Evaluación del desempeño computacional de un algoritmo de descompresión de trazas sísmicas sobre una GPU.

JAIRO ALBERTO CASTELAR MORA

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERIAS FISICO-MECANICAS
ESCUELA DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y DE TELECOMUNICACIONES
Bucaramanga
2015

Evaluación del desempeño computacional de un algoritmo de descompresión de trazas sísmicas sobre una GPU.

JAIRO ALBERTO CASTELAR MORA

Trabajo de Grado para optar al título de

Ingeniero Electrónico

Director

Ing. Carlos A. Angulo Julio

Co-director

Phd(c). Carlos Augusto Fajardo Ariza

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERIAS FISICO-MECANICAS
ESCUELA DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y DE TELECOMUNICACIONES
Bucaramanga
2015

ACKNOWLEDGMENTS

This work is supported by Colombian Oil Company ECOPETROL and COLCIENCIAS as a part of the research project grants No. 0266-2013 and 511/2010. The authors gratefully acknowledge CPS research group of Industrial University of Santander.

CONTENT

LIST OF FIGURES

LIST OF TABLES

RESUMEN

Título:

EVALUACIÓN DEL DESEMPEÑO COMPUTACIONAL DE UN ALGORITMO DE DESCOMPRESIÓN DE TRAZAS SÍSMICAS SOBRE UNA GPU.[1]

Autor: Jairo Alberto Castelar Mora[2].

Palabras Claves: Transformada Wavelet Discreta, esquema Lifting, Descompresión, GPU.

El proceso utilizado por la industria del petróleo para construir imágenes del subsuelo con el fin de localizar las reservas de hidrocarburos requiere procesar una gran cantidad de información, lo que genera el desafío de procesar grandes volúmenes de datos, los cuales deben ser transmitidos hasta las estaciones de cómputo locales para su respectivo procesamiento. Así mismo, la cantidad de datos con los que se debe trabajar aumenta con el paso del tiempo y es imprescindible mantener la productividad sin afectar los resultados. Por otro lado, la velocidad a la que se transfieren los datos sísmicos no es lo suficientemente alta como para aprovechar de una mejor manera los recursos en las unidades de procesamiento. Es por esto que la compresión de datos se hace deseable puesto que proporciona una reducción en términos de almacenamiento y ancho de banda de transmisión.

En este proyecto se desarrolló un algoritmo de descompresión de datos sísmicos que consta de tres etapas: la decodificación Huffman, cuantificación uniforme inversa y una transformación wavelet bidimensional inversa, basada en el esquema lifting y utilizando un filtro biortogonal 5/3. El algoritmo fue desarrollado en e CUDA y luego implementado en una GPU GeForce GTX 660 con capacidad CUDA de 3.0.

[1] Trabajo de Grado modalidad en investigación

[2] Facultad de Ingenierías Físico Mecánicas. Escuela de Ingenierías Eléctrica, Electrónica y Telecomunicaciones. Director: M.Sc Carlos Angulo Julio. PhDc. Codirector: Carlos A. Fajardo.

ABSTRACT

TITLE:

COMPUTATIONAL PERFOMANCE EVALUATION OF A SEISMIC DATA DECOMPRESISION ALGORITHM INTO A GPU[3]

AUTHOR: Jairo Alberto Castelar Mora[4].

KEYWORDS: Discrete Wavelet transform, Lifting scheme, Decompression, GPU.

The process used by the oil industry to build images of the subsurface with the aim to locate hydrocarbon reserves requires processing a lot of information, generating the challenge to process high data volumes, which must be transmitted to local stations for their respective processing. Likewise, the amount of data that must be worked increases over time and is essential to maintain the productivity without affecting the results. Furthermore, the speed at which the seismic data is transferred is not high enough to take advantage in a better way the resources on processing units. Data compression is desirable because it provides a reduction in terms of storage and transmission bandwidth.

In this project was developed a decompression algorithm of seismic data comprising of three stages: Huffman decoding, inverse uniform quantization and Two-dimensional Inverse Wavelet Transform based on the lifting scheme using a 5/3 biorthogonal filter. The algorithm was developed in CUDA and then implemented on a GPU GeForce GTX 660 with CUDA capability 3.0.

[3] Degree Project

[4] Faculty of Physics Mechanics Engineering. Electrical, Electronics Engineering and Telecommunications School. Director: M.Sc Carlos Angulo Julio. Codirector: PhDc. Carlos A. Fajardo.

INTRODUCTION

The oil industry uses diverse methods, based on the analysis of high-resolution subsoil images, to determine the presence of mineral deposits [1] [2]. Therefore, a lot of information is produced, which is reflected in the volume of the data (known as seismic traces) in the order of Terabytes [2] [3].

Now, since larger data volumes are required to be processed, the seismic traces transfer speed is not high enough compared to its processing speed (limitation known as memory wall) causing that the process is inefficient [3], so it is necessary to look for transfer alternatives such the compression of seismic traces.

Compression is a method used to represent the original information with less data, allowing a better use of the space where the information is stored and reducing its transmission time [2] [4]. Seismic data compression has been favored with wavelet analysis because offers mathematical constructions with great potential in statistical methodology [1]. The Discrete Wavelet Transform (DWT) is know thanks to the work developed in various applications, as noise removal, image analysis and data compression [1] [5].

This paper presents an implementation of a decompression algorithm of seismic traces into a GPU based on Huffman Decoding, inverse Uniform Quantification and 2D Inverse Discrete Wavelet Transform using a 5/3 biorthogonal filter. The paper is organized as follows: The second section presents the algorithm used to perform the data decompression. In the third section the hardware and software used in this project are mentioned. In the fourth section the proposed algorithm is shown. In the fifth section five the results obtained in the tests are presents and finally, in sixth section the conclusions obtained with the proposed strategy are exposed.

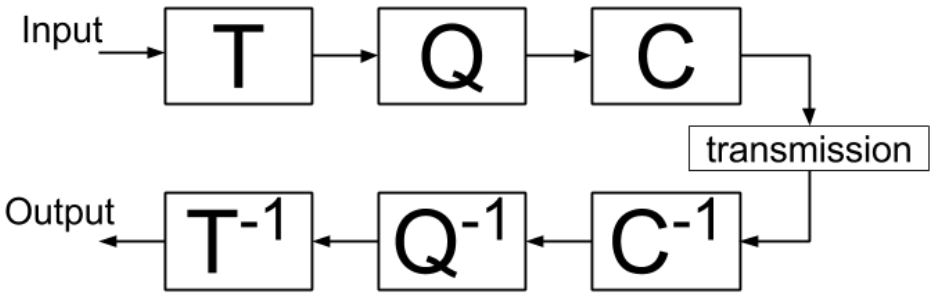
1. DATA COMPRESSION

Data compression is a process in which the same information is represented using fewer bits, thereby reducing the space used to store the information and the time needed to transfer it [6]. The compression is compounded of three steps:

- Transformation: allows the decorrelation of seismic data to concentrate its power in less data than the original information.
- Quantification: reduces the number of bits used to represent each datum. At this stage some loss information occurs.
- Encoding: decreases the amount of bits needed to represent all the data set.

After performing these three stages and transferring the information to the processing device, the data have to be converted back to its original form, so it is necessary to do (in the shortest time possible) the reverse process to compression know as decompression. Figure 1 shows the scheme for data compression and decompression.

