PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

USO DE PROCESADORES DE TARJETAS GRÁFICAS PARA VALIDACIÓN DE FORTALEZAS DE ALGORITMOS CRIPTOGRÁFICOS

VÍCTOR FELIPE PACHECO LATOJA

INFORME FINAL DEL PROYECTO
PARA OPTAR AL TÍTULO PROFESIONAL DE
INGENIERO CIVIL EN INFORMÁTICA

MARZO 2013

PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

USO DE PROCESADORES DE TARJETAS GRÁFICAS PARA VALIDACIÓN DE FORTALEZAS DE ALGORITMOS CRIPTOGRÁFICOS

VÍCTOR FELIPE PACHECO LATOJA

Profesor Guía: **Jaime Briggs Luque**

Carrera: **Ingeniería Civil Informática**

MARZO 2013

A mi madre, mi hermana y mi familia materna
por su apoyo incondicional y su entrega de
valores durante toda mi vida.

Índice

Glosario de Términos i
Lista de Abreviaturas iii
Lista de Figuras iv
Lista de Tablas vii
Resumen ix
Abstract x
1 Presentación del Tema 1
1.1 Introducción 1
1.2 Origen del Tema 2
1.3 Definición del Problema 2
1.4 Objetivos 3
1.4.1 Objetivo general 3
1.4.2 Objetivos específicos 3
1.5 Alcances o Ámbitos del Proyecto 4
1.6 Metodología y Plan de Trabajo 4
2 Evolución de las Tarjetas Gráficas (GPU) 6
2.1 Historia 6
2.1.1 Primera Generación de GPUs 6
2.1.2 Segunda Generación de GPUs 6
2.1.3 Tercera Generación de GPUs 7
2.1.4 Cuarta Generación de GPUs 7
3 Lenguajes GPGPU 10
3.1 Brook para GPUs 10
3.2 CTM – Close to the Metal 10
3.3 CUDA – Compute Unified Device Architecture 11
4 Proyectos Desarrollados en CUDA 12
4.1 Folding@home 12
4.2 Solving Dense Linear Systems on Graphics Processors 12
4.3 Badaboom Media Converter 13
4.4 A Neural Network on GPU 13
4.5 Gnort: High Performance Network Intrusion Detection Using Graphics Processors 14
4.6 CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography 15
4.7 ElcomSoft Distributed Password Recovery 16
5 Tarjetas Gráficas: Arquitectura y Funcionamiento 18
5.1 Arquitectura Lógica y Funcionamiento del Procesador de la Tarjeta Gráfica ... 18

5.1.1 Parallel Computing Architecture 19
5.1.2 Graphics Processing Architecture 20
5.1.3 Thread Processing Cluster – Texture Processing Cluster (TPC) 21
5.1.4 Sistema e Interfaz de Memoria 22
5.1.5 Arquitectura SIMT 23
5.1.6 Mayor Número de Núcleos de Procesamiento 23
5.1.7 Gran Número de Hebras 24
5.1.8 Soporte para Doble Precisión 24
5.2 Arquitectura Física de la Tarjeta Gráfica 24
5.2.1 Procesador de la Tarjeta Gráfica 25
5.2.2 Memorias RAM 26
5.2.3 Sistema de Refrigeración 26
5.2.4 Administración de Consumo Energético y Temperatura 27

6 Tarjetas Gráficas: Programación 29

6.1 Modelo de Programación 30
6.1.1 Kernels 30
6.1.2 Thread Hierarchy – Jerarquía de Hebras 31
6.1.3 Memory Hierarchy – Jerarquía de la Memoria 33
6.1.4 Host and Device – Anfitrión y Dispositivo 34
6.1.5 Capacidad de Cómputo 35
6.2 C para CUDA 36
6.2.1 Extensiones del Lenguaje 36
6.2.1.1 Tipos de Clasificadores de Función 37
6.2.1.2 Tipos de Clasificadores de Variables 38
6.2.1.3 Configuración de la Ejecución 40
6.2.1.4 Variables Incorporadas 41
6.2.2 Compilación con NVCC 42
6.2.3 Componente Común en Tiempo de Ejecución 43
6.2.3.1 Tipos de Vectores Incorporados 43
6.2.3.2 Tipo dim3 44
6.2.4 Ejemplo: Multiplicación de Matrices 44

7 Criptografía y Criptoanálisis 49

7.1 Definición de Criptografía y Criptoanálisis 49
7.2 Tipos de Criptografía 50
7.2.1 Criptografía Simétrica 50
7.2.2 Criptografía Asimétrica 51
7.2.3 Funciones Hash 52
7.3 Tipos de Ataques 53
7.3.1 Brute Force Attack – Ataque de Fuerza Bruta 53
7.3.2 Ciphertext-only Attack – Ataque de Sólo Texto Cifrado 54
7.3.3 Known-plaintext Attack – Ataque de Texto Claro Conocido 54
7.3.4 Chosen-ciphertext Attack – Ataque de Texto Cifrado Elegido 55
7.3.5 Chosen-plaintext Attack – Ataque de Texto Claro Elegido 55
7.3.6 Criptoanálisis Diferencial 56
7.3.7 Criptoanálisis Lineal 56

7.4 Propuesta del Algoritmo a Probar ... 57
7.4.1 Algoritmo Simétrico de Cifrado de Flujo: RC4 ... 57
7.4.2 Resumen de la Propuesta ... 58

8 Diseño de la Solución al Problema ... 60

8.1 Diseño Conceptual ... 60
8.2 Diseño en UML ... 61
8.2.1 Diagramas de Casos de Uso ... 61
8.2.2 Diagramas de Actividad ... 64

9 Configuración de las Pruebas ... 66

9.1 Características de los Archivos y Claves ... 66
9.2 Características de las Pruebas ... 68
9.3 Plataforma para Pruebas ... 69

10 Ataque de Fuerza Bruta en el Procesador ... 70

10.1 Porción de Código Fuente que Realiza el Ataque de Fuerza Bruta ... 70
10.2 Ataque de Fuerza Bruta para Claves Seleccionadas ... 72
10.3 Ataque de Fuerza Bruta para Claves Aleatorias ... 75

11 Ataque de Fuerza Bruta en la Tarjeta Gráfica ... 77

11.1 Porción de Código Fuente que Realiza el Ataque de Fuerza Bruta ... 77
11.2 Ataque de Fuerza Bruta para Claves Seleccionadas ... 79
11.2.1 Ataque de Fuerza Bruta con Una Hebra ... 79
11.2.2 Ataque de Fuerza Bruta con 10 Hebras ... 81
11.2.3 Ataque de Fuerza Bruta con 50 Hebras ... 83
11.2.4 Ataque de Fuerza Bruta con 100 Hebras ... 85
11.2.5 Ataque de Fuerza Bruta con 200 Hebras ... 87
11.3 Ataque de Fuerza Bruta para Claves Aleatorias ... 89
11.3.1 Ataque de Fuerza Bruta con Una Hebra ... 89
11.3.2 Ataque de Fuerza Bruta con 10 Hebras ... 91
11.3.3 Ataque de Fuerza Bruta con 50 Hebras ... 93
11.3.4 Ataque de Fuerza Bruta con 100 Hebras ... 95
11.3.5 Ataque de Fuerza Bruta con 200 Hebras ... 97

12 Conclusiones ... 99

Anexo A. Código Fuente del Ataque de Fuerza Bruta en el Procesador ... 103

Anexo B. Código Fuente del Ataque de Fuerza Bruta en la Tarjeta Gráfica ... 108

Anexo C. Resultados del Ataque de Fuerza Bruta en el Procesador ... 113

C.1 Resultados del Ataque de Fuerza Bruta en el Procesador para el Archivo: textoplano2.txt ... 113
C.1.1 Ataque de Fuerza Bruta para Claves Seleccionadas ... 113
C.1.2 Ataque de Fuerza Bruta para Claves Aleatorias ... 114
C.2 Resultados del Ataque de Fuerza Bruta en el Procesador para el Archivo: textoplano3.txt ... 116
C.2.1 Ataque de Fuerza Bruta para Claves Seleccionadas ... 116

C.2.2 Ataque de Fuerza Bruta para Claves Aleatorias 117
Anexo D. Resultados del Ataque de Fuerza Bruta en la Tarjeta Gráfica 119
D.1 Resultados del Ataque de Fuerza Bruta en la Tarjeta Gráfica para el Archivo: textoplano2.txt 119
D.1.1 Ataque de Fuerza Bruta para Claves Seleccionadas 119
D.1.1.1 Ataque de Fuerza Bruta con Una Hebra 119
D.1.1.2 Ataque de Fuerza Bruta con 10 Hebras 120
D.1.1.3 Ataque de Fuerza Bruta con 50 Hebras 122
D.1.1.4 Ataque de Fuerza Bruta con 100 Hebras 123
D.1.1.5 Ataque de Fuerza Bruta con 200 Hebras 125
D.1.2 Ataque de Fuerza Bruta para Claves Aleatorias 126
D.1.2.1 Ataque de Fuerza Bruta con Una Hebra 126
D.1.2.2 Ataque de Fuerza Bruta con 10 Hebras 128
D.1.2.3 Ataque de Fuerza Bruta con 50 Hebras 129
D.1.2.4 Ataque de Fuerza Bruta con 100 Hebras 131
D.1.2.5 Ataque de Fuerza Bruta con 200 Hebras 132
D.2 Resultados del Ataque de Fuerza Bruta en la Tarjeta Gráfica para el Archivo: textoplano3.txt 134
D.2.1 Ataque de Fuerza Bruta para Claves Seleccionadas 134
D.2.1.1 Ataque de Fuerza Bruta con Una Hebra 134
D.2.1.2 Ataque de Fuerza Bruta con 10 Hebras 135
D.2.1.3 Ataque de Fuerza Bruta con 50 Hebras 137
D.2.1.4 Ataque de Fuerza Bruta con 100 Hebras 138
D.2.1.5 Ataque de Fuerza Bruta con 200 Hebras 140
D.2.2 Ataque de Fuerza Bruta para Claves Aleatorias 141
D.2.2.1 Ataque de Fuerza Bruta con Una Hebra 141
D.2.2.2 Ataque de Fuerza Bruta con 10 Hebras 143
D.2.2.3 Ataque de Fuerza Bruta con 50 Hebras 144
D.2.2.4 Ataque de Fuerza Bruta con 100 Hebras 146
D.2.2.5 Ataque de Fuerza Bruta con 200 Hebras 147
13 Referencias 149

Glosario de Términos

AGP (Accelerated Graphics Port): puerto desarrollado por Intel en 1996 el cual era de uso exclusivo de las tarjetas gráficas. Logró alcanzar velocidades de hasta 533 MHz con tasas de transferencia de 2 GB/s.

Arithmetic Intensity: es la razón entre las operaciones aritméticas y el ancho de banda de la memoria.

Command-Processor: es un componente de la plataforma DPVM que acepta comandos empaquetados por una aplicación. Es también el responsable de la programación de los procesadores para ejecutar una determinada tarea cuando recibe el comando específico.

Computación Gráfica: estudia los conceptos y algoritmos relacionados con la edición y producción de gráficas por computador.

Convolutional Neural Network: es un tipo especial de redes neuronales multicapas. Al igual que casi todas las otras redes neuronales se entrena con una versión del algoritmo "back-propagation", donde difiere es en su arquitectura. Este tipo de redes se han diseñado para reconocer patrones visuales directamente desde las imágenes de píxeles con un mínimo de pre-procesamiento.

Data Parallelism: es una técnica de programación para dividir un conjunto de datos de gran tamaño en trozos pequeños que pueden ser operados en paralelo. Después que los datos son procesados, se combinan formando nuevamente un conjunto de datos.

Data-Parallel Processor: es el componente computacional de la plataforma DPVM que permite desarrollar los cálculos de una aplicación a partir de los datos de entrada, entregando datos de salida (resultados).

DPVM (Data Parallel Virtual Machine): es una plataforma que presenta a la GPU como tres componentes mayores, los cuales son el Command-Processor, el Memory Controller y el Data-Parallel Processor.

Memory Controller: es un componente de la plataforma DPVM que se encarga del manejo de la memoria, ya sea la memoria local propia de la tarjeta gráfica, como también la memoria global del computador.

PCI (Peripheral Component Interconnect): bus que permite conectar dispositivos periféricos directamente a la placa madre, estos dispositivos pueden ser tarjetas de expansión o circuitos integrados incorporados dentro de ésta.

PCI-E (PCI-Express): es un desarrollo del bus PCI pero para uso exclusivo de tarjetas gráficas. Fue el reemplazante del puerto AGP y en la actualidad es un estándar. En su última versión alcanza un ancho de banda del bus de datos de 16 GB/s con una tasa de transferencia de 5 GB/s.

Procesador – CPU: componente principal de un computador que interpreta las instrucciones y procesa los datos provenientes de los programas en ejecución.

Snort: es un IDS/IPS (Intrusion Detection System/Instrusion Prevention System) Open-Source que utiliza un lenguaje de normas, el cual combina los beneficios de la firma, el protocolo y los métodos de inspección basados en anomalías. Es ampliamente empleado en la industria llegando a convertirse así en el estándar de facto.

