

# 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, without none loss in compression ratio.

**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, sin ninguna pérdida en la relación de compresión.

**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 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.

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].

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 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 (Run Length Encoding), 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       frequency ← 1
4       while i + 1 < n and in[i] = in[i + 1]
5           frequency ← frequency + 1
6           i ← i + 1
7       end while
8       symbolsOut[index] ← in[i]
9       countsOut[index] ← frequency
10      index ← index + 1
11  end for

```

Fig. 1 Pseudocódigo de la Estructura Básica de RLE [12]

### B. Algoritmo Lempel-Ziv-78 (LZ78)

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 ( $S = \text{Null}$  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  $S \cdot X$  (concatenación de  $S$  y  $X$ ) en el diccionario, si la cadena  $S \cdot X$  se encuentra en el diccionario,  $S$  es ahora  $S \cdot X$  y se lee un nuevo símbolo  $X$ . Nuevamente, se busca  $S \cdot X$  en el diccionario y si la cadena se encuentra, se vuelve a leer otro símbolo de entrada y el proceso se repite buscando nuevamente  $S \cdot X$  en el diccionario. Si la cadena  $S \cdot X$  no se encuentra en el diccionario, se guarda la cadena  $S \cdot X$  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].

```

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;
9         else
10            CodeWordForPrefix = DictionaryIndex;
11            Output: (CodeWordForPrefix, Char);
12            InsertinDictionary (( DictionaryIndex, Prefix + Char ));
13            DictionaryIndex ++;
14            Prefix = NULL;
15 if(!isEmpty(Prefix))
16     CodeWordForPrefix = DictionaryIndex for Prefix;
17     Output: (CodeWordForPrefix, );

```

Fig. 2 Algoritmo de Codificación en LZ78. [10]

### C. Arquitectura Unificada de Dispositivos de Cómputo (CUDA)

CUDA es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la gran 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 [16].

El flujo de procesamiento en CUDA se aprecia en la fig. 3, primero se copian los datos desde la memoria principal a la memoria de la GPU (1). La CPU designa el proceso a la GPU (2). La GPU se encarga de la ejecución en paralelo con cada núcleo (3). Finalmente se copian los resultados de la memoria de la GPU a la memoria principal (4).

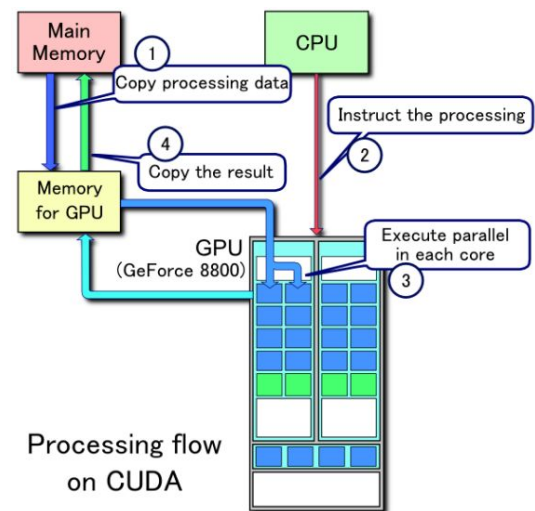


Fig. 3 Flujo de Procesamiento en CUDA. [17]

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) la 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 [18].

#### 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 RLE 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 [15], 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 aprecia 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.

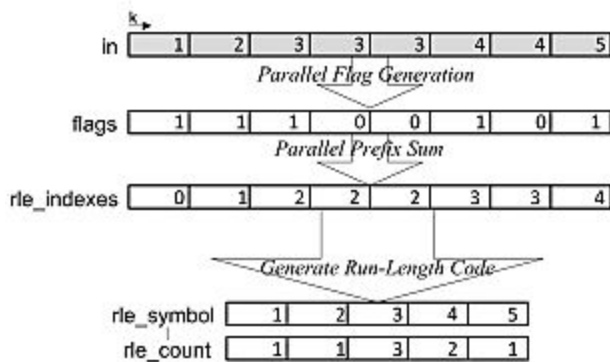


Fig. 4 Perspectiva de RLE paralelo [15]

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 segun 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 Belavic, 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 *simbolos* 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 (LZ78) 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 Figura 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 Figura 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 = DiccionarioIndex;