Figure 1 Data compression scheme



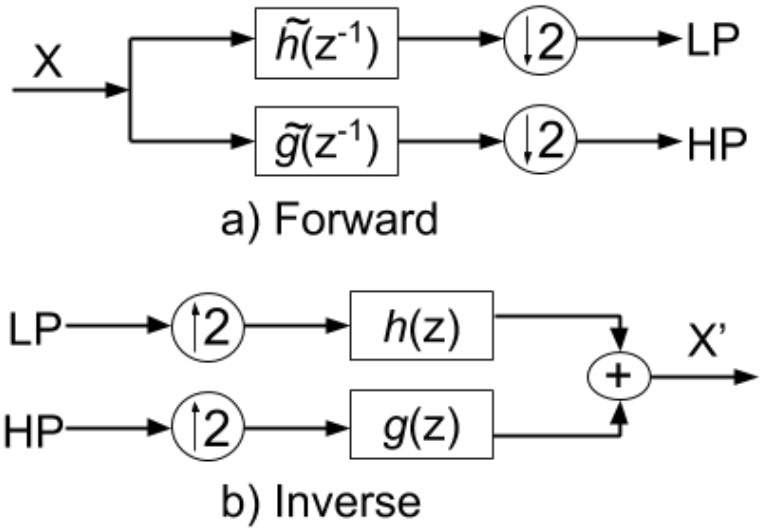
1.1. DISCRETE WAVELET TRANSFORM (DWT)

Wavelet transform is a tool to analyze signals used in various fields, including data compression [6]. The aim of the transformation stage in a compression algorithm is

to compact the data for betters results in the later stages. Two main options exist for the implementing of DWT: the traditional method, based on convolution and the lifting-based method [7].

1. *Traditional DWT:* Consists in applying the analysis filter pair $\tilde{h}$ - $\tilde{g}$ (low-pass and high-pass respectively), followed by an oversample. This results the decomposition of the input signal in approximation *(LP)* and detail *(HP)* coefficients (Figure 2a). The inverse transform (IDWT) consist upsampling *LP* and *HP* coefficients before applying the synthesis filters $h(z)$ and $g(z)$ (Figure 2b) [8].

Figure 2 Discrete Wavelet Transform. Adapted from [8].



2. *DWT based on lifting scheme:* Lifting scheme reduces the computational transformation requirements by factoring the polyphase matrix of wavelet filters at elementary matrices. Polyphase matrix analysis is defined in Z domain as follows [7]:

$$\tilde{P}(z^{-1}) = \begin{bmatrix} H_e(z) & H_o(z) \\ G_e(z) & G_o(z) \end{bmatrix} \quad (1)$$

where $H_e(z)$ and $H_o(z)$ denote the Type-I even and odd components of the low-pass analysis filter, and $G_e(z)$ and $G_o(z)$ denote the Type-I even and odd components of the high-pass analysis filter. Using equation (1), decomposition of input X can be written as:

$$\begin{bmatrix} LP(z) \\ HP(z) \end{bmatrix} = \tilde{P}(z^{-1}) \begin{bmatrix} X_e(z) \\ X_o(z) \end{bmatrix} \quad (2)$$

where $X_e(z)$ and $X_o(z)$ denote the Type-I even and odd components of signal $X(z)$.

In [8] has been proven that any pair of complementary filters $\tilde{h}$ - $\tilde{g}$ can be factorized into a sequence of triangular matrices – upper and lower – and a diagonal matrix. The polyphase matrix representation of a wavelet filter bank is given by:

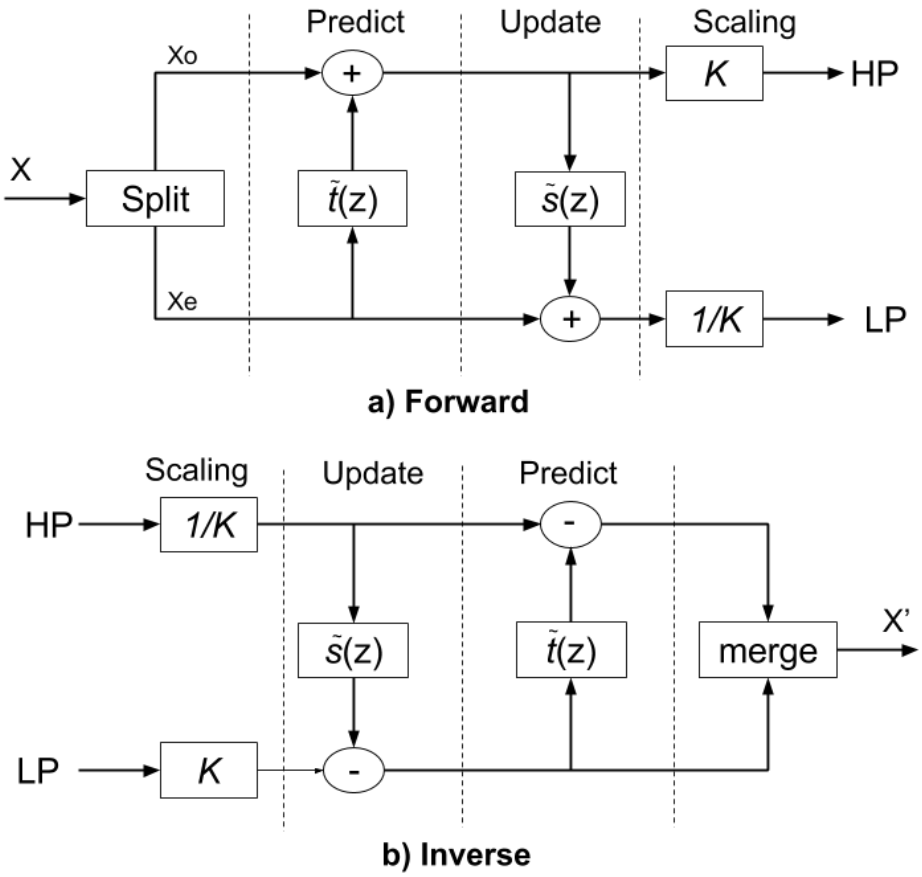
$$\tilde{P}(z^{-1}) = \begin{bmatrix} \tilde{h}_e(z) & \tilde{h}_o(z) \\ \tilde{g}_e(z) & \tilde{g}_o(z) \end{bmatrix} = \begin{bmatrix} K & 0 \\ 0 & \frac{1}{K} \end{bmatrix} \prod_{i=1}^{m} \left(\begin{bmatrix} 1 & \tilde{s}_i(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0) \\ \tilde{t}_i(z) & 1 \end{bmatrix} \right) \quad (3)$$

where $\tilde{s}_i(z)$ and $\tilde{t}_i(z)$ are Laurent polynomials, m is the number of lifting steps and K is a non-zero scaling factor.

The lifting scheme for forward DWT consists of the following stages (Figure 3a):

- Split: consists in separating the input signal x into even x_e and odd x_o samples.
- Predict: the even samples are multiplied by the coefficients of the polynomial $\tilde{t}_i(z)$ and added with the odd samples.
- Update: the new odd samples are multiplied by the coefficients of the polynomial $\tilde{s}_i(z)$ and added with the original even samples.
- Scaling: both samples, odd and even, are multiplied by 1/K and K respectively. This stage can be skipped if the input data consist of integer samples, such transform is known as integer wavelet transform.

