Improved runtime performance of compression algorithms on CPU and GPU using CUDA

Milagros Mayta-Rosas[1], Henry Talavera-Díaz[1], Gonzalo Quispe-Huanca[1] , José Sulla-Torres, Mg. [1]
[1] Escuela Profesional de Ingeniería de Sistemas, Universidad Nacional de San Agustín, Arequipa, Perú.
ml.mayta.rosas@gmail.com, hen.talavera@gmail.com, gonzqh@gmail.com, jsulla@unsa.edu.pe

Abstract– In this paper we present a parallel implementation of Lempel-Ziv (LZ78) and Run Length Encoding (RLE) algorithms, originally sequential, using the parallel programming model and Compute Unified Device Architecture (CUDA), on a NVIDIA-branded GPU device. It presents a comparison between the execution time of the algorithms in CPU and in GPU demonstrating a significant improvement in the execution time of the process of data compression on the GPU in comparison with the implementation based on the CPU in both algorithms.

Keywords- CUDA, GPU, LZ78, Run Length Encoding, Lossless compression algorithms

Digital Object Identifier (DOI):http://dx.doi.org/10.18687/LACCEI2018.1.1.44
ISBN: 978-0-9993443-1-6
ISSN: 2414-6390

Mejora de rendimiento en tiempo de ejecución de los Algoritmos de Compresión en CPU y GPU utilizando CUDA

Milagros Mayta-Rosas[1], Henry Talavera-Díaz[1], Gonzalo Quispe-Huanca[1], José Sulla-Torres, Mg.[1]
[1]Escuela Profesional de Ingeniería de Sistemas, Universidad Nacional de San Agustín, Arequipa, Perú.
ml.mayta.rosas@gmail.com, hen.talavera@gmail.com, gonzqh@gmail.com, jsulla@unsa.edu.pe

***Abstract*— Currently users handle large amounts of data that are increasing, consequently the compression of these introduces an additional overhead and the performance of the hardware can be reduced, therefore must take into account the execution time as a key element to choose properly the algorithm perform this action. In this paper we present a parallel implementation of Lempel-Ziv (LZ78) and Run Length Encoding (RLE) algorithms, originally sequential, using the parallel programming model and Compute Unified Device Architecture (CUDA), on a NVIDIA-branded GPU device. It presents a comparison between the execution time of the algorithms in CPU and in GPU demonstrating a significant improvement in the execution time of the process of data compression on the GPU in comparison with the implementation based on the CPU in both algorithms.**

***Keywords*— CUDA, GPU, LZ78, Run Length Encoding, Lossless compression algorithms.**

***Resumen*— Actualmente los usuarios manejan grandes cantidades de datos que van en incremento, en consecuencia, la compresión de estos introduce una sobrecarga adicional y el rendimiento del hardware puede reducirse, por lo tanto, se debe tomar en cuenta el tiempo de ejecución como elemento clave para escoger adecuadamente el algoritmo que realice esta acción. En este artículo presentamos una implementación paralela de los algoritmos de compresión de datos sin pérdida Lempel-Ziv (LZ78) y Run Length Encoding (RLE), originalmente secuenciales, mediante el uso del modelo de programación paralela y la herramienta CUDA (Compute Unified Device Architecture), sobre un dispositivo GPU de marca NVIDIA. Se presenta una comparación entre el tiempo de ejecución de los dos algoritmos en CPU y en GPU demostrando una mejora significativa en el tiempo de ejecución del proceso de compresión de datos sobre la GPU en comparación con la implementación basada en la CPU en ambos algoritmos.**

***Palabras Clave*— CUDA, GPU, LZ78, Run Length Encoding, Algoritmos de Compresión sin pérdida.**

I. Introducción

En la actualidad la gran cantidad de datos que manejan los usuarios los obligan a utilizar métodos de compresión que permitan reducir el tamaño de estos sin tener pérdida de información en el proceso, el uso de algoritmos de compresión de datos es una tendencia cada vez más popular que conlleva una búsqueda del algoritmo de compresión más conveniente y rápido según el tipo de datos que se desea manejar.

Las tarjetas gráficas GPU (Graphics Processor Units) en la actualidad no tienen limitaciones para su uso en la investigación científica gracias a la creación de herramientas con este fin, entre ellas la herramienta CUDA (Compute Unified Device Architecture) de NVIDIA, que permite utilizar todo el potencial de las GPU mediante su modelo de programación paralela haciéndolas completamente programables para aplicaciones científicas además de añadir soporte para lenguajes de alto nivel como C y C++ [1].