Streaming Processors: es el encargado de ejecutar un kernel sobre todos los elementos de un stream de entrada, y colocar los resultados en un stream de salida.

T&L (Transformation and Lighting): término utilizado en computación gráfica. La transformación (Transformation) se refiere a la tarea de conversión de coordenadas en el espacio (movimientos de objetos en 3D y conversión de coordenadas 3D en 2D de la pantalla). La iluminación (Lighting) se refiere a la tarea de situar objetos que irradien luz en una escena virtual, a los cuales se les calculan las sombras y colores resultantes al incidir la luz en ellos.

Transistor: es un dispositivo electrónico semiconductor que se utiliza como amplificador o conmutador de una corriente eléctrica. Es un componente clave en toda la electrónica moderna, donde es ampliamente utilizado.

Lista de Abreviaturas

CUDA: Compute Unified Device Architecture.

GPU: Graphics Processing Unit.

CPU: Central Processing Unit.

GPGPU: General-Purpose computation on Graphics Processing Units.

RC4: Rivest Cipher 4.

XOR: Exclusive Or.

Lista de Figuras

Figura 5.1: Características del procesador GT200 y sus modos de funcionamiento. 19
Figura 5.2: Estructura en el modo Parallel Computing Architecture.............................20
Figura 5.3: Estructura en el modo Graphics Processing Architecture. 21
Figura 5.4: Estructura detallada de una unidad TPC. 22
Figura 5.5: Sistema de memoria principal de la tarjeta gráfica. 22
Figura 5.6: Tarjeta gráfica modelo NVIDIA GeForce GTX285. 25
Figura 5.7: Parte inferior del procesador GT200…………………..……………………26
Figura 5.8: Parte superior del procesador GT200. 26
Figura 5.9: Sistema de refrigeración de los modelos GeForce GTX200. 27
Figura 6.1: CUDA es diseñado para soportar varios lenguajes de programación. 29
Figura 6.2: Grid de thread blocks..33
Figura 6.3: Jerarquía de la Memoria. 34
Figura 6.4: Ejecución de un programa C en el anfitrión y el dispositivo. 35
Figura 6.5: Proceso de compilación CUDA..43
Figura 6.6: Multiplicación de matrices. 45
Figura 7.1: Proceso simplificado de encriptación y desencriptación.............................50
Figura 7.2: Criptografía asimétrica utilizada para cifrar mensajes o para firma digital. .52
Figura 7.3: Funcionamiento de forma abstracta de una función Hash o Message Digest Function. 53
Figura 7.4: Un ataque de fuerza bruta o ataque de búsqueda de claves..........................54
Figura 8.1: Diagrama de Caso de Uso de Alto Nivel (Forma Gráfica). 62
Figura 8.2: Diagrama de Caso de Uso “Realizar Ataque de Fuerza Bruta” (Forma Gráfica). 63
Figura 8.3: Diagrama de Caso de Uso “Mostrar Resultados del Ataque de Fuerza Bruta” (Forma Gráfica). 64
Figura 8.4: Diagrama de Actividad de la aplicación..65
Figura 10.1: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta para claves seleccionadas para el archivo textoplano.txt. 74
Figura 10.2: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta para claves aleatorias para el archivo textoplano.txt. 76
Figura 11.1: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con una hebra para claves seleccionadas para el archivo textoplano.txt. 80
Figura 11.2: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 10 hebras para claves seleccionadas para el archivo textoplano.txt. 82
Figura 11.3: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 50 hebras para claves seleccionadas para el archivo textoplano.txt. 84
Figura 11.4: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 100 hebras para claves seleccionadas para el archivo textoplano.txt. 86
Figura 11.5: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 200 hebras para claves seleccionadas para el archivo textoplano.txt. 88
Figura 11.6: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con una hebra para claves aleatorias para el archivo textoplano.txt. 90
Figura 11.7: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 10 hebras para claves aleatorias para el archivo textoplano.txt......................92

Figura 11.8: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 50 hebras para claves aleatorias para el archivo textoplano.txt.....................94
Figura 11.9: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 100 hebras para claves aleatorias para el archivo textoplano.txt...................96
Figura 11.10: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 200 hebras para claves aleatorias para el archivo textoplano.txt...................98
Figura C.1: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta para claves seleccionadas para el archivo textoplano2.txt. 114
Figura C.2: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta para claves aleatorias para el archivo textoplano2.txt.......................................115
Figura C.3: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta para claves seleccionadas para el archivo textoplano3.txt. 117
Figura C.4: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta para claves aleatorias para el archivo textoplano3.txt.......................................118
Figura D.1: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con una hebra con claves seleccionadas para el archivo textoplano2.txt...........120
Figura D.2: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 10 hebras para claves seleccionadas para el archivo textoplano2.txt..........121
Figura D.3: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 50 hebras para claves seleccionadas para el archivo textoplano2.txt..........123
Figura D.4: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 100 hebras para claves seleccionadas para el archivo textoplano2.txt........124
Figura D.5: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 200 hebras para claves seleccionadas para el archivo textoplano2.txt........126
Figura D.6: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con una hebra para claves aleatorias para el archivo textoplano2.txt. 127
Figura D.7: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 10 hebras para claves aleatorias para el archivo textoplano2.txt.................129
Figura D.8: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 50 hebras para claves aleatorias para el archivo textoplano2.txt.................130
Figura D.9: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 100 hebras para claves aleatorias para el archivo textoplano2.txt...............132
Figura D.10: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 200 hebras para claves aleatorias para el archivo textoplano2.txt...............133
Figura D.11: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con una hebra para claves seleccionadas para el archivo textoplano3.txt..........135
Figura D.12: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 10 hebras para claves seleccionadas para el archivo textoplano3.txt..........136
Figura D.13: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 50 hebras para claves seleccionadas para el archivo textoplano3.txt..........138
Figura D.14: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 100 hebras para claves seleccionadas para el archivo textoplano3.txt........139
Figura D.15: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 200 hebras para claves seleccionadas para el archivo textoplano3.txt........141
Figura D.16: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con una hebra para claves aleatorias para el archivo textoplano3.txt. 142

Figura D.17: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 10 hebras para claves aleatorias para el archivo textoplano3.txt.................144
Figura D.18: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 50 hebras para claves aleatorias para el archivo textoplano3.txt.................145
Figura D.19: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 100 hebras para claves aleatorias para el archivo textoplano3.txt................147
Figura D.20: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 200 hebras para claves aleatorias para el archivo textoplano3.txt................148

Lista de Tablas

Tabla 5.1: Número de núcleos de procesamiento. 23
Tabla 5.2: Comparación entre CPUs actuales y el procesador GT200. 23
Tabla 5.3: Número máximo de hebras. 24
Tabla 10.1: Tiempos empleados en cada ataque de fuerza bruta para claves seleccionadas para el archivo textoplano.txt. 73
Tabla 10.2: Tiempos empleados en cada ataque de fuerza bruta para claves aleatorias para el archivo textoplano.txt. 75
Tabla 11.1: Tiempos empleados en cada ataque de fuerza bruta con una hebra para claves seleccionadas para el archivo textoplano.txt. 80
Tabla 11.2: Tiempos empleados en cada ataque de fuerza bruta con 10 hebras para claves seleccionadas para el archivo textoplano.txt. 81
Tabla 11.3: Tiempos empleados en cada ataque de fuerza bruta con 50 hebras para claves seleccionadas para el archivo textoplano.txt. 83
Tabla 11.4: Tiempos empleados en cada ataque de fuerza bruta con 100 hebras para claves seleccionadas para el archivo textoplano.txt. 85
Tabla 11.5: Tiempos empleados en cada ataque de fuerza bruta con 200 hebras para claves seleccionadas para el archivo textoplano.txt. 87
Tabla 11.6: Tiempos empleados en cada ataque de fuerza bruta con una hebra para claves aleatorias para el archivo textoplano.txt. 89
Tabla 11.7: Tiempos empleados en cada ataque de fuerza bruta con 10 hebras para claves aleatorias para el archivo textoplano.txt. 91
Tabla 11.8: Tiempos empleados en cada ataque de fuerza bruta con 50 hebras para claves aleatorias para el archivo textoplano.txt. 93
Tabla 11.9: Tiempos empleados en cada ataque de fuerza bruta con 100 hebras para claves aleatorias para el archivo textoplano.txt. 95
Tabla 11.10: Tiempos empleados en cada ataque de fuerza bruta con 200 hebras para claves aleatorias para el archivo textoplano.txt. 97
Tabla C.1: Tiempos empleados en cada ataque de fuerza bruta para claves seleccionadas para el archivo textoplano2.txt. 113
Tabla C.2: Tiempos empleados en cada ataque de fuerza bruta para claves aleatorias para el archivo textoplano2.txt. 115
Tabla C.3: Tiempos empleados en cada ataque de fuerza bruta para claves seleccionadas para el archivo textoplano3.txt. 116
Tabla C.4: Tiempos empleados en cada ataque de fuerza bruta para claves aleatorias para el archivo textoplano3.txt. 118
Tabla D.1: Tiempos empleados en cada ataque de fuerza bruta con una hebra para claves seleccionadas para el archivo textoplano2.txt. 119
Tabla D.2: Tiempos empleados en cada ataque de fuerza bruta con 10 hebras para claves seleccionadas para el archivo textoplano2.txt. 121
Tabla D.3: Tiempos empleados en cada ataque de fuerza bruta con 50 hebras para claves seleccionadas para el archivo textoplano2.txt. 122
Tabla D.4: Tiempos empleados en cada ataque de fuerza bruta con 100 hebras para claves seleccionadas para el archivo textoplano2.txt. 124

Tabla D.5: Tiempos empleados en cada ataque de fuerza bruta con 200 hebras para claves seleccionadas para el archivo textoplano2.txt. 125
Tabla D.6: Tiempos empleados en cada ataque de fuerza bruta con una hebra para claves aleatorias para el archivo textoplano2.txt. 127
Tabla D.7: Tiempos empleados en cada ataque de fuerza bruta con 10 hebras para claves aleatorias para el archivo textoplano2.txt. 128
Tabla D.8: Tiempos empleados en cada ataque de fuerza bruta con 50 hebras para claves aleatorias para el archivo textoplano2.txt. 130
Tabla D.9: Tiempos empleados en cada ataque de fuerza bruta con 100 hebras para claves aleatorias para el archivo textoplano2.txt. 131
Tabla D.10: Tiempos empleados en cada ataque de fuerza bruta con 200 hebras para claves aleatorias para el archivo textoplano2.txt. 133
Tabla D.11: Tiempos empleados en cada ataque de fuerza bruta con una hebra para claves seleccionadas para el archivo textoplano3.txt. 134
Tabla D.12: Tiempos empleados en cada ataque de fuerza bruta con 10 hebras para claves seleccionadas para el archivo textoplano3.txt. 136
Tabla D.13: Tiempos empleados en cada ataque de fuerza bruta con 50 hebras para claves seleccionadas para el archivo textoplano3.txt. 137
Tabla D.14: Tiempos empleados en cada ataque de fuerza bruta con 100 hebras para claves seleccionadas para el archivo textoplano3.txt. 139
Tabla D.15: Tiempos empleados en cada ataque de fuerza bruta con 200 hebras para claves seleccionadas para el archivo textoplano3.txt. 140
Tabla D.16: Tiempos empleados en cada ataque de fuerza bruta con una hebra para claves aleatorias para el archivo textoplano3.txt. 142
Tabla D.17: Tiempos empleados en cada ataque de fuerza bruta con 10 hebras para claves aleatorias para el archivo textoplano3.txt. 143
Tabla D.18: Tiempos empleados en cada ataque de fuerza bruta con 50 hebras para claves aleatorias para el archivo textoplano3.txt. 145
Tabla D.19: Tiempos empleados en cada ataque de fuerza bruta con 100 hebras para claves aleatorias para el archivo textoplano3.txt. 146
Tabla D.20: Tiempos empleados en cada ataque de fuerza bruta con 200 hebras para claves aleatorias para el archivo textoplano3.txt. 148

Resumen

Las tarjetas gráficas a lo largo de su historia han sido diseñadas para procesar imágenes y videojuegos con una mayor calidad y fluidez, pero desde hace algunos años estas se han utilizado para otros fines, desarrollando aplicaciones de propósito general sobre ellas para aprovechar su alta capacidad de cómputo. La aparición del entorno de programación CUDA ha permitido que los desarrolladores de aplicaciones puedan resolver problemas de toda índole con la tecnología de las tarjetas gráficas. Debido a esto es que el presente proyecto busca incluir esta tecnología en pruebas, que se realizan sobre algoritmos criptográficos para validar su fortaleza. Estas pruebas, conocidas como ataques de fuerza bruta, son definidas claramente antes de realizarse, en beneficio de determinar si las tarjetas gráficas o los procesadores de computador son más eficientes para la ejecución de estas tareas, finalizando con los resultados que decidirán que dispositivo es el idóneo para realizar este tipo de pruebas.