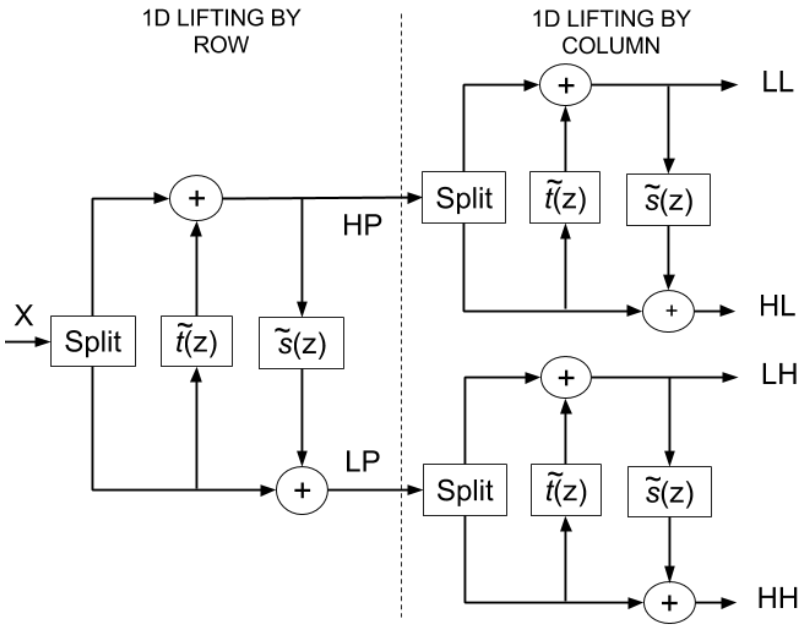
Figure 3 Lifting Scheme. Adapted from [9]



The inverse transform is performed by flipping the signs of the coefficients in the Laurents polynomials $\tilde{s}_i(z)$ and $\tilde{t}_i(z)$ and changing the scaling factor from K to 1/K (Figure 3b).

It is necessary to perform a two dimensional transformation when the input data correspond to images. In these cases, the wavelet decomposition consist on four different subbands *LL, HL, LH* and *HH* (known as approximation coefficients and horizontal, vertical and diagonal detail coefficients respectively). The 2D DWT involves making one 1D DWT by rows and two 1D DWT by columns (Figure 4).

Figure 4 Lifting Scheme for 2D DWT. Adapted from [7]



3. *Wavelet filters:* The 5/3 and 9/7 biorthogonal wavelet filters are a set of filters widely used in the implementation of the DWT [7]. We decided to work with the

5/3 filter, because it requires less lifting steps and there are only two multiplications involved (by 1/2 and 1/4) that can be implemented with displacements.

The analysis filters for the 5/3 wavelet filter are [7]:

$$\tilde{h}(z) = -\frac{1}{8}z^{-2} + \frac{1}{4}z^{-1} + \frac{3}{4}z^{0} + \frac{1}{4}z^{1} - \frac{1}{8}z^{2}$$

$$\tilde{g}(z) = -\frac{1}{2}z^{-2} + z^{-1} - \frac{1}{2}z^{0} \quad (4)$$

Using equation (3), its polyphase matrix is:

$$\tilde{P}(z) = \begin{bmatrix} -\frac{1}{8}z^{-1} + \frac{3}{4} - \frac{1}{8}z & \frac{1}{4} + \frac{1}{4}z \\ -\frac{1}{2}z^{-1} - \frac{1}{2} & 1 \end{bmatrix} \quad (5)$$

Which can be factorized as:

$$\tilde{P}(z) = \begin{bmatrix} \tilde{h}_e^{new}(z) & \frac{1}{4} + \frac{1}{4}z \\ \tilde{g}_e^{new}(z) & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \tilde{t}(z) & 1 \end{bmatrix} \quad (6)$$

Now, it is necessary to find Laurent polynomials $\tilde{t}(z)$, $\tilde{h}_e^{new}(z)$ and $\tilde{g}_e^{new}(z)$ such that: Which can be factorized as:

$$-\frac{1}{8}z^{-1} + \frac{3}{4} - \frac{1}{8}z = \tilde{h}_e^{new}(z) + \tilde{t}(z)\left(\frac{1}{4}z^{1} + \frac{1}{4}z\right)$$

$$-\frac{1}{2}z^{-1} - \frac{1}{2} = \tilde{g}_e^{new}(z) + \tilde{t}(z)\left(-\frac{1}{2}\right) \quad (7)$$

The polynomials $\tilde{h}_e^{new}(z)$ and $\tilde{t}(z)$ are obtained by dividing $\tilde{h}_e(z)$ between $\tilde{h}_o(z)$, where the ratio corresponds to $\tilde{t}(z)$ and the residue to $\tilde{h}_e^{new}(z)$. Having achieved the polynomial $\tilde{t}(z)$, the same procedure is performed to find $\tilde{g}_e^{new}(z)$ resulting:

$$\tilde{t}(z) = -\frac{1}{2}z^{-1} - \frac{1}{2}$$

$$\tilde{h}_e^{new}(z) = 1 \qquad (8)$$

$$\tilde{g}_e^{new}(z) = 0$$

Finally, the polyphase matrix is:

$$\tilde{P}(z) = \begin{bmatrix} 1 & \frac{1}{4} + \frac{1}{4}z \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -\frac{1}{2}z^{-1} - \frac{1}{2} & 1 \end{bmatrix} \qquad (9)$$

Hence, for the 5/3 biorthogonal wavelet filter we have

$$\tilde{t}(z) = -\frac{1}{2}(z^{-1} + 1)$$

(10)

$$\tilde{s}(z) = \frac{1}{4}(1 + z)$$

1.2. UNIFORM QUANTIFICATION

The quantization step reduces the number of bits needed to represent each datum but adds noise to each sample, for this reason is considered a lossy compression technique. The process is based on approximating to a single value every datum on a specific range of values. The procedure for the quantization consists of the following steps:

- The maximum *(max)* and minimum *(min)* value of the input vector X are calculated.
- The number of bit of quantification *(n)* is selected.
- Each input datum is subtracted from the minimum value and is operated as show in equation (11).

$$X_{quantified} = round\left(\frac{X_s * (2^n - 1)}{max}\right) \qquad (11)$$

Where $X_s = X - min$

The inverse quantization calculation proceeds as follows:

$$X^* = \frac{X_{quantified} \ *max}{(2^n - 1)} + min \qquad (12)$$

1.3. HUFFMAN ENCODING

The Huffman encoding is an entropy algorithm used for lossless data compression [10]. This is a method that encodes and compresses a given data set by using a variable-length code. A binary tree is created based on the occurrence probability of each value [11]. The table I shows an example in which the frequency is determined from a data set.

Figure 5 shows the binary tree produced from table I, where the left and right branches of a node are assigned as 0 and 1 respectively [12]. For this is necessary that symbols with lower frequencies have to be placed at the bottom nodes while symbols with higher frequencies at the top nodes. Paths from the root node (the one with the highest frequency) to the lower nodes are generated with the aim of creating the codes that will be assigned to each symbol as show in table II.

Table 1 Symbols frequency. Adapted from [12]

Symbols	Frequency
0	8
1	4
2	16
3	5
4	5
5	4
6	5
7	5

Figure 5 Huffman binary tree. Adapted from [12].

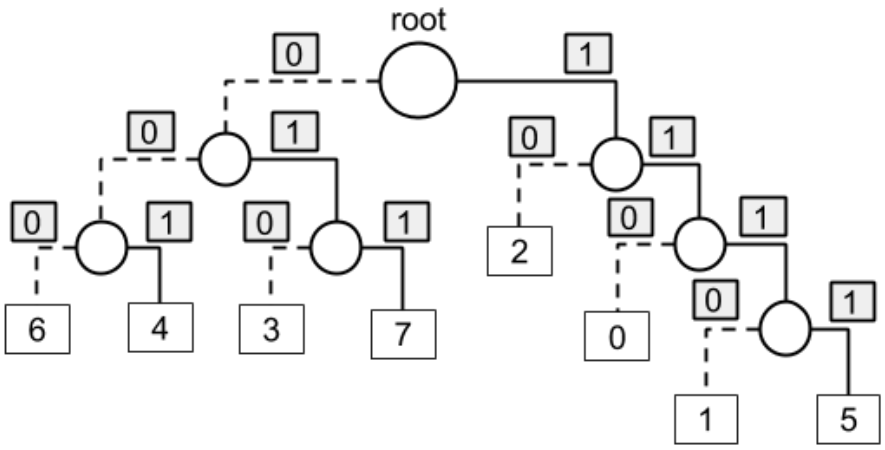


Table 2 Huffman encoded data. Adapted from [12]

Symbol	Codeword	Length
0	110	3
1	1110	4
2	10	2
3	010	3
4	001	3
5	1111	4
6	000	3
7	011	3

2. GRAPHICS PROCESSING UNIT

Graphics processing units (GPU) are coprocessors dedicated to graphics processing to relieving the load of a CPU processor. Its purpose is to run applications significantly faster. This is achieved by offloading compute-intensive portions of the application to the GPU, while the remainder code still runs on the CPU.

