Comparación de Rendimiento en Tiempo de Ejecución de los Algoritmos de Compresión en CPU y GPU Utilizando CUDA.

Mayta Rosas Milagros Lizet, Talavera Díaz Henry Abraham, Gonzalo Emiliano Quispe Huanca, Sulla Torres Jose

Escuela Profesional de Ingeniería de Sistemas, Facultad de Ingenierías de Producción y Servicios, Universidad Nacional de San Agustín, Arequipa, Perú.

ml.mayta.rosas@gmail.com
hen.talavera@gmail.com
gonzqh@gmail.com
josullato@gmail.com

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.

Key words — **CUDA, GPU, LZ78, Run Length Encoding, Algoritmos de 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, 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 reducción 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. Trabajos Relacionados

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

En [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.

Patel, Zhang, Mak, Davidson y Owens presentan algunos algoritmos paralelos e implementaciones de un esquema de compresión de datos sin pérdidas tipo bzip2 para arquitecturas de GPU, su enfoque paraleliza tres etapas principales en la tubería de compresión bzip2: transformación de Burrows-Wheeler (BWT), transformación de movimiento a frente (MTF) y codificación de Huffman [3].

Gilchrist describe en [4], una implementación paralela del programa de compresión sin pérdidas bzip2 block-sorting, comparando el rendimiento de la implementación paralela con el programa bzip2 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 bzip2 paralelo en sistemas con múltiples procesadores.

En [5], los autores 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.

En [6], los autores 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, Curry, Ward, SKjellum y Bangalore, en [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 [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 en [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.

III. Materiales y Métodos

A. Algoritmo Run Length Encoding

RLE, pertenece a la clase de algoritmos de diccionario adaptativo (Ziv y Lempel, 1977) 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 [12].

```
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 [10].

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 [11]. 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 [10].

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 [15].

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.

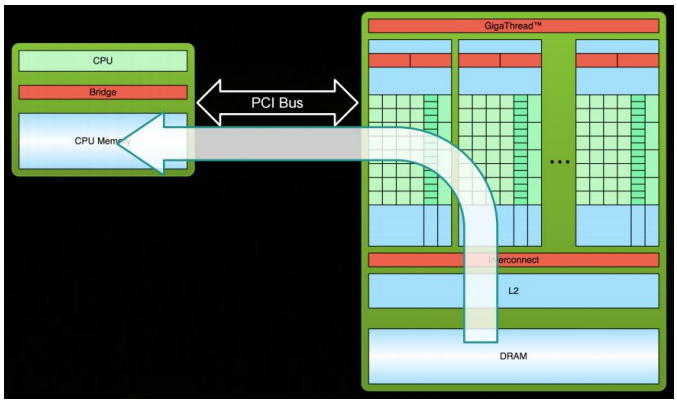

Fig. 3 Flujo de Procesamiento en CUDA. [16]

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 [13], 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 [17].

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

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 Ana Balevic [14], es una modificación del algoritmo RLE cuyo primer enfoque para calcular los códigos se basa en el uso de otra primitiva paralela, la reducción, para resumir el número de veces que un símbolo apareció en su ejecución. En lugar de acumular el número de ocurrencias para cada símbolo en paralelo, los índices de los elementos de la línea se determinan en base a las banderas. Estos valores se utilizan para calcular el número total de elementos que aparecen entre estas ubicaciones: el recuento resultante corresponde al número de veces que un elemento apareció en su ejecución.

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.

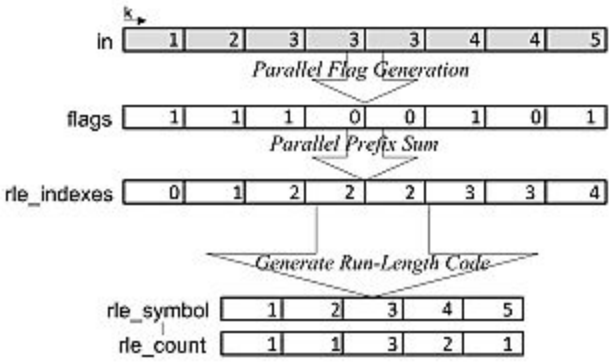

Fig. 4 Perspectiva de RLE paralelo [14].

La implementación de este algoritmo se detalla en las siguientes imágenes, que muestran el pseudocódigo de cada paso del mismo; en la fig. 5, se expone el pseudocódigo de la creación del arreglo de banderas *backwardMask* a partir del arreglo de entradas *in,* cada iteración del ciclo *for* de este paso 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,* existen muchas implementaciones de suma de prefijos y utilizamos 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,* en este último arreglo se tendrá toda lo necesario para crear los dos arreglos de salida del algoritmo de Ana Balevic, se detalla la creación de este arreglo, llamado *compactedBackwardMask,* en el pseudocódigo de la fig. 6, en

la que al igual que en el primer paso cada iteración del ciclo *for* es un hilo independiente.