Palabras Claves: Tarjetas Gráficas, GPU, CUDA, Algoritmos Criptográficos, Pruebas, Ataque de Fuerza Bruta.

Abstract

Throughout history, the graphics cards have been designed for processing images and video games with higher qualities and fluidity. However, this graphics cards have been used for other goals some years ago, developing a general purpose applications in order to take advantage of their high computing capacity. The CUDA programming environment has allowed developers of the applications to solve all kind of graphics cards technology problems. Due to this situation, the current project includes technology in testing about algorithms to validate its strength. These tests known as brute force attack are clearly defined before being performed to determine if the graphic cards or computer processors are more efficient to execute these tasks. The results will allow a decision to be made on the appropiate device used for this kind of testing.

Keywords: Graphics Cards, GPU, CUDA, Cryptographic Algorithms, Tests, Brute Force Attack.

1 Presentación del Tema

1.1 Introducción

Las tarjetas gráficas, históricamente han evolucionado en su arquitectura y poder de cómputo para llevar a cabo la tarea de procesar imágenes y videojuegos de una manera más fluida y con un mayor realismo, pero desde hace algún tiempo lo fabricantes de estas se han desviado de ese propósito para replantearse la utilización y el enfoque de esta tecnología, y ampliar el espectro de uso de las tarjetas gráficas. Por esta razón las tarjetas gráficas dejan de ser simplemente un dispositivo hardware que se emplea en computación gráfica, y pasan a ser componentes que pueden ser utilizados para ejecutar cálculos complejos provenientes de aplicaciones que trabajan sobre ellas. Es así como estas se vuelven más programables y permiten que los desarrolladores de aplicaciones de diferentes áreas puedan emplearlas para codificar algoritmos que las utilicen para llevar a cabo cálculos de gran envergadura debido a su gran potencia y velocidad de cómputo.

El nuevo enfoque de las tarjetas gráficas, ha impulsado el desarrollo de diferentes lenguajes y entornos de programación que faciliten al desarrollador de aplicaciones la codificación de algoritmos que puedan funcionar sobre ellas. Es así como en el presente existe un entorno de programación llamado CUDA, el cual posee variadas características, dentro de las más destacables se encuentran su propio modelo de programación y la capacidad de utilizar el lenguaje de programación C como un lenguaje de alto nivel añadiéndole un conjunto de extensiones para que pueda soportar la programación de las tarjetas gráficas.

Herramientas como CUDA permiten resolver problemas de toda índole, es por esto que en el presente proyecto la inclusión de la tecnología de las tarjetas gráficas busca resolver un problema específico en el área de la ciencia de la criptografía. Este problema se relaciona con pruebas puntuales, llamadas ataques de fuerza bruta sobre algoritmos criptográficos, las cuales toman largos períodos de tiempo en desarrollarse y a la vez consumen una gran cantidad de recursos, por ende resulta atractivo incorporar una tecnología de estas características para buscar reducir el consumo de tiempo y de recursos en la ejecución de este tipo de pruebas.

Como en todo proyecto, es importante ir conociendo poco a poco las aristas de este. Es por esto que en el primer capítulo se identifica el origen del tema y la definición del problema a resolver, a grandes rasgos. En el segundo capítulo se hace un resumen de la historia de las tarjetas gráficas y como estas han evolucionado en el tiempo. El tercer capítulo presenta los lenguajes de programación para tarjetas gráficas clasificados como GPGPU, y para concluir el estado del arte, en el cuarto capítulo se enumeran algunos proyectos exitosos desarrollados con la tecnología.

Luego de este conjunto de capítulos que presentan la base histórica y evolutiva de las tarjetas gráficas, se presentan dos secciones claves: el capítulo número cinco se sumerge en la arquitectura lógica y física de las tarjetas gráficas, y a continuación, el sexto capítulo expone todos aquellos puntos importantes de la programación de estas. Por otro lado en el presente proyecto, se encuentra la criptografía y el criptoanálisis, los que son abordados en el capítulo

siete, para entender a cabalidad en que área del conocimiento será puesta a prueba la tecnología de las tarjetas gráficas.

Los capítulos ocho y nueve, son concisos y puntuales. En ellos se propone un diseño a la solución del problema, utilizando diagramas UML, y se determinan la configuración de las pruebas con sus respectivos supuestos y elementos que utilizarán.

En la parte final del proyecto, se encuentran los capítulos que presentan los resultados de las pruebas realizadas. En el capítulo número diez, se exponen los resultados de las pruebas en el procesador, mientras que en el capítulo número once, se entregan los resultados de las pruebas en la tarjeta gráfica. En ambas secciones, los resultados son presentados en forma de tablas y gráficos, para que el lector pueda tener un entendimiento sencillo de estos. Para finalizar, se presentan las conclusiones que son obtenidas principalmente de los dos capítulos anteriormente citados.

Al finalizar, se espera que el lector haya recorrido los puntos más importantes de la tecnología de las tarjetas gráficas, ya sea a un nivel del estado del arte, como también de la tecnología misma, de su programación y arquitectura, y como está puede ser aplicada a la ciencia de la criptografía u a otras áreas del conocimiento que presenten problemas que pueden ser resueltos de mejor forma con la inclusión de la tecnología de las tarjetas gráficas.

1.2 Origen del Tema

El proyecto "Uso de Procesadores de Tarjetas Gráficas para Validación de Fortalezas de Algoritmos Criptográficos", nace del interés del alumno por estudiar una tecnología que fue concebida para otros propósitos, y que hoy en día ha extendido su uso a otras áreas, más allá de la computación gráfica. Lo anterior obedece a la idea de adquirir nuevos conocimientos y a la vez aplicarlos en el área de seguridad informática, en particular en la ciencia de la criptografía. Si bien el estudio del funcionamiento y programación de las tarjetas gráficas podría ser aplicado a cualquier proyecto, el alumno ha elegido el área antes mencionada debido a que esta representa una gran motivación para él y para su formación como profesional.

1.3 Definición del Problema

Los avances de la tecnología en el área de producción de hardware han evolucionado a pasos agigantados en los últimos años. Es común encontrar en cualquier tienda que se dedique a la venta de hardware, diferentes componentes de un alto poder de procesamiento a un bajo precio para un computador de escritorio, es por esto que en cualquier hogar se puede encontrar un computador que en su interior contenga piezas que muchas veces no son aprovechadas a su máxima capacidad por parte del usuario, pero aun así estos exigen cada día un aumento en el poder del hardware. Debido a las razones anteriores es que componentes como el procesador y la tarjeta gráfica han aumentado su performance para satisfacer las altas demandas de procesamiento de datos y gráficos, respectivamente.

Las tarjetas gráficas han mejorado su arquitectura y capacidad de procesamiento por variadas razones, pero la razón principal tiene relación con los videojuegos y los exigentes consumidores de tarjetas gráficas que siempre quieren un poco mas de rendimiento. A partir de este aumento de rendimiento, surgen otras interrogantes sobre estos dispositivos hardware, que llevan a querer extender su uso hacia otras áreas. Debido a lo anterior, es que nace la necesidad de poder programar las tarjetas para utilizarlas en aplicaciones de propósito general, por lo tanto se necesitan entornos que provean al desarrollador de herramientas sencillas de ocupar y lenguajes de una baja curva de aprendizaje. En los tiempos presentes estos entornos y lenguajes existen, debido a que los fabricantes de tarjetas gráficas han tratado de acercar a los desarrolladores de aplicaciones a la tecnología, entregándoles diferentes herramientas que hacen que las tarjetas se puedan utilizar para realizar procesamientos de datos de gran envergadura en aplicaciones de cualquier tipo. Dentro de estas soluciones se encuentra CUDA que pertenece a la prestigiosa empresa NVIDIA, el cual es un entorno que permite un sencillo manejo de la programación de las tarjetas gráficas y es el que se destaca en el presente por ser una de las herramientas más avanzadas para desarrollar aplicaciones.

La problemática ahora viene por el lado de los investigadores y desarrolladores, estos deben pensar cómo aprovechar esta tecnología para resolver problemas de cualquier índole, y como optimizar las aplicaciones que funcionan en un procesador, para que funcionen en una tarjeta gráfica obteniendo mejores resultados de tiempo y procesamiento de datos que en una CPU. En particular para este proyecto, esa problemática se enfoca en estudiar la tecnología y ver cómo esta puede adaptarse a la ciencia de la criptografía.

1.4 Objetivos

1.4.1 Objetivo general

Estudiar la tecnología de las tarjetas gráficas para aplicarla en los procesos de validación de seguridad de algoritmos criptográficos con el propósito de optimizar pruebas que por su complejidad resultan de un mayor consumo de capacidad de procesamiento y por ende de un elevado tiempo de ejecución.

1.4.2 Objetivos específicos

- Estudiar el funcionamiento y programación de las tarjetas gráficas NVIDIA.

- Aplicar el estudio del funcionamiento y programación de las tarjetas gráficas NVIDIA a un determinado algoritmo que permita realizar un ataque de fuerza bruta a un algoritmo criptográfico previamente seleccionado.

- Implementar o reutilizar un algoritmo de ataque de fuerza bruta que funciona en el procesador para que este pueda ser implementado mediante CUDA, y funcione correctamente en una tarjeta gráfica.

- Documentar el tiempo de procesamiento de una tarjeta gráfica versus un procesador al realizar la misma tarea.

- Verificar que los algoritmos criptográficos se mantienen sin problemas de seguridad al utilizar un mayor poder de procesamiento en un ataque de fuerza bruta.

1.5 Alcances o Ámbitos del Proyecto

El proyecto se concentra principalmente en el estudio de la tecnología de las tarjetas gráficas, debido a lo anterior se investiga el material que provee el fabricante de las tarjetas gráficas para conocer y estudiar su funcionamiento, y sus detalles arquitectónicos, para luego pasar a investigar las herramientas que permiten la programación de estas. La mayor parte del tiempo de desarrollo del proyecto se abocará a lo anteriormente expuesto.

Al transcurrir el proyecto, se espera que el nivel de entendimiento del alumno sobre la tecnología sea avanzado como para poder ser aplicado a un problema en específico, para el caso del proyecto, este vendría a ser el desarrollo de un ataque en específico (ataque de fuerza bruta) a ciertos algoritmos criptográficos previamente seleccionados. Se debe dejar en claro que en ningún punto del proyecto se pretende hacer un estudio de criptoanálisis de algún algoritmo criptográfico, ni menos estudiar algún algoritmo a fondo para determinar cuáles son sus fortalezas, ya que esto corresponde a una investigación que escapa del alcance del proyecto.

Finalmente se espera que se puedan implementar sin inconvenientes los algoritmos en las tarjetas gráficas para poder llevar a cabo el ataque de fuerza bruta y puedan ser documentados sus resultados, tanto en la ejecución de este en el procesador como en la tarjeta gráfica.

1.6 Metodología y Plan de Trabajo

Como metodología y plan de trabajo se pretende seguir los siguientes pasos:

- Definición del campo de estudio y problema. Se pretende definir cuál es el problema que se intenta resolver y cuáles son los límites de este.

- Determinación de los conceptos involucrados en el problema a resolver.

- Búsqueda e investigación del material referente al proyecto, basándose en los conceptos determinados en el punto anterior.

- Desarrollo del estado del arte, con la información recolectada se construye una base de conocimientos para desarrollar el proyecto.

- Investigar y estudiar a fondo la tecnología de las tarjetas gráficas, abarcando su arquitectura, funcionamiento y programación.

- Investigar dos aristas importantes, los cuales abarcan los algoritmos criptográficos y sus posibles ataques (uno de ellos el de fuerza bruta), y por otro lado la implementación de este.

- Determinar el algoritmo criptográfico que resulte atractivo para el proyecto, el que posteriormente será puesto a prueba.

- Implementación del algoritmo de ataque de fuerza bruta o búsqueda de código fuente que pueda ser reutilizado para el proyecto, y que presenten las características que permitan su implementación en la tarjeta gráfica respectiva.

- Pruebas sobre el algoritmo criptográfico seleccionado, las cuales se llevarán a cabo en el procesador y la tarjeta gráfica para realizar comparaciones.

- Documentar y analizar los resultados provenientes de las pruebas realizadas.

- Conclusiones sobre la tecnología de las tarjetas gráficas, las cuales son presentadas como el resultado del proyecto.

2 Evolución de las Tarjetas Gráficas (GPU)

Las tarjetas gráficas a través de los años han evolucionado en muchos aspectos, estas se han vuelto más poderosas y eficientes para procesar las imágenes provenientes de los videojuegos u otras aplicaciones, como también han cambiado en su componente arquitectónico, utilizando GPUs (Graphics Processing Unit - Unidad de Procesamiento Gráfico) de tamaños más pequeños, mayor capacidad de memoria RAM dedicada en la propia placa de la tarjeta gráfica, y sistemas de enfriamiento más complejos y aparatosos los cuales permiten mantener al procesador gráfico y las memorias RAM a bajas temperaturas de trabajo para un funcionamiento óptimo. Además han adquirido un grupo de características más ricas en lo referente a su programación, lo que hace que estas sean más fáciles de manipular a la hora de desarrollar aplicaciones que utilicen la GPU. Desde este punto en adelante se utilizarán las palabras tarjetas gráficas y GPU indistintamente para referirse al mismo dispositivo.