El objetivo de esta investigación es realizar una comparación entre el tiempo de ejecución que toman los algoritmos de compresión sin Pérdida *Run Length Encoding* (RLE) y *Lempel Ziv* - 78 (LZ78) implementados de forma paralela tanto en CPU como en GPU mediante el uso de la herramienta CUDA de NVIDIA, con el fin de demostrar una mejora significativa en el tiempo de compresión de ambos algoritmos.

El resto de este artículo está estructurado de la siguiente manera. En la sección II se presentan algunos de los trabajos relacionados con un análisis de las diferentes investigaciones análogas a este trabajo. La sección III Materiales y Métodos, proporciona información sobre los algoritmos de compresión de datos analizados y la arquitectura CUDA. En la sección IV Diseño e Implementación, se presenta la descripción de la implementación paralela de los algoritmos en la arquitectura CUDA. La sección V Resultados, presentación cuantitativa de los tiempos de ejecución obtenidos en GPU y CPU de los algoritmos analizados con múltiples datos de entrada. Finalmente, en la sección VI Conclusiones, se presenta el análisis de los resultados y apreciaciones finales del trabajo realizado.

II. Estado del arte

Se han presentado diversos artículos científicos relacionados con el uso de CUDA y la paralelización de algoritmos secuenciales.

En el trabajo de Ozsoy y Swany [2], los autores presentan una implementación del algoritmo de compresión de datos sin pérdida de Lempel-ZivStorer-Szymanski (LZSS) mediante el uso del framework CUDA (Compute Unified Device Architecture) de NVIDIA GPU, muestran una mejora significativa en el rendimiento del proceso de compresión en comparación con la implementación basada en la CPU.

Digital Object Identifier (DOI): http://dx.doi.org/10.18687/LACCEI2018.1.1.44
ISBN: 978-0-9993443-1-6
ISSN: 2414-6390

Patel et al. [3], presentan algunos algoritmos paralelos e implementaciones de un esquema de compresión de datos sin pérdidas tipo *bzip*2 para arquitecturas de GPU, su enfoque paraleliza tres etapas principales en la tubería de compresión *bzip*2: transformación de *Burrows-Wheeler* (BWT), transformación de movimiento a frente (MTF) y codificación de *Huffman*.

Gilchrist [4], describe una implementación paralela del programa de compresión sin pérdidas *bzip*2 *block-sorting*, comparando el rendimiento de la implementación paralela con el programa *bzip*2 secuencial que se ejecuta en varias arquitecturas paralelas de memoria compartida. Sus resultados muestran que se logra una aceleración casi lineal significativa utilizando el programa *bzip*2 paralelo en sistemas con múltiples procesadores.

En el trabajo de Mielikainen, Honkanen, Toivanen y Huang [5], presentan la implementación de un método de compresión de datos de imágenes espectrales llamado *Linear Prediction* con Coeficientes Constantes (LP-CC) usando la arquitectura CUDA de computación paralela de NVIDIA, su implementación de la GPU se compara experimentalmente con la implementación nativa de la CPU mostrando resultados aceptables.

Patel, Wongm Tatikonda y Marczewski [6], exploran las posibles mejoras de rendimiento que podrían obtenerse mediante el uso de técnicas de procesamiento de GPU dentro de la arquitectura CUDA para el algoritmo de compresión JPEG. La elección de algoritmos de compresión como el foco se basó en ejemplos de paralelismo de nivel de datos que se encuentran dentro de los algoritmos y un deseo de explorar la eficacia de la gestión de algoritmos cooperativos entre el sistema de CPU y una GPU disponible.

Cloud et al. [7], presentan una modificación del algoritmo de *Huffman* que permite que los datos sin comprimir se descompongan en bloques independientes comprimibles y descomprimibles, permitiendo la compresión y descompresión concurrentes en múltiples procesadores, modificado en una GPU NVIDIA, mostrando un rendimiento favorable de GPU para casi todas las pruebas.

En el trabajo de Fang, He y Luo [8], los autores implementan nueve esquemas de compresión ligeros en la GPU y estudian las combinaciones de estos esquemas para una mejor relación de compresión. Diseñan un planificador de compresión para encontrar la combinación óptima y sus experimentos demuestran que la compresión basada en GPU y la descompresión alcanzaron una velocidad de procesamiento de hasta 45 y 56 GB/s, respectivamente.

La investigación de Franco, Bernabé, Fernández y Acacio [9], nos presentan la paralelización en CUDA de una transformada wavelet en 2D en una tarjeta gráfica la NVIDIA Tesla C870, con la cual, logran alcanzar una aceleración de 20.8 para un tamaño de 8192 x 8192 en comparación con la implementación en *OpenMP*. Estos resultados lo consideran aceptable dentro de los objetivos de la investigación.