NVIDIA[5] GPUs are based on a set of multi-core units called streaming multiprocessors (*SM*). Each *SM* can run in parallel with the others *SM* in order to optimize the application runtime. Each core can execute one sequential thread, but the cores are executed in *SIMT*[6] fashion, that is, all cores in the same group execute the same instruction at the same time [13].

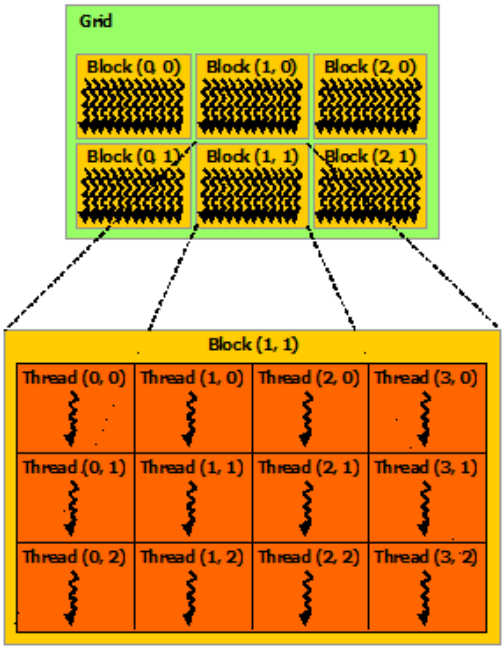
CUDA (*Compute Unified Device Architecture*) offers a data parallel programming model that is supported by NVIDIA GPUs. It allows increasing the performance of data processing by taking advantage of GPU capabilities using the parallelism that multi-core units offer. In this model, the host program launches a sequence of GPU kernels and these kernels can spawn sub-kernels.

The kernel execution is performed by threads grouped into blocks that are organized into a grid as illustrated on Figure 6. The block size can be one-dimensional, two-dimensional or three-dimensional, while the grid can be one-dimensional or two-dimensional. Typically, each kernel completes the execution before the next kernel begins, with an implicit barrier synchronization between kernels.

Each thread has a unique local index in its block, and each block has a unique index in the grid as show in figure 6. Kernels can use these indices to compute array subscripts. Threads in a single block will be executed on a single multiprocessor, sharing the software data cache, and can synchronize and share data with threads in the same block. Considering the above, a decompression algorithm based on Wavelet transformation was developed. It allows the algorithm to run in parallel, reducing execution time compared to the implementation of the algorithm on a CPU.

[5] NVIDIA is an worldwide technology company that manufactures GPUs.
[6] Single Instruction - Multiple Threads.

Figure 6 Thread Hierarchy. Taken from [14]



3. PROPOSED ALGORITHM

The implementation of data compression method was divided in two parts: first, the compression was performed on CPU using C++; then, the decompression was implemented on a GPU using CUDA. For both cases, the scheme presented in Figure 1 was used.

3.1. COMPRESSION PROCESS

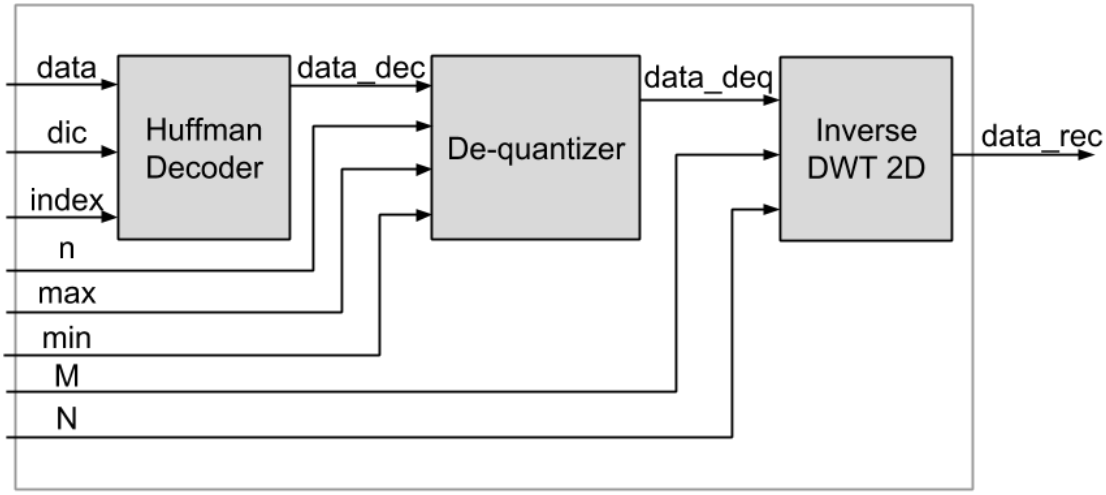
Initially, the software MATLAB was used to perform the compression and decompression of seismic traces. The scripts for the decompression process were developed without the use of proprietary functions to identify independent processes and to establish strategies for its implementation on the GPU. Then, scripts were developed in C ++ to evaluate its computational performance. We

worked with 2D seismic data, so it was developed a 2D DWT script to carry out the step of inverse transformation in the decompression algorithm.

3.2. GPU decompression process

A computational algorithm is proposed for the purpose of decompressing a 2D dataset into a GPU (Figure 7). The dataset correspond to *M* seismic traces with *N* samples per trace.

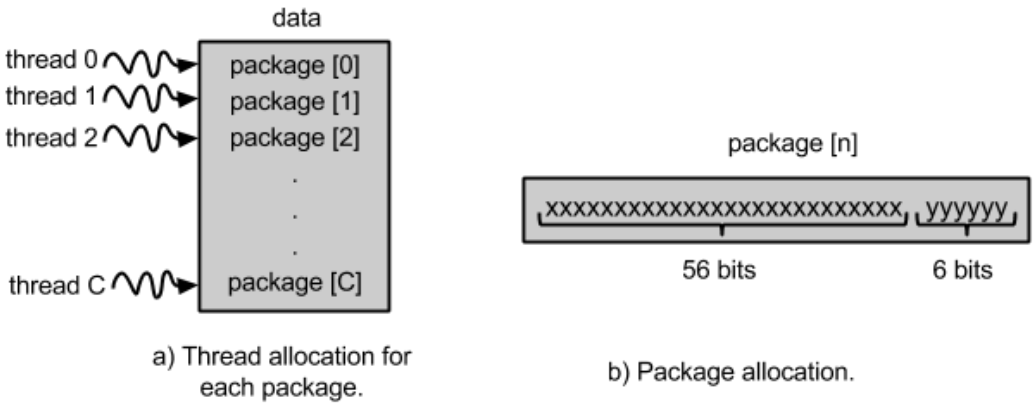
Figure 7 Proposed architecture



1. *Huffman decoder algorithm*: The following information is needed to decode the compressed data:
   - Encoded data *(data)*: compressed information organized in 64-bit packages. In each package, the first 58 bits correspond to compressed data and the last 6 bits represent how many data are in the package.
   - Dictionary *(dic)*: Code-words used in the compression process and their corresponding symbols.
   - *Index*: Initial position for each encoded datum in every 64 bit package.