2.1 Historia

2.1.1 Primera Generación de GPUs

La primera generación de tarjetas gráficas modernas que podían desplegar imágenes en 2D y 3D comienza a finales de 1998 y principios de 1999, en donde aparecen varios modelos de diferentes compañías los cuales son: NVIDIA TNT y TNT2, ATI Rage y 3DFX Voodoo3, entre otras. Previo a la aparición de esta última la compañía 3DFX comercializaba los modelos Voodoo y Voodoo2, pero estas no clasifican como tarjetas gráficas 2D/3D ya que sólo permitían desplegar imágenes en 3D y necesitaban de una tarjeta extra para mostrar imágenes en 2D lo cual encarecía su adquisición y uso. Las tarjetas gráficas de esta generación, como la NVIDIA TNT2 era capaz de mover 9 millones de triángulos por segundo y contaba con 32 MB en video frente a la 3DFX Voodoo3 que era capaz de mover 8 millones de triángulos por segundo y contaba con 16 MB en video. La cantidad de transistores que se podían encontrar en las tarjetas de esta época era alrededor de 10 millones [1], [2].

2.1.2 Segunda Generación de GPUs

La segunda generación de tarjetas gráficas modernas aparece a finales del año 1999 y principios del año 2000. Dentro de las tarjetas que pertenecen a esta generación se pueden encontrar los modelos NVIDIA GeForce256 y GeForce2, ATI Radeon 7500 y S3 Savage3D, entre otras. En esta época NVIDIA adquirió a 3DFX lo cual le permitió comenzar a dominar el mercado, esto trajo consigo que NVIDIA se enfocara a la creación de nuevas y mejores tarjetas gráficas, y es en este momento cuando comienza a denominarles GPUs (Graphics Processing Unit - Unidad de Procesamiento Gráfico) las cuales tienen como función principal realizar las operaciones matemáticas lo más rápido posible para desplegar las imágenes que se requieran. Una mejora considerable es la liberación de una función que se realizaba antes en la CPU, llamada T&L (Transformation and Lighting), debido a que esta es una operación que se efectuaba recurrentemente lo cual permite que la CPU se encuentre más libre de carga para realizar otros tipos de cálculos. Otro cambio relevante ocurre en la forma de conectar las

GPUs a la placa madre, ya que hasta entonces utilizaban el slot PCI (Peripheral Component Interconnect) pero este comenzó a verse limitado en ancho de banda y velocidad, es por esto que Intel desarrolla un nuevo slot llamado AGP (Accelerated Graphics Port - Puerto de Gráficos Acelerado) el cual fue diseñado para el uso exclusivo de las GPUs, con este avance se solucionó los cuellos de botella entre el procesador y la GPU. En esta generación, las tarjetas gráficas contenían aproximadamente 25 millones de transistores [1], [2].

2.1.3 Tercera Generación de GPUs

La tercera generación de tarjetas gráficas llega en el año 2001. Entre los modelos más relevantes de la época se encuentran la NVIDIA GeForce3 y la ATI Radeon 4500, entre otras. Ya a estas alturas las tarjetas gráficas eran más complejas y permitían un mayor nivel de programación que sus antecesoras, debido a esto es que se comienzan a realizar los primeros intentos de utilizar las GPUs para realizar otros tipos de aplicaciones de propósito general. Las tarjetas gráficas al ser mas programables y más poderosas, requieren a su vez de un mejoramiento y aumento en su memoria RAM, es por esto que los modelos antes mencionados traen consigo memorias tipo DDR que mejoraron la velocidad y capacidad de almacenamiento de las tarjetas gráficas de ese entonces, por ejemplo la NVIDIA GeForce3 venía con 64 MB de memoria. En esta época las tarjetas gráficas tenían cerca de 60 millones de transistores [2], [3].

2.1.4 Cuarta Generación de GPUs

La cuarta generación de tarjetas gráficas aparece en el año 2002 y se extiende hasta el presente. Desde el comienzo de esta generación hasta el día de hoy el mercado ha sido dominado por las empresas NVIDIA y ATI que han desarrollado numerosas tarjetas gráficas. Aparecen en el comienzo de esta época, nuevas GPUs como lo son los modelos NVIDIA GeForce4 y ATI Radeon 9700, siendo esta última un acierto de la compañía ATI que logró entregar a sus consumidores un producto mucho más potente que su competencia y la dejó en el liderazgo de las compañías fabricantes de tarjetas gráficas. Las GPUs en este año tenían cerca de 65 millones de transistores [4].

En el año 2003 aparecen las NVIDIA GeForce FX, en especial el modelo 5950 que compite codo a codo con la ATI Radeon 9800, sucesora de la Radeon 9700. Nuevamente las dos compañías quedan al tope del liderazgo de los fabricantes de tarjetas gráficas luchando constantemente por tomar el primer lugar. En este año las GPUs estaban compuestas de alrededor de 125 millones de transistores.

El año 2004 muestra ya dos series de tarjetas algo desgastados, por lo tanto se necesitan modelos nuevos para los demandantes consumidores de GPUs, es por eso que aparecen los modelos NVIDIA GeForce 6800 (parte de la serie GeForce 6 de la compañía) y ATI X800. Las GPUs de este año tenían cerca de 220 millones de transistores. Un cambio importante comienza a ocurrir, el slot AGP lentamente es relevado por un nuevo slot que aparece con las nuevas placas madres, llamado PCI-E (PCI-Express) que al igual que el AGP era un slot dedicado exclusivamente a dar soporte a las tarjetas gráficas. Otra novedad de estos años fue la aparición de la tecnología SLI de NVIDIA que permitía, a las placas madres que tuvieran

dos slots PCI-E, conectar dos GPUs en paralelo para aumentar el poder de procesamiento en aplicaciones de alta demanda, como los videojuegos. ATI también lanza una tecnología similar llamada Crossfire [4].

El año 2005 entrega nuevos modelos de tarjetas gráficas, como lo son todos los modelos de la serie GeForce 7 de NVIDIA, una de ellas la GeForce 7800, y la serie Radeon X1000 de ATI, la cual albergaba la Radeon X1800. Ambas series lanzaron modelos de tarjetas gráficas que utilizaban el slot AGP como el slot PCI-E para no dejar de lado a los consumidores que querían actualizar sus GPUs pero que no disponían del reciente slot PCI-E en sus placas madres. En este año las GPUs contenían alrededor de 300 millones de transistores.

Los años 2006 y 2007 traen consigo la aparición de nuevas series de tarjetas gráficas, por el lado de NVIDIA aparece la serie GeForce 8, con su modelo top de la línea la GeForce 8800 Ultra, mientras que ATI lanza la Radeon X1900 que se sumaba a la ya conocida serie Radeon X1000. Posterior a la salida de esta última, aparece la serie Radeon HD 2000 que viene a luchar directamente con la nueva serie de la competencia, pero ATI no se duerme y vuelve a lanzar otra serie en estos años, la Radeon HD 3000 que incorpora entre sus filas la primera tarjeta gráfica, de esta compañía, que está compuesta por dos procesadores gráficos, la Radeon HD 3870X2. NVIDIA al lanzar la serie GeForce 8 cambia su arquitectura revolucionando el mercado, ahora se trata de una GPU con una arquitectura unificada la cual posee una unidad de sombreado que permite asignar de forma dinámica los recursos necesarios para el procesamiento de operaciones de geometría, física o sombreado de vértices y píxeles, lo que lleva a esta serie a proporcionar el doble de rendimiento en videojuegos que las series anteriores. Por estos años ya el slot PCI-E se había vuelto un estándar dejando atrás el vetusto slot AGP, sobrepasándolo y llegando a tener el doble de su velocidad. Las GPUs por estos años tenían alrededor de 680 millones de transistores [1], [2], [3].

En el año 2008 aparecen nuevos modelos de tarjetas gráficas, NVIDIA lanza la serie GeForce 9 y ATI la serie Radeon HD 4000, a la cual pertenece la Radeon HD 4870X2 que tal como su predecesora, la Radeon HD 3870X2, tiene dos procesadores gráficos la cual la hizo merecedora del título de la GPU más rápida del mercado por un largo tiempo. Para competir con la tarjeta gráfica de 2 procesadores de ATI, NVIDIA entrega a sus consumidores la GeForce 9800GX2, pero esta sólo logra superar la potencia de la ATI Radeon HD 3870X2. Las GPUs en este año contienen alrededor de 755 millones de transistores [1].

A fines del año 2008 y principios del año 2009, NVIDIA lanza al mercado la serie GeForce GTX200, una serie basada en la arquitectura de las series GeForce 8 y 9 pero con mejoras significativas. Las primeras en aparecer fueron la GeForce GTX260 y GeForce GTX280, las cuales dieron el primer golpe en el mercado debido a su potencia y tamaño. A continuación aparece la GeForce GTX285 que se posiciona como la tarjeta de un solo procesador gráfico más rápida del mercado, junto a esta aparece la GeForce GTX295 que posee dos procesadores gráficos y que lleva a NVIDIA a sobrepasar el poder de cómputo de la ATI Radeon HD 4870X2 que tenía el liderazgo en cuanto a velocidad en el segmento de las tarjetas de doble procesador gráfico, con esto NVIDIA se pone nuevamente a la cabeza del mercado con una serie que sobrepasa a su competencia en todas las pruebas de rendimiento. Las tarjetas gráficas GeForce GTX200 tienen la capacidad de ser configuradas en triple SLI, es decir tres GPUs conectadas en paralelo, como también traen soporte para la tecnología

CUDA (Compute Unified Device Architecture) que es un entorno basado en el lenguaje de programación C el cual permite desarrollar aplicaciones para resolver problemas computacionales complejos aprovechando la capacidad de procesamiento de las GPUs. Ambas características nombradas anteriormente se hacían presentes en la serie GeForce 9 pero es con las GTX200 que se clarifica el horizonte, en especial en el tema referente a la programación de las GPUs con CUDA, lo cual ha revolucionado el mercado ya que no solamente se adquieren tarjetas gráficas para jugar en un computador, sino también para hacer pesados cálculos en diferentes áreas de la salud, matemáticas, etc. En el presente las GPUs están compuestas de 1400 millones de transistores, lo cual da cuenta del tamaño y complejidad de las tarjetas gráficas en la actualidad [1], [2], [3], [4].

3 Lenguajes GPGPU

El desarrollo de aplicaciones que utilicen la GPU para hacer cálculos con otros propósitos, diferentes a los cómputos de primitivas gráficas, es conocido como GPGPU o General-Purpose computation on Graphics Processing Units. Bajo este concepto se encuentran ciertos lenguajes de programación que están orientados a facilitar el desarrollo de aplicaciones para los programadores, haciendo que estos se enfoquen en los algoritmos y no en la implementación de estos, ocupando operaciones comunes simplificadas y sin necesidad de utilizar lenguajes de programación, que clásicamente, están destinados a la computación gráfica como lo son OpenGL, Cg o HLSL. En esta sección se presentan algunos de los lenguajes de programación más importantes que se encuentran bajo el paraguas del concepto GPGPU.

3.1 Brook para GPUs

Brook fue desarrollado por la Universidad de Stanford inicialmente como un lenguaje de programación para "Streaming Processors", tal como el Stanford's Merrimac Streaming Supercomputer y el procesador Imagine. Brook fue adaptado posteriormente a las capacidades del hardware gráfico, siendo así el primer lenguaje de propósito general para las GPUs. Los elementos claves en Brook son el Stream, el cual es una colección de registros que requieren de un cálculo similar, y los Kernels, que son funciones aplicadas a cada elemento de un Stream. Para clarificar un Stream es similar a un arreglo de datos, con la diferencia que un Stream no tiene un índice y la dependencia de elementos no está permitida, y el Kernel son todas las operaciones que se pueden hacer sobre un Stream que permiten entregar un resultado final [5].

El lenguaje se diseñó bajo los conceptos de Data Parallelism, que permite utilizar mejor los recursos ya que se fragmentan los datos y las operaciones se hacen en paralelo aprovechando mejor los recursos, y Arithmetic Intensity que permite un mejor uso del ancho de banda de la memoria, con esto la GPU se mantiene procesando constantemente los datos de entrada evitando tiempos de ocio como ocurre en la CPU muchas veces. Brook no ha sido mantenido desde finales del año 2004, es por esto que al momento de desarrollar aplicaciones con este lenguaje se debe tener claro de antemano todo lo que se utilizará, ya sea el sistema operativo, el modelo y fabricante de la tarjeta gráfica, y la API [5].

