SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO
Instituto Tecnológico de La Paz

INSTITUTO TECNOLÓGICO DE LA PAZ
DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN
MAESTRÍA EN SISTEMAS COMPUTACIONALES

COMPRESIÓN SIN PÉRDIDA UTILIZANDO GPGPU

QUE PARA OBTENER EL GRADO DE
MAESTRO EN SISTEMAS COMPUTACIONALES

PRESENTA:
ING. ÁNGEL CÉSAR VEJAR ROMERO

DIRECTOR DE TESIS:
DR. MARCO ANTONIO CASTRO LIERA

LA PAZ, BAJA CALIFORNIA SUR, MÉXICO, DICIEMBRE 2017.

SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO
Instituto Tecnológico de La Paz

"Año del Centenario de la Promulgación de la Constitución Política de los Estados Unidos Mexicanos"

La Paz, Baja California Sur, 04/diciembre/2017

OFICIO No. DEPI/474/2017

ASUNTO: Autorización de impresión

C. ING. ÁNGEL CÉSAR VEJAR ROMERO,
ESTUDIANTE DE LA MAESTRÍA EN
SISTEMAS COMPUTACIONALES,
P R E S E N T E .

Con base en el dictamen de aprobación emitido por el Comité Tutorial de la Tesis denominada: "**COMPRESIÓN SIN PÉRDIDA UTILIZANDO GPGPU**", mediante la opción de tesis (Proyectos de Investigación), entregado por usted para su análisis, le informamos que se **AUTORIZA** la impresión

A T E N T A M E N T E
"CIENCIA ES VERDAD, TÉCNICA ES LIBERTAD"

MC. MAURICIO SALVADOR RODRÍGUEZ OJEDA,
JEFE DE LA DIV. DE EST. DE POSGRADO E INV.

INSTITUTO TECNOLÓGICO DE LA PAZ
DIVISIÓN DE ESTUDIOS DE POSGRADO
E INVESTIGACIÓN

c.c.p. Archivo.
MSRO/icl*

SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

Tecnológico Nacional de México
Instituto Tecnológico de La Paz

DICTAMEN DEL COMITÉ TUTORIAL

SUBDIRECCIÓN ACADÉMICA
DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN.

La Paz, B.C.S., **04/diciembre/ 2017**

C. M.C. MAURICIO SALVADOR RODRÍGUEZ OJEDA,
JEFE DE LA DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN,
P R E S E N T E.

Por medio del presente, enviamos a usted dictamen del Comité Tutorial de tesis para la obtención del grado de Maestro, con los siguientes datos generales:

No. de Control M15310019	Nombre ÁNGEL CÉSAR VEJAR ROMERO
Maestría en:	SISTEMAS COMPUTACIONALES
Título de la tesis: COMPRESIÓN SIN PÉRDIDA UTILIZANDO GPGPU	
DICTAMEN: Se autoriza el trabajo de investigación, en virtud de que realizó las correcciones correspondientes conforme a las observaciones planteadas por este Comité Tutorial.	

A t e n t a m e n t e .
El Comité Tutorial

M.C. JESÚS ANTONIO CASTRO

MSC. ILIANA CASTRO LIERA

DR. MARCO ANTONIO CASTRO LIERA

c.c.p. Coordinador de la Maestría.
c.c.p. Departamento de Servicios Escolares.
c.c.p. Estudiante.

Dedicatoria

A mi madre, abuela y mis hermanos, por todo su amor y apoyo incondicional.

Agradecimientos

A todas las instituciones que me brindaron su apoyo para hacer esto posible, a mi director de tesis, mis profesores, mi familia, a los compañeros y amigos que me dedicaron parte de su invaluable tiempo y esfuerzo, ustedes saben bien quienes son, a todos ellos muchas gracias.

Resumen

El presente trabajo describe la implementación del algoritmo de compresión sin pérdida de Huffman que fue ejecutado sobre una unidad de procesamiento gráfico de propósito general (GPGPU) e implementado sobre la arquitectura de procesamiento paralelo CUDA.
Los resultados que obtenidos al realizar la comparativa de velocidad entre la versión secuencial y la versión paralelizada fueron alentadores. Se obtubieron mejoras significativas en los tiempos de ejecución para la compresión de conjuntos de datos de distintos tamaños sobre distintos equipos de cómputo.

Abstract

This work describes the implementation of Huffman's lossless compression algorithm wich was executed on a general-purpose computing on graphic pocessing unit (GPGPU) and implemented on parallel processing architecture called CUDA.

The results obtained in the speed comparative between sequential version and parallel version were encouraging. Significant improvements were obtained in execution times achieved on the compression of data sets of different sizes on differents computers.

Índice general

1. Introducción **1**

1.1. Descripción del problema 3

1.2. Objetivos 3

1.2.1. Objetivo general 3

1.2.2. Objetivos específicos 3

1.3. Justificación 4

1.4. Limitaciones y alcance 4

2. Compresión de datos **5**

2.1. Tipos 5

2.2. Algoritmos de compresión sin pérdida 6

2.2.1. *Run Lenght Encoding* (RLE) 6

2.2.2. De árbol de contexto 6

3. Procesamiento en paralelo **9**

3.1. Cómputo paralelo 9

3.2. Arquitecturas 10

3.2.1. Arquitectura de memoria compartida 10

3.2.2. Memoria de acceso uniforme (UMA *Uniform Memory Access*) 11

3.2.3. Memoria de acceso no uniforme (NUMA *Non-Uniform Memory Access*) . 11

3.2.4. Arquitectura de memoria distribuida 12

3.2.5. Arquitectura de memoria compartida distribuida híbrida 13

3.2.6. CUDA(*Compute Unified Device Architecture*) 13

4. Compresión de propósito general en CUDA 22

4.1. Fase de conteo y ordenamiento 23

4.1.1. Generación de la tabla de frecuencias en paralelo 23

4.2. Creación del árbol de contexto 24

4.3. Creación de la tabla de códigos de longitud variable 26

4.4. Compresión en paralelo 27

4.4.1. Tasa de compresión 27

4.4.2. Generación de los archivos de prueba 28

4.4.3. Paqueterias para compilar 29

4.4.4. Compilación y ejecución 29

5. Resultados 31

5.1. Validación de resultados 31

6. Conclusiones y trabajo futuro **35**

A. Archivos de cabecera **36**

A.0.1. params.h 36

A.0.2. qsort.h 37

B. Compresión de datos secuencial **39**

B.0.1. comprimir.c 39

C. Descompresión de datos secuencial **48**

C.0.1. descomprimir.c 48

D. Compresión de datos paralela **55**

D.0.1. comprimir.cu 55

Bibliografía **66**

Índice de figuras

2.1. Ejemplo de compresión utilizando RLE [6]. 6

2.2. Ejemplo de árbol de descripción mínima [6]. 8

3.1. Procesamiento secuencial. 9

3.2. Procesamiento en paralelo. 10

3.3. Memoria de acceso uniforme(UMA). 11

3.4. Memoria de acceso no uniforme (UMA). 12

3.5. Memoria distribuida. 12

3.6. Memoria distribuida. 13

3.7. Comparativa de unidades de procesamiento de CPU contra las de GPU [15]. . . 16

3.8. Representación del procesamiento del lado del *host* y *device* [15]. 17

4.1. Diagrama de compresión por bloques. 22

4.2. Distribución de memoria por bloques e hilos. 23

4.3. Sumatoria de conteos parciales por hilo. 24

4.4. Creación de árbol de contexto de longitud mínima. 25

4.5. Criterio de árbol MDL para obtener el código de longitud variable. 26

Índice de tablas

2.1. Ejemplo de datos para código Huffman. 7

3.1. Configuración del equipo de pruebas 1. 20

3.2. Configuración del equipo de pruebas 2. 21

4.1. Ejemplo de tabla de frecuencia del *Host*. 25

4.2. Ejemplo de tabla de código de longitud variable. 26

4.3. Estructura del archivo. 27

5.1. Tiempos de ejecución en secuencial y paralelo para conteo de caracteres. 31

5.2. Tiempos de ejecución en secuencial y paralelo para conteo de caracteres. 32

5.3. Tiempos de ejecución para compresión con GPU Nvidia GeForce GT730. 32

5.4. Tiempos de ejecución para compresión con GPU Nvidia GeForce GT730. 33

5.5. Tiempos de ejecución para compresión con GPU GTX Titan X. 33

5.6. Tiempos de ejecución para compresión con GPU GTX Titan X. 34

Capítulo 1

Introducción

En épocas anteriores a las tecnologías computacionales cualquier comercio, institución o individuo que necesitaba hacer un conglomerado de datos para su posterior manipulación se encontraba con el problema de cómo mantener dichos datos organizados de manera que al momento de necesitarla supiera donde buscar y no perder demasiado tiempo ni recursos en dicha labor, para eso utilizaban libretas, o listas en las cuales se llenaban formatos para poder llevar un control adecuado; tiempo después, con el desarrollo computacional se crearon sistemas capaces de efectuar dicha tarea.

Con el tiempo se fueron desarrollando las tecnologías de la computación y fue cuando se crearon las bases de datos, con las cuales se podía llevar todo el control que se hacía de una forma más sencilla, se ocupaba menos espacio y que la información no podía ser dañada o verse afectada por diversos factores físicos, teniendo en cuenta dicha ventaja, distintos formatos fueron apareciendo, y con ellos maneras para manejarlos optimizadamente. Audio, video e imagen entre otros tantos; pese a esto, aun existían temas que debían ser tratados para optimizar los recursos computacionales con los que se contaba, cómo el espacio que ocupaba la información en disco o al ser transmitida.

La compresión de datos, en las ciencias de la computación, se define como la reducción del tamaño que ocupa determinada información en disco. Esto se logra gracias a un algoritmo que se encarga de recodificar la información para que este propósito se logre. A su vez, para poder

acceder a dicha información posteriormente, se lleva a cabo un proceso de descompresión, lo que significaría decodificar o devolver a su estado original dicho paquete de datos, para poder lograr esto se usa un algoritmo distinto e inverso al de compresión.

Actualmente existen distintos programas informáticos que desempeñan la tarea del manejo de compresión de datos en diferentes plataformas para los más diversos tipos de usuarios. Estas aplicaciones son muy eficientes y se usan en tareas cotidianas con un rendimiento lo suficientemente aceptable, por otro lado, algunos de ellos sólo trabajan con base en la velocidad del microprocesador y algunos de forma secuencial, dado que sólo usan un núcleo del CPU para desenvolverse, por lo que los tiempos de ejecución son mejores que si se hiciera de manera conjunta por todos sus núcleos (paralelamente).

El procesamiento de datos mediante el uso de tarjetas gráficas es algo que ha evolucionado increíblemente, comenzando desde las épocas donde los investigadores tenían que arreglárselas para interpretar los resultados de sus procesos que eran mostrados en pantalla, hasta el uso de los lenguajes de programación especialmente adaptados para poder utilizarlos en conjunto con el hardware de video [16]. La necesidad de programar en este tipo de ambiente se debe a los beneficios que estos aportan, comúnmente el microprocesador de una computadora convencional tiene un número de unidades de procesamiento (también llamadas núcleos) menor al que pueden llegar a manejar las unidades gráficas, aunado a esto, la velocidad con la que puede ejecutar una tarea es teóricamente mayor que la del CPU, siempre y cuando dicha tarea sea paralelizable, lo cual es un factor muy importante a tener en cuenta.

Existen diversas disciplinas en donde se puede aplicar este tipo de tecnología, unos cuantos ejemplos de esto son la computación de alto rendimiento (también llamado supercómputo) [18], las simulaciones biomoleculares ya sea de proteínas, secuencias de ADN o simplemente gráficos moleculares, también se usa en el campo de las matemáticas e investigación potenciando el cálculo numérico, en la geoexploración para hacer cálculo del movimiento de las mareas, simulación de sismos, entre otros y en la criptografía, comúnmente se le conoce por relacionarse con la seguridad de un canal o del almacenamiento de información por medio de la codificación de contraseñas o volúmenes de datos como lo serian bases de datos, pero en este proyecto se enfocara en otro aspecto que forma parte de los elementos criptográficos que son el manejo de

grandes volúmenes de datos para la compresión de archivos sin pérdida.

1.1. Descripción del problema

La mayoría de las aplicaciones de compresión de datos de propósito general pueden llegar a usar como máximo cada uno de los núcleos del microprocesador de la computadora donde se alojen, lo cual es bastante conveniente para un usuario que no tiene demandas muy grandes, sin embargo, cuando llega el momento de realizar compresión o descompresión de volúmenes de datos grandes la capacidad que tiene el microprocesador de manipular esa información se ve opacada debido a que el manejar tantos datos le exige demasiado y por tanto se tarda más en llevar a cabo dicha tarea. Una alternativa prudente para tratar este problema es el cómputo de propósito general sobre unidades de procesamiento de gráficos (GPGPU por sus siglas en inglés), de esta manera el microprocesador sólo serviría como intermediario entre el almacenamiento y la tarjeta gráfica para que la labor más demandante la pueda realizar el GPU. Aún teniendo una herramienta poderosa como lo es el GPGPU es necesario evaluar conforme a las características que presentan, los distintos algoritmos de compresión sin pérdida de datos e implementarlo de forma paralelizada.