III. Materiales y Métodos

A. *Algoritmo Run Length Encoding*

RLE, pertenece a la clase de algoritmos de diccionario adaptativo propuesto por Ziv y Lempel, (1978) con los datos almacenados como pares de frecuencia y valor. Existen numerosas variantes, en la Fig. 1, se muestra el pseudocódigo de la estructura básica [10].

```
runLengthEncoding (in, n, symbolsOut, countsOut)
 1    index ← 0
 2    for i ← 0 : n
 3      frecuency ← 1
 4      while i + 1 < n and in[i] = in[i +1]
 5        frecuency ← frecuency+1
 6        i ← i + 1
 7      end while
 8      symbolsOut[index] ← in[i]
 9      countsOut[index] ← frecuency
10      index ← index + 1
11    end for
```

Fig. 1 Pseudocódigo de la Estructura Básica de RLE.

B. *Algoritmo Lempel-Ziv-78*

LZ77 y LZ78 son dos algoritmos de compresión de datos sin pérdidas publicados por Abraham Lempel y Jacob Ziv en 1977 y 1978. También se les conoce como LZ1 y LZ2 respectivamente. Estos dos algoritmos forman la base de muchas variaciones, incluyendo LZW, LZSS, LZMA y otros.

Ambos son teóricamente codificadores de diccionario. LZ77 mantiene una ventana deslizante durante la compresión. Esto demostró ser equivalente al diccionario explícitamente construido por LZ78, sin embargo, sólo son equivalentes cuando toda la información está destinada a ser descomprimida [11].

LZ78 tiene un diccionario que contiene las cadenas que han ocurrido previamente. El diccionario está vacío inicialmente y su tamaño está limitado por la memoria disponible. Para ilustrar la forma en la que el método funciona, considérese un diccionario (arreglo lineal), de N localidades con la capacidad de almacenar una cadena de símbolos en cada una de ellas. El diccionario se inicializa guardando en la posición cero del diccionario la cadena vacía. El algoritmo de codificación se muestra en la Fig. 2.

El proceso es iterativo y termina cuando ya no existen más símbolos a la entrada para codificar. En cada iteración S se inicializa a Null (S = Null indica una cadena vacía que siempre se encuentra en la posición cero del diccionario).

El símbolo X del archivo de entrada se lee y se busca la cadena S·X (concatenación de S y X) en el diccionario, si la cadena S·X se encuentra en el diccionario, S es ahora S·X y se lee un nuevo símbolo X. Nuevamente, se busca S·X en el diccionario y si la cadena se encuentra, se vuelve a leer otro símbolo de entrada y el proceso se repite buscando nuevamente S·X en el diccionario. Si la cadena S·X no se encuentra en el diccionario, se guarda la cadena S·X en una posición disponible en el diccionario y se escribe al archivo de salida la posición de S dentro del diccionario y el símbolo X

[12]. En la Fig. 2, se muestra el pseudocódigo de la estructura básica del algoritmo LZ78.

```
1   Dictionary; Prefix; DictionaryIndex = 1;
2   while(!isEmpty(characterStream))
3           Char  = next_character in characterStream;
4           if (Prefix + Char exist in Dictionary)
5                   Prefix = Prefix + Char;
6           else
7                   if (isEmpty(Prefix))
8                           CodeWordForPrefix  = 0;
8                   else
9                           CodeWordForPrefix =  DictionaryInc
10                          Output: (CodeWordForPrefix, );
11                          InsertinDictionary  ((  DictionaryInde
                    Char ));
12                          DictionaryIndex ++;
13                          Prefix = NULL;
14  if(!isEmpty(Prefix))
15          CodeWordForPrefix = DictionaryIndex for Prefix;
16          Output: (CodeWordForPrefix, );
```

Fig. 2 Pseudocódigo del Algoritmo de Codificación en LZ78 [11].