3.2 CTM – Close to the Metal

CTM o Close to the Metal es una plataforma desarrollada por ATI. Esta plataforma es una DPVM (Data Parallel Virtual Machine) que permite la comunicación directa con las GPUs ATI a lo largo de su API gráfica. Este enfoque impone muchas restricciones, incluyendo la habilidad de leer, modificar y escribir la memoria en un solo programa, acceder directamente a la memoria principal, o entre los formatos emitidos sin explícitamente copiar los datos. CTM es distribuida como una librería que permite la apertura, el uso y el cierre de "conexiones administradas" a una de las tres unidades del hardware gráfico: el Command Processor es programado vía un lenguaje independiente de la arquitectura, el Data-Parallel

Processor es programado vía un conjunto de instrucciones nativas (dependientes de la arquitectura) y el Memory Controller permite el acceso directo a la GPU y a la memoria principal [6].

Como su nombre lo dice, CTM es utilizado para acceder al hardware gráfico a un muy bajo nivel, por lo tanto ha sido diseñado para optimizar las funcionalidades basadas en la GPU, y no para el uso diario [6]. Además, la aplicación es la responsable de todas las dificultades que puedan ocurrir en la programación de la tarjeta gráfica, lo cual genera problemas de muy bajo nivel que pueden ser difíciles de detectar, por consiguiente incrementa la complejidad del desarrollo y los costos.

3.3 CUDA – Compute Unified Device Architecture

El entorno CUDA o Compute Unified Device Architecture, es introducido por NVIDIA a finales del año 2006 y es similar a Brook. Por ejemplo el lenguaje de programación C estándar es extendido para soportar tipos de Streaming y sus correspondientes operaciones. El entorno CUDA genera ejecutables completos, a diferencia de Brook que genera archivos C++ intermedios, y puede ser utilizado como un "entorno unificado" para desarrollar aplicaciones tanto para GPUs como CPUs. Su principal ventaja se sustenta en que se accede directamente al hardware gráfico de NVIDIA por lo tanto soporta características únicas de un desarrollo sobre una GPU [6].

CUDA también incluye librerías para algebra lineal llamada CUBLAS, la cual es una implementación de BLAS (Basic Linear Algebra Subprograms), y para procesamiento de señales digitales llamada CUFFT, la que proviene de la librería FFT (Fast Fourier Transform). Ambas pueden ser utilizadas fuera del lenguaje [7].

La utilización de CUDA ha tenido éxito en su aplicación en numerosos problemas de procesamiento de señales, administración de base de datos, simulación física, etc.

4 Proyectos Desarrollados en CUDA

La aparición de CUDA ha hecho que se incentive el uso de GPUs NVIDIA para realizar cálculos intensivos en aplicaciones de propósito general, es por esto que cada día más y más desarrolladores se interesan en estudiar la tecnología de las tarjetas gráficas y su programación. Ya se han desarrollado una buena cantidad de aplicaciones para las GPUs en diferentes áreas. A continuación se presentan algunas iniciativas y proyectos desarrollados en CUDA, con un énfasis en desarrollos en el área de seguridad informática.

4.1 Folding@home

Folding@home es un proyecto de computación distribuida desarrollado por la Universidad de Stanford, que estudia el plegamiento proteico normal y anormal, la agregación proteica y las enfermedades relacionadas. Utiliza métodos de computación distribuida a gran escala, para simular escalas de tiempo miles a millones de veces mayores que las previamente obtenidas. Esto permite que se pueda simular el plegado proteico por primera vez, y a su vez dirige la investigación al estudio de las enfermedades relacionadas.

El proyecto viene operando desde el 1° de Octubre del año 2000 y ya han sido aproximadamente 400.000 computadores los que han participado. El funcionamiento de Folding@home se basa en un programa cliente que el usuario descarga de la página oficial y que instala en su computador, convirtiéndolo en una máquina que desarrolla un procesamiento distribuido vía Internet ayudando a entregar resultados para cumplir los objetivos del proyecto. Estos programas cliente funcionan bajo varias plataformas, es decir hay clientes para diferentes sistemas operativos, los cuales a su vez se diversifican utilizando diferentes tipos de hardware. Existen clientes para Microsoft Windows, Linux y MacOSX, como también clientes de alta performance que pueden trabajar en alguno de los tres sistemas operativos pero utilizando GPUs de los fabricantes NVIDIA y ATI, o CPUs de más de un núcleo, incluso en esta categoría existe un cliente para la consola de videojuegos PlayStation 3 [8].

Folding@home postula que al agregarse un nuevo computador al proyecto, este incrementa considerablemente la velocidad de simulación de proteínas debido a los algoritmos que se utilizan, obteniendo así resultados a más corto plazo.

4.2 Solving Dense Linear Systems on Graphics Processors

El título del proyecto se puede traducir como "Solución de Sistemas Lineales Densos sobre Procesadores Gráficos". Este proyecto presenta una serie de métodos que permiten calcular la solución de sistemas lineales utilizando la GPU, haciendo uso de la librería CUBLAS de CUDA.

Principalmente, el proyecto abarca dos métodos para la resolución de sistemas lineales, los cuales son el método de Cholesky y la Factorización LU. Ambos métodos son implementados en la GPU utilizando CUDA y luego son optimizados para obtener una mejor performance. Por ejemplo, una de las optimizaciones aplicadas es llamada "Hybrid

Algorithm" donde se traspasan ciertos cálculos que se llevan a cabo en la GPU a la CPU para explotar las habilidades de este tipo de procesador, dentro de las que se encuentran un eficiente cálculo de matrices pequeñas y un fácil cómputo de raíces cuadradas, dejando a la GPU los cálculos más pesados [9].

Los métodos y su implementación, más las optimizaciones llevadas a cabo, demuestran el mejoramiento en la performance para resolver sistemas lineales y revelan que utilizando una GPU se pueden resolver problemas complejos con mayor velocidad que utilizando un procesador.

4.3 Badaboom Media Converter

El software Badaboom Media Converter es una solución comercial. Permite transformar diferentes formatos de video y audio para que puedan ser reproducidos en diferentes dispositivos. Por ejemplo, la aplicación toma un archivo de video en formato MPEG2 y lo transforma a formato MP4 el cual puede ser reproducido en dispositivos como el IPhone y el IPod.

Badaboom presenta una novedad en la forma de llevar a cabo la tarea de transformación de formatos. Principalmente, la aplicación tiene un funcionamiento diferente que las soluciones de la competencia, ya que utiliza las tarjetas gráficas NVIDIA que soportan CUDA para realizar la conversión de archivos [10], llevando a cabo la tarea en un menor tiempo que las soluciones que utilizan puramente la CPU, con esto entrega un valor agregado al liberar al procesador de una tarea tan pesada como lo es la codificación de video, haciendo que el usuario pueda seguir utilizando el computador para otras tareas como navegar por internet, utilizar las herramientas de ofimática, etc. Por lo tanto, Badaboom Media Converter entrega la capacidad de conversión de archivos de video y audio a una gran velocidad y con una alta calidad en el formato de salida, sin comprometer los recursos del computador.

4.4 A Neural Network on GPU

Una Red Neuronal Artificial (RNA) o Artificial Neural Networks (ANN) es un método de procesamiento de información que está inspirado en cómo funciona los sistemas nerviosos biológicos, como el cerebro, para procesar la información. Se compone de un gran número de elementos de procesamientos altamente interconectados (neuronas) que trabajan al unísono para resolver problemas específicos. Las RNA han sido ampliamente utilizadas en clasificaciones de señales análogas, incluyendo la escritura a mano, voz y reconocimiento de imágenes. También pueden ser usadas en videojuegos, permitiendo a estos aprender adaptativamente el comportamiento del jugador. Esta técnica ha sido empleada en videojuegos de carreras, de manera que los automóviles oponentes controlados por computadoras pueden aprender cómo conducir gracias a los jugadores humanos.

Dado que una RNA requiere de un número considerable de operaciones vectoriales y matriciales para obtener resultados, se ajusta bastante para ser aplicada en un modelo de programación paralela y así poder ejecutarse en GPUs. Otra razón importante para la implementación es que el entrenamiento de la RNA y su ejecución son dos procesos

separados. La red implementada es del tipo Convolutional Neural Network de 5 capas, la cual es especial para reconocer escrituras hechas a mano [11].

La principal ventaja en el uso de una GPU para la implementación de una RNA es que se logra robustez. El resultado entregado es prometedor en comparación con la implementación en una CPU. Los tiempos que se logran en la ejecución de la RNA en una CPU son bastante más extensos que en la GPU. Según los autores del proyecto el tiempo de ejecución en la CPU es de aproximadamente 16 milisegundos, mientras que en la GPU es solo de 0.059 milisegundos, es decir alrededor de 270 veces más rápido. Para entregar resultados más exactos, se debe considerar el tiempo de entrada/salida de la versión implementada en la GPU, aun así es por lo menos 10 veces más rápido que la versión implementada en la CPU [11].

4.5 Gnort: High Performance Network Intrusion Detection Using Graphics Processors

Gnort es un prototipo del conocido Snort, un NIDS (Network Intrusion Detection System) Open-Source ampliamente utilizado. El funcionamiento de un NIDS, a grandes rasgos, consiste en tomar un paquete que entra a la red, revisa su cabecera y lo compara con alguna de las reglas que tiene instauradas de antemano, con esto verifica si el paquete proviene de un lugar confiable o si es un paquete malicioso que puede atentar contra la seguridad de los recursos de la red (información, datos, hardware, etc.). El costo de utilización de la CPU en el proceso de verificación de paquetes es alto (cerca del 75% de los recursos de esta), ya que debe hacer constantemente comparaciones con las reglas implementadas.

La arquitectura del prototipo del proyecto está separada en tres diferentes tareas: la transferencia de los paquetes a la GPU, el proceso de comparación de los paquetes con las reglas en la GPU, y finalmente la transferencia de los resultados a la CPU [12]. Con esta arquitectura se libera de la tarea de reconocimiento de paquetes a la CPU, ya que esta solo recolecta los paquetes y los clasifica para entregárselos a la GPU, permitiendo que se encuentre libre para realizar otras tareas.

La implementación de Gnort fue realizada en una tarjeta gráfica de la serie GeForce 8, utilizando CUDA. En todas las pruebas, con diferentes enfoques y algoritmos los resultados fueron positivos, encontrándose una mejora en la velocidad de salida de los paquetes del doble o a veces el triple de velocidad que las versiones de Snort para CPU. Este solo logró superar a la GPU en las pruebas con paquetes pequeños de no más de 100 bytes [12]. Cabe destacar que Gnort no es un prototipo que funcione enteramente en la GPU, sino que solo realiza el trabajo de reconocimiento de paquetes frente a las reglas, el cual es el trabajo más pesado que lleva a cabo Snort, todas las demás tareas son llevadas a cabo en la CPU, es por esto que esta es una implementación que emplea la GPU como un co-procesador que ayuda al procesador central.

4.6 CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography

El algoritmo criptográfico público AES (Advanced Encryption Standard) fue introducido el año 2001 por el NIST (National Institute of Standards and Technology) en respuesta a las preocupaciones del envejecimiento de DES, el cual ya había sido vulnerado en un par de ocasiones permitiendo que nuevos "atacantes" se aprovecharan de las falencias para obtener beneficios de los sistemas donde estaba implementado este. AES está basado en el algoritmo Rijndael desarrollado por dos criptógrafos belgas, Joan Daemen y Vincent Rijmen, y acepta un tamaño de bloque de 128 bits, y largo de claves de 128, 192 o 256 bits. El largo de clave determina el número de rondas que el algoritmo llevará a cabo; con 128 bits se ejecutan 10 rondas, con 192 bits se ejecutan 12 rondas y con 256 se ejecutan 14 rondas. Cada ronda es una secuencia de 4 escenarios: *AddRoundKey, SubBytes, ShiftRows, MixColumns*.

La implementación de AES en este proyecto tiene dos diferentes enfoques. Ambos no se concentran en el cálculo de las claves ya que esta tarea es limitada y no explota el verdadero poder de procesamiento de las tarjetas gráficas, es por esto que esa tarea se desarrolla en la CPU dejando a la GPU las operaciones más complejas y de mayor cómputo. El primer enfoque está basado en OpenGL, pero este presenta problemas sobre todo para realizar cálculos (como la función XOR) ya que sigue ocupando la CPU para algunos cómputos. El segundo enfoque está basado en CUDA, el cual presenta beneficios desde un comienzo al entregar un modelo de programación más libre para el desarrollador permitiendo asignar mejor los recursos de la tarjeta gráfica, ocupando una GPU de 32 bits lo que permite que AES funcione de forma optimizada ya que fue diseñado para procesadores de ese largo de palabra, y altas velocidades de salida al ejecutar la implementación [13].