1.2. Objetivos

1.2.1. Objetivo general

Desarrollar una aplicación de compresión de datos sin pérdida para escritorio que utilice la tecnología GPGPU de la plataforma CUDA.

1.2.2. Objetivos específicos

- Mejorar el tiempo de compresión de archivos masivos de datos, mediante el uso de las GPU

- Hacer comparativa entre la ejecución secuencial y paralela del algoritmo en cuanto al tiempo

1.3. Justificación

La importancia de la compresión de datos en la vida diaria es muy grande, cada vez se generan datos en volúmenes mayores y mantenerlos alojados demanda más recursos, por eso, una excelente opción para el usuario promedio es poder contar con una alternativa diferente que le permita el manejo de grandes volúmenes de datos, para ello se utiliza la arquitectura de computo paralelo CUDA que aprovecha la potencia de cálculo del GPU.
El uso de esta tecnología para la compresión de datos sin pérdida supone la disminución en gran medida del tiempo que tarda en realizar el microprocesador de una computadora convencional este mismo procedimiento, lo cual, brinda una ventaja significativa a la hora de manejar volúmenes grandes de información, motivo por el cual la aplicación es beneficiosa para el usuario promedio, ya que se le brinda una alternativa potente y hasta el momento escasa en el mercado.

1.4. Limitaciones y alcance

- El algoritmo que se utilizará para realizar compresión se podrá ejecutar únicamente en la arquitectura CUDA la cual pertenece a la empresa de tarjetas gráficas NVIDIA
- El tipo de compresión que realizará la aplicación será únicamente sin pérdida de información
- No se pretende mejorar la tasa de compresión del algoritmo

Capítulo 2

Compresión de datos

2.1. Tipos

Cuando se comprimen datos siempre hay una clasificación en la que debe estar el algoritmo de compresión en cuestión, dependiendo de cómo maneje la información se puede categorizar como compresión sin pérdida y compresión con pérdida [5] [17].

La compresión sin pérdida (también llamada de propósito general) es aquella en que el manejo de la información es muy sensible pues generalmente hace manipulación de archivos que por su misma naturaleza no pueden ser modificados al momento de ser comprimidos, haciendo que el archivo o conjunto de datos al descomprimir tenga que ser idéntico al que fue comprimido, pues si alguna parte del archivo es cambiada impediría que el archivo funcionara correctamente al ser descomprimido, hasta incluso poder a llegarlo a estropear completamente. Algunos ejemplos de compresión sin pérdida son los documentos de texto como libros, códigos informáticos, bases de datos, entre otros.

La compresión con pérdida se basa en manipular los datos de manera que algunos datos o *bits* del archivo puedan ser eliminados o cambiados sin afectar en esencia el funcionamiento del archivo original, esto con el propósito de que gracias a esa discriminación se pueda comprimir en mayor medida. Comúnmente se utilizan estas técnicas para conjuntos de datos multimedia,

algunos ejemplos de compresión con pérdida son los archivos de audio *.mp3, los archivos de imagen *.jpg y video en *.mp4.

2.2. Algoritmos de compresión sin pérdida

2.2.1. *Run Lenght Encoding* (RLE)

El principio básico de este algoritmo es observar la repetición de caracteres que se encuentran agrupados consecutivamente abreviando la ocurrencia de ellos e indicandola para hacer el archivo mas ligero. Un ejemplo de ello sería:

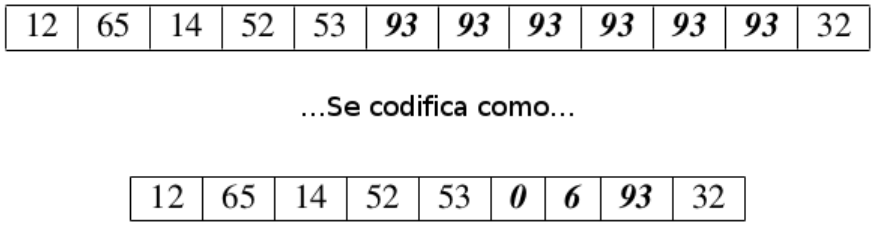

Figura 2.1: Ejemplo de compresión utilizando RLE [6].

En la fig. 2.1 se puede apreciar que el carácter que se repite es el 93, por tanto esa subsecuencia de caracteres debe ser comprimida. En la linea que muestra como se compone el archivo comprimido ya no existen todos esos símbolos repetidos, pero existen nuevos que tienen la función de indicar en donde se encuentra una secuencia comprimida, el número cero indica en donde comienza una secuencia que se encuentra comprimida, el seis indica el número de veces que se repite el símbolo y el número 93 indica el símbolo que se comprimió.

2.2.2. De árbol de contexto

2.2.2.1. Huffman

David Albert Huffman (9 de Agosto de 1925 – 7 de Octubre de 1979) fue el creador del popular algoritmo en el año 1951, como el resultado final de un trabajo de doctorado que realizaba en

el Instituto de Tecnología de Massachusetts (MIT), el cual, fue descrito en su trabajo *A Method For the Construction Minimum-Redundancy Codes* [8].

Este algoritmo se encarga de realizar la compresión de datos utilizando un alfabeto y la frecuencia de aparición de cada símbolo, la peculiaridad de este método es la implementación de un árbol binario también llamado árbol de contexto de descripción mínima o *Minimun Description Lenght(MDL) context tree*, para con base en el poder crear lo que se conoce como código de longitud variable o *Variable Lenght Code (VLC)* que consiste en crear cadenas más cortas de *bits* para los caracteres que tienen una frecuencia mayor (se encuentran más cerca del nodo raíz) y cadenas más largas de *bits* para aquellos caracteres que tienen una menor frecuencia de aparición (se encuentran más lejos del nodo raíz), esta técnica es conocida por la buena tasa de compresión que genera respecto a otros algoritmos [11], desde el año en que se publicó ha sido puesta a prueba y estudiada bajo diferentes condiciones por diversos investigadores alrededor del mundo, siendo combinada con numerosos métodos para crear mejoras [1] [7].

Carácter	Frecuencia	Código
22	4	00
43	2	01
17	1	100
32	1	101
48	1	110
49	1	111

Tabla 2.1: Ejemplo de datos para código Huffman.

En la tabla 2.1 se muestra la información que es necesaria para poder crear el árbol de descripción que es la frecuencia de cada símbolo, una vez creado como se muestra en la fig. 2.2, se puede llegar a la creación del VLC recorriendo las ramas del árbol hacia cada hoja que es representada por un símbolo del alfabeto.

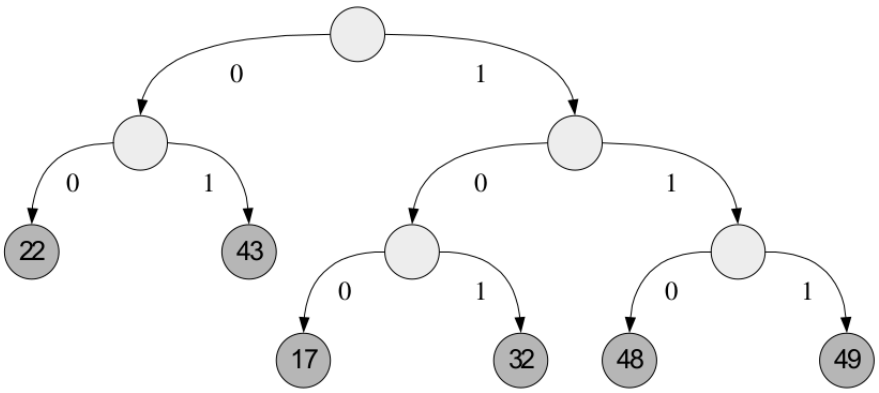

Figura 2.2: Ejemplo de árbol de descripción mínima [6].

Capítulo 3

Procesamiento en paralelo

3.1. Cómputo paralelo

Es un estilo de computación en el que se da solución a un problema dividiéndolo en varias partes para ser solucionadas simultáneamente, uniendo después de cada cálculo los resultados de todas sus partes, esto se realiza con el principal propósito de aminorar el tiempo elevado de cómputo que demanda una tarea al ejecutarse secuencialmente.

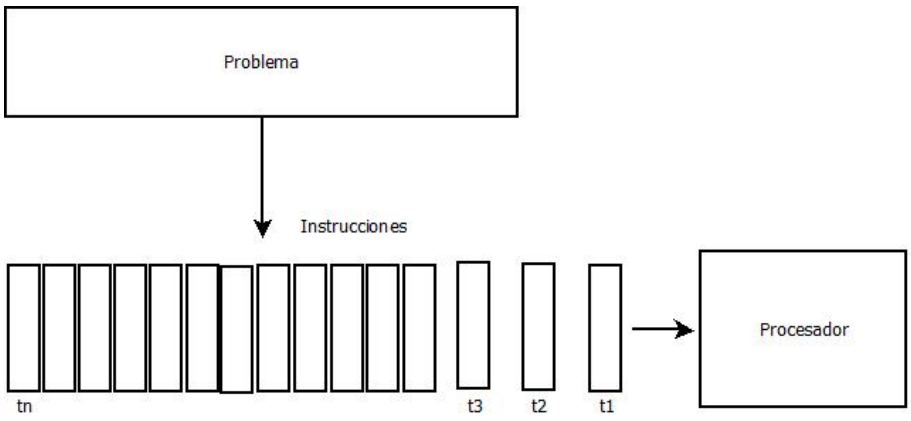

Figura 3.1: Procesamiento secuencial.

En la programación secuencial, las partes del problema se van resolviendo una a una por el procesador, de manera que cuando un subproblema está siendo resuelto, las demás partes del problema esperan su turno, cada parte ocupa un instante de tiempo.

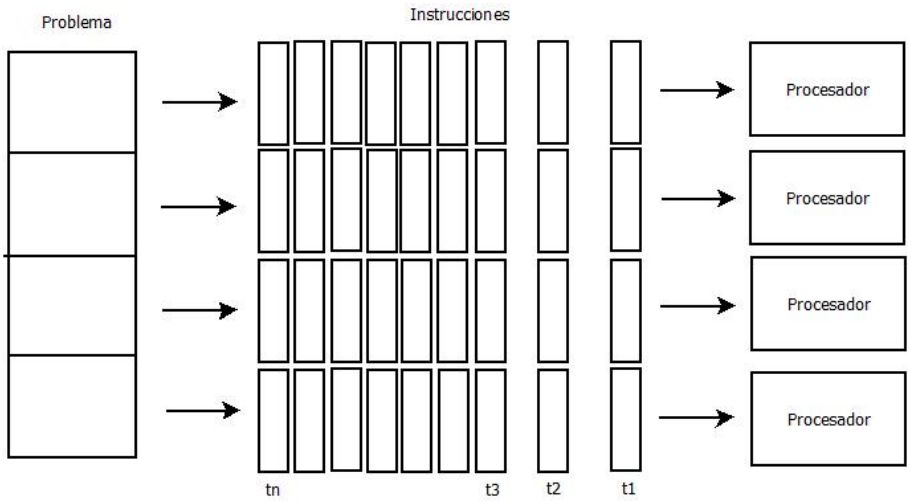

Figura 3.2: Procesamiento en paralelo.

En la programación paralela se pueden solucionar varias partes de un problema en el mismo instante de tiempo. Esto nos sirve para resolver una gran variedad de problemas, unos ejemplos serían las simulaciones de fenómenos naturales, donde existen múltiples factores a considerar, otro ejemplo seria la línea de ensamble, donde múltiples partes o piezas son puestas en su lugar para armar un artefacto, existen un gran número de disciplinas como la física o la ingeniería donde este tipo de cómputo puede ser implementado.

3.2. Arquitecturas

3.2.1. Arquitectura de memoria compartida

Una de las características más generales de esta arquitectura es que cada uno de los procesadores que posee la capacidad de accesar a toda la memoria como si se tratara de un espacio de memoria global, los procesadores que posee pueden funcionar de manera independiente, pero siempre comparten la misma capacidad de memoria.

Otra característica importante es que si un procesador hace algún cambio en memoria, este cambio puede ser visible para los demás procesadores. Dependiendo del tiempo que tienen para accesar a la memoria esta arquitectura se divide en dos clases UMA y NUMA.

3.2.2. Memoria de acceso uniforme (UMA *Uniform Memory Access*)

Comúnmente representada por procesadores simétricos, esta categoría cuenta con la característica de poseer los mismos tipos de multiprocesador los cuales pueden accesar a la misma memoria con los mismos tiempos de acceso. Algunas veces se le llama CC-UMA (UMA de cache coherente) pues, cuando un procesador realiza algún cambio en memoria, los demás multiprocesadores son alertados del cambio realizado.