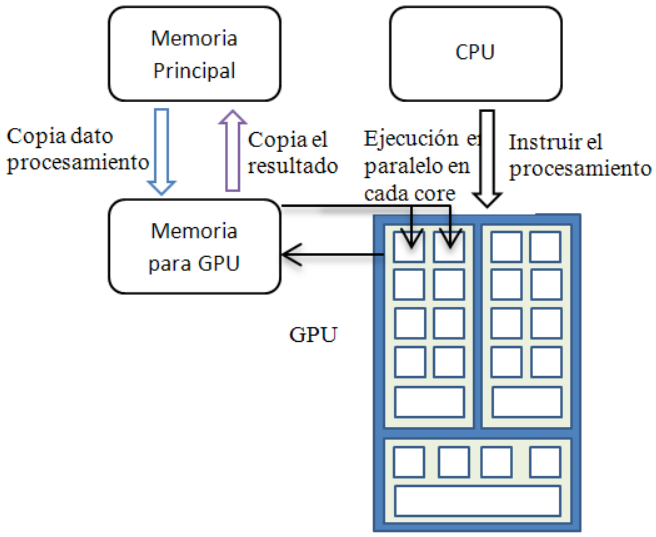

Fig. 3 Flujo de Procesamiento en CUDA

C. *Arquitectura Unificada de Dispositivos de Cómputo*

CUDA es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la potencia de la GPU, La plataforma de computación CUDA se extiende desde los 1000 procesadores de computación de uso general que figuran en la arquitectura de computación de la GPU NVIDIA, extensiones de computación paralela a muchos lenguajes populares, poderosas bibliotecas aceleradas para convertir aplicaciones clave y aplicaciones de computación basadas en la nube. CUDA se extiende más allá del popular CUDA Toolkit y el lenguaje de programación CUDA C / C ++ [13].

CUDA utiliza un modelo de programación paralela diseñado para cubrir por completo el incremento de los núcleos de las GPU y manteniendo la accesibilidad a los programadores familiarizados con los lenguajes C y C++. Su núcleo posee tres abstracciones clave: Una jerarquía de grupos de hilos, memorias compartidas y sincronización de barreras. Estas abstracciones proporcionan paralelismo de datos de grano fino y paralelismo de hilos, anidados dentro del paralelismo de datos de grano grueso y paralelismo de tareas. Estas guían al programador para dividir el problema en subproblemas que pueden ser resueltos independientemente en paralelo por bloques de hilos y cada sub-problema en piezas más finas que pueden ser resueltas cooperativamente en paralelo por todos los hilos dentro del bloque, cada bloque de subprocesos puede programarse en cualquiera de los multiprocesadores disponibles dentro de una GPU [14].

El flujo de procesamiento en CUDA se aprecia en la Fig. 3, primero se copian los datos de entrada de la memoria de la CPU a la memoria de la GPU. Se carga el programa en la GPU y se ejecuta ubicando datos en caché para mejorar el rendimiento.

En este trabajo se utilizó CUDA, por la fiabilidad de las herramientas y la disponibilidad de hardware de NVIDIA. Además se utilizó CUDA Toolkit 8.0 [15], que proporciona un entorno de desarrollo integral para desarrolladores de C y C ++ que crean aplicaciones aceleradas por GPU, incluye un compilador para GPUs NVIDIA, bibliotecas matemáticas y herramientas para depurar y optimizar el rendimiento de las aplicaciones. Para la paralelización del algoritmo LZ78, se utilizó un estándar OpenACC (Para Aceleradores Abiertos) el cual es un estándar de programación para la informática paralela desarrollada por Cray, CAPS, NVIDIA y PGI. El estándar está diseñado para simplificar la programación paralela de sistemas heterogéneos de CPU/GPU [16].

D. *Hardware Graphics Processor Unit (GPU) y CPU*

La unidad de procesamiento gráfico utilizada en esta investigación es una GPU NVIDIA GeForce 840m. sobre un procesador Intel Core i5 4210u.

IV. Desarrollo e Implementación

Para el desarrollo de la propuesta se implementó en primera instancia el algoritmo RLE utilizando CUDA y posteriormente se implementó el algoritmo *Lempel-Ziv-78* también con CUDA; luego se ejecutó las pruebas con cifras de diferentes cantidades completamente aleatorias y convenientemente comprimibles, para finalmente observar los resultados comparando el mejoramiento del tiempo de ejecución entre los algoritmos.

A. *Implementación Paralela del Algoritmo Run Length Encoding utilizando CUDA.*

Para la paralelización de RLE, se debe calcular los índices de los elementos que deben ser almacenados y sus símbolos, esta propuesta, original de la autora Balevic [17].

Como se muestra en la Fig. 4, el enfoque de esta modificación de RLE crea a partir del arreglo de entrada un arreglo de banderas que indica el inicio de una nueva cadena de símbolos y a partir de este último un nuevo arreglo con los índices de aparición de cada símbolo en el arreglo de salida.