13              Output: (CodeWordforPrefix, );
14              InsertinDictionary (( DiccionarioIndex 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 tabla 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	179 0	175 9	1764	1769	1760	1768
200000	380 9	355 8	3800	3532	3586	3657
500000	923 2	896 0	9119	8910	9186	9082
1000000	179 73	176 77	1826 1	1826 5	1812 3	18060
2000000	350 58	358 21	3609 1	3627 6	3609 0	35867
5000000	896 58	900 24	8973 2	9037 1	9107 5	90172
10000000	178 590	177 094	1775 95	1785 44	1799 25	178349
20000000	358 274	359 912	3629 48	3581 72	3575 47	359370
30000000	528 378	536 170	5359 76	5328 10	5381 53	534297

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

Canti dad 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	2987 1	23236	2325 8	5421 1	5069 5	36254.2
1000	1518 41	14204 9	1521 69	1701 02	1423 91	151710.4
2000	9712 61	96286 7	9152 57	9142 37	9123 35	935191.4
5000	1077 3146	11766 811	1179 0423	1173 8288	1173 9350	1156160 3.6

<b>10000</b>	1080 3435 8	10417 1963	1059 0800 0	1056 2176 9	1063 0115 2	1060074 48.4
<b>15000</b>	3479 6023 9	35008 8465	3291 1160 6	3312 0246 0	3388 5853 9	3394442 61.8

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
<b>10000</b>	696	696	696	696	696	696
<b>50000</b>	136 0	1360	1361	1361	1361	1361
<b>100000</b>	213 9	2125	2137	2138	2138	2136
<b>200000</b>	363 0	3553	3645	3594	3640	3613
<b>500000</b>	779 8	7636	7823	7742	7814	7763
<b>1000000</b>	144 29	1443 4	1443 1	1443 0	1443 2	14431
<b>2000000</b>	283 71	2836 9	2836 1	2836 5	2837 0	28367
<b>5000000</b>	704 84	7049 9	7047 9	7049 2	7047 7	70486
<b>10000000</b>	141 108	1410 83	1410 88	1410 78	1411 36	141099
<b>20000000</b>	282 600	2825 90	2826 01	2825 58	2825 70	282584
<b>30000000</b>	424 932	4249 53	4249 42	4248 49	4249 62	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
<b>100</b>	166	554	299	556	234	361.8
<b>200</b>	2526	2276	1133	2491	1275	1940.2
<b>500</b>	1425 8	7718	2417 4	1040 9	1858 4	15028.6
<b>1000</b>	5797 5	48122	4983 0	5252 6	6995 8	55682.2
<b>2000</b>	3273 85	31197 1	2940 99	2992 60	3276 03	312063.6
<b>5000</b>	3756 615	37662 83	3774 235	3733 878	3775 680	3761338.2
<b>10000</b>	2678 0511	26950 798	2750 0394	2778 3067	2716 0028	2723495.9.6
<b>15000</b>	7578 8511	75676 139	7538 8334	7538 8686	7574 6747	7559768.3.4

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 CONVENIENTEMENTE COMPRIMIBLES

Cantidad de Cifras	T1 (ms)	T2 (ms)	T3 (ms)	T4 (ms)	T5 (ms)	Promedio
<b>10000</b>	160	159	159	160	159	159
<b>50000</b>	832	798	798	800	799	806
<b>100000</b>	1604	1597	159 8	1601	1603	1601
<b>200000</b>	3188	2946	319 5	3197	3191	3143
<b>500000</b>	7866	8013	803 9	8022	8078	8004
<b>1000000</b>	1600 9	1626 8	161 94	1617 0	1616 7	16162
<b>2000000</b>	3242 3	3216 4	323 25	3286 2	3273 1	32501
<b>5000000</b>	8102 9	8103 7	811 08	8090 3	8101 4	81018
<b>10000000</b>	1617 03	1623 92	161 859	1614 36	1610 82	161695
<b>20000000</b>	3240 40	3228 22	324 175	3243 55	3183 31	322745
<b>30000000</b>	4850 00	4869 53	487 519	4849 41	4845 97	485802

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
<b>100</b>	1389	1634	1464	1502	1470	1491.8
<b>200</b>	5450	6681	7513	7089	6321	6610.8
<b>500</b>	5179 4	27522	2128 5	5235 6	2010 4	34612.2
<b>1000</b>	1118 33	11223 9	1122 31	1276 57	1140 74	115606.8
<b>2000</b>	5802 01	60516 7	5891 84	5799 12	6543 00	601752.8