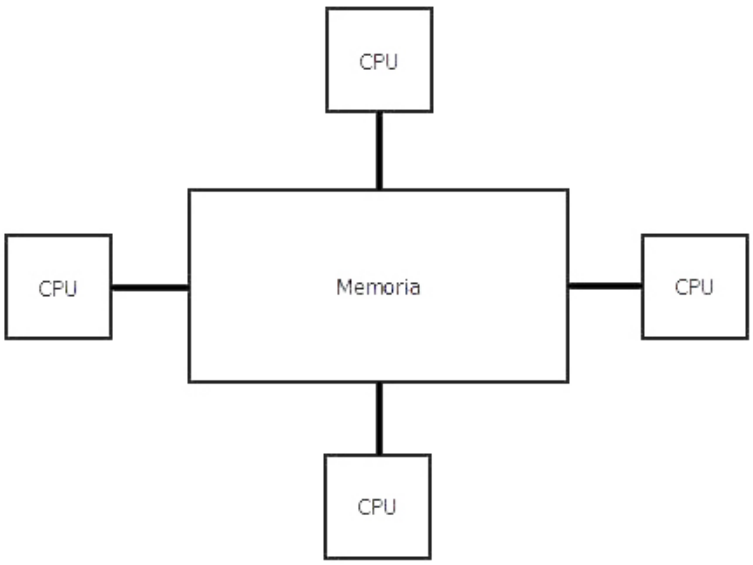

Figura 3.3: Memoria de acceso uniforme(UMA).

3.2.3. Memoria de acceso no uniforme (NUMA *Non-Uniform Memory Access*)

Comúnmente hecha con la unión de dos máquinas SMP(*Symmetric Multiprocessor*) en donde un SMP puede accesar directamente a la memoria de otro SMP, a diferencia de los UMA no todos los multiprocesadores tienen el mismo tiempo de acceso a memoria, otra diferencia es que el acceso a memoria a través del enlace es más lento y si la coherencia de la memoria cache se mantiene se le puede llamar también CC-NUMA (NUMA de cache coherente).

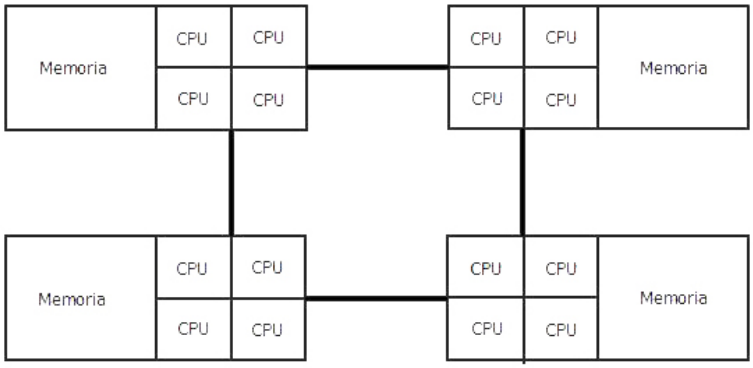

Figura 3.4: Memoria de acceso no uniforme (UMA).

3.2.4. Arquitectura de memoria distribuida

Existen diversos tipos de sistemas de memoria distribuida, sin embargo siguen compartiendo las características básicas, los procesadores poseen su propia memoria local, cuando un procesador hace un cambio en su memoria, los demás procesadores no se dan por enterados del acontecimiento pro lo que no hay memoria global entre los procesadores en cuestión.

Por ende, los procesadores tienen su propia memoria individual y cada procesador funciona independientemente de los demás multiprocesadores haciendo que no exista coherencia de memoria *cache*. Cuando algún procesador necesita acceso a la información de otro procesador se debe especificar por el programador para que se puedan sincronizar y pasar los datos.

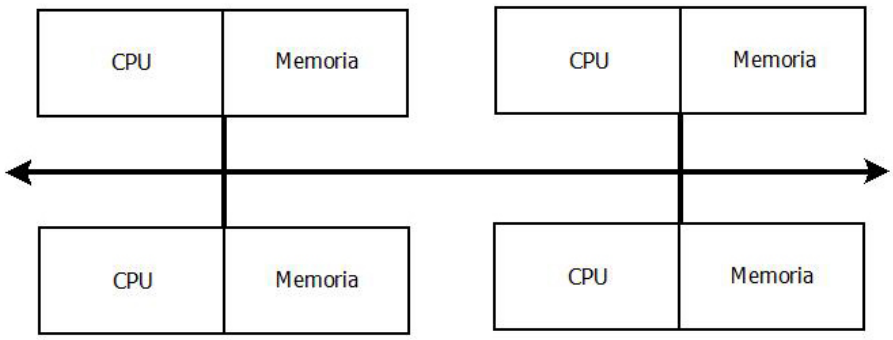

Figura 3.5: Memoria distribuida.

3.2.5. Arquitectura de memoria compartida distribuida híbrida

Los componentes de memoria compartida pueden llegar a ser la propia memoria de la maquina o unidades de procesamiento gráficos (GPU), el componente de la memoria distribuida es la conexión en red de las diversas máquinas de memoria compartida que sólo conocen su propia memoria y no la de alguna otra máquina, por lo tanto, se requieren las comunicaciones en red para poder disponer de los datos entre las máquinas conectadas a la red.
Las computadoras más grandes y más rápidas actualmente usan este tipo de arquitectura, por lo que parece esta arquitectura seguirá prevaleciendo en los siguientes años.

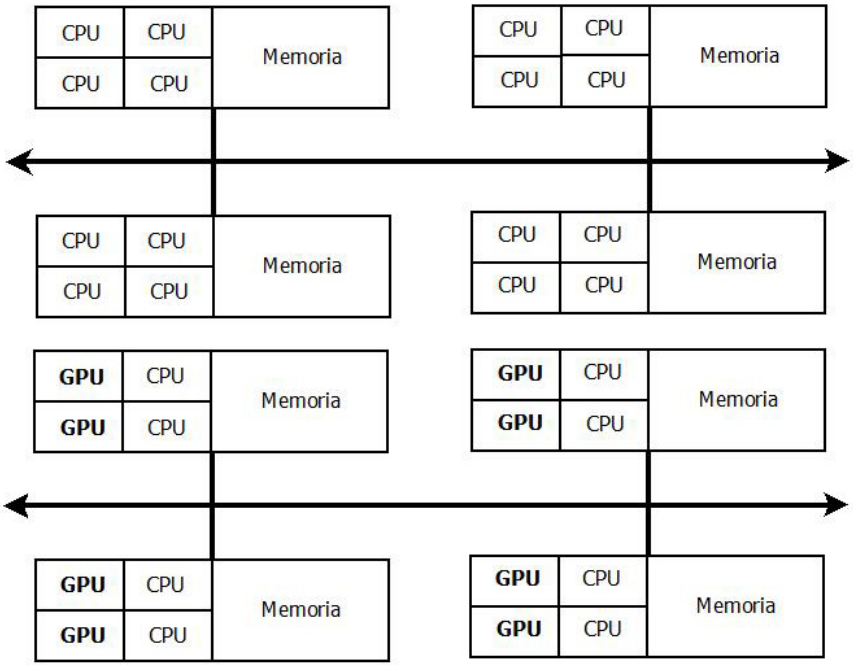

Figura 3.6: Memoria distribuida.

3.2.6. CUDA(*Compute Unified Device Architecture*)

Compute Unified Device Architecture (CUDA) es el nombre de la arquitectura desarrollada por la empresa NVIDIA para las tarjetas gráficas de su marca que tiene como propósito hacer accesible la tecnología para implementar cómputo de propósito general por medio de lenguajes de alto nivel. CUDA brinda los beneficios de los lenguajes de alto nivel como son C, OpenCL,

Fortran y C++ a los que se agregan algunas instrucciones sencillas.
Las GPU están conformadas por varios multiprocesadores y unidades de memoria que son capaces de manipular información con los núcleos de cómputo que poseen. Los núcleos de cómputo (CUDA *Cores*) se agrupan en bloques y se encargan de ejecutar las instrucciones por medio de hilos; estas instrucciones se pueden categorizar en las de tipo SIMD (*Single Instruction Multiple Data*) donde cada unidad de procesamiento realiza la misma función sobre datos diferentes [10].

3.2.6.1. Relación del procesamiento paralelo con CUDA

La idea principal del procesamiento paralelo es usar todas las unidades de procesamiento disponibles para poder realizar operaciones sobre los datos de manera simultanea [9]. Este procedimiento puede ser realizado por todos los nucleos que posee un microprocesador dandole a cada uno un conjunto de datos para que sean calculados, una ventaja del microprocesador sobre la GPU es que generalmente tiene una velocidad de reloj superior a la de la tarjeta gráfica, la única desventaja que tiene son sus escasas unidades de procesamiento. Los microprocesadores de hoy en día pueden llegar a tener hasta diez nucleos, sin embargo la tarjeta de video posee muchos más nucleos de procesamiento, esto se debe a que dichas tarjetas están diseñadas específicamente para poder procesar la información gráfica de la computadora a diferencia del procesamiento de propósito general para el que fueron diseñados los microprocesadores.

La tecnologia CUDA se encarga de brindarle herramientas al público para poder realizar calculo de proposito general de la informacion sobre la tarjeta de graficos de una manera sencilla. la numerosa cantidad de nucleos que posee la GPU le permiten dividir el conjunto de datos en partes mas manejables para cada unidad de procesamiento y hacer los calculos pertinentes.

3.2.6.2. Beneficios a la comunidad

En los inicios de la computación de datos de propósito general con tarjetas gráficas las necesidades que existían eran muchas, uno de los grandes retos a superar era el formato en que se manipulaba la información que era mandada a dichas tarjetas. la mejor manera de poder

procesar datos era por medio de OpenGL y como su nombre lo indica, es una libreria de graficos de código abierto, debido a eso, los investigadores y programadores de esos días tenían que arreglarselas para poder manipular sus datos para adaptarlos al formato de openGL y asi poder interpretarlos nuevamente cuando el procesamiento hubiera concluido, en otras palabras, tenian que procesar datos de proposito general como si fueran datos de gráficos por computadora.

Cuando se supo de la tecnología GPGPU generó una grán espectativa pues era algo que muchos programadores anhelaban, programar sin tener que adaptarlo a algun gráfico. Después de que se difundiera tanto el GPGPU en las distintas comunidades de científicos e investigadores, se pusieron a trabajar en diversos problemas que pertenecen a distintas ramas del conocimiento de una manera mas eficiente [4]. Algunas de las disciplinas que se fueron beneficiadas con esta tecnología fueron:

- Imágenes médicas
- Dinámica de fluidos
- Ciencias ambientales
- Computación e ingeniería
- Administración y finanzas

3.2.6.3. Arquitectura

Como ya se ha mencionado la diferencia mas remarcable entre el CPU y GPU son las unidades aritmetico lógicas de procesamiento o ALU por sus siglas en inglés como se muestra en la figura 3.7, estas unidades son las que realizan las operaciones a los datos. Hay ocasiones en que es mejor realizar el procesamiento enteramente con el microprocesador de la computadora. Un ejemplo de ello sería realizar el procesamiento de una cantidad de datos mínima, la tarjeta de video podria realizar las operaciones rápidamente como es normal, sin embargo, podria ralentizarse el proceso de manipulación de datos debido al tiempo que se usa para transmitir los datos hacia la GPU.

Generalmente este tipo de casos no es tan frecuente, ya que en su mayoría se utilizan volúmentes de información bastante grandes como lo son registros de bases de datos de miles de tuplas o cientos de cálculos matemáticos para poder reconocer algun elemento que se encuentre en una imagen y poder utilizar la menor cantidad de tiempo posible para hacer dichos cálculos.

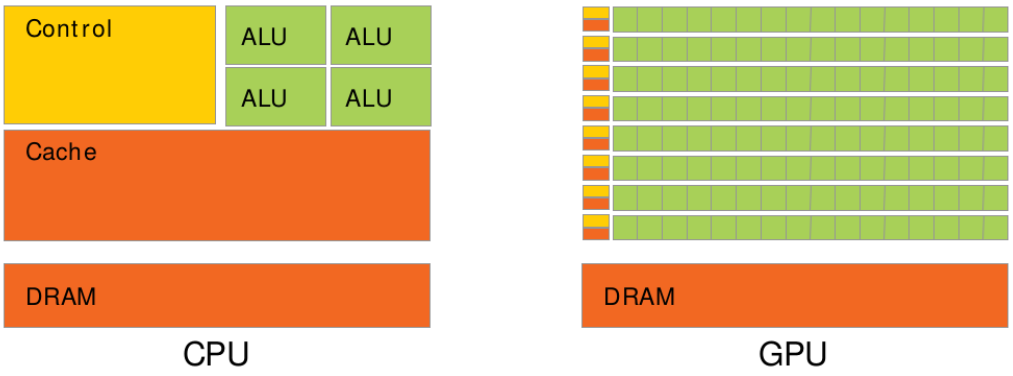

Figura 3.7: Comparativa de unidades de procesamiento de CPU contra las de GPU [15].

3.2.6.4. La plataforma de cálculo

Un reto muy grande que ha tenido que superar esta tecnología es que conforme avanza el tiempo van cambiando tanto los microprocesadores y las tarjetas gráficas con un numero mayor de unidades de procesamiento, esto exige que el lenguaje que se deba implementar sea escalable a las necesidades. El modelo de procesamiento paralelo de CUDA esta diseñado para funcionar de esa manera, con una serie de extensiones al lenguaje C/C++ [20] la curva de aprendizaje no es tan marcada.