La performance del algoritmo criptográfico se vio mejorada al ser implementada en una GPU, ya que comparada con una implementación clásica en una CPU, esta presenta un ahorro de tiempo cercano a 5 veces menos que el tiempo utilizado en un enfoque clásico. Por ejemplo con un archivo de 8 MB la GPU se demora cerca de 28 ms. en realizar el proceso total de cifrado, es decir, tomar los datos y la inicialización de claves desde el procesador, realizar la encriptación y devolver los datos al procesador. Por su parte la CPU se demora cerca de 148 ms. en realizar el proceso total. Todo lo anterior fue documentado poniendo a prueba un algoritmo AES-256 (los tiempos con un AES-128 fueron similares) [13]. Una observación importante tiene que ver con el tamaño de los archivos, ya que cada vez que se aumentaban, la GPU tenía un mejor rendimiento en el tiempo de encriptado/desencriptado y una mayor velocidad de salida. Además por primera vez la GPU se encarga de encriptar y desencriptar los datos de entrada sin tener que utilizar a la CPU, a diferencia de proyectos anteriores relacionados con este trabajo.

4.7 ElcomSoft Distributed Password Recovery

La empresa Elcomsoft fue fundada en el año 1990 en Moscú, Rusia. Siempre relacionada con el área de seguridad informática, se ha convertido en líder en soluciones de recuperación de claves de sistemas y análisis forenses. Dentro de sus prestigiosos clientes se encuentra IBM, Cisco, Adobe, Sony Entertainment, AT&T, entre muchos más.

Elcomsoft ha desarrollado una aplicación comercial, llamada Distributed Password Recovery, que permite recuperar contraseñas de variados sistemas, como por ejemplo sistemas operativos Windows, documentos de Microsoft Office, archivos Adobe PDF, hashes MD5, redes Wireless WPA/WPA2, etc., empleando como ayuda al procesador las tarjetas gráficas NVIDIA para lograr su objetivo. Como muchas aplicaciones que utilizan a la GPU como co-procesador, esta libera de carga a la CPU al realizar las operaciones de cómputo más pesadas en la tarjeta gráfica.

La aplicación se basa en un ataque de fuerza bruta (Brute Force Attack) para obtener las contraseñas que el usuario desee, por otro lado necesita que varios computadores conectados en una red trabajen en conjunto para llevar a cabo la tarea de recuperación, de ahí proviene su nombre. Puede realizar varios tipos de búsqueda las cuales se pueden personalizar con diferentes opciones que provee el software, como por ejemplo: determinación del largo máximo de la clave, eliminación de símbolos dispensables, utilización de solo minúsculas/mayúsculas, etc., además permite realizar una lista con tareas la cual puede ser abortada por el administrador en cualquier momento. Dentro de los beneficios más claros se encuentran [14]:

- Amplio rango de aplicaciones que soporta.
- Escalabilidad. Puede ser usado en una red de cualquier tamaño.
- Carga mínima de la red. Los datos son enviados comprimidos con esto se minimiza la carga de la red.
- Los “Agentes” (estaciones de trabajo) pueden ajustar las tareas a gusto.
- Utiliza todos los computadores disponibles. Incluso los computadores más débiles pueden ser empleados debido a la capacidad de configuración de tareas.
- Trabaja con múltiples documentos. Cualquier número de documentos puede ser listado para una búsqueda de contraseña.
- Se puede elegir el orden de procesamiento de los documentos.

Gracias a todos estos beneficios y la forma de trabajo que tiene esta aplicación, es que logra tiempos asombrosos de recuperación de contraseñas. Por ejemplo, al intentar recuperar un hash MD5, un procesador Intel Core 2 Q6600 el cual tiene 4 núcleos, logra probar 70 millones de contraseñas por segundo, versus una tarjeta gráfica NVIDIA GTX295 que logra

probar 920 millones de contraseñas por segundo, y con GPUs aun más poderosas la cantidad de contraseñas por segundo aumenta a números mayores (hasta 2000 millones de contraseñas por segundo con un sistema NVIDIA Tesla S1070) [14].

La licencia del software se puede adquirir por US$ 599 para 20 clientes, es decir puede ser instalada en veinte estaciones de trabajo de una red sin tener que pagar ni un dólar más. Los valores aumentan en la medida que crece la cantidad de clientes a los cuales se les quiera instalar la aplicación, llegando a pagar US$ 4,999 para 2500 clientes.

5 Tarjetas Gráficas: Arquitectura y Funcionamiento

Los capítulos anteriores presentaban los temas relacionados con las tarjetas gráficas para dar una base teórica actualizada al trabajo a desarrollar. Desde este punto en adelante, la investigación se concentra en temas concretos referentes al proyecto, para dar un paso más hacia la solución del problema tratado en el mismo.

El presente capítulo busca exponer la arquitectura y funcionamiento de las tarjetas gráficas, presentando y explicando sus componentes claves para comprender más a fondo la tecnología.

5.1 Arquitectura Lógica y Funcionamiento del Procesador de la Tarjeta Gráfica

A través de los años, NVIDIA ha lanzado al mercado diferentes modelos de GPUs. Estos modelos se agrupan en familias las cuales se denominan *Series*. La actual serie de la compañía NVIDIA se llama GeForce GTX200 (o simplemente 200), y está compuesta por los modelos GTS250, GTX260, GTX275, GTX280, GTX285 y GTX295. Los modelos de una serie tienen algo en común, su procesador. A través de los diferentes modelos, el procesador cambia en características que se traducen en un mayor o menor rendimiento y desempeño, por ende, basándose en la serie actual, la tarjeta gráfica GTS250 es la que tiene el menor rendimiento de la serie, mientras que la GTX295 es la que tiene el mayor rendimiento, esta ley se cumple en todas las series de tarjetas gráficas de NVIDIA. Para cada familia de tarjetas gráficas existe una nomenclatura para denominar al procesador. En la serie GeForce GTX200, su procesador tiene el nombre de GT200 o GT200b que es una mejora del primero, esta ley también se cumple para las series anteriores, por ejemplo, la serie GeForce 8 tiene el procesador G80 y la serie 9 el procesador G92.

La arquitectura de la serie actual puede funcionar de dos formas, la primera en el modo *Graphics Processing Architecture (Arquitectura de Procesamiento Gráfico)* la que es conocida hasta el día de hoy, y se utiliza para el procesamiento de imágenes, figuras, etc. en videojuegos y aplicaciones gráficas. Por otro lado, se encuentra el modo *Parallel Computing Architecture (Arquitectura de Cómputo Paralelo)* el cual está destinado al procesamiento bruto de datos, similar a lo que realiza una CPU en la actualidad.

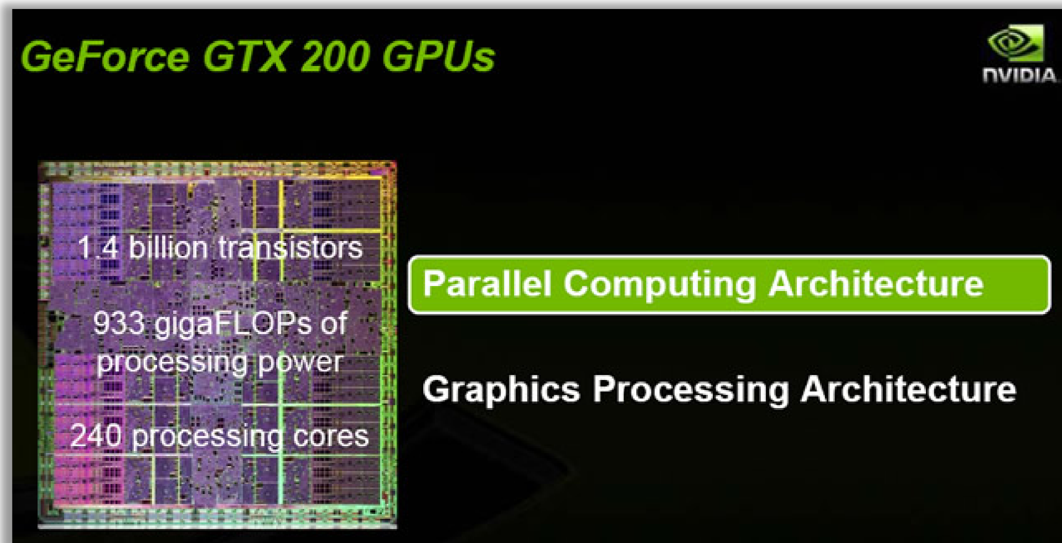

Figura 5.1: Características del procesador GT200 y sus modos de funcionamiento.

El procesador GT200 cuenta con 1400 millones de transistores y 240 núcleos de procesamiento (análogo a lo que se conoce como multi-núcleos en las CPUs modernas), doblando las características del procesador G92 que posee 754 millones de transistores y 128 núcleos de procesamiento. Estas características le dan al GT200 una capacidad de procesamiento de 933 GigaFLOPS (Giga Floating point Operations Per Second – Giga Operaciones en punto Flotante Por Segundo), siendo el procesador de tarjetas gráficas más grande, complejo y poderoso fabricado hasta el momento por la compañía NVIDIA [15].

5.1.1 Parallel Computing Architecture

Las GPUs de la serie GTX200 en el modo Parallel Computing Architecture permiten realizar cálculo paralelo, es decir pueden llevar a cabo cómputo intensivo en aplicaciones de propósito general, similar a lo que hace una CPU en un computador de escritorio, notebook o servidor. Algunas de las tareas que se pueden realizar en este modo son: computación distribuida (como es el caso de Folding@home), cálculo de físicas, aceleración de vídeo HD, aceleración de procesos científicos, etc.

La estructura del procesador de la tarjeta gráfica es similar para ambos modos de trabajo, y se denomina *SPA (Scalable Processor Array)*. Esta estructura tiene 10 bloques llamados *Thread Processing Clusters (TPCs)* y en su parte inferior se encuentra el sistema de memoria principal, que puede variar según el modelo de tarjeta gráfica de la serie, pudiendo tener desde 512 MB hasta 1792 MB. Por ejemplo, para el modelo NVIDIA GTX285, se tiene 16 bloques de memoria de 64 MB cada uno, sumando un total de 1024 MB (1 GB). Entre los TPCs y el sistema de memoria principal se encuentran las unidades llamadas *Tex L2*, las cuales son una caché de textura que se utilizan para combinar accesos a memoria más eficientes y un mayor ancho de banda de memoria en operaciones de lectura/escritura. Acompañando a estas se encuentran las unidades especiales llamadas *Atomic*, las cuales tienen la habilidad de llevar a cabo operaciones atómicas de lectura, modificación y escritura, por otro lado permiten tener un acceso granular a las localidades de la memoria principal, y facilitan las reducciones

paralelas y la administración de estructuras de datos paralelas. En la parte superior de la estructura se encuentra el *Thread Scheduler,* el que se encarga de administrar las hebras de ejecución de las aplicaciones a través de los TPCs [15], [16].

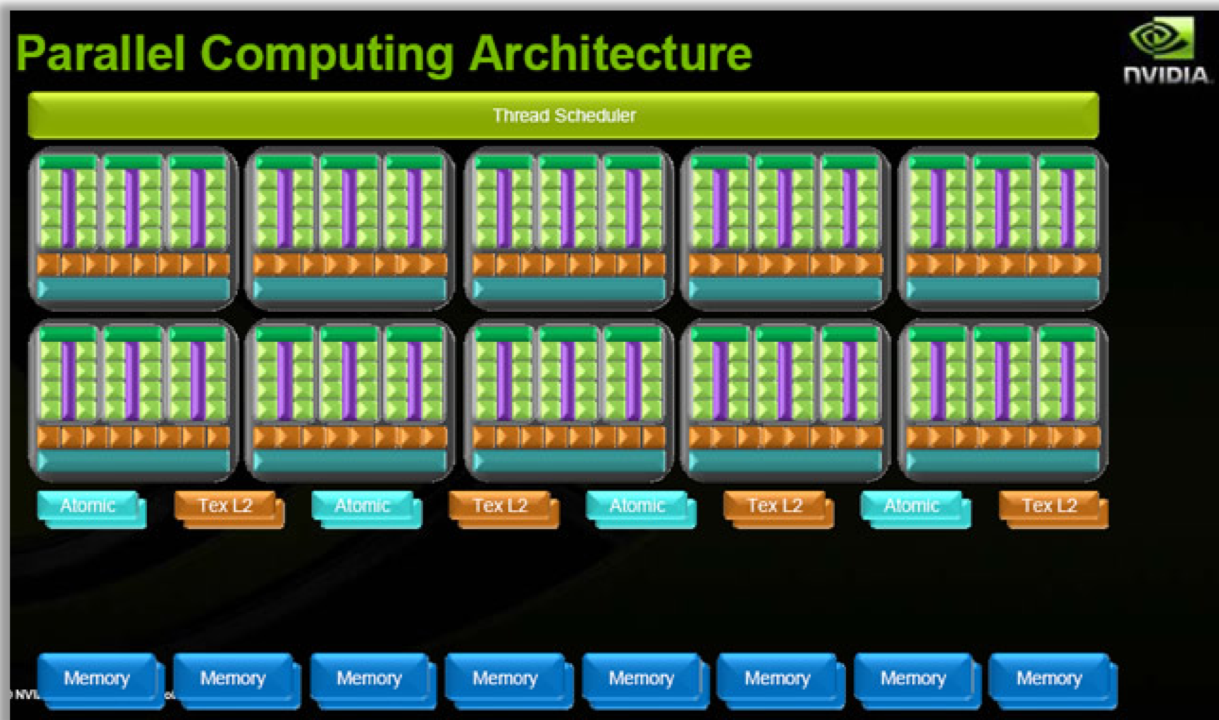

Figura 5.2: Estructura en el modo Parallel Computing Architecture.