```
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,* con un procedimiento simple, colocando en el arreglo símbolos 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 posiciones 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 visto en la fig. 2.

El algoritmo Paralelo LZ78 es análogo al original, con diferencias en el la ejecución del *loop*, el cual mediante el OpenAcc se ejecuta de forma paralela y con un array de entrada copiado 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 tablas I y II se presentan los resultados en microsegundos del tiempo de ejecución necesarios para comprimir cifras aleatorias de datos en CPU.

TABLA I
RESULTADOS DE LA COMPRESIÓN EN RUN LENGTH EN ARREGLOS DE CIFRAS COMPLETAMENTE ALEATORIAS

Cantidad de Cifras	T1 (ms)	T2 (ms)	T3 (ms)	T4 (ms)	T5 (ms)	Promedio
10000	177	178	178	179	178	178
50000	883	890	883	897	932	897
100000	1790	1759	1764	1769	1760	1768
200000	3809	3558	3800	3532	3586	3657
500000	9232	8960	9119	8910	9186	9082
1000000	17973	17677	18261	18265	18123	18060
2000000	35058	35821	36091	36276	36090	35867
5000000	89658	90024	89732	90371	91075	90172
10000000	178590	177094	177595	178544	179925	178349
20000000	358274	359912	362948	358172	357547	359370
30000000	528378	536170	535976	532810	538153	534297

TABLA II
RESULTADOS DE LA COMPRESIÓN EN LZ78 EN ARREGLOS DE CIFRAS COMPLETAMENTE ALEATORIAS

Cantidad de Cifras	T1 (ms)	T2 (ms)	T3 (ms)	T4 (ms)	T5 (ms)	Promedio
100	1592	1585	1487	1170	1279	1422.6
200	6635	3615	7303	7263	5832	6129.6
500	29871	23236	23258	54211	50695	36254.2
1000	151841	142049	152169	170102	142391	151710.4
2000	971261	962867	915257	914237	912335	935191.4
5000	10773146	11766811	11790423	11738288	11739350	11561603.6
10000	108034358	104171963	105908000	105621769	106301152	106007448.4
15000	347960239	350088465	329111606	331202460	338858539	339444261.8

En las tabla III y IV se presentan los resultados en microsegundos del tiempo de ejecución necesarios para comprimir cifras aleatorias de datos en GPU.

TABLA III
RESULTADOS DE LA COMPRESIÓN EN RUN LENGTH EN ARREGLOS DE CIFRAS COMPLETAMENTE ALEATORIAS

Cantidad de Cifras	T1 (ms)	T2 (ms)	T3 (ms)	T4 (ms)	T5 (ms)	Promedio
10000	696	696	696	696	696	696
50000	1360	1360	1361	1361	1361	1361
100000	2139	2125	2137	2138	2138	2136
200000	3630	3553	3645	3594	3640	3613
500000	7798	7636	7823	7742	7814	7763
1000000	14429	14434	14431	14430	14432	14431
2000000	28371	28369	28361	28365	28370	28367
5000000	70484	70499	70479	70492	70477	70486
10000000	141108	141083	141088	141078	141136	141099
20000000	282600	282590	282601	282558	282570	282584
30000000	424932	424953	424942	424849	424962	424927

TABLA IV
RESULTADOS DE LA COMPRESIÓN EN LZ78 EN ARREGLOS DE CIFRAS COMPLETAMENTE ALEATORIAS

Cantidad de Cifras	T1 (ms)	T2 (ms)	T3 (ms)	T4 (ms)	T5 (ms)	Promedio
100	166	554	299	556	234	361.8
200	2526	2276	1133	2491	1275	1940.2
500	14258	7718	24174	10409	18584	15028.6
1000	57975	48122	49830	52526	69958	55682.2
2000	327385	311971	294099	299260	327603	312063.6
5000	3756615	3766283	3774235	3733878	3775680	3761338.2
10000	26780511	26950798	27500394	27783067	27160028	27234959.6
15000	75788511	75676139	75388334	75388686	75746747	75597683.4

En las tabla V y VI se presentan los resultados en microsegundos del tiempo de ejecución necesarios para comprimir convenientemente comprimibles de datos en CPU.