Para poder utilizar la paquetería que NVIDIA facilita desde su pagina web y comenzar a realizar programas que usen GPGPU se debe tener conocimeinto sobre algunos puntos principales que conforman la arquitectura.

3.2.6.4.0.1. *Host* y *device*

Para comenzar a planificar una aplicación que funciona paralelamente se debe tener muy presente cúales son las partes críticas del programa que deben ser paralelizadas. La función principal

siempre iniciará del lado del *host* utilizando el microprocesador de la computadora y cuando sea requerido el procesamiento de la GPU se mandan los datos necesarios hacia el *device*, por tanto se da por entendido que el procesamiento del *host* es toda accion realizada gracias al CPU de la computadora, mientras que el *device* se encarga de realizar las operaciones en paralelo gracias a la tarjeta gráfica como se muestra en la fig. 3.8.

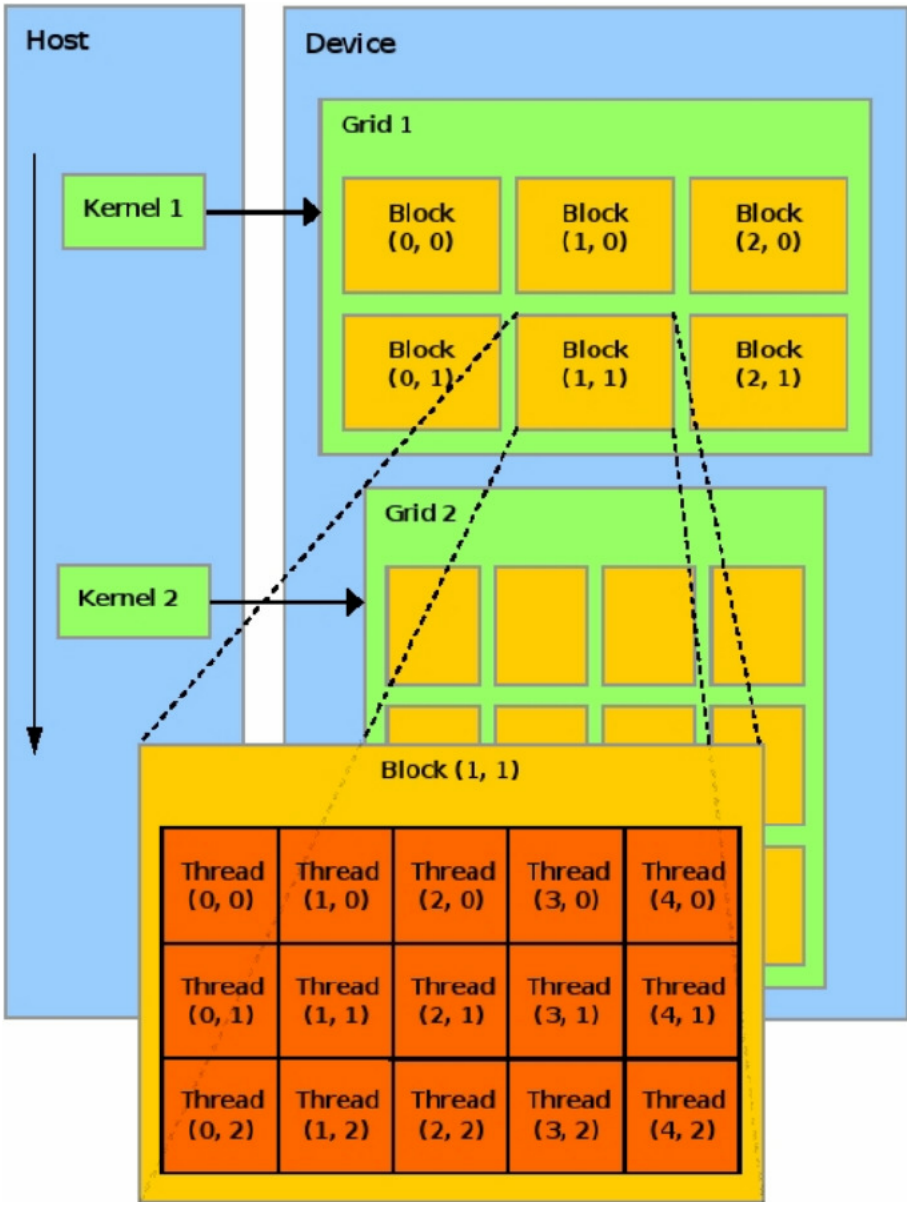

Figura 3.8: Representación del procesamiento del lado del *host* y *device* [15].

3.2.6.4.0.2. *Kernel*

Se le llama *kernel* a todas las funciones que se ejecutan únicamente del lado del *device*, en un programa CUDA pueden existir distintas llamadas a *kernels*, cuando se manda ejecutar una de estas funciones el *kernel* se ejecuta las veces que sea necesaria en cada uno de las unidades de procesamiento que conforman la tarjeta gráfica, los parámetros más importantes que se deben considerar para poder ejecutar un *kernel* son el número de bloques, el número de hilos y los parametros que tendrá.

3.2.6.4.0.3. Hilos de ejecución

Los hilos de ejecución son la unidad mínima de procesamiento de un programa CUDA, estos hilos se encargan de realizar las operaciones que se definen en el *kernel* y se encuentran agrupados por bloques de modo que existe un número limitado de hilos en un bloque. Normalmente se pueden utilizar haciendo referencia a una sola dimensión X, pero se pueden tratar dentro del programa como si fueran arreglos bidimensionales o tridimensionales, todo dependiendo de la abstracción para la solución que queramos darle al problema en cuestión.

3.2.6.4.0.4. Bloques de ejecución

Se implementan los bloques de ejecución para poder organizar la manipulación de los hilos, estos se pueden utilizar de manera similar que los hilos, ya que pueden ser configurados para operar linealmente, bidimensional o tridimensionalmente.

3.2.6.4.0.5. Arreglo de bloques

En la fig. 3.8 se puede apreciar del lado del *device* el *grid* o enmallado que contiene diversos bloques, el enmallado sirve para poder manipular el conjunto de bloques de manera multidimensional, un detalle que se debe tener en cuenta es que las mallas unicamente pueden realizar la ejecución de un *kernel* por vez.

3.2.6.4.0.6. Compilación de un programa CUDA

Se tiene el siguiente código para compilar:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <cuda_runtime.h>
4  #define blk 2
5  #define thr 100
6  __device__ int arreglo_d[blk*thr] = {0};
7  __global__ void funcion()
8  {
9          int tid = threadIdx.x + (blockDim.x*blockIdx.x);
10         arreglo_d[tid] = tid;
11 }
12 int main(int argc, char *argv[])
13 {
14         int arreglo[blk*thr];
15         funcion<<<blk,thr>>>();
16         cudaMemcpyFromSymbol(arreglo, arreglo_d, sizeof(arreglo));
17         int x;
18         for(x=0; x<blk*thr; x++)
19         {
20                 printf("\tarreglo[%03d] = %03d\n",x , arreglo[x]);
21         }
22         return 0;
23 }
```

La función que realiza el programa es asignar el valor del id que tiene el hilo al campo del arreglo correspondiente, devolver los valores al *host* y mostrarlos en pantalla. Para poder compilar el programa sobre Linux se debe tener instalado con anterioridad el *software* de NVIDIA llamado *CUDA toolkit*.

En la terminal nos posicionamos en el directorio en que se encuentra nuestro codigo de programa `ejemplo.cu`, después escribimos el nombre del compilador `nvcc` y asignamos los parametros de como se muestra:

```
nvcc ejemplo.cu -o ejemplo
```

La estructua en que estan organizados los parametros de la linea de compilación son muy similares al estilo de compilación del lenguaje C/C++. Una vez compilado el programa sólo resta ejecutarlo mediante:

```
./ejemplo
```

3.2.6.4.0.7. Especificaciones del equipo de cómputo

Para este proyecto se han utilizado dos equipos de cómputo con las caracteristicas que muestra la tabla 3.1 y 3.2.

Nombre	Descripción
Sistema operativo	Linux Mint 17.3 Cinnamon 64 bit
Procesador	Intel Core^{TM} i7-4790 CPU @ 3.60 GHz x 8
Disco duro	Kingston (240GB) estado sólido SSD NOW
Tarjeta gráfica	GTX Titan X (12 GB)
Placa base	Gigabyte Technology Co.
Versión de gcc	4.8.4
Versión de CUDA	7.5

Tabla 3.1: Configuración del equipo de pruebas 1.

Nombre	Descripción
Sistema operativo	Linux Mint 18.2 Cinnamon 64 bit
Procesador	Intel Core^{TM} 2 Duo CPU E7500 @ 2.93 GHz x 2
Disco duro	Seagate Barracuda 7200.12 500GB
Tarjeta gráfica	Nvidia Corporation [GeForce GT730]
Placa base	BIOSTAR G3ID-M7.
Versión de gcc	5.4.0
Versión de CUDA	9.0

Tabla 3.2: Configuración del equipo de pruebas 2.

Capítulo 4

Compresión de propósito general en CUDA

El algoritmo de Huffman se compone de varias etapas, las cuales se presentan en la figura 4.1.

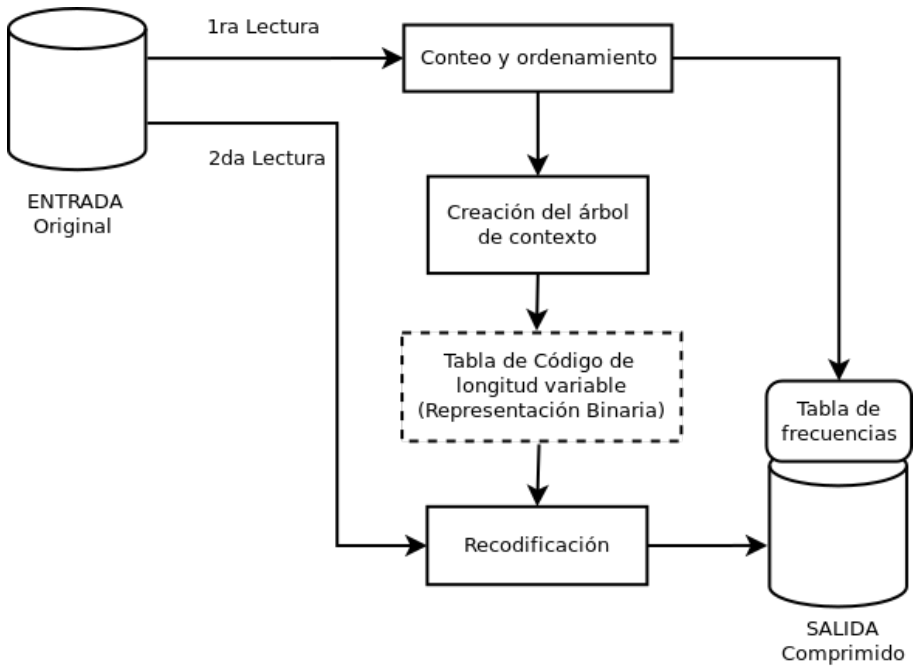

Figura 4.1: Diagrama de compresión por bloques.

4.1. Fase de conteo y ordenamiento

En esta fase se lee y se cuenta por primera vez cada símbolo que posee el archivo original, creando una tabla donde se especifica la frecuencia de aparición de cada símbolo.

La estrategia que se siguió para realizar la fase de conteo fue dividir el archivo a comprimir en el número total de hilos disponibles en la tarjeta (el producto del número de bloques por el número de hilos en cada bloque) para que a cada hilo de ejecución se le asigne la misma cantidad de *bytes* a manipular, como se muestra en la figura 4.2. Como no en todos los casos se pueden tener todos los hilos con la misma cantidad de carga de trabajo en el peor de los casos un hilo trabajaría con menos carga que el resto, de tal suerte que no los retrase.

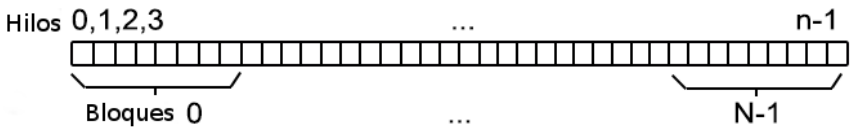

Figura 4.2: Distribución de memoria por bloques e hilos.

4.1.1. Generación de la tabla de frecuencias en paralelo

Al inicio de la ejecución se obtiene la longitud total de caracteres a procesar y esta cantidad se copia a una variable global del *device* junto con el contenido del archivo. Del lado del *host* se crea un arrelgo de enteros de dimensión 256 (los posibles valores de cada *byte*) donde se guarda el conteo total de cada símbolo y del lado del *device* se crea un arreglo que tiene dimensiones del número de bloques por el número de hilos en cada bloque por 256, para, en primera instancia, realizar un conteo parcial.

Cuando se ejecuta el *kernel* que realiza los conteos, a cada hilo se le asigna una porción del archivo cuya longitud se calcula mediante 4.1, donde l es la longitud en *bytes* de la porción a contabilizar por cada hilo, f es el tamaño total del archivo, b el número de bloques con que se ejecutó el programa y t el número de hilos de cada bloque. De esta forma, cada hilo genera su

propia tabla de conteo parcial.

$$l = ceil\left(\frac{f}{b * t}\right) \tag{4.1}$$

Una vez que se ha concluido la ejecución de todos los hilos de la primera fase, en un segundo *kernel*, que se ejecuta con un bloque y 256 hilos se realiza la sumatoria de cada una de las tablas de conteos parciales, como lo muestra la figura 4.3. Primero se suman las tablas de los hilos por cada bloque y después las de cada uno de los bloques, ubicando el conteo total en la primera tabla del arreglo de sumas parciales.

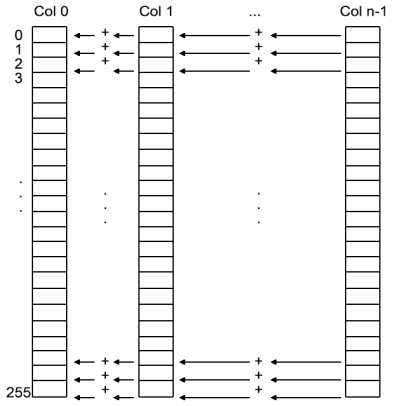

Figura 4.3: Sumatoria de conteos parciales por hilo.

Finalizados los dos *kernels*, debido a la diferencia de memoria entre las variables del *host* y del *device* se copia sólo el segmento de memoria que tiene el conteo total y se aloja en un arreglo del lado del *host*, resultando como se ejemplifica en la tabla 4.1 una tabla de frecuencias de mínimo tamaño en comparación a la creada en el *device*.

4.2. Creación del árbol de contexto

Una vez creada la tabla de frecuencias, como se muestra en la figura 4.4 se procede a crear una lista ligada donde cada elemento de la lista albergará el símbolo y la frecuencia que está asociada a él. Después, se toman los dos primeros elementos de dicha lista (los cuales en el primer caso serán aquellos que posean la menor frecuencia de la tabla de frecuencias) se crea

Carácter	Frecuencia
n	1
u	1
s	2
a	2
e	2
c	3

Tabla 4.1: Ejemplo de tabla de frecuencia del *Host*.

un nodo padre que apunta a cada uno de ellos, de manera que tendrá un nodo hijo izquierdo y derecho, la frecuencia del nodo padre será la suma de sus hijos creando así, un árbol binario; después, este árbol se ordena en la lista y se repite el procedimiento hasta que finalmente se obtenga un árbol binario con todos los símbolos colocados como hojas del mismo, este árbol es conocido también como árbol de longitud de descripción mínima o MDL por sus siglas en inglés [19].

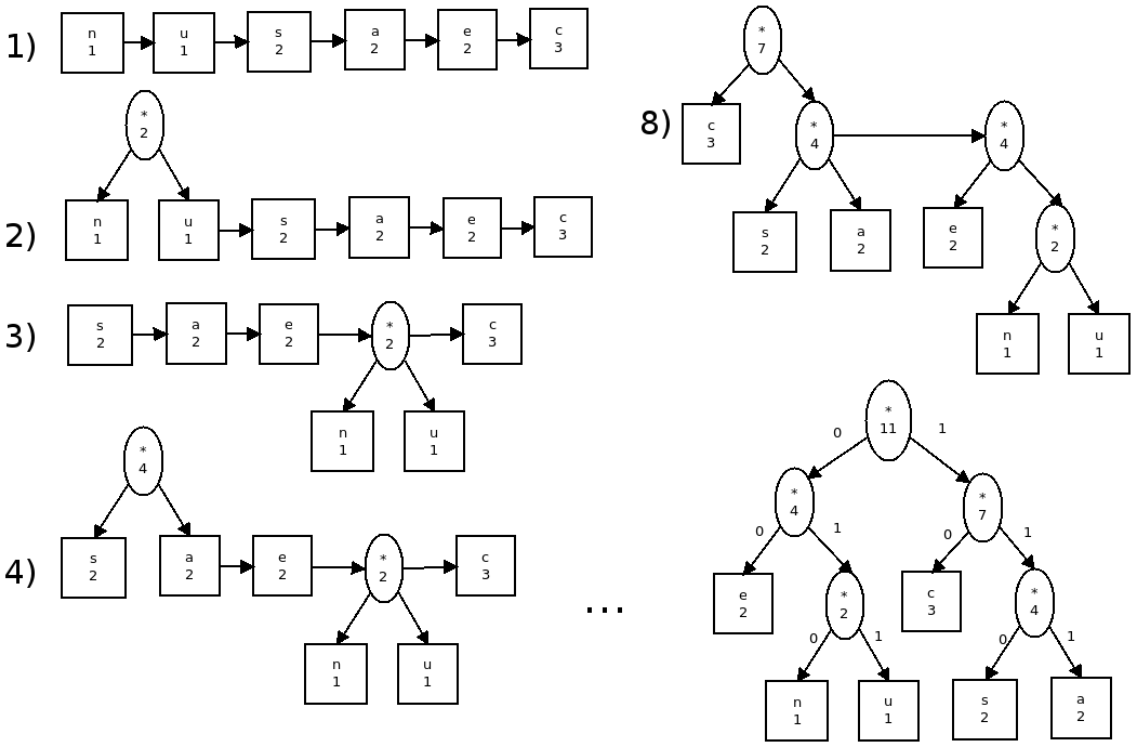

Figura 4.4: Creación de árbol de contexto de longitud mínima.

4.3. Creación de la tabla de códigos de longitud variable

Se procede a crear la tabla de códigos de longitud variable o VLC por sus siglas en inglés, mostrada en la figura 4.2, el código VLC se utiliza para reemplazar cada caracter sin comprimir por una serie de *bits* que representaran al mismo cuando se cree el archivo comprimido. Los códigos de cada caracter se obtienen haciendo recorridos por el árbol MDL desde el nodo padre hacia cada una de las hojas, anotando los saltos hacia sus hijos, recordando el valor de cada salto ya sea izquierda o derecha, si el salto es hacia el nodo hijo de la izquierda se anota un *bit* de valor cero y si el salto es hacia la derecha se anota un *bit* de valor uno, como se muestra en la figura 4.5.

Caracter	Código VLC
n	010
u	011
s	110
a	111
e	00
c	10

Tabla 4.2: Ejemplo de tabla de código de longitud variable.

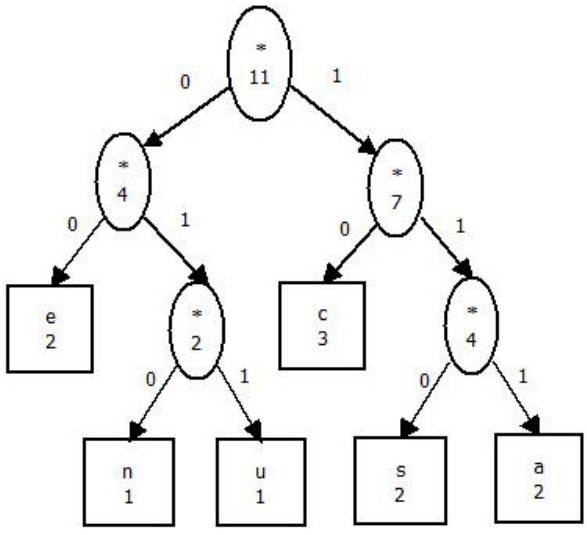

Figura 4.5: Criterio de árbol MDL para obtener el código de longitud variable.

4.4. Compresión en paralelo

Se realiza una segunda lectura al archivo original y se hace la división del archivo como se menciona en la sección 4.1, después, sustituyen los caracteres de cada hilo por el código de longitud variable que le corresponde [2] [12] [13] [14], realizar este procedimiento tiene el riesgo, si el caracter a sustituir tiene un código muy largo existe una posibilidad de que se pueda sobre escribir el siguiente caracter que aun no se ha leído; Para mitigar ese riesgo, se usa un arreglo como buffer donde se van colocando los codigos que se van obteniendo y cuando no se corre el riesgo de que exista la sobreescritura antes mencionada el dato es escrito.
Al guardar el archivo comprimido se le colocan como cabeceras algunos datos que son tomados en cuenta para realizar el proceso de descompresión, tal acomodo se muestra en la tabla 4.3.

Dato	Tipo de dato	Tamaño en Bytes
Tabla de frecuencias	unsigned int	1024
Número de hilos	long int	4
Longitud del archivo original	unsigned int	4
Número de caracteres escritos por hilo	unsigned int	4
Datos comprimidos	char	Variable

Tabla 4.3: Estructura del archivo.

Se puede apreciar claramente que se esta añadiendo más de un *Kilobyte* al archivo comprimido, aunque parece un poco excesivo vale la pena, ya que para volúmenes de datos grandes la tasa de compresión debe superar con creces ese tamaño extra.

4.4.1. Tasa de compresión

Cuando un algoritmo comprime un conjunto de datos como mínimo se espera que genere a lo sumo un archivo que tenga menos tamaño que el original, en caso de que dicho archivo sea de igual tamaño o incluso mayor se puede decir que el desempeño de ese algoritmo es pobre o que no es efectivo por su baja tasa de compresión. La tasa de compresión es un factor con el cual se puede medir el desempeño de distintos algoritmos de esta naturaleza, generalmente se expresa

en términos de porcentaje y se calcula como muestra la ecuación 4.2.

$$\text{Tasa de compresión} = \frac{\text{Tamaño del archivo de salida}}{\text{Tamaño del archivo de entrada}} \tag{4.2}$$

Donde el la tasa de compresión depende de dos datos que son variables en la mayoria de los casos, el archivo de entrada se refiere al documento o los datos que se someterán al proceso de compresión (archivo original) y el archivo de salida hace referencia al tamaño que tendrán los datos una vez que sean codificados (archivo comprimido), el porcentaje que se maneja comúnmente va desde cero hasta cien, siendo cien el tamaño original del archivo, aunque existen casos donde se puede existir una compresión negativa, en este caso el porcentaje seria mayor a 100 % [3]. En el caso hipotetico de que un archivo midiera 1000MB y su archivo comprimido midiera 800MB siguiendo la ecuación su tasa de compresión sería de valor 0.8 (80 %).
Como se menciona en la sección 1.4, sólo se adaptará el algoritmo de huffman a una versión paralelizada, por tanto, los resultados de la tasa de compresión no representan un factor importante para someter a prueba y mencionar en el capítulo 5.

4.4.2. Generación de los archivos de prueba

Las pruebas de velocidad de procesamiento se realizaron utilizando distintos archivos de prueba que fueron creados por un programa que se encarga de agregar caracteres a un archivo, los cuales eran creados de manera aleatoria con el objetivo de no hacer pruebas con datos que pudieran estar comprimidos previamente.
El inconveniente de esto es que los caracteres que conforman un archivo comprimido tienen un codigo de longitud variable grande debido a la uniformidad de frecuencia de sus caracteres, que ocasiona que la profundidad en el árbol MDL sea suficiente para generar un *byte* completo y la compresión se vea mermada.

4.4.3. Paqueterias para compilar

Para poder ejecutar sin problemas el código que se encuentra en los anexos se deben tener instaladas las siguientes paqueterias para la plataforma Linux:

- NVIDIA CUDA Toolkit (1.6GB aproximadamente)
- Build essential (20KB)

El único requisito de *hardware* que se debe cumplir para poder ejecutar la versión paralela es poseer una tarjeta de video capaz de soportar la tecnología CUDA.

4.4.4. Compilación y ejecución

Antes de comenzar a compilar el código hay que posicionarnos en el directorio donde se encuentra el archivo que queremos compilar, después se debe entrar en modo root para que no suceda ningún problema con la linea de comandos y se tengan los privilegios requeridos. Para comprimir datos de manera secuencial se usa el archivo `comprimir.c`, en la linea de comandos se escribe:

```
gcc comprimir.c -lm -o comprimir
```

El parametro `-l` esta ahí debido a que el programa usa la librería `math.h` que se usa para realizar operaciones matemáticas y se utiliza el compilador gcc pues la codificación esta escrita en lenguaje C. Para ejecutar el programa se escribe:

```
./comprimir [archivo]
```

Donde `archivo` hace referencia al nombre del conjunto de datos que se desean comprimir. Para usar la descompresión secuencial se lleva a cabo el mismo procedimiento con la única diferencia que se utiliza el archivo `descomprimir.c`.

Si se desea compilar el programa paralelizado se deberá escribir:

```
nvcc comprimir.cu -lm -o comprimir
```

En esta ocasión se usa un compilador distinto al gcc, es compilador especial para código CUDA `nvcc`, para ejecutar el programa se debe escribir en la terminal:

```
./comprimir [archivo]
```

De igual manera que en la versión secuencia el parametro `archivo` hace referencia al nombre del archivo que se desea comprimir.

Capítulo 5

Resultados

5.1. Validación de resultados

Al inicio del proyecto se realizó una comparación preliminar entre la versión secuencial y la paralela del conteo de caracteres para distintos tamaños de archivos con el propósito de obtener un punto de referencia en cuanto a la aceleración que se podria alcanzar en las pruebas finales, los resultados de dicha prueba se muestran en las tablas 5.1 y 5.2.

Cabe mencionar que estas pruebas de procesamiento fureron realizadas únicamente con el equipo mencionado anteriormente en la tabla 3.1.

Tamaño(MB)	Secuencial (s)	Paralelo (s)	Aceleración (s)
100	1.35	0.23	5.87
200	2.46	0.33	7.45
400	4.89	0.50	9.78
600	7.20	0.69	10.43
800	9.51	0.88	10.81

Tabla 5.1: Tiempos de ejecución en secuencial y paralelo para conteo de caracteres.

Tamaño(GB)	Secuencial (s)	Paralelo (s)	Aceleración (s)
1.0	12.17	1.13	10.77
1.2	14.59	1.39	10.50
1.4	17.09	1.70	10.05
1.6	19.52	2.05	9.52
1.8	21.77	2.26	9.63
2.0	24.36	2.60	9.37

Tabla 5.2: Tiempos de ejecución en secuencial y paralelo para conteo de caracteres.

Se puede observar que los tiempos de ejecución obtenidos en las pruebas de conteo, para comparar la versión secuencial y la paralela, fueron prometedores. Al aumentar el tamaño del archivo, se incrementa la diferencia entre los tiempos, consiguiéndose una aceleración cercana a 10 veces menos tiempo en paralelo que en secuencial.

En las pruebas realizadas con la tarjeta GeForce GT730 cuyo desempeño se muestra en las tablas 5.3 y 5.4, es posible apreciar un aumento progresivo en la aceleración, pues inicia con valores similares y aumenta hasta llegar a obtener una mejoría en la velocidad de compresión mayor al 300 % comparando el tamaño de archivo más grande con el más pequeño. Se tiene también que pasa de ser alrededor de seis veces más rápido a ser casi diecinueve veces más veloz que el algoritmo secuencial.

Tamaño(MB)	Secuencial (s)	Paralelo (s)	Aceleración (s)
100	22.73	3.72	6.11
200	45.18	7.22	6.26
400	91.03	13.11	6.94
600	134.88	13.66	9.87
800	182.55	14.22	12.77

Tabla 5.3: Tiempos de ejecución para compresión con GPU Nvidia GeForce GT730.

Tamaño(GB)	Secuencial (s)	Paralelo (s)	Aceleración (s)
1.0	229.32	18.36	12.49
1.2	275.84	20.302	13.59
1.4	323.84	19.70	16.44
1.6	371.67	20.50	18.12
1.8	415.98	23.76	17.50
2.0	460.28	24.24	18.99

Tabla 5.4: Tiempos de ejecución para compresión con GPU Nvidia GeForce GT730.

Comparando los resultados del equipo que usa la tarjeta Titan X que se encuentran en las tablas 5.5 y 5.6 se observa que los tiempos de ejecución del algoritmo secuencial son más cortos que el del equipo que usa la GeForce 730, esto es debido a que la velocidad de reloj del microprocesador i7 es mayor que la del Core 2 Duo, haciendo así, que la aceleración por parte de la tarjeta gráfica se vea un poco disminuida. Sin embargo, aún teniendo ese factor en cuenta, se puede apreciar que la aceleracion llega a tener mínimamente un valor poco mayor de diez, lo cual es un resultado remarcable.

Tamaño(MB)	Secuencial (s)	Paralelo (s)	Aceleración (s)
100	11.98	0.86	13.91
200	24.48	1.55	15.76
400	48.78	2.92	16.68
600	73.55	4.57	16.09
800	98.09	6.69	14.66

Tabla 5.5: Tiempos de ejecución para compresión con GPU GTX Titan X.

Tamaño(GB)	Secuencial (s)	Paralelo (s)	Aceleración (s)
1.0	122.50	8.94	13.70
1.2	145.72	11.02	13.22
1.4	170.887	13.77	12.41
1.6	190.52	16.70	11.40
1.8	217.61	19.49	11.16
2.0	235.77	22.54	10.46

Tabla 5.6: Tiempos de ejecución para compresión con GPU GTX Titan X.

Capítulo 6

Conclusiones y trabajo futuro

Hasta el tamaño de 2.0 GB, se puede constatar que el tiempo de transferencia de los datos del host al device, no constituyó un cuello de botella para la fase de generación de la tabla de frecuencias. En la fase de recodificación no es necesario hacer una nueva transferencia del *host* al *device*, dado que los datos permanecen en la memoria de la tarjeta gráfica.

En las pruebas de codificacion de archivos se puede observar que el comportamiento de los tiempos es muy similar al de las pruebas preliminares dado que los tiempos de ejecución guardan una aceleración aproximada de diez veces más rápido a favor de la versión paralelizada del algoritmo, con esto se puede afirmar que es muy recomendable realizar la compresión de datos en forma paralela.

Como trabajo futuro se recomiendan realizar los siguientes puntos:

- Implementar el algoritmo de descompresión.
- Adecuar el algoritmo para que pueda ejecutarse sobre cualquier tarjeta gráfica.
- Realizar pruebas y desarrollo de la aplicación para otras plataformas.

Apéndice A

Archivos de cabecera

A.0.1. params.h

```
#define N_CAR 256
/////////////////////////////////////////////////////
//ESTRUCTURA DE LA TABLA DE FRECUENCIAS YA ORDENADA //
/////////////////////////////////////////////////////
typedef struct
{
        unsigned int caracter;
        unsigned int cantidad;
} tab_asc;
/////////////////////////////////////////////////
//ESTRUCTURA DE LOS NODOS DEL ARBOL DE CONTEXTO //
/////////////////////////////////////////////////
typedef struct _nodo
{
   unsigned int datocar;
   unsigned int datocan;
   struct _nodo *sig;
   struct _nodo *der;
```

```
19     struct _nodo *izq;
20 } tiponodo;
```

A.0.2. qsort.h

```
1  ////////////////////////////////////////////////////////////
2  // ORDENAMIENTO ASCENDENTE DE LA LISTA SIMPLEMENTE LIGADA  //
3  ////////////////////////////////////////////////////////////
4  void asc(tab_asc *p, int lim_iz, int lim_de)
5  {
6      int izq, der;
7      tab_asc temp, piv;
8      izq = lim_iz;
9      der = lim_de;
10     piv = p[(izq + der)/2];
11     do{
12         while(p[izq].cantidad < piv.cantidad && izq < lim_de)
13         {
14             izq++;
15         }
16         while(piv.cantidad < p[der].cantidad && der > lim_iz)
17         {
18             der--;
19         }
20         if(izq <= der)
21         {
22             temp = p[izq];
23             p[izq] = p[der];
24             p[der] = temp;
25             izq++;
26             der--;
27         }
```

```
28      }
29      while(izq <= der);
30      if(lim_iz < der)
31      {
32          asc(p, lim_iz, der);
33      }
34      if(lim_de > izq)
35      {
36          asc(p, izq, lim_de);
37      }
38  }
```

Apéndice B

Compresión de datos secuencial

B.0.1. comprimir.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <string.h>
5  #include <math.h>
6  #include "params.h"
7  #include "qsort.h"
8  typedef tiponodo *pnodo;
9  typedef tiponodo *arbol;
10 ////////////////////////////////////////////////////////////
11 //ESTRUCTURA DONDE SE GUARDA EL CODIGO DE LONGITUD VARIABLE  //
12 //Y LA PROFUNDIDAD A LA QUE SE ENCUENTRA EL CARACTER         //
13 ////////////////////////////////////////////////////////////
14 typedef struct
15 {
16    unsigned short int code;
17    char deep;
18 } codestruct;
```

```
19 codestruct codes[256];
20 int cadindex = 0;
21 ////////////////////////////////////////////////////////////////
22 //SE ENCARGA DE GENERAR EL CARACTER A GUARDAR CON BASE EN LOS  //
23 //CODIGOS DE LONGITUD VARIABLE QUE POSEA EN EL MISMO CARACTER  //
24 ////////////////////////////////////////////////////////////////
25 void bintocar(char *binbits, FILE * archivo)
26 {
27     int indice,j,sum, totalbyte;
28     unsigned char caracter;
29     for(indice = 0; indice < 8; indice++ )
30     {
31         j=indice%8;
32         if(j==0)
33         {
34             sum = 128;
35             totalbyte = 0;
36         }
37         if(binbits[indice] == '1')
38         {
39             totalbyte += sum;
40         }
41         sum/=2;
42         if(sum == 0)
43         {
44             caracter = (char)totalbyte;
45             fwrite(&caracter,1,1, archivo);
46         }
47     }
48 }
49 ////////////////////////////////////////////////////////////
50 //GENERA EL ARBOL DE CONTEXTO A PARTIR DE LA LISTA ORDENADA //
51 ////////////////////////////////////////////////////////////
```

```
52 void crea_arbol(arbol *huff)
53 {
54     arbol cabeza = *huff, p, q, aux;
55     if(cabeza->sig != NULL)
56     {
57         while(cabeza->sig !=NULL)
58         {
59             p = cabeza;
60             q = p->sig;
61             aux = (arbol)malloc(sizeof(tiponodo));
62             aux->datocan = p->datocan + q->datocan;
63             aux->sig = NULL;
64             aux->izq = p;
65             aux->der = q;
66             if(q->sig != NULL)
67             {
68                 cabeza = q->sig;
69             }
70             else
71             {
72                 cabeza->sig = NULL;
73                 cabeza = aux;
74                 break;
75             }
76             p->sig = NULL;
77             q->sig = NULL;
78             p = cabeza;
79             q = p->sig;
80             if(cabeza->sig == NULL)//para el ultimo caso donde solo
                   quedan dos nodos de la lista
81             {
82                 if(aux->datocan < cabeza->datocan)
83                 {
```

```
84                      aux->sig = cabeza;
85                      cabeza = aux;
86                  }
87                  else if(aux->datocan >= cabeza->datocan)
88                  {
89                      cabeza->sig = aux;
90                  }
91              }
92              else if(aux->datocan < p->datocan)//para el inicio de la
                    lista
93              {
94                  aux->sig = p;
95                  cabeza = aux;
96              }
97              else if( aux->datocan >= q->datocan && q->sig == NULL)//
                    para el final de la lista
98              {
99                  q->sig = aux;
100             }
101             else
102             {
103                 while(aux->datocan >= q->datocan && q->sig != NULL)
104                 {
105                     p = q;
106                     q = q->sig;
107                 }
108                 if(aux->datocan < q->datocan)
109                 {
110                     p->sig = aux;
111                     aux->sig = q;
112                 }
113                 else
114                 {
```

```
115                     q->sig = aux;
116                 }
117             }
118         }
119     }
120     else
121     {
122         printf("La lista se encuentra vacia\n");
123     }
124     *huff = cabeza;
125 }
126 ////////////////////////////////////////////////////////////////////
127 //CREA UN NUEVO NODO EN LA LISTA, SI LA LISTA NO EXISTE LA CREA  //
128 ////////////////////////////////////////////////////////////////////
129 void inserta_nodo(arbol *huff, arbol *nuevo_nodo)//crea la lista
       ligada ordenada
130 {
131     arbol p = *huff;
132     arbol aux = *nuevo_nodo;
133     if( p == NULL )
134     {
135         *huff = (arbol)malloc(sizeof(tiponodo));
136         (*huff)->datocar = aux->datocar;
137         (*huff)->datocan = aux->datocan;
138         (*huff)->izq = (*huff)->der = NULL;
139     }
140     else
141     {
142         while(p->sig != NULL)
143         {
144             p = p->sig;
145         }
146         p->sig = aux;
```

```
147     }
148 }
149 ////////////////////////////////////////
150 //GENERA EL CODIGO DE LONGITUD VARIABLE //
151 ////////////////////////////////////////
152 void dbits(char codigo, char deep, char *arr_bits, int bandera, FILE *
        archivo)
153 {
154    char i;
155    int mask=1;
156    mask = mask << (deep-1);
157    for (i=0; i<deep; i++)
158    {
159       arr_bits[cadindex] = ((codigo&mask)!=0)+'0';
160       cadindex++;
161       mask= mask>>1;
162       if(cadindex >= 8)
163       {
164                 bintocar(arr_bits, archivo);
165                 cadindex = 0;
166       }
167    }
168    if(bandera)
169    {
170                 while(cadindex < 8 && cadindex != 0)
171                 {
172                         arr_bits[cadindex] = '0';
173                         cadindex++;
174                 }
175                 bintocar(arr_bits, archivo);
176    }
177 }
178 ////////////////////////////////////////////////
```

```
179 //OBTIENE EL CODIGO DE LONGITUD VARIABLE Y    //
180 //LO GUARDA EN LA ESTRUCTURA CORRESPONDIENTE  //
181 ////////////////////////////////////////////////
182 void genera_codigo(arbol a, int codigo, char deep)//obtiene el codigo
       binario de cada caracter en el arbol y la profundidad
183 {
184     if(a != NULL)
185     {
186         if(a->izq==NULL && a->der==NULL)
187         {
188             codes[a->datocar].code = codigo;
189             codes[a->datocar].deep = deep;
190         }
191         if(a->izq!=NULL)
192         {
193             genera_codigo(a->izq, codigo<<1, deep+1);
194         }
195         if(a->der!=NULL)
196         {
197             genera_codigo(a->der, (codigo<<1)+1, deep+1);
198         }
199     }
200 }
201 int main(int argc, char *argv[])
202 {
203     unsigned long int longitud,original;
204     if(argv[1] == NULL)
205     {
206         perror("Entrada invalida, nombre del archivo erroneo.\n\n");
207         exit(1);
208     }
209     char bits[8];
210     unsigned char caracter;
```

```
211     int indice, sum_car = 0, contador = 1;
212     tab_asc tabla[N_CAR];
213     for(indice=0; indice<N_CAR; indice++)//inicializa los valores del
           arreglo
214     {
215         tabla[indice].caracter = indice;
216         tabla[indice].cantidad = 0;
217     }
218     FILE *arch_entrada, *arch_salida;
219     arch_entrada = fopen(argv[1], "r");
220     while(fread(&caracter,1,1,arch_entrada))
221     {
222         tabla[caracter].cantidad++;
223         sum_car++;
224     }
225         fclose(arch_entrada);
226         int arr_freq[N_CAR], arr_aux[N_CAR];
227         for(indice = 0; indice < N_CAR; indice++)
228         {
229                 arr_freq[indice] = tabla[indice].cantidad;
230         }
231         FILE *f;
232         f = fopen("frecuencias.huff", "wb");
233         fwrite(arr_freq, sizeof(int), sizeof(arr_freq), f);
234         fclose(f);
235     asc(tabla,0,N_CAR-1);
236     arbol arbol_huff = NULL, nuevo;//declaracion de las variables tipo
           arbol
237     for(indice=0; indice<N_CAR; indice++)//crea la lista ligada
           ordenada
238     {
239         if(tabla[indice].cantidad > 0)
240         {
```

```
241             nuevo = (arbol)malloc(sizeof(tiponodo));
242             nuevo->datocar = tabla[indice].caracter;
243             nuevo->datocan = tabla[indice].cantidad;
244             nuevo->sig = NULL;
245             nuevo->izq = nuevo->der = NULL;
246             inserta_nodo(&arbol_huff, &nuevo);
247         }
248     }
249     crea_arbol(&arbol_huff);
250     genera_codigo(arbol_huff,0,0);
251     arch_entrada = fopen(argv[1], "r");
252     arch_salida = fopen(strcat(argv[1],".huff"), "a");
253     while(fread(&caracter,1,1,arch_entrada))
254     {
255         if(contador < sum_car)
256         {
257                 dbits(codes[caracter].code, codes[caracter].deep, bits
                        , 0, arch_salida);
258         }
259         else if(contador == sum_car)
260         {
261                 dbits(codes[caracter].code, codes[caracter].deep, bits
                        , 1, arch_salida);
262         }
263         contador++;
264     }
265     fclose(arch_salida);
266     fclose(arch_entrada);
267     return 0;
268 }
```