5.1.2 Graphics Processing Architecture

La arquitectura para este modo permite el funcionamiento clásico de la tarjeta gráfica, es decir, el procesador de esta se utiliza para la generación de figuras y aceleración gráfica en videojuegos. La estructura es similar a la que se encuentra en el modo Parallel Computing Architecture, pero la diferencia está en que en este modo de funcionamiento se deben agregar unas unidades llamadas *ROPs (Raster Operations Processors – Procesadores de Operaciones de Barrido)* que se componen de 8 clusters que incluyen 16 unidades cada uno, sumando un total de 128 ROPs, los cuales se encuentran entre las unidades Tex L2 y la memoria principal de la tarjeta gráfica, por otro lado desaparecen las unidades Atomic en esta forma de funcionamiento. En la parte superior de la estructura, se encuentran las unidades de *Vertex Shader, Píxel Shader y Setup/Raster*, más la unidad *Geometry Shader*, las que conforman la arquitectura unificada de segunda generación de NVIDIA, la cual es una mejora de la arquitectura unificada que se encuentra en las GPUs de las series GeForce 8 y 9. Un detalle importante es la nomenclatura de los bloques superiores de la estructura, en el modo Parallel Computing Architecture eran llamados Thread Processing Clusters, en cambio en este modo se denominan *Texture Processing Clusters*. Si bien para ambos su configuración es la misma y utilizan el mismo acrónimo (TPCs), en cada modalidad de funcionamiento se refieren a nombres diferentes [15], [16].

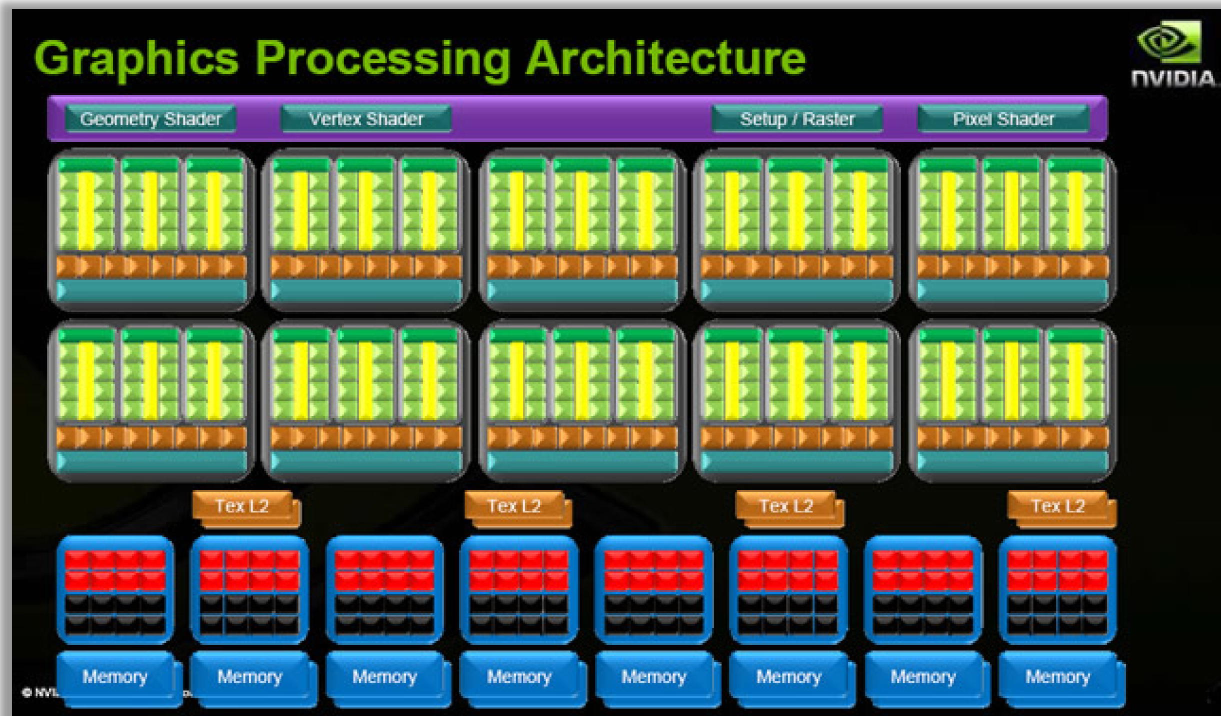

Figura 5.3: Estructura en el modo Graphics Processing Architecture.

5.1.3 Thread Processing Cluster – Texture Processing Cluster (TPC)

El componente más importante de la estructura es la unidad TPC, el cual está presente en el modo de procesamiento gráfico y en el modo de cómputo paralelo. Dentro de este cluster o unidad, se encuentran tres bloques denominados *Streaming Multiprocessors (SMs),* los que están compuestos de 8 núcleos (cores) conocidos también como *Streaming Processors (SPs)*, en el modo de procesamiento gráfico, o *Thread Processors*, en el modo de cómputo paralelo. Estos núcleos suman un total de 24 por cada TPC, por lo tanto el procesador GT200 tiene 240 núcleos, los cuales pueden ser más o menos en cantidad según el modelo de tarjeta gráfica, similar a lo que sucede con el caso de la memoria principal en una GPU. Por ejemplo, una GPU modelo GeForce GTX260 tiene 216 núcleos, mientras que el modelo GeForce GTX285 posee 240 núcleos [15], [17].

Un TPC, aparte de contener una cierta cantidad de núcleos, posee también 8 bloques de textura llamados *Texture Filtering Processors (TF)*, los que se utilizan en el modo de procesamiento gráfico, pero que a su vez son útiles para varias operaciones en el modo de cómputo paralelo. Dentro de cada SM se encuentra una memoria local compartida (*Local Memory*) de 16k que permite comunicar a los núcleos para que estos puedan compartir datos sin tener que realizar operaciones de lectura y escritura en subsistemas externos de memoria, con esto se mejora la velocidad de cómputo y la eficiencia en variados algoritmos. Para completar el bloque TPC se encuentra una memoria caché llamada *L1 Cache* la que permite la compartición de datos entre cada unidad SM [15], [16].

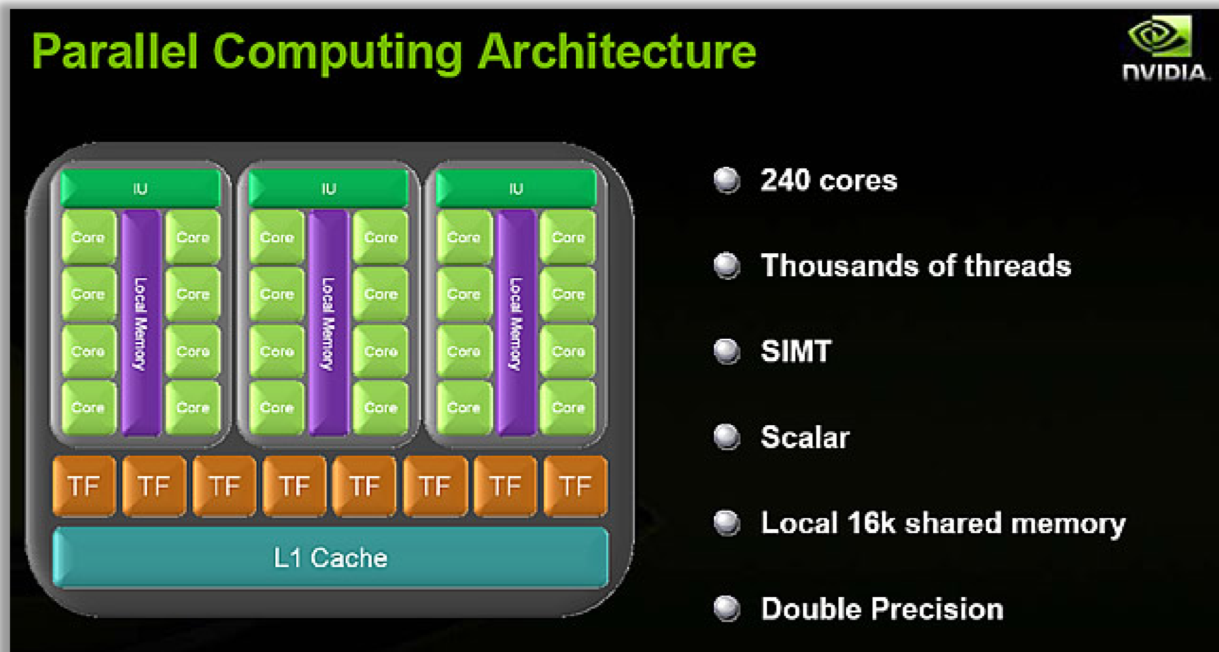

Figura 5.4: Estructura detallada de una unidad TPC.

5.1.4 Sistema e Interfaz de Memoria

La memoria principal que se encuentra en las GPUs de la serie GTX200 es del tipo GDDR3. Por ejemplo, en los modelos GeForce GTX280 y GTX285 se encuentran 8 bloques de memoria por cada lado de la tarjeta gráfica, ubicados en pares, sumando un total de 16 bloques, tal como se muestra en la Figura 5.5. Cada par de bloques de memoria tiene una interfaz de comunicación de 64 bits, entregando un total de 512 bits de interfaz [15], una mejora considerable con respecto a las tarjetas gráficas de las series anteriores que solo tenían una interfaz de memoria de 384 bits. Por otra parte, estas 16 unidades de memoria de 64 MB de capacidad cada una, forman un frame buffer de 1024 MB, cantidad necesaria para responder a los altos requerimientos de aplicaciones gráficas y de cómputo general en la actualidad.

La relación del ancho de banda entre las texturas y el frame buffer (TEX:FB) ha sido mejorada al incorporar una interfaz más amplia, permitiendo soportar de mejor forma la carga de procesamiento de las aplicaciones actuales y futuras. Adicionalmente, el procesador GT200 incluye un componente hardware que permite comprimir los datos que viajan desde y hacia la memoria principal de la tarjeta gráfica, ayudando a ocupar más eficientemente la interfaz ya que permite enviar más datos por unidad de tiempo.

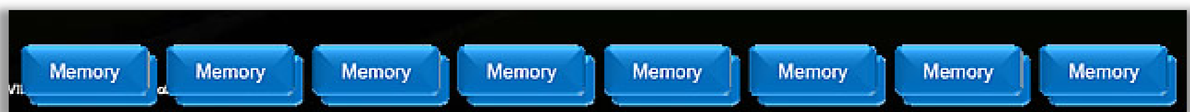

Figura 5.5: Sistema de memoria principal de la tarjeta gráfica.

5.1.5 Arquitectura SIMT

Los modos de funcionamiento utilizan dos modelos diferentes de procesamiento. Para la ejecución a través de los TPCs, la arquitectura es MIMD (Multiple Instruction, Multiple Data) la cual permite obtener paralelismo, ya que para este caso, cada TPC puede ser visto como un procesador el que funciona de forma asíncrona e independiente sobre un conjunto de datos. Para la ejecución a través de cada SM, la arquitectura es SIMT (Single Instruction, Multiple Thread) la cual es una mejora a la arquitectura SIMD (Single Instruction, Multiple Data), permitiendo obtener un mejor rendimiento y una programación más sencilla. Por otra parte SIMT asegura que todos los núcleos de procesamiento estén completamente ocupados todo el tiempo.

Desde el punto de vista del desarrollador de aplicaciones, SIMT permite que cada hebra (thread) tome su propio camino, puesto que los desvíos (branching) son manejados por el hardware, sin necesidad de ser manejados manualmente [15].

5.1.6 Mayor Número de Núcleos de Procesamiento

La nueva arquitectura SPA de segunda generación del procesador GT200 mejora el desempeño comparada con su generación previa, la que se encuentra en los procesadores G80 y G92. Sus mejoras en el diseño se pueden apreciar en dos niveles: primero, incrementa el número de SMs por TPC de 2 a 3, y segundo, incrementa el número máximo de TPCs por chip (procesador) de 8 a 10. Estas mejoras entregan una cantidad de 240 núcleos de procesamiento, lo que se traduce en una mayor capacidad de procesamiento.

Procesador	TPCs	SMs por TPC	SPs por SM	Total de SPs
Series GeForce 8 (G80) & 9 (G92)	8	2	8	128
Serie GeForce GTX200 (GT200)	10	3	8	240

Tabla 5.1: Número de núcleos de procesamiento.

A continuación, en la Tabla 5.2, se presenta una comparación sencilla entre la cantidad de núcleos de procesamiento y GFLOPS del chip GT200, y las CPUs actuales que se pueden encontrar en un computador de escritorio.