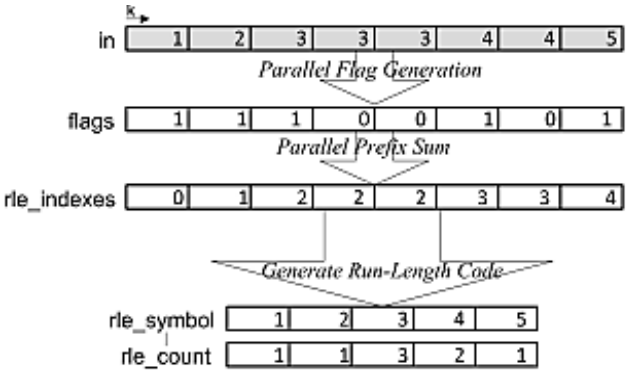

Fig. 4 Perspectiva de RLE paralelo [11].

En la Fig. 5, se muestra el pseudocódigo de la creación del arreglo de banderas *backwardMask* a partir del arreglo de entradas in, cada iteración del ciclo *for* de todos los pasos del algoritmo es ejecutado en hilo diferente, ya que en ningún momento de la ejecución se requiere de resultados futuros de los otros hilos.

```
maskKernel (in, backwardMask, n)
1     for i ← 0 : n
2       if i = 0 then
3         backwardMask[i] ← 1
4       else
5         if in[i] = in[i - 1] then
6           backwardMask[i] ← 0
7         else
8           backwardMask[i] ← 1
9         end if
10      end if
11    end for
```

Fig 5. Pseudocódigo de maskKernel.

El segundo arreglo, *scannedBackwardMask*, consiste en una suma de prefijos de *BackwardMask*, se utilizó la incluida en la librería de primitivas paralelas desarrollada por Markus Billeter y su equipo, "*chag::pp*", que según sus autores es la más rápida en existencia.

Como último paso crítico del algoritmo creamos un arreglo a partir de *scannedBackwardMask*, que contendrá la posición del inicio de una secuencia de símbolos repetidos en el arreglo de entrada in, se detalla la creación de este arreglo, llamado *compactedBackwardMask*, en el pseudocódigo de la Fig. 6.

```
compactKernel (sBM, compactedBackwardMask, totalRuns, n)
1     for i ← 0 : n
2       if i = n - 1 then
3         compactedBackwardMask[sBM[i]] ← i + 1
4         totalRuns ← sBM[i]
5       end if
6       if i = 0 then
7         compactedBackwardMask[0] ← 0
8       else
9         if sBM[i] != sBM[i - 1] then
10          compactedBackwardMask[sBM[i] - 1] ← i
11        end if
12      end if
13    end for
```

Fig. 6 Pseudocódigo de compactKernel (sBM es el arreglo scannedBackwardMask).

En base al último arreglo creado podemos, mediante *scatterKernel*, crear los arreglos de salida *symbolsOut* y *countsOut*, colocando en el primero el símbolo del arreglo de entrada que corresponde a la posición indicada por cada elemento de *compactedBackwardMask*, y en el arreglo de contadores la resta de cada posición de *compactedBackwardMask* con la anterior; este procedimiento se aprecia en la Fig. 7.

```
scatterKernel (cBM, totalRuns, in, symbolsOut, countsOut)
1     n ← totalRuns
2     for i ← 0 : n
3       a ← cBM[i]
4       b ← cBM[i + 1]
5       symbolsOut[i] = in[a]
6       countsOut[i] = b – a
7     end for
```

Fig. 7 Pseudocódigo de scatterKernel (cBM es el arreglo compactedBackWardMask)

B. Implementación Paralela del Algoritmo Lempel-Ziv-78 utilizando CUDA.

Para la implementación de este algoritmo basado en diccionarios, se hizo uso del lenguaje C, el cual es un lenguaje básico de CUDA, siguiendo el pseudocódigo básico del algoritmo LZ78.

El algoritmo Paralelo LZ78 es análogo al original, con diferencias en la ejecución del *loop*, el cual mediante el *OpenAcc* se ejecuta de forma paralela y con un arreglo de entrada replicado en memoria global de la GPU con la finalidad de reducir el tiempo de procesamiento al intercambiar datos entre la GPU y la CPU mediante el bus PCI-Express. La cadena de datos original se divide entre tantos bloques nos permite los arreglos de caracteres en CUDA. En la Fig. 8, se presenta el pseudocódigo de LZ78 con el uso de las Directivas *OpenAcc*.

```
1   Dictionary: Prefix; DictionaryIndex =1;
2   #pragma acc data copy(characterStream)
3   #pragma acc kernels
4   While(!isEmpty(characterStream))
5               Char = next_character in characterStream;
6               If( Prefix + Char exist in Dictionary )
7                           Prefix = Prefix + Char;
8               Else
9                           If ( isEmpty(Prefix) )
10                          CodeWordForPrefix = 0;
11                          Else
12                          CodeWordforPrefix = DiccionaryIndex;
13                          Output: (CodeWordforPrefix, );
14                          InsertinDictionary (( DiccionarIndex Char ));
15                          DictionaryIndex++;
16                          Prefix = NULL;
17  If( !isEmpty(Prefix) )
18              CodeWordForPrefix  = DictionaryIndex for Prefix;
19              Output: (codeWordForPrefix);
```