Apéndice C

Descompresión de datos secuencial

C.0.1. descomprimir.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <math.h>
#include "params.h"
#include "qsort.h"
typedef tiponodo *pnodo;
typedef tiponodo *arbol;
typedef struct
{
   unsigned short int code;
   char deep;
} codestruct;
////////////////////////////////////////////////////////////////
//UTILIZA EL CONTENIDO DEL ARCHIVO COMPRIMIDO USANDO EL ARBOL DE   //
//CONTEXTO PARA GENERAR NUEVAMENTE EL ARCHIVO ORIGINAL             //
////////////////////////////////////////////////////////////////
```

```
19 void cartobin( FILE *datos_comprimidos, FILE *datos_descomprimidos,
      arbol huff, int sumatoria)
20 {
21    int i = 0,j,num;
22    unsigned char caracter,mask;
23    arbol padre,actual;
24    padre = actual = huff;
25     while(fread(&caracter,1,1,datos_comprimidos))
26     {
27       mask=128;
28       for(j=0;j<8;j++)
29       {
30                   if(i < sumatoria)
31                   {
32                      num = (caracter&mask) ? 1:0;
33                      if(num == 0)
34                      {
35                         actual = actual->izq;
36                      }
37                      else if(num == 1)
38                      {
39                             actual = actual->der;
40                      }
41                      if(actual->izq == NULL && actual->der == NULL &&
                           i < sumatoria)
42                      {
43                         num = actual->datocar;
44                         fwrite(&num,1,1, datos_descomprimidos);
45                         i++;
46                         actual = padre;
47                      }
48                          mask/=2;
49                  }
```

```
50        }
51     }
52  }
53  ////////////////////////////////////////////////////////////
54  //GENERA EL ARBOL DE CONTEXTO A PARTIR DE LA LISTA ORDENADA //
55  ////////////////////////////////////////////////////////////
56  void crea_arbol(arbol *huff)
57  {
58      arbol cabeza = *huff, p, q, aux;
59      if(cabeza->sig != NULL)
60      {
61          while(cabeza->sig !=NULL)
62          {
63              p = cabeza;
64              q = p->sig;
65              aux = (arbol)malloc(sizeof(tiponodo));
66              aux->datocan = p->datocan + q->datocan;
67              aux->sig = NULL;
68              aux->izq = p;
69              aux->der = q;
70
71              if(q->sig != NULL)
72              {
73                  cabeza = q->sig;
74              }
75              else
76              {
77                  cabeza->sig = NULL;
78                  cabeza = aux;
79                  break;
80              }
81              p->sig = NULL;
82              q->sig = NULL;
```

```
83              p = cabeza;
84              q = p->sig;
85              if(cabeza->sig == NULL)//para el ultimo caso donde solo
                   quedan dos nodos de la lista
86              {
87                  if(aux->datocan < cabeza->datocan)
88                  {
89                      aux->sig = cabeza;
90                      cabeza = aux;
91                  }
92                  else if(aux->datocan >= cabeza->datocan)
93                  {
94                      cabeza->sig = aux;
95                  }
96              }
97              else if(aux->datocan < p->datocan)//para el inicio de la
                   lista
98              {
99                  aux->sig = p;
100                 cabeza = aux;
101             }
102             else if( aux->datocan >= q->datocan && q->sig == NULL)//
                   para el final de la lista
103             {
104                 q->sig = aux;
105             }
106             else
107             {
108                 while(aux->datocan >= q->datocan && q->sig != NULL)
109                 {
110                     p = q;
111                     q = q->sig;
112                 }
```

```
113                 if(aux->datocan < q->datocan)
114                 {
115                     p->sig = aux;
116                     aux->sig = q;
117                 }
118                 else
119                 {
120                     q->sig = aux;
121                 }
122             }
123         }
124     }
125     else
126     {
127         printf("La lista se encuentra vacia\n");
128     }
129     *huff = cabeza;
130 }
131 ////////////////////////////////////////////////////////////////////
132 //CREA UN NUEVO NODO EN LA LISTA, SI LA LISTA NO EXISTE LA CREA  //
133 ////////////////////////////////////////////////////////////////////
134 void inserta_nodo(arbol *huff, arbol *nuevo_nodo)
135 {
136     arbol p = *huff;
137     arbol aux = *nuevo_nodo;
138     if( p == NULL )
139     {
140         *huff = (arbol)malloc(sizeof(tiponodo));
141         (*huff)->datocar = aux->datocar;
142         (*huff)->datocan = aux->datocan;
143         (*huff)->sig = (*huff)->izq = (*huff)->der = NULL;
144     }
145     else
```

```
146     {
147         while(p->sig != NULL)
148         {
149             p = p->sig;
150         }
151         p->sig = aux;
152     }
153 }
154 int main(int argc, char *argv[])
155 {
156         tab_asc tabla[N_CAR];
157         int arr_frec[N_CAR], sum_car = 0;
158         FILE * frecuencias = fopen("frecuencias.huff", "rb");
159         fread(arr_frec, sizeof(int), 256, frecuencias);
160         fclose(frecuencias);
161         int a;
162         for(a=0; a<N_CAR; a++)
163         {
164             tabla[a].caracter = a;
165             tabla[a].cantidad = arr_frec[a];
166             sum_car = sum_car + tabla[a].cantidad;
167         }
168         asc(tabla,0,N_CAR-1);
169         arbol arbol_huff = NULL, nuevo;//declaracion de las variables
               tipo arbol
170         int indice;
171     for(indice=0; indice<N_CAR; indice++)//crea la lista ligada
           ordenada
172     {
173         if(tabla[indice].cantidad > 0)
174         {
175             nuevo = (arbol)malloc(sizeof(tiponodo));
176             nuevo->datocar = tabla[indice].caracter;
```