<b>5000</b>	5688 695	57201 01	5726 362	5705 794	5687 335	5705657.4
<b>10000</b>	4009 4300	41580 569	4076 5273	4078 9974	4081 9879	4080999.9
<b>15000</b>	9726 1077	98814 774	9809 8827	9786 7837	9779 6357	9796777.4.4

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.

TABLA VII  
RESULTADOS DE LA COMPRESIÓN EN RUN LENGTH EN  
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	120 5	1206	120 5	120 5	1205
100000	1845	184 6	1845	184 5	184 5	1845
200000	3130	313 1	3131	313 1	313 1	3131
500000	7004	700 4	7005	700 6	700 4	7005
1000000	1344 7	134 48	1344 6	134 51	134 48	13448
2000000	2636 4	263 62	2636 8	263 67	263 63	26365
5000000	6523 7	652 33	6524 1	652 32	652 38	65236
10000000	1301 70	130 170	1301 63	130 194	130 187	130177
20000000	2601 98	260 231	2602 20	260 179	260 193	260204
30000000	3902 48	390 235	3902 31	390 273	390 225	390242

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

Canti dad 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	1695 9	18719	1803 8	1615 3	1796 6	17567
1000	4067 3	60388	3699 2	3477 5	3879 9	42325.4
2000	1926 02	24490 9	1912 49	2092 56	1859 41	204791.4
5000	1859 110	18163 36	1822 625	1833 595	1868 377	1840008. 6
10000	4975 938	53578 49	5357 463	5437 844	5313 491	5288517
15000	9337 591	93360 96	9371 718	9427 403	9505 666	9395694. 8

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.

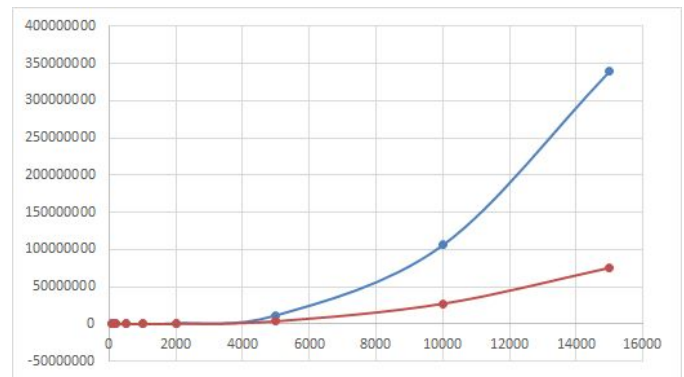


Fig. 10 Resultados de la Compresión LZ78 sobre datos completamente aleatorios.

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.

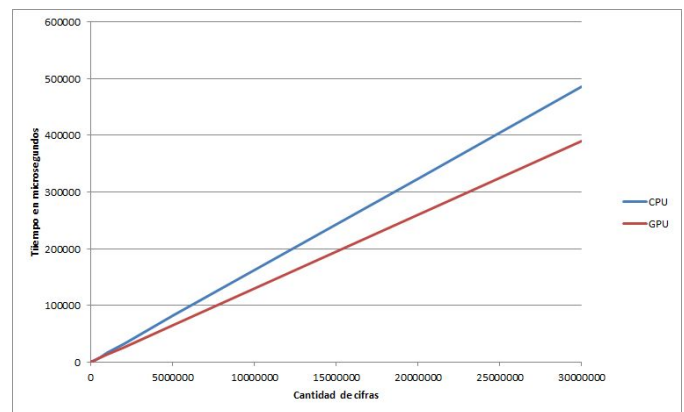


Fig. 11 Resultados de la Compresión Run Length sobre datos convenientemente comprimibles.

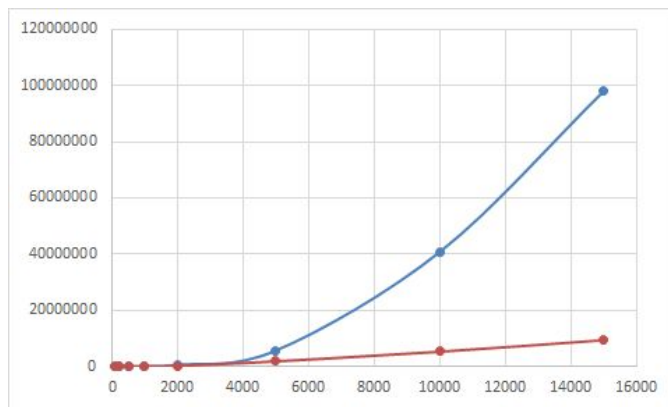


Fig. 12 Resultados de la Compresión LZ78 sobre datos convenientemente comprimibles.

Como se puede observar en las figuras 8 y 10 el tiempo de ejecución de RLE mantiene una relación lineal con la cantidad de datos de entrada, por otro lado, en la fig. 9 y la fig. 11, 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 Run Length y Lempel-Ziv-78 (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 Run Length Encoding (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 Run Length 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 Lempel-Ziv-78 (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 del GPU.

## REFERENCIAS

- [1] C. Represa, J. Cámara, P. Sánchez, "Introducción a la Programación en CUDA" Universidad de Burgos.
- [2] A. Ozsoy, M. Swamy, "CULZSS: LZSS lossless data compression on CUDA" Universidad de Delaware
- [3] R. Pathel, Y. Zhang, "Parallel Lossless Data Compression on the GPU" Universidad de California
- [4] J. Gilchrist, "Parallel Data Compression with BZIP2" Universidad de Carleton
- [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. Bernabe "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. Duncalf, 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] Wilt, Nicholas. "The cuda handbook: A comprehensive guide to gpu programming". Pearson Education, 2013.
- [15] A. Balevic, "Fine-grain Parallelization of Entropy Coding on GPGPUs".
- [16] 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>
- [17] "Processing flow on CUDA", [Online]. Available: [https://es.wikipedia.org/wiki/CUDA#/media/File:CUDA\\_processing\\_flow\\_\(En\).PNG](https://es.wikipedia.org/wiki/CUDA#/media/File:CUDA_processing_flow_(En).PNG)
- [18] "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.