Fig. 8 Pseudocódigo de LZ78 con el uso de Directivas OpenAcc.

V. RESULTADOS

Los algoritmos paralelos se ejecutaron para cadenas de datos enteros de distinta longitud, los cuales se dividen en los siguientes casos de evaluación:

- Datos aleatorios, son una cadena de datos de longitud finita que contienen cifras aleatorias de 0 a 9.
- Datos convenientemente comprimibles, son una cadena de datos de longitud finita que contienen cifras numéricas de 0 a 9, con la regla de contener cifras del mismo valor por bloques incrementales de tipo 000011112222...9999.

En las Fig. 9 y Fig. 10, se presentan gráficos comparativos de tiempos de ejecución a partir de los resultados obtenidos en las pruebas realizadas sobre datos completamente aleatorios.

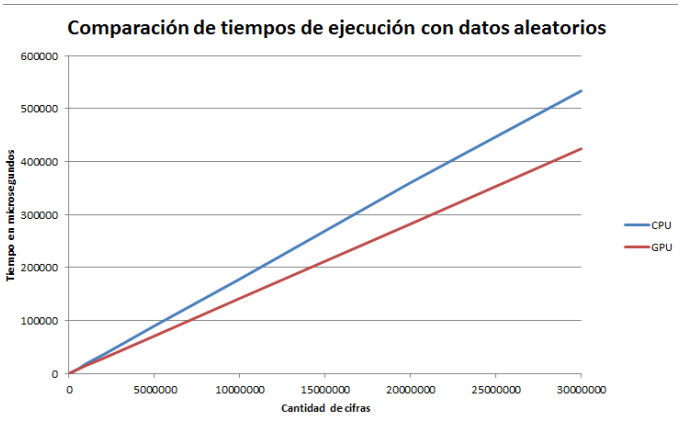

Fig. 9 Resultados de la Compresión Run Length sobre datos completamente aleatorios.

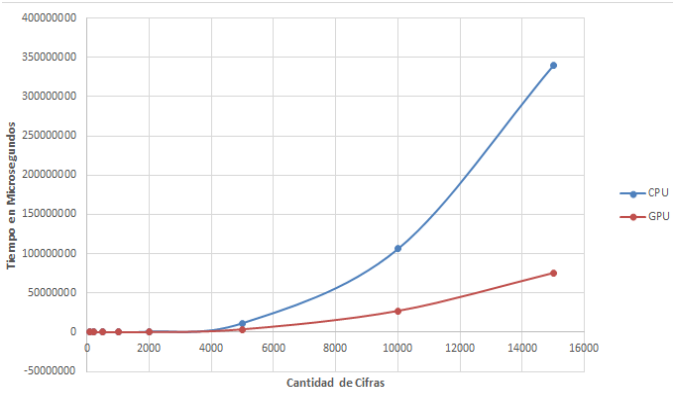

Fig. 10 Resultados de la Compresión LZ78 sobre datos completamente aleatorios.

En las fig. 11 y fig. 12 se presentan gráficos comparativos de tiempos de ejecución a partir de los resultados obtenidos en las pruebas realizadas sobre datos convenientemente comprimibles.

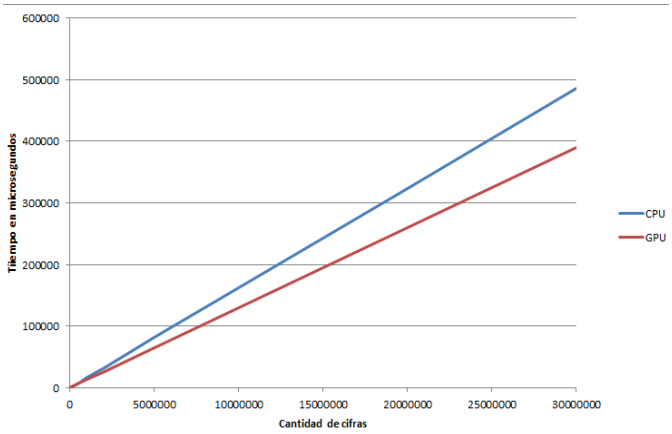

Fig. 11 Resultados de la Compresión Run Length sobre datos convenientemente comprimibles.

