para tratar de aprovechar la ejecución multidimensional que permite la GPU, utilizando el lenguaje CUDA. Esto es posible al organizar los datos de forma vectorial, matricial o en vectores tridimensionales [14]. Al implementar el kernel con hilos en 2D, se observó un menor rendimiento comparado con la versión en 1D, dado que el tiempo de ejecución en matrices de 16x16 variaba entre 1.88 ms y 3.92 ms.

Figura 17. Tiempo de decodificación en la GPU. Versión 2D.



La tabla 3 muestra los resultados obtenidos al ejecutar el algoritmo de decodificación Huffman en la CPU y el tiempo de decodificación obtenido al ejecutar el algoritmo paralelo en la GPU.

Tabla 3. Tiempos de decodificación.

DISPARO	**1**	**2**	**3**	**4**	**5**	**6**
CPU	57.08	58.93	46.54	46.81	47.45	46.65
GPU	3.02	3.08	2.77	2.51	3.24	2.7
DISPARO	**7**	**8**	**9**	**10**	**11**	**12**
CPU	44.21	43.23	47.27	43.81	40.52	26.89
GPU	2.73	2.67	3.03	3	2.42	1.65

3.4 DISCUSIÓN DE RESULTADOS

En la figura 15, se observa un que el disparo 9 con factor de compresión de 9.22 no sigue la tendencia observada en la gráfica; este pico se presenta porque este disparo tiene en su diccionario de codificación 1.399 datos, 372 más que el promedio de símbolos obtenidos en los diccionarios, por ende, a pesar de tener un buen factor de compresión tiene un diccionario muy extenso y esto ralentiza el proceso de búsqueda y por lo tanto, todo el proceso de decodificación.

La GPU ejecuta un algoritmo en hilos agrupados en paquetes de 32, llamados warps, que se ejecutan de forma simultánea. Esta condición de funcionamiento genera *divergencia* al ejecutar algoritmos con bucles condicionales [15]. La divergencia es ocasionada por hilos ociosos, que permanecen inactivos cuando la

condición del bucle no se cumple, lo cual ralentiza el proceso en ejecución. Lo ideal es que los 32 hilos que conforman cada warp, trabajen de forma simultánea utilizando el 100% de los recursos disponibles en la GPU [16].

La utilización de una GPU se hace eficiente, siempre y cuando cientos de dispositivos realizan la misma operación sobre distintos datos. Según Cook [17], las GPUs tienen mejor rendimiento cuando más bloques de hilos se ejecutan en un mismo streaming multiproccessor (SM) y para poder generar más bloques dentro de un mismo SM, no se deben asignar todos los hilos disponibles al mismo bloque para evitar hilos ociosos, adicionalmente no es seguro que al ejecutar un algoritmo paralelo sobre todos los hilos disponibles en una GPU, este algoritmo se ejecutara de forma más rápida. Por esto, se probó la ejecución del algoritmo de decodificación paralelo de Huffman en la GPU, utilizando 128, 256, 512 y 1024 hilos por SM y los resultados mostraron que la GPU decodifica la información de forma mas rápida, utilizando 256 hilos por SM.

La implementación en paralelo del algoritmo de decodificación Huffman, presenta divergencia [18], dado que la cantidad de datos que contiene cada paquete no es igual. Para ejecutar el algoritmo en la GPU, se asigna a cada hilo un paquete de datos codificados las veces que sea necesario y la GPU ejecuta la misma operación sobre cada hilo, los hilos que tengan mayor cantidad de datos codificados, ralentizan el proceso de decodificación, dado que habrán hilos ociosos que terminaron primero su proceso de decodificación [19]. Dichos hilos se generan porque habrá más datos en unos paquetes que en otros y el algoritmo de búsqueda de símbolos en el diccionario de codificación, que es el más costoso en su ejecución, tendrá que ejecutarse más veces en unos hilos que en otros, incidiendo en la demora para la finalización del algoritmo.

La búsqueda en el diccionario de codificación es otro factor que afecta la velocidad de procesamiento del algoritmo de decodificación y se da porque a mayor longitud de los símbolos en un paquete, mayor tiempo empleara la búsqueda. Lo anterior es el resultado de organizar los datos en el diccionario de forma descendente, en función de la probabilidad de aparición; por lo cual, si un paquete contiene pocos datos, estos datos serán poco probables, teniendo en cuenta la codificación Huffman (ver capítulo 1), en consecuencia, una búsqueda con mayor tiempo de ejecución.

Caso contrario sucede con los paquetes que contienen muchos datos. Al ser estos datos del grupo de los probables, su posición en el diccionario será en la parte superior, lo cual hace que el proceso de búsqueda sea más rápido [20].

4. CONCLUSIONES

En la presente investigación, se diseñó e implemento un algoritmo de codificación-decodificación en una CPU. Adicionalmente se diseñó e implemento un algoritmo de decodificación en una GPU. Utilizamos técnicas de programación de bajo nivel, con el fin de utilizar la codificación Huffman y una técnica de decodificación, que permitía decodificar un símbolo por cada operación en la GPU.