Each package in *data* is assigned to a thread in order to decode it (Figure 8). The decodification is done by comparing the codes in the dictionary (one at time) with the bits in the package. The output of the algorithm *(data_dec)* corresponds to the decoded data.

Figure 8 Thread allocation for Huffman decoder



2. *Uniform Quantification inverse algorithm*: This stage is performed by implementing the equation (12) over the decoded data *(data_dec)* obtained from *Huffman algorithm* and using the compression parameters *n*, *max* and *min*.
3. *2D IDWT algorithm*: Accordingly Figure 3b, it must be developed a 1D IDWT by columns over *LL* and *HL* to obtain *LP*, another one over *LH* and *HL* to obtain HP, and then one 1D DWT by rows over *LP* and *HP*.

   To perform the 1D IDWT by columns of *LL* and *HL* subbands the following steps are needed:

   - Update: It is created a copy of the *LL* subband (named *LL'*) with an additional row of zeros at the end. We call the extra data as Boundary Values (*BV*). Each GPU thread accesses two consecutive positions of *LL'* and a position of subband *HL* simultaneously to perform the update operation of LL as shown in Figure 9a.

- Predict: It is created a copy of the updated *LL* (named *LL_u*) with a BV row at the beginning, then the predict operation is performed in a similar way as the update operation (Figure 9b).
- Merge: the results of the update stage are interspersed by rows with the results of the predict stage to obtain *LP*.

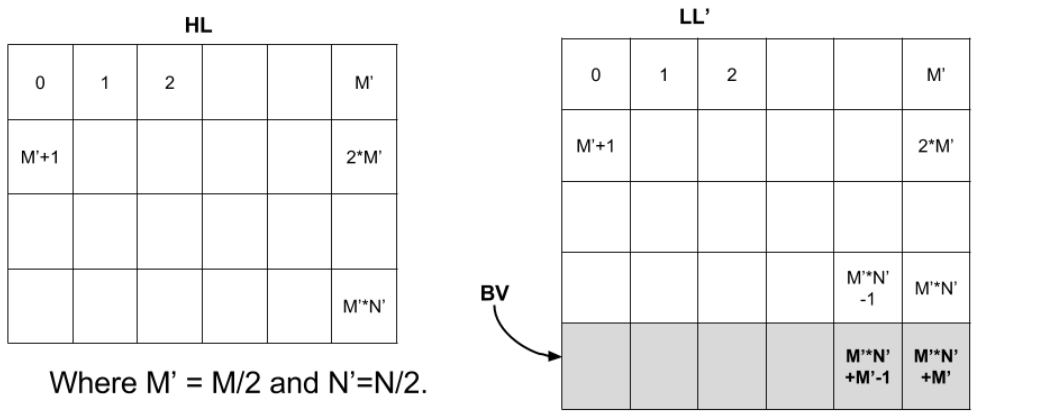
The same process is made with *LH* and *HH* subbands to perform its 1D IDWT by columns and to obtain *HP* (Figure 9b).

For the 1D IDWT by rows, the following changes from the 1D IDWT by columns are required:

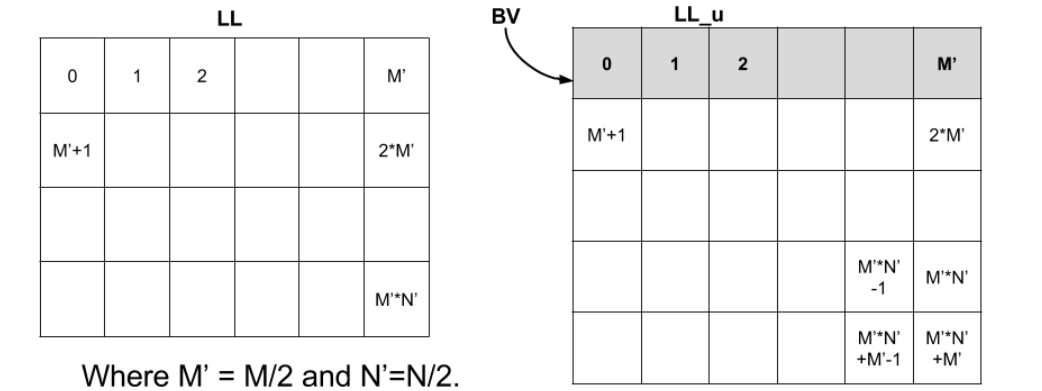
- The *BV* in the update stage are in the last column (Figure 10a).
- The *BV* in the predict stage are in the first column (Figure 10b).
- The alternation in the merge stage is made by columns

Upon completion this stage, the decompression process has been finished, being *data_rec* the retrieved information (i.e the decompressed data).

Figure 9 Threads allocation for 1D IDWT by columns.

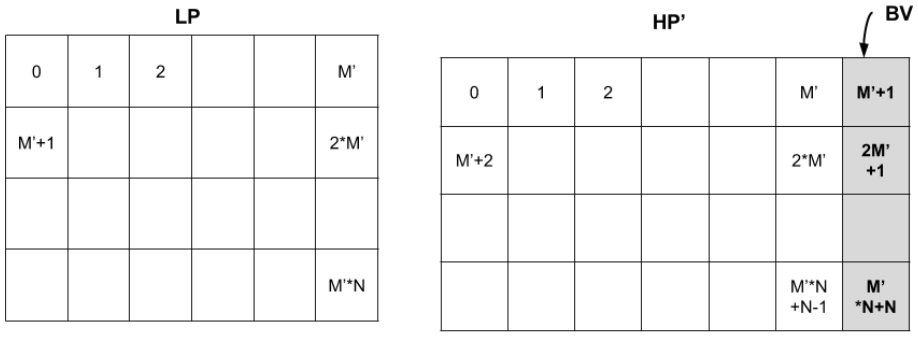


a) Update



b) Predict

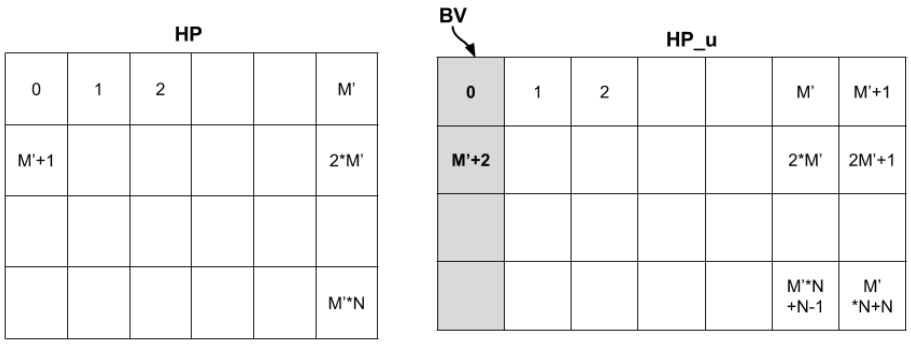
Figure 10 Threads allocation for 1D IDWT by rows



Where M' = M/2.

thread 0 ⇝ (HP'[0]+HP'[M'+1])*0.25 - LP[0]
thread 1 ⇝ (HP'[1]+HP'[M'+2])*0.25 - LP[1]
⋮
thread M'+1 ⇝ (HP'[M'+1]+HP'[2*M+1])*0.25 -LP[M'+1]
thread M'+2 ⇝ (HP'[M'+2]+HP'[2*M'+3])*0.25 -LP[M'+2]
⋮
thread M'*N-1 ⇝ (HP'[M'*N-1]+HP'[M'*N+M'-1])*0.25-LP[M'*N-1]
thread M'*N ⇝ (HP'[M'*N]+HP'[M'*N+M'])*0.25 - LP[M'*N]

a) Update



Where M' = M/2.