TABLA V
RESULTADOS DE LA COMPRESIÓN EN RUN LENGTH EN ARREGLOS DE CIFRAS COMPLETAMENTE ALEATORIAS

ARREGLOS DE CIFRAS CONVENIENTEMENTE COMPRIMIBLES

Cantidad de Cifras	T1 (ms)	T2 (ms)	T3 (ms)	T4 (ms)	T5 (ms)	Promedio
10000	160	159	159	160	159	159
50000	832	798	798	800	799	806
100000	1604	1597	1598	1601	1603	1601
200000	3188	2946	3195	3197	3191	3143
500000	7866	8013	8039	8022	8078	8004
1000000	16009	16268	16194	16170	16167	16162
2000000	32423	32164	32325	32862	32731	32501
5000000	81029	81037	81108	80903	81014	81018
10000000	161703	162392	161859	161436	161082	161695
20000000	324040	322822	324175	324355	318331	322745
30000000	485000	486953	487519	484941	484597	485802

ARREGLOS DE CIFRAS CONVENIENTEMENTE COMPRIMIBLES

Cantidad de Cifras	T1 (ms)	T2 (ms)	T3 (ms)	T4 (ms)	T5 (ms)	Promedio
10000	645	645	645	645	645	645
50000	1205	1205	1206	1205	1205	1205
100000	1845	1846	1845	1845	1845	1845
200000	3130	3131	3131	3131	3131	3131
500000	7004	7004	7005	7006	7004	7005
1000000	13447	13448	13446	13451	13448	13448
2000000	26364	26362	26368	26367	26363	26365
5000000	65237	65233	65241	65232	65238	65236
10000000	130170	130170	130163	130194	130187	130177
20000000	260198	260231	260220	260179	260193	260204
30000000	390248	390235	390231	390273	390225	390242

TABLA VI
RESULTADOS DE LA COMPRESIÓN EN LZ78 EN ARREGLOS DE CIFRAS CONVENIENTEMENTE COMPRIMIBLES

Cantidad de Cifras	T1 (ms)	T2 (ms)	T3 (ms)	T4 (ms)	T5 (ms)	Promedio
100	1389	1634	1464	1502	1470	1491.8
200	5450	6681	7513	7089	6321	6610.8
500	51794	27522	21285	52356	20104	34612.2
1000	111833	112239	112231	127657	114074	115606.8
2000	580201	605167	589184	579912	654300	601752.8
5000	56886955	5720101	57263625	57057945	56873355	5705657.4
10000	40094300	41580569	40765273	40789974	40819879	40809999
15000	97261077	98814774	98098827	97867837	97796357	97967774.4

TABLA VIII
RESULTADOS DE LA COMPRESIÓN EN LZ78 EN ARREGLOS DE CIFRAS CONVENIENTEMENTE COMPRIMIBLES

Cantidad de Cifras	T1 (ms)	T2 (ms)	T3 (ms)	T4 (ms)	T5 (ms)	Promedio
100	296	500	374	565	499	446.8
200	1752	2038	2618	2600	1150	2031.6
500	16959	18719	18038	16153	17966	17567
1000	40673	60388	36992	34775	38799	42325.4
2000	192602	244909	191249	209256	185941	204791.4
5000	18591110	1816336	18226625	18335595	18683777	1840008.6
10000	49759938	5357849	53574463	54375844	53134491	5288517
15000	93375591	9336096	93711718	94273403	95056666	9395694.8

En las tabla VII y VIII se presentan los resultados en microsegundos del tiempo de ejecución necesarios para comprimir convenientemente comprimibles de datos en GPU.

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.

TABLA VII
RESULTADOS DE LA COMPRESIÓN EN RUN LENGTH EN

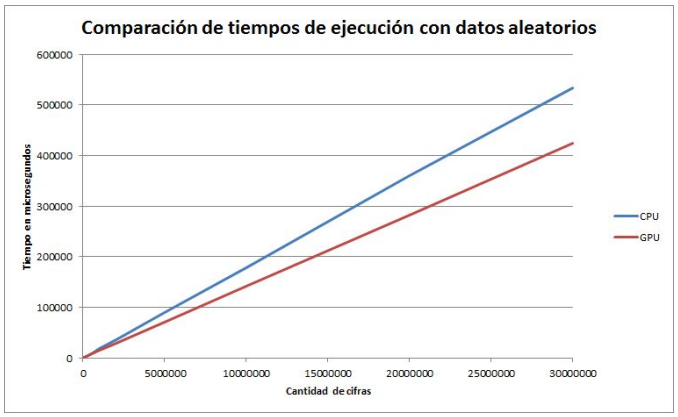

Fig. 9 Resultados de la Compresión Run Length sobre datos completamente aleatorios.