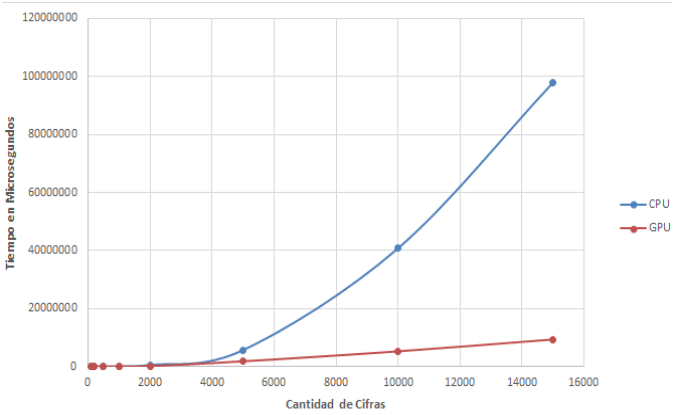

Fig. 12 Resultados de la Compresión LZ78 sobre datos convenientemente comprimibles.

Como se puede observar en las Fig. 9 y Fig. 11, el tiempo de ejecución de RLE mantiene una relación lineal con la cantidad de datos de entrada, por otro lado, en la Fig. 10 y la

Fig. 12, el tiempo de ejecución del algoritmo LZ78 en CPU y GPU crece en función de la cantidad de datos de entrada, siendo el tiempo de ejecución paralela en GPU menor a la ejecución en CPU.

En la Tabla I se presentan los resultados del tiempo de ejecución, en promedio de microsegundos, que tomó comprimir cifras aleatorias de datos en CPU y GPU con el algoritmo Run Lenght.

TABLA I
RESULTADOS DEL TIEMPO DE COMPRESIÓN DE DATOS CONVENIENTEMENTE COMPRIMIBLES Y ALEATORIOS UTILIZANDO LA COMPRESIÓN RUN LENGTH

	Algoritmo Run Length			
	Datos Completamente Aleatorios		Datos Convenientemente Comprimibles	
Cantidad de Cifras	Ejecución en CPU Promedio (ms)	Ejecución en GPU Promedio (ms)	Ejecución en CPU Promedio (ms)	Ejecución en GPU Promedio (ms)
10000	178	696	159	645
50000	897	1361	806	1205
100000	1768	2136	1601	1845
200000	3657	3613	3143	3131
500000	9082	9082	8004	7005
1000000	18060	18060	16162	13448
2000000	35867	35867	32501	26365
5000000	90172	90172	81018	65236
10000000	178349	178349	161695	130177
20000000	359370	359370	322745	260204
30000000	534297	534297	485802	390242

Se puede apreciar en la Tabla I, que de las pruebas realizadas los mejores resultados se obtiene cuando se tienen los datos convenientemente comprimibles y con una cantidad mayor a las 200000 cifras.

En la Tabla II se presentan los resultados del tiempo de ejecución, en promedio de microsegundos, que tomó comprimir cifras aleatorias de datos en CPU y GPU con el algoritmo LZ78.

TABLA II
RESULTADOS DEL TIEMPO DE COMPRESIÓN DE DATOS CONVENIENTEMENTE COMPRIMIBLES Y ALEATORIOS UTILIZANDO LA COMPRESIÓN LZ78

	Algoritmo LZ78			
	Datos Completamente Aleatorios		Datos Convenientemente Comprimibles	
Cantidad de Cifras	Ejecución en CPU Promedio (ms)	Ejecución en GPU Promedio (ms)	Ejecución en CPU Promedio (ms)	Ejecución en GPU Promedio (ms)
100	1422.6	361.8	1491.8	446.8
200	6129.6	1940.2	6610.8	2031.6
500	36254.2	15028.6	34612.2	17567
1000	151710.4	55682.2	115606.8	42325.4
2000	935191.4	312063.6	601752.8	204791.4
5000	11561604	3761338.2	5705657.4	1840008.6
10000	106007448	27234960	40809999	5288517
15000	339444262	75597683	97967774	9395694.8

De manera similar, la interpretación de la Tabla II, muestra que los mejores resultados se obtienen cuando se tienen los datos aleatorios y convenientemente comprimibles siendo el tiempo de ejecución superiores con una cantidad mayor a las 1000 cifras.