thread 0 ⇝ (HP_u[0]+HP_u[M'+1])*-0.5 - HP[0]
thread 1 ⇝ (HP_u[1]+HP_u[M'+2])*-0.5 - HP[1]
⋮
thread M'+1 ⇝ (HP_u[M'+1]+HP_u[2*M+1])*-0.5 - HP[M'+1]
thread M'+2 ⇝ (HP_u[M'+2]+HP_u[2*M'+3])*-0.5 - HP[M'+2]
⋮
thread M'*N-1 ⇝ (HP_u[M'*N-1]+HP_u[M'*N+M'-1])*-0.5 - HP[M'*N-1]
thread M'*N ⇝ (HP_u[M'*N]+HP_u[M'*N+M'])*-0.5 - HP[M'*N]

b) Predict

4. RESULTS

The algorithm was developed in CUDA 6.5 and implemented on a GeForce GTX 660 GPU with 3.0 CUDA capability and the decompression algorithm was applied to different shots that were supplied by ECOPETROL. Each shot has 96 traces (N) and 3584 samples per trace (M). The results obtained by the GPU algorithm were compared with the results of the decompression using MATLAB to ensure the proper operation.

Figure 11 Original and recovered shot.

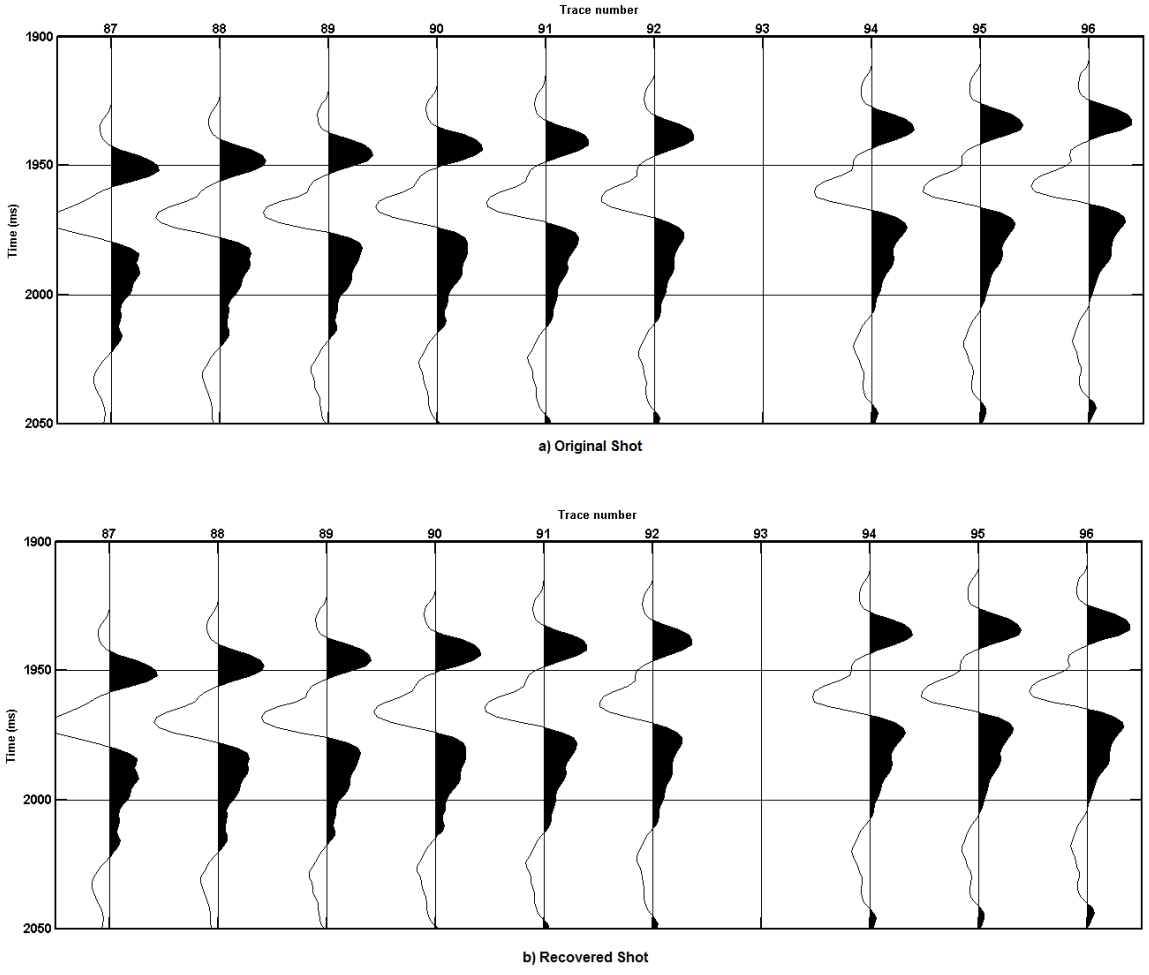


Figure 11 shows the comparison between the original and recovered shot. In this, is possible seen that the graphs obtained are similar, but for better comparison, the quality of the reconstructed signal was measured.

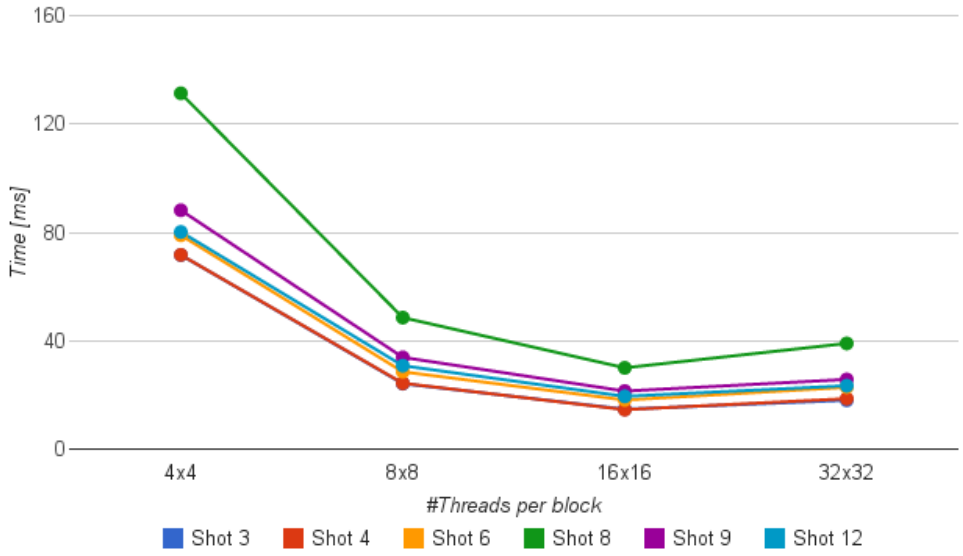
We compressed different seismic traces with different bits of quantization. Table III summarizes the results of compression, in terms of signal to noise ratio (SNR), Pearson product-moment correlation coefficient (*r*) and compression ratio (CR), for different data sets using 11 and 12 bits of quantization. We can see that if the number of bits increases, the SNR and *r* increases, but the CR decreases. For a better data processing in latter stages, is desirable to have a SNR higher than 40 dB and a *r* most nearly to 1, so we decided to work with 12 bits.

Table 3 SNR and CR for different datasets.