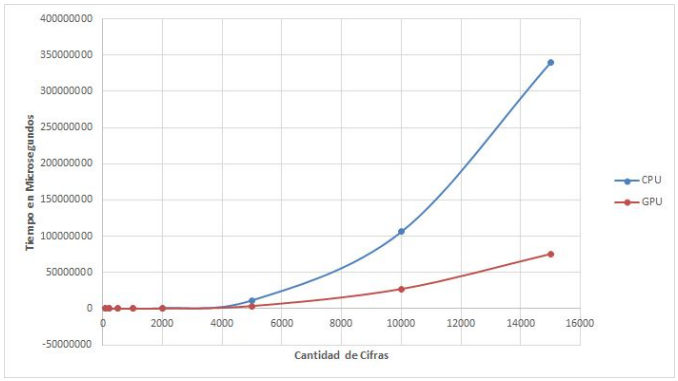

Fig. 10 Resultados de la Compresión LZ78 sobre datos completamente aleatorios.

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.

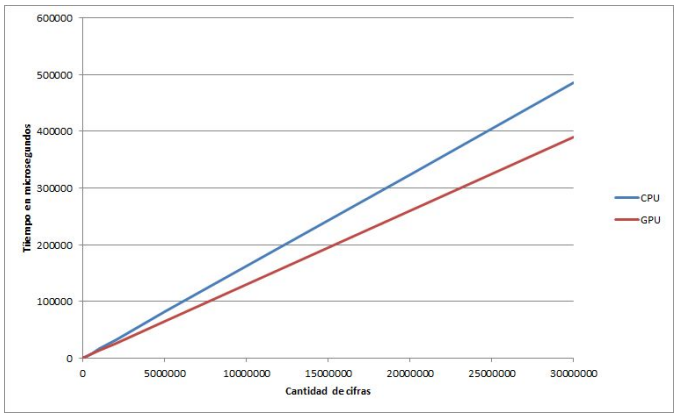

Fig. 11 Resultados de la Compresión Run Length sobre datos convenientemente comprimibles.

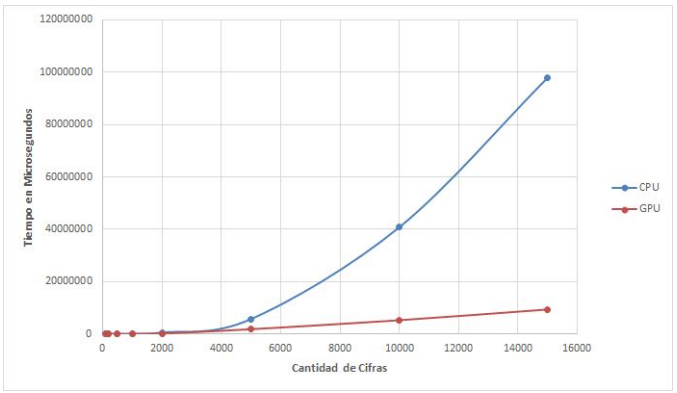

Fig. 12 Resultados de la Compresión LZ78 sobre datos convenientemente comprimibles.

Como se puede observar en las figuras 9 y 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.

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” Universidad de Burgos.
[2] A. Ozsoy, M.Swany, “CULZSS: LZSS lossless data compression on CUDA” Universidad de Delaware
[3] R. Patel, Y.Zhang, “Parallel Lossless Data Compression on the GPU” Universidad de California
[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, Noviembre 2004.
[5] J. Mielikainen “GPUs for data parallel spectral image compression” Proceedings of SPIE
[6] P. Patel, J. Wong, M. Tatikonda, J. Marczewski “JPEG Compression Algorithm Using CUDA” Universidad de Toronto
[7] R.L. Cloud, M.L. Curry “Accelerating Lossless Data Compression with GPUs”
[8] Q. Luo “Database Compression on Graphics Processors ” Universidad de Hong Kong
[9] J. Franco, G. Bernabé, J. Fernández and M. Acacio, “A Parallel Implementation of the 2D Wavelet Transform Using CUDA” Universidad de Murcia
[10] J. Ziv, A. Lempel, “Compression of Individual Sequences via Variable-Rate Coding” IEEE Transactions on Information Theory, Vol, IT-24, N° 5, Septiembre 1978
[11] M. Morales, “Notas sobre Compresión de Datos”, INAOE, 2003.
[12] 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.
[13] NVidia, “CUDA”, [Online]. Available: http://www.nvidia.es/object/cuda-parallel-computing-es.html
[14] A. Balevic, “Fine-grain Parallelization of Entropy Coding on GPGPUs”.
[15] 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
[16] C. Zeller, NVIDIA Corporation, “Supercomputing 2011 tutorial”, [Online]. Available: http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf
[17] “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.