```
177             nuevo->datocan = tabla[indice].cantidad;
178             nuevo->sig = NULL;
179             nuevo->izq = nuevo->der = NULL;
180             inserta_nodo(&arbol_huff, &nuevo);
181         }
182     }
183     crea_arbol(&arbol_huff);
184     FILE *arch_comprimido, *arch_descomprimido;
185     arch_comprimido = fopen(argv[1], "r");
186     char nombre_arch[120];
187     indice = 0;
188     while( indice < strlen(argv[1])-5 )
189     {
190         nombre_arch[indice] = argv[1][indice];
191         if(indice+1 == strlen(argv[1])-5)
192         {
193                 nombre_arch[indice+1] = '\0';
194         }
195         indice++;
196     }
197     arch_descomprimido = fopen(nombre_arch,"a");
198     cartobin(arch_comprimido ,arch_descomprimido, arbol_huff, sum_car)
        ;
199     fclose(arch_comprimido);
200     fclose(arch_descomprimido);
201     return 0;
202 }
```

Apéndice D

Compresión de datos paralela

D.0.1. comprimir.cu

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <cuda_runtime.h>
5  #include "params.h"
6  #include "qsort.h"
7  #define blk 24
8  #define thr 128
9  ////////////////////////////////////////////////////////////
10 //ESTRUCTURA DONDE SE GUARDA EL CODIGO DE LONGITUD VARIABLE  //
11 //Y LA PROFUNDIDAD A LA QUE SE ENCUENTRA EL CARACTER         //
12 ////////////////////////////////////////////////////////////
13 typedef struct
14 {
15    unsigned short int code;
16    char deep;
17 }codestruct;
18 typedef tiponodo *pnodo;
```

```
19 typedef tiponodo *arbol;
20 codestruct codes[256];//TABLA PARA CODIGO VLC
21 __device__ unsigned int tabla_freq_d[N_CAR*thr*blk] = {0};
22 __device__ unsigned int len_d;
23 // ESTRUCTURA PARA EL CONTEO DE ESCRITURA DE CADA HILO
24 __device__ unsigned int cont_esc_d[blk*thr] = {0};
25 ////////////////////////////////////////////////
26 //KERNEL QUE REALIZA EL CONTEO DE CARACTERES     //
27 ////////////////////////////////////////////////
28 __global__ void conteo(unsigned char *archivo)
29 {
30         int tid;
31         tid = threadIdx.x + (blockIdx.x * blockDim.x);
32         int tam_aux,base;
33         tam_aux = ceilf((float)len_d/((float)blk*(float)thr));
34         base = (blockIdx.x*thr+threadIdx.x)*N_CAR;
35         for(int x=tid*tam_aux; x<(tid+1)*tam_aux && x<len_d; x++)
36         {
37                 tabla_freq_d[base+archivo[x]]++;
38         }
39 }
40 ////////////////////////////////////////////////////
41 //KERNEL QUE SE ENCARGA DE SUMAR EL CONTEO PARCIAL   //
42 //DE CADA HILO AL PRIMER HILO                        //
43 ////////////////////////////////////////////////////
44 __global__ void acumula_conteo()
45 {
46         for(int b=0;b<blk;b++)
47         {
48             for(int t=0;t<thr;t++)
49             {
50                if(t!=0 || b!=0)
51                {
```

```
52                    tabla_freq_d[threadIdx.x] += tabla_freq_d[(b*thr+t)*
                          N_CAR+threadIdx.x];
53                }
54            }
55        }
56 }
57 ////////////////////////////////////////////////////////////////
58 //SE ENCARGA DE GENERAR EL CARACTER A GUARDAR CON BASE EN LOS  //
59 //CODIGOS DE LONGITUD VARIABLE QUE POSEA EN EL MISMO CARACTER  //
60 ////////////////////////////////////////////////////////////////
61 __device__ void bintocar(char *binbits , unsigned char *archivo, int
      cont)
62 {
63         int thr_id = threadIdx.x + (blockIdx.x * blockDim.x);
64         int indice,sum, totalbyte;
65         unsigned char caracter;
66         sum = 128;
67         totalbyte = 0;
68         for(indice = 0; indice < 8; indice++)
69         {
70                 if(binbits[indice] == '1')
71                 {
72                         totalbyte += sum;
73                 }
74                 sum/=2;
75         }
76         caracter = (char)totalbyte;
77         archivo[cont] = caracter;
78         cont_esc_d[thr_id]++;
79 }
80
81
82 ////////////////////////////////////////
```

```
83  //GENERA EL CODIGO DE LONGITUD VARIABLE //
84  ////////////////////////////////////////
85  __global__ void dbits(unsigned char *archivo, codestruct *tab_vlc) //
        GENERA CODIGO DE ARCHIVO COMPRIMIDO
86  {
87          int tid;
88          tid = threadIdx.x + (blockIdx.x * blockDim.x);
89          int tam_aux;
90          tam_aux = ceilf((float)len_d/((float)blk*(float)thr));  //PARA
                SABER EN QUE SECCION DEBE TRABAJAR EL PROGRAMA
91          int cadindex = 0;
92          //contador_d INICIA EN CADA PRIMER ELEMENTO DEL HILO POR
                ANALIZAR
93          //PARA LLEVAR EL CONTROL DE LAS INSERCIONES DE CARACTERES
94          int contador_d = tid*tam_aux;
95          int x;
96          char i;
97          char bits[8]={0};//BYTE QUE SE INSERTA DESPUES DE LA
                CODIFICACION VLC
98          int mask=1;
99          for(x = tid*tam_aux; x < (tid+1)*tam_aux && x < len_d; x++) //
                RECORRE LAS SECCIONES DEL ARCHIVO
100         {
101                 mask=1;
102                 mask = mask << (tab_vlc[archivo[x]].deep-1);
103                 for (i = 0; i < tab_vlc[archivo[x]].deep; i++)
104                 {
105                         bits[cadindex] = ((tab_vlc[archivo[x]].code&
                                mask)!=0)+'0';
106                         cadindex++;
107                         mask = mask>>1;
108                         if(cadindex >= 8)
109                         {
```

```
110                                 bintocar(bits, archivo, contador_d);
111                                 contador_d++;
112                                 cadindex = 0;
113                         }
114                 }
115         }
116         while(cadindex < 8)
117         {
118                 bits[cadindex] = '0';
119                 cadindex++;
120         }
121         bintocar(bits, archivo, contador_d);
122 }
123 ////////////////////////////////////////////////////////////////////
124 //CREA UN NUEVO NODO EN LA LISTA, SI LA LISTA NO EXISTE LA CREA  //
125 ////////////////////////////////////////////////////////////////////
126 void inserta_nodo(arbol *huff, arbol *nuevo_nodo)//AGREGA UN NODO MAS
       AL FINAL DE LA LISTA ORDENADA
127 {
128     arbol p = *huff;
129     arbol aux = *nuevo_nodo;
130     if( p == NULL )
131     {
132                 *huff = *nuevo_nodo;
133     }
134     else
135     {
136         while(p->sig != NULL)
137         {
138             p = p->sig;
139         }
140         p->sig = aux;
141     }
```

```
142 }
143 void crea_arbol(arbol *huff)
144 {
145     arbol cabeza = *huff, p, q, aux;
146     if(cabeza->sig != NULL)
147     {
148         while(cabeza->sig !=NULL)
149         {
150             p = cabeza;
151             q = p->sig;
152             aux = (arbol)malloc(sizeof(tiponodo));
153             aux->datocan = p->datocan + q->datocan;
154             aux->sig = NULL;
155             aux->izq = p;
156             aux->der = q;
157             if(q->sig != NULL)
158             {
159                 cabeza = q->sig;
160             }
161             else
162             {
163                 cabeza->sig = NULL;
164                 cabeza = aux;
165                 break;
166             }
167             p->sig = NULL;
168             q->sig = NULL;
169             p = cabeza;
170             q = p->sig;
171             if(cabeza->sig == NULL)
172             {
173                 if(aux->datocan < cabeza->datocan)
174                 {
```

```
175                     aux->sig = cabeza;
176                     cabeza = aux;
177                 }
178                 else if(aux->datocan >= cabeza->datocan)
179                 {
180                     cabeza->sig = aux;
181                 }
182             }
183             else if(aux->datocan < p->datocan)
184             {
185                 aux->sig = p;
186                 cabeza = aux;
187             }
188             else if( aux->datocan >= q->datocan && q->sig == NULL)
189             {
190                 q->sig = aux;
191             }
192             else
193             {
194                 while(aux->datocan >= q->datocan && q->sig != NULL)
195                 {
196                     p = q;
197                     q = q->sig;
198                 }
199                 if(aux->datocan < q->datocan)
200                 {
201                     p->sig = aux;
202                     aux->sig = q;
203                 }
204                 else
205                 {
206                     q->sig = aux;
207                 }
```

```
208             }
209         }
210     }
211     else
212     {
213         printf("La lista se encuentra vacia\n");
214     }
215     *huff = cabeza;
216 }
217 
218 ////////////////////////////////////////////////
219 //OBTIENE EL CODIGO DE LONGITUD VARIABLE Y    //
220 //LO GUARDA EN LA ESTRUCTURA CORRESPONDIENTE  //
221 ////////////////////////////////////////////////
222 void genera_codigo(arbol a, int codigo, char deep)
223 {
224     if(a != NULL)
225     {
226         if(a->izq==NULL && a->der==NULL)
227         {
228             codes[a->datocar].code = codigo;
229             codes[a->datocar].deep = deep;
230         }
231         if(a->izq!=NULL)
232         {
233             genera_codigo(a->izq, codigo<<1, deep+1);
234         }
235         if(a->der!=NULL)
236         {
237             genera_codigo(a->der, (codigo<<1)+1, deep+1);
238         }
239     }
240 }
```

```
241
242 int main(int argc, char *argv[])
243 {
244         //ABRE EL ARCHIVO Y CUENTA SU LONGITUD
245         FILE *archivo = fopen(argv[1], "rb");
246         fseek(archivo, 0, SEEK_END);
247         unsigned int len = ftell(archivo);
248         unsigned char *arreglo = (unsigned char*)malloc(len);
249         fseek(archivo, 0, SEEK_SET);
250         fread(arreglo, 1, len, archivo);
251         fclose(archivo);
252         unsigned int tabla_freq[N_CAR];
253         unsigned char *arreglo_d;
254         //COPIA EL ARCHIVO AL DEVICE
255         cudaMalloc((void **)&arreglo_d, len);
256         cudaMemcpy(arreglo_d, arreglo, len, cudaMemcpyHostToDevice);
257         cudaMemcpyToSymbol(len_d, &len, sizeof(len));
258         //CREA TABLA DE FRECUENCIAS
259         conteo<<<blk,thr>>>(arreglo_d);
260         acumula_conteo<<<1,256>>>();
261         //COPIA LA TABLA DE FRECUENCIAS AL HOST
262         cudaMemcpyFromSymbol(tabla_freq, tabla_freq_d, sizeof(
              tabla_freq));
263         //CREA ARCHIVO QUE CONTIENE LA TABLA DE FRECUENCIAS
264         FILE *f;
265         f = fopen("archivo_comprimido.huff", "wb");
266         fwrite(tabla_freq, sizeof(unsigned int), N_CAR, f);//ESCRIBE
              LA TABLA DE FRECUENCIAS
267         fclose(f);
268         //INICIALIZA LA ESTRUCTURA PARA TABLA DE FRECUENCIAS AUXILIAR
269         tab_asc tab_frec[N_CAR];
270         int x;
271         for(x=0; x<N_CAR; x++)
```

```
272         {
273                 tab_frec[x].caracter = x;
274                 tab_frec[x].cantidad = tabla_freq[x];
275         }
276         //ORDENA LA TABLA DE FRECUENCIAS AUXILIAR
277         asc(tab_frec,0,N_CAR-1);//ORDENA ASCENDENTEMENTE LA LISTA
278         //CREA LA LISTA LIGADA
279         arbol arbol_huff = NULL, nuevo;//DECLARACION DE LAS VARIABLES
               TIPO ARBOL
280         for(x=0; x<N_CAR; x++)
281         {
282                 if(tab_frec[x].cantidad > 0)
283                 {
284                         nuevo = (arbol)malloc(sizeof(tiponodo));
285                         nuevo->datocar = tab_frec[x].caracter;
286                         nuevo->datocan = tab_frec[x].cantidad;
287                         nuevo->sig = NULL;
288                         nuevo->izq = nuevo->der = NULL;
289                         inserta_nodo(&arbol_huff, &nuevo);
290                 }
291         }
292         //CREA EL ARBOL MDL
293         crea_arbol(&arbol_huff);
294         //CREA TABLA DE LONGITUD DE CODIGO VARIABLE
295         genera_codigo(arbol_huff,0,0);
296         //COPIA LA TABLA VLC AL DEVICE
297         codestruct *tab_codigos_d;
298         cudaMalloc((void **)&tab_codigos_d, sizeof(codestruct)*N_CAR);
299     cudaMemcpy(tab_codigos_d,&codes,sizeof(codestruct)*N_CAR,
           cudaMemcpyHostToDevice);
300     //GENERA EL CODIGO DE ARCHIVO COMPRIMIDO
301     dbits<<<blk,thr>>>(arreglo_d, tab_codigos_d);
```

```
302     //COPIA DE REGRESO EL ARRCHIVO SOBREESCRITO CON EL CODIGO DE
            COMPRESION
303     cudaMemcpy(arreglo, arreglo_d, len, cudaMemcpyDeviceToHost);
304         //COPIA EL CONTENIDO DEL ARREGLO CONT_ESC(CONTADOR DE
                ESCRITURA) DEL DEVICE AL HOST
305         unsigned int cont_esc[blk*thr] = {0};
306         cudaMemcpyFromSymbol(cont_esc, cont_esc_d, sizeof(cont_esc));
307         int hils, tamaux;
308         long int nthr =  blk*thr;
309         tamaux = ceilf((float)len/((float)blk*(float)thr));
310         f = fopen("archivo_comprimido.huff", "ab");
311         fwrite(&nthr, sizeof(long int), 1, f);//ESCRIBE EL NUMERO DE
                HILOS
312         fwrite(&len, sizeof(unsigned int), 1, f);//ESCRIBE LAS
                DIMENSIONES DEL ARCHIVO ORIGINAL
313         for(hils = 0; hils < nthr; hils++)//RECORRE EL ID DE CADA HILO
314         {
315                 fwrite(cont_esc+hils, sizeof(unsigned int), 1, f);//
                        ESCRIBE EL # DE CARACTERES ESCRITOS POR EL HILO
316         }
317         for(hils = 0; hils < nthr; hils++)//RECORRE EL INDICE DE CADA
                HILO
318         {
319                 fwrite(arreglo+(hils*tamaux), sizeof(unsigned char),
                        cont_esc[hils], f);//ESCRIBE LOS DATOS COMPRIMIDOS
                        POR HILO
320         }
321         fclose(f);
322         return 0;
323 }
```

Bibliografía

[1] Rabia Arshad, Adeel Saleem, and Danista Khan. Performance comparison of Huffman Coding and Double Huffman Coding. In *2016 6th International Conference on Innovative Computing Technology, INTECH 2016*, pages 361–364, 2017.

[2] Ph. D. Departament of Electrical Dror Baron and Computer Engineering. *Fast Parallel Algorithms For Universal Lossless Source Coding*. PhD thesis, 2003.

[3] A.G Fallis. *Data Compression: The Complete Reference*, volume 53. 2013.

[4] Iván Omar Rivera G and Miguel Vargas-Lombardo. Principios y Campos de Aplicación en CUDA Programación paralela y sus potencialidades, 2012.

[5] M. Nelson Gailly and J. *The Data Compression Book*. 2nd editio edition, 1995.

[6] Marcus Geelnard. Basic Compression Library Manual API version 1.2. 2006.

[7] Oscar Herrera Alcántara and Francisco Javier Zaragoza Martínez. Incompresibilidad y compresión de datos sin pérdidas: Un acercamiento con descubrimiento de patrones. *Computación y Sistemas*, 13(1):45–60.

[8] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[9] Wen-mei Hwu and David Kirk. *Programmin Massively Parallel Processors: A Hands-on Approach*. 2016.

[10] J. Cheng, M. Grossman and T. KcKercher. *Professional CUDA C Programming*. Wrox, 2014.

[11] S R Kodituwakku and U S Amarasinghe. Comparison of Lossless Data Compression Algorithms. *Indian Journal of Computer Science and Engineering*, 1(4):416–425, 2010.

[12] Nikhil Krishnan and Dror Baron. Performance of Parallel Two-Pass MDL Context Tree Algorithm.

[13] Nikhil Krishnan and Dror Baron. A universal parallel two-pass MDL context tree compression algorithm. *IEEE Journal on Selected Topics in Signal Processing*, 9(4):741–748, 2015.

[14] Nikhil Krishnan, Dror Baron, and Mehmet Kıvanç Mıhçak. A Parallel Two-Pass MDL Context Tree Algorithm for Universal Source Coding.

[15] Nvidia. CUDA C Programming Guide. *Design Guide*, (August):228, 2014.

[16] Jason Sanders and Edward Kandrot. *CUDA by Example*, volume 54. 2010.

[17] Khalid Sayood. Huffman Coding. *Introduction to Data Compression (Fourth Edition)*, pages 43–89, 2012.

[18] Top500. TOP500 Supercomputer Sites.

[19] Peter Wayner. *Compression Algorithms for Real Programmers*. 2000.

[20] Nicholas Wilt. *The CUDA Handbook*. Number 1. 2013.