Shot	11 bits			12 bits		
	SNR (dB)	CR	*R*	SNR (dB)	CR	*R*
1	34.25	5.92	0.99980	40.77	4.58	0.99996
2	35.17	5.88	0.99984	41.39	4.49	0.99996
3	36.70	4.74	0.99989	42.73	3.79	0.99997
4	35.46	4.73	0.99985	41.49	3.75	0.99996
5	38.33	6.19	0.99992	44.78	4.76	0.99998
6	37.50	6.58	0.99991	43.97	5.08	0.99998
7	36.88	6.21	0.99989	43.55	4.86	0.99998
8	37.32	6.40	0.99990	44.00	4.96	0.99998
9	38.98	7.75	0.99993	42.46	5.66	0.99997
10	38.49	6.69	0.99992	45.44	5.29	0.99998
11	38.78	7.43	0.99992	42.78	5.29	0.99997
12	31.62	9.95	0.99966	37.92	7.78	0.99992

As a starting point, the three stages of decompression were implemented into a single kernel. This kernel was implemented with a 2D indexing because it was necessary for the transform stage. Figure 12 shows the time taken by this kernel and shows that the best performance occurs when blocks of 16x16 threads are used.

Figure 12 Decompression time vs threads per block.



In order to improve the computational performance, we proceeded to work every decompression stage independently, that is, each stage will run on a different kernel. For Huffman decoder and de-quantizer algorithms, the kernel was done with a 1D indexing because its input correspond to vectors, while kernel for transform stage has 2D indexing because the data correspond to a 2D matrix.

According to Tables IV and V, best performance for Huffman decoder and de-quantizer algorithms was obtained by working 256x1 threads per block and 128x1 threads per block respectively.

Table 4 Execution time comparison for Huffman.

Shot	Time [ms]				
	64 threads	128 threads	256 threads	512 threads	1024 threads
3	3.95	3.05	3.10	3.15	3.38
4	3.80	2.90	3.02	3.22	3.40
6	3.59	2.92	2.74	2.93	3.29
8	3.71	2.91	2.68	2.84	3.41
9	3.04	2.49	2.42	2.61	2.84
12	2.06	1.66	1.65	1.74	1.92

Table 5 Execution time comparison for De-quantizer.

Shot	Time [us]				
	64 threads	128 threads	256 threads	512 threads	1024 threads
3	59.94	35.97	36.32	37.92	41.28
4	60.00	35.78	36.35	37.47	42.53
6	59.97	36.00	36.29	37.57	41.98
8	59.97	35.84	36.06	37.54	42.43
9	60.00	36.03	36.64	37.38	41.86
12	60.06	36.29	36.54	38.11	41.50

The 2D IDWT was performed by a kernel with a grid size calculated as follows:

$$(Grid_{row}, Grid_{col}) = \left(ceil\left[\frac{M/2}{thread}\right], ceil\left[\frac{N}{thread}\right]\right) \quad (13)$$

where *thread* is the number of threads per block used for the implementation.

The number of threads used equals the product between the number of blocks launched (i.e the grid size) and the number of threads per block (i.e the block size). Using equation (13) and taking into account that in our tests the data size *(MxN)* is divisible by the block size, the total number of threads used is $M/2 * N$. However, half of threads launched are wasted because the size of the sub-bands used in the 1D IDWT by columns is $M * N/4$. For this reason we decided to reduce the size of the grid to have a better distribution of the threads in the transform algorithm.

The grid size in the second version of kernel was calculated dividing *N* by two as shown below:

$$(Grid_{row}, Grid_{col}) = \left(ceil\left[\frac{M/2}{thread}\right], ceil\left[\frac{N/2}{thread}\right]\right) \quad (14)$$

Furthermore, in version 1 of kernel, some copies of the information input were performed in order to separate the data and to work easily. For version 2 we decided not to make these copies and access the information directly from the

input data. Additionally, the algorithm was compiled using the CUDA instruction set architecture. For our GPU, the instruction used was *-arch=compute_30.*

The time spent by each version of the 2D IDWT kernel using different block sizes is showed in Table VI and we can see that the time used in version 2 is approximately 60% lower compared to version 1. These times are similar for the different datasets because all of them have the same size.

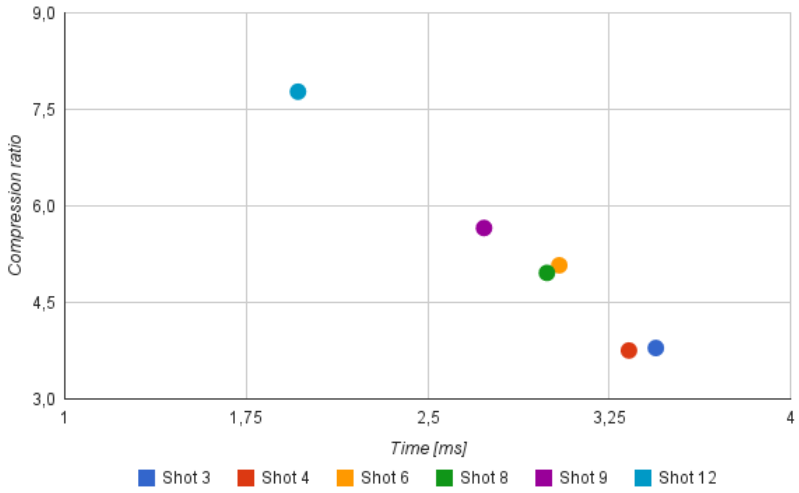
According to table VI, best performance for the IDWT is obtained by using a *8x8* block size. The algorithm proposed for the transformation can process up *8x8xSM* of data simultaneously.

Table 6 IDWT algorithm execution time comparison.

Threads per block	Time [ms] Version 1	Time [ms] Version 2
4x4	1720	684.335
8x8	671.1	267.010
16x16	699.71	278.393
32x32	705.91	280.860

Figure 13 shows the compression ratio and the time taken to decompress some datasets. It shows that the relationship between compression ratio and decompression time tends to be inversely proportional, that is, as CR increase, time spent decreases, and vice versa.

Figure 13 Compression ratio vs decompression time.



The algorithm was implemented using different bits of quantization and we observe that if we use less bits, the decompression time decreases, this due that the compression factor increases, so the size of data to decompress decreases (Figure 14).

Figure 14 Decompression time for different shots and different quantification bits.

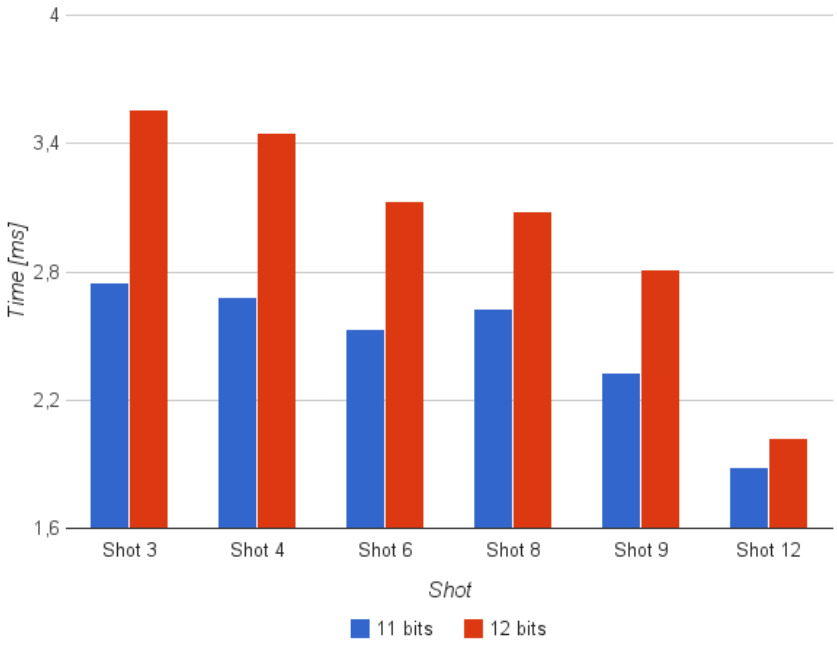


Table VII shows the GPU resources used by each algorithm in the decompression of seismic traces.