VI. CONCLUSIONES

Se ha comparado el tiempo de ejecución de los Algoritmos de compresión sin pérdida RLE y LZ78 en CPU y GPU con diferentes números de datos. El Algoritmo con menor tiempo de ejecución tanto en CPU como en GPU es el Algoritmo RLE mostrando un tiempo de ejecución lineal a diferencia de LZ78 el cual describe una semiparábola. Ambos algoritmos en su versión paralela en GPU obtuvieron un tiempo de ejecución menor con respecto a sus implementaciones convencionales en CPU, logrando en RLE reducir un 21% de tiempo de ejecución en 30 millones de datos aleatorios y un 20% en la misma cantidad de datos convenientemente comprimibles y en LZ78 se logró reducir en promedio un 31% de tiempo de ejecución en datos aleatorios y un promedio de 29% en datos convenientemente comprimibles.

Respecto a la paralelización de RLE aplicada en esta comparación, es destacable que depende elementalmente de la cantidad de hilos que pueden generarse en la creación de cada arreglo descrito la sección IV, por ende es deducible que a mayor cantidad de núcleos hayan disponibles en la GPU para la creación de hilos, mayor sería el nivel de paralelización y menor el tiempo de ejecución, guardando así una relación proporcionalmente inversa entre el tiempo de ejecución y la cantidad de núcleos de la GPU.

REFERENCIAS

[1] C. Represa, J. Cámara, P. Sánchez, "Introducción a la Programación en CUDA: v. 3.1"., Universidad de Burgos, 2016.

[2] A. Ozsoy, M.Swany, "CULZSS: LZSS lossless data compression on CUDA" En Cluster Computing (CLUSTER), 2011 IEEE International Conference on. IEEE. p. 403-411, 2011.

[3] R. Patel, Y.Zhang, J. Mark, A. Davidson y J. Owens. "Parallel Lossless Data Compression on the GPU" IEEE, 2012.

[4] J. Gilchrist, "Parallel Data Compression with BZIP2", Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems, Vol. 16, pp. 559-554, 2004.

[5] J. Mielikainen, R. Honkanen, P. Toivanen y B. Huang, "GPUs for data parallel spectral image compression", Proceedings of SPIE Optical Engineering Applications. International Society for Optics and Photonics, p. 74550C-74550C-8, 2009.

[6] P. Patel, J. Wong, M. Tatikonda y J. Marczewski, "JPEG Compression Algorithm Using CUDA" Department of Computer Engineering, University of Toronto, Course Project for ECE, vol. 1724, 2009.

[7] R. Cloud, M. Curry, H. Ward, A. Skjellum y P. Bangalope, "Accelerating Lossless Data Compression with GPUs" arXiv preprint arXiv:1107.1525, 2011.

[8] W. Fang, B. He, y Q. Luo. "Database compression on graphics processors", Proceedings of the VLDB Endowment, vol. 3, no 1-2, p. 670-680, 2010.

[9] J. Franco, G. Bernabé, J. Fernández and M. Acacio, "A Parallel Implementation of the 2D Wavelet Transform Using CUDA" En Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on. IEEE, p. 111-118, 2009.

[10]G. Davis, L. Lau, R. Young, F. Duncalfe, and L. Brebber, “Parallel Run Length Encoding Compression: Reducing I/O in Dynamic Environmental Simulations ”, International Journal of High Performance Computing Applications, Vol. 12, N° 4, pp. 396 - 410. 1998.
[11]J. Ziv, A. Lempel, “Compression of Individual Sequences via Variable-Rate Coding” IEEE Transactions on Information Theory, Vol, IT-24, N° 5, 1978.
[12]M. Morales, “Notas sobre Compresión de Datos”, INAOE, 2003.
[13]NVidia, “CUDA”, [Online]. Available: http://www.nvidia.es/object/cuda-parallel-computing-es.html
[14]NVidia “CUDA C Programming guide 7.50”, [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#from-graphics-processing-to-general-purpose-parallel-computing
[15]C. Zeller, NVIDIA Corporation, “Supercomputing 2011 tutorial”, [Online]. Available: http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf
[16] “Nvidia, Cray, PGI, and CAPS launch ‘OpenACC' programming standard for parallel computing”,[Online]. Available: http://www.theinquirer.net/inquirer/news/2124878/nvidia-cray-pgi-caps-launch-openacc-programming-standard-parallel-computing, The Inquirer. 4 de noviembre de 2011.
[17]A. Balevic, “Fine-grain Parallelization of Entropy Coding on GPGPUs”, 2008.