Encontramos que la rapidez de ejecución del algoritmo de decodificación Huffman en paralelo, se ve afectada por la longitud variable de los datos comprimidos en cada paquete, lo cual genera una divergencia en la GPU.

Uno los factores que más afectan el rendimiento del algoritmo de decodificación Huffman, es la búsqueda en el diccionario de codificación y los múltiples accesos a este, procesos necesarios para recuperar la información original. Esto se da por la extensión del diccionario y la probabilidad de que el símbolo que se desea decodificar, se encuentre en la parte inicial o final del diccionario, por lo que, una búsqueda extensa en el diccionario ralentiza la ejecución del algoritmo de decodificación y aún más si estos símbolos de baja probabilidad, son recurrentes en los paquetes de codificación.

Una posible área de trabajo futura, seria restringir la cantidad de datos que contiene cada paquete de codificación e implementar una estrategia de búsqueda en paralelo para el diccionario. Reduciendo así los efectos de la divergencia y por ende, mejorar el tiempo de ejecución del algoritmo de decodificación Huffman en la GPU.

REFERENCIAS

[1] Aarti Departmentof CSE ACET. Performance analysis of huffman coding algorithm. International Journal of Advance Research in Computer Science and Software Engineering, 3, 2013.

[2] A. Baaziz and L. Quoniam. How to use big data technologies to optimize operations in upstream petroleum industry. Aix-Marseille University, 2014.

[3] Carlos A. Fajardo, Oscar M. Reyes, and Ana Ramirez. Seismic Data Compression Using 2D LiftingWavelet Algorithms. Ingener´ıa y Ciencia, 2015.

[4] Carlos Angulo, Carlos Fajardo, Oscar Reyes, and Javier Castillo. FPGA implementation of a Huffman decoder for high speed seismic data decompression. In 2014 Data Compression Conference, page 396, Salt Lake, United States., 2014. IEEE Comput. Soc.

[5] L. Duval and T. Rosten. Filter bank decomposition of seismic data with application to compression and denoising. SEG Annual International Meeting Society Exploration Geophysicists, pages 2055– 2058, 2006.

[6] J.M. Mendel. White-noise estimators for seismic data processing in oil exploration. Automatic Control, IEEE Transactions on, 694–706, Oct 1977.

[7] C. Shannon. A mathematical theory of communication. The Bell System Technical Journal, pages 379– 423, 1948.

[8] D.A Huffman. A method for the construction of minimum-redundancy codes. I.R.E., 40:1098–1101, 1952.

[9] David Salomon. Data Compression. The Complete Reference. Springer, 3 edition, 2004.

[10] NVIDIA Corporation. Nvidia's next generation: Cuda compute arquitecture kepler. Technical report, 2012.

[11] NVIDIA Corporation. Nvidia's next generation: Cuda compute arquitecture fermi. Technical report, 2009.

[12] Hyesoom Kim. Design and programming tesla gpu. Technical report, 2011.

[13] NVIDIA. CUDA C Programming Guide. NVIDIA Corporation, 5.5 edition, 2013.

[14] NVIDIA Corporation. Technology overview. nvidia geforce gtx. Technical report, 2012.

[15] J. Ajanovic. Pci express 3.0 accelerator features. Technical report, 2008.

[16] W.L. Wilson, I. Sham, G. Yuan, and T. M. Aamdot. Dynamic warp formation and scheduling for efficient gpu control flow. University of British Columbia, 2007.

[17] S. Cook. Cuda Programming. A Developer's Guide to Parallel Computing with GPUs. Elsevier, 1 edition, 2013.

[18] B. Coutinho, D. Sampaio, F. Magno, and W. Meira. Divergence analysis and optimizations. Department of Computer Science - UFMG - Brazil, 2013.

[19] Z. Cui, Y. Liang, and K. Rupnow. An accurate gpu performance model for effective control flow divergence optimization. Advanced Digital Sciences Center at Singapore, 2012.

[20] P. Suri and M. Goel. Ternary tree and memory-efficient huffman decoding algorithm. IJCSI International Journal of Computer Sciences Issues, 8, 2011.

BIBLIOGRAFIA

ANGULO, Carlos, FAJARDO, Carlos, REYES, Oscar. y CASTILLO, Jairo.. FPGA Implementation of a Huffman Decoder for High Speed Seismic Data Decompression. 2014 Data Compression Conference, 2014. IEEE Computing Society.

COOK, Shane. CUDA Programming: A Developer's Guide to Parallel Computing With GPUs. Elsevier. 2013

FAJARDO, Carlos, REYES, Oscar y RAMIREZ, Ana. Seismic Data Compression Using 2D Lifting Wavelet Algorithms. Ingeniera y Ciencia. 2015.

HUFFMAN, David. A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the I.R.E. Pages:1098-1101. 1952.

NVIDIA. CUDA C Programming Guide. Nvidia Corporation. 2013.

SALOMON, David. Data Compression The Complete Reference. Springer, 2001.