Table 7 GPU resources employed

Stage	index	Block size	Grid size	Globlal Memory used (MB)
Huffman Decoder	1D	128x1	335x1	1.84
De-quantizer	1D	256x1	1344x1	1.31
2D IDWT	2D	8x8	448x12	9.22

The algorithm was implemented using different bits of quantization and we observe that if we use less bits, the decompression time decreases, this due that the compression factor increases, so the size of data to decompress decreases.

The implementation of the transformation requires to reserve 16 memory spaces occupying 9.22MB. The GeForce GTX 660 has 2GBytes of memory, we estimate that the device is capable of decompressing a set of seismic traces corresponding to more than 21300 traces each one with 3584 samples.

5. CONCLUSIONS

We implemented an algorithm that performs the decompression of seismic traces into a GPU, using a specific data compression strategy. The implementation performs the Huffman Decoding, Inverse Uniform Quantification and the 2D IDWT using the lifting scheme.

The speed in decompression process is related to the compression ratio and performance of the algorithm on the GPU. The relationship between the compression ratio and decompression time tends to be inversely proportional because, while the CR increases, the amount of resulting compressed data is smaller so that the number of threads used to decode it decreases, giving a faster decompression.

The bottleneck in the decompression process is the Huffman decoding because this process depends on the amount of data represented in each package resulting from compression, while that the transformation was designed to be processed each thread simultaneously.

REFERENCES

[1] M. Khene and S. Abdul-Jauwad, “Adaptive seismic compression by wavelet shrinkage,” in Statistical Signal and Array Processing, 2000. Proceedings of the Tenth IEEE Workshop on. IEEE, 2000, pp. 544– 548.

[2] W. Wu, Z. Yang, Q. Qin, and F. Hu, “Adaptive Seismic Data Compression Using Wavelet Packets,” 2006 IEEE Int. Symp. Geosci. Remote Sens., no. 3, pp. 787–789, Jul. 2006.

[3] C. Fajardo and J. Castillo, “Reducción de los tiempos de computo de la Migración Sísmica usando FPGAs y GPGPUs: Un artículo de revisión,” vol. 9, no. 17, pp. 261–293, 2013.

[4] M. Al-Moohimeed, “Towards an Efficient Compression Algorithm for Seismic Data,” Radio Science Conference, 2004. Proceedings. 2004 Asia-Pacific, pp. 550–553, 2004.

[5] A. Z. Averbuch, F. Meyer, J. Stromberg, R. Coifman, and A. Vassiliou, “Low Bit-Rate Efficient Compression for Seismic Data,” Image Processing, IEEE Transactions on, vol. 10, no. 12, pp. 1801–1814, Dec. 2001.

[6] D. Salomon, Data Compression The Complete Reference, 4th ed., 2007.

[7] M. E. Angelopoulou, K. Masselos, P. Y. Cheung, and Y. Andreopoulos, “Implementation and Comparison of the 5/3 Lifting 2D Discrete Wavelet Transform Computation Schedules on FPGAs ,” J. Signal Process. Syst., vol. 51, no. 1, pp. 3–21, Oct. 2008.

[8] I. Daubechies and W. Sweldens, “Factoring Wavelet transforms into lifting steps,” Nov. 1997.

[9] K. Andra, C. Chakrabarti, and T. Acharya, “A VLSI Architecture for Lifting-Based Forward and Inverse Wavelet Transform,” vol. 50, no. 4, pp. 966–977, 2002.

[10] L. Acasandrei and M. Neag, “A Fast Parallel Huffman Decoder por FPGA implementation,” vol. 49, no. 1, p. 8, 2009.

[11] H.-A. Pham, V.-H. Bui, and A.-V. Dinh-Duc, “An Adaptive Huffman Decoding Algorithm for MP3 Decoder,” 2010 Fifth IEEE Int. Symp. Electron. Des. Test Appl., pp. 153–157, 2010.

[12] S. Beak, B. Hieu, H. Lee, S. Choi, I. Kim, K. Lee, Y. Lee, and T. Jeong, “Novel binary tree Huffman decoding algorithm and field programmable gate array implementation for terrestrial-digital multimedia broadcasting mobile handheld ,” IET Sci. Meas. Technol, vol. 6, no. 6, pp. 527–528, 2012.

[13] M. Wolfe and P. C. Engineer, “Understanding the cuda data parallel threading model a primer,” The Portland Group Technical News, 2010.

[14] Nvidia. Cuda c programming guide: Design guide. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide

BIBLIOGRAPHY

A. Z. Averbuch, F. Meyer, J. Stromberg, R. Coifman, and A. Vassiliou, “Low Bit-Rate Efficient Compression for Seismic Data,” Image Processing, IEEE Transactions on, vol. 10, no. 12, pp. 1801–1814, Dec. 2001.

C. Fajardo and J. Castillo, “Reducción de los tiempos de computo de la Migración Sísmica usando FPGAs y GPGPUs: Un artículo de revisión,” vol. 9, no. 17, pp. 261–293, 2013.

D. Salomon, Data Compression The Complete Reference, 4th ed., 2007.

H.-A. Pham, V.-H. Bui, and A.-V. Dinh-Duc, “An Adaptive Huffman Decoding Algorithm for MP3 Decoder,” 2010 Fifth IEEE Int. Symp. Electron. Des. Test Appl., pp. 153–157, 2010.

I. Daubechies and W. Sweldens, “Factoring Wavelet transforms into lifting steps,” Nov. 1997.

K. Andra, C. Chakrabarti, and T. Acharya, “A VLSI Architecture for Lifting-Based Forward and Inverse Wavelet Transform,” vol. 50, no. 4, pp. 966–977, 2002.

L. Acasandrei and M. Neag, “A Fast Parallel Huffman Decoder por FPGA implementation,” vol. 49, no. 1, p. 8, 2009.

M. Al-Moohimeed, “Towards an Efficient Compression Algorithm for Seismic Data,” Radio Science Conference, 2004. Proceedings. 2004 Asia-Pacific, pp. 550–553, 2004.

M. E. Angelopoulou, K. Masselos, P. Y. Cheung, and Y. Andreopoulos, “Implementation and Comparison of the 5/3 Lifting 2D Discrete Wavelet Transform Computation Schedules on FPGAs ,” J. Signal Process. Syst., vol. 51, no. 1, pp. 3–21, Oct. 2008.

M. Khene and S. Abdul-Jauwad, “Adaptive seismic compression by wavelet shrinkage,” in Statistical Signal and Array Processing, 2000. Proceedings of the Tenth IEEE Workshop on. IEEE, 2000, pp. 544– 548.

M. Wolfe and P. C. Engineer, “Understanding the cuda data parallel threading model a primer,” The Portland Group Technical News, 2010.

Nvidia. Cuda c programming guide: Design guide. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide

S. Beak, B. Hieu, H. Lee, S. Choi, I. Kim, K. Lee, Y. Lee, and T. Jeong, “Novel binary tree Huffman decoding algorithm and field programmable gate array implementation for terrestrial-digital multimedia broadcasting mobile handheld ,” IET Sci. Meas. Technol, vol. 6, no. 6, pp. 527–528, 2012.

W. Wu, Z. Yang, Q. Qin, and F. Hu, “Adaptive Seismic Data Compression Using Wavelet Packets,” 2006 IEEE Int. Symp. Geosci. Remote Sens., no. 3, pp. 787–789, Jul. 2006.