	Core 2 Duo E8400 (Dual Core)	Core 2 Extreme 9650 (Quad Core)	Serie GeForce GTX200 (Multi-Core)
Número de Núcleos de Procesamiento	2	4	240
Número de GFLOPS	48	96	933

Tabla 5.2: Comparación entre CPUs actuales y el procesador GT200.

5.1.7 Gran Número de Hebras

Las GPUs de la serie GeForce GTX200 soportan sobre 30 mil hebras de ejecución. El Thread Scheduler se encarga de mantener a todos los núcleos de procesamiento ocupados casi al 100%. La arquitectura del procesador de la tarjeta gráfica es tolerante a la latencia, por ejemplo, si una hebra en particular está esperando por un acceso a la memoria, este inmediatamente cambia a otra hebra para procesarlo, de esta forma se mantiene una actividad constante sin tener que sacrificar un núcleo por una tarea o hebra en espera.

La unidad SIMT dentro de un SM crea, administra, organiza y ejecuta las hebras en grupos de 32 hebras paralelos llamados *Warps*[15]. Un SM soporta hasta 32 Warps, versus 24 Warps soportados por un SM en los procesadores de las series GeForce 8 y 9, por lo tanto cada SM puede manejar un máximo de 1024 (32*32) hebras concurrentes, soportando un total de 30720 hebras concurrentes si se tienen 30 SMs. Los valores se detallan en la Tabla 5.3, que se presenta a continuación.

Procesador	TPCs	SMs por TPC	Hebras por SM	Hebras Totales Por Procesador
Series GeForce 8 (G80) & 9 (G92)	**8**	**2**	**768**	**12288**
Serie GeForce GTX200 (GT200)	**10**	**3**	**1024**	**30720**

Tabla 5.3: Número máximo de hebras.

5.1.8 Soporte para Doble Precisión

Una característica muy importante del procesador GT200 es su doble precisión, que soporta cálculo de Punto Flotante (Floating Point) de 64-bit, lo que beneficia a aplicaciones de cómputo científicas, ingenieriles y financieras, o cualquier tarea de cálculo que necesite de una exactitud alta en sus resultados. A pesar de que los procesadores G80 y G92 soportan este tipo de cálculo, solo llegan a la precisión simple de 32-bit de Punto Flotante [15], [16].

Los bloques SM incorporan una unidad matemática de doble precisión de Punto Flotante de 64-bit, para un total de 30 núcleos de procesamiento de doble precisión de 64-bit.

5.2 Arquitectura Física de la Tarjeta Gráfica

La arquitectura física es un componente importante, que determina en algunos casos, un rendimiento inferior o superior de la tarjeta gráfica. Las partes que constituyen la GPU permiten que el procesador de esta, funcione a más o menos temperatura, con un mayor o menor nivel de ruido, que ocupe un mayor espacio dentro del computador, o que consuma más o menos energía eléctrica.

En la técnica del *overclocking*, la que se emplea para aumentar las frecuencias de reloj del procesador y las memorias de la tarjeta gráfica para obtener un incremento de potencia en

esta, los componentes arquitectónicos son decisivos para mantener estables los aumentos de ciclos de reloj sin acortar drásticamente la vida útil del hardware debido a las altas temperaturas, producto de una exigencia mayor de la tarjeta gráfica. Esta técnica es ampliamente utilizada por los entusiastas de los videojuegos, y no solo se lleva a cabo a nivel de GPUs, sino también de otros tipos de hardware, como lo son las placas madres, memorias RAM, etc.

Figura 5.6: Tarjeta gráfica modelo NVIDIA GeForce GTX285.

Las características de los elementos constituyentes de la arquitectura pueden variar según el fabricante de la tarjeta gráfica. NVIDIA produce los diferentes modelos de GPUs como referencia y las comercializa entre los diferentes fabricantes de hardware gráfico, los cuales transforman estos modelos, entregándoles cada uno su sello propio a las diferentes características físicas de la tarjeta gráfica, pero nunca cambiando el procesador por otro ya que este representa el cerebro de la GPU. Por ejemplo, algunas compañías entregan a sus consumidores modelos de tarjetas gráficas con disipadores de calor mejorados en comparación con los modelos de referencia entregados por NVIDIA, o comercializan modelos con overclocking de fábrica, etc. Entre las empresas fabricantes más importantes de tarjetas gráficas se encuentran Palit Microsystems Ltd., EVGA Corporation, ASUSTek Computer Inc., GIGABYTE Technology Company Ltd., etc.

A continuación se presentan algunas características claves de la arquitectura física de la tarjeta gráfica.

5.2.1 Procesador de la Tarjeta Gráfica

El procesador GT200, como se ha revisado hasta este punto, es el más reciente fabricado por NVIDIA. Este chip tiene un tamaño de aproximadamente 470 mm^2 debido a su proceso de manufactura de 55 nm (nanómetros) llevado a cabo por TSMC (Taiwán Semiconductor Manufacturing Corp.). El resultado de emplear un proceso de manufactura de un menor número de nanómetros se traduce en la obtención de un producto que tiene mejores capacidades térmicas (emite menos calor) y de consumo energético, por ende es más escalable en frecuencias de reloj y mejora su rendimiento productivo, todo esto en un chip de menores dimensiones. Por otro lado, con este tipo de fabricación, es posible que un chip incluya una mayor cantidad de transistores, es por esto que el procesador GT200 tiene 1400 millones de

transistores, además por su parte inferior posee 2485 pines [18], tal como se muestra en la Figura 5.7. Debido a las características expuestas anteriormente es que este procesador puede alcanzar velocidades de hasta aproximadamente 648 MHz, en los modelos top de la serie.

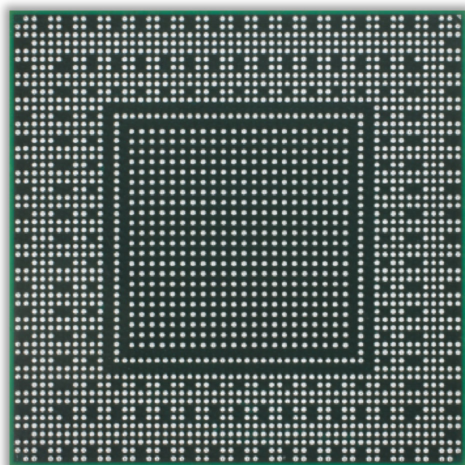

Figura 5.7: Parte inferior del procesador GT200.

Figura 5.8: Parte superior del procesador GT200.

5.2.2 Memorias RAM

Las memorias RAM son otro componente clave en la arquitectura física de la GPU, por ende se necesita que estas sean lo más rápidas posible para permitir un traspaso fluido de los datos desde y hacia el procesador GT200. Los modelos de la serie GeForce GTX200 vienen equipados con memorias del tipo GDDR3, las cuales son especialmente diseñadas para las tarjetas gráficas y son ampliamente utilizadas por los fabricantes de estas, es por esto que en el mercado actual todas las memorias de las GPUs son GDDR3, o incluso GDDR5 en algunos modelos de otros fabricantes. La cantidad de memoria principal de estas tarjetas gráficas varía entre 512 MB y 1792 MB según el modelo que se desee adquirir, y las velocidades de estas se mueven en un rango de 999 MHz a 1242 MHz, con posibilidades de overclock tal como el chip gráfico.

5.2.3 Sistema de Refrigeración

El sistema de refrigeración es importantísimo a la hora de exigir un funcionamiento elevado de la tarjeta gráfica, este permite mantener al procesador gráfico y las memorias RAM a bajas temperaturas en las etapas de carga de esta, y a su vez, proporciona al usuario un mayor margen de overclocking dependiendo de la calidad y complejidad del sistema de refrigeración. Hoy en día existen dos tipos de refrigeración que pueden ser adquiridos en el mercado, el primero es la clásica refrigeración constituida por *Heatpipes* (tuberías) mas radiadores que en su conjunto son llamados disipadores, y uno o más ventiladores, y la segunda es la solución de enfriamiento por agua, siendo esta más compleja y más costosa que la primera, pero a su vez más efectiva.

Los fabricantes muchas veces transforman este componente por variadas razones, por ejemplo, son mejorados para obtener un mayor margen de aumento de frecuencias en la

práctica del overclocking, disminuir el ruido del ventilador en la fase de carga de la tarjeta gráfica (que es cuando más rápido gira el ventilador), etc.

Arquitectónicamente, el sistema de enfriamiento está constituido por un disipador de gran tamaño, hecho de una combinación de materiales como cobre y aluminio para conducir el calor hacia el exterior de la tarjeta gráfica, y por uno o más ventiladores de un material plástico resistente que funciona a más o menos revoluciones según la carga que tenga en un determinado momento la GPU, y que permiten disminuir y eliminar el calor proveniente desde el disipador. Los modelos de la serie GeForce GTX200 son bastante grandes y aparatosos debido a su refrigeración que debe cubrir toda la placa de la tarjeta gráfica, por ende ocupan un espacio considerable dentro del gabinete que contiene los componentes hardware del computador. Otro detalle importante, es que el disipador no toca directamente el chip, entre ellos existe una pasta térmica que permite una mejor conducción del calor desde el procesador hacia el disipador, para su posterior disminución y eliminación. En la Figura 5.9 se muestra un sistema de refrigeración clásico que se puede encontrar en cualquier modelo de la serie GTX200, ya que todas estas tarjetas son muy parecidas visualmente debido a que la refrigeración es casi idéntica en cada una de ellas.

Figura 5.9: Sistema de refrigeración de los modelos GeForce GTX200.

5.2.4 Administración de Consumo Energético y Temperatura

Las GPUs GeForce GTX200 incluyen una arquitectura para administrar el consumo energético más flexible y dinámica que las pasadas generaciones de tarjetas gráficas de NVIDIA. Según datos de la compañía, estas GPUs consumen cerca de 25 W cuando se encuentran en reposo (modo idle), es decir cuando simplemente despliegan imágenes por el monitor y no hacen uso intensivo de los recursos de estas, y aproximadamente 236 W cuando están con una carga total desplegando imágenes en 3D. Estos valores, en algunos estudios empíricos realizados por entusiastas del overclocking y los videojuegos, aumentan casi hasta llegar a los 200 W, en modo idle, y a los 435 W, en modo de carga total, debido a que los modelos de mayor potencia o con overclocking de fábrica tienen mayores frecuencias de reloj en el chip gráfico y en las memorias, es por esto que se debe poseer una fuente de poder de a lo menos 600 W para darle energía a estas tarjetas gráficas de gran envergadura y al resto de

componentes hardware de un computador [16], [18]. Otra característica que es destacable es que ciertos sectores son “apagados” cuando la tarjeta gráfica no está en su carga máxima, lo que permite un menor consumo de energía eléctrica.

Cabe destacar, que la aplicación que se desea implementar en la tarjeta gráfica utiliza intensivamente esta, por lo tanto se espera tener una alta velocidad de procesamiento, por ende, la aplicación trabajará por un largo periodo de tiempo en modo de carga total de la GPU, es por esto que se necesita, tal como se expuso anteriormente, de una fuente de poder de a lo menos 600 W para que los componentes de este tipo de tarjeta gráfica y el resto de hardware del computador no sufran bajas en la energía eléctrica que puedan disminuir su desempeño o hacer que este simplemente se apague.

Las temperaturas son un tema delicado, ya que a mayor temperatura, los componentes pueden tener una vida útil más corta que lo que propone el fabricante de la tarjeta gráfica. Según experimentos, en reposo estas GPU emiten cerca de 45° C y en carga total aproximadamente 85° C [16], [18]. Estos valores pueden variar un poco debido a la complejidad del sistema de refrigeración que incluya la tarjeta gráfica y también del grado de overclocking a que se someta.

6 Tarjetas Gráficas: Programación

Las tarjetas gráficas NVIDIA poseen su propia plataforma de programación llamada CUDA (Compute Unified Device Architecture), la que se presentó brevemente en el Capítulo 3 (Sección 3.3). Este entorno presenta características propias que permiten al desarrollador de aplicaciones explotar las cualidades presentes en las GPUs, centrando la mirada en el diseño de los algoritmos por sobre la implementación de estos, por ende la tarea de programación se facilita en gran medida debido a que la codificación de los algoritmos pasa a ser un trabajo que puede ser desarrollado por cualquier programador que conozca la plataforma a fondo.

La idea principal detrás de CUDA es permitir que los desarrolladores ocupen C como un lenguaje de programación de alto nivel, esperando en el futuro la inclusión de otros lenguajes de programación como C++, FORTRAN, OpenCL, etc., posibilitando con esto mantener una baja curva de aprendizaje ya que se utilizan lenguajes de programación que son ampliamente conocidos por los desarrolladores de software. Por otra parte el entorno, a través de un pequeño conjunto de extensiones del lenguaje de programación, permite un alto nivel de paralelismo de datos y de hebras, lo que se traduce en que los problemas pueden ser resueltos de forma más eficiente ya que estos pueden ser divididos en sub-problemas menores que pueden ser ejecutados independientemente gracias a la arquitectura propia de las tarjetas gráficas, obteniendo un ahorro de tiempo en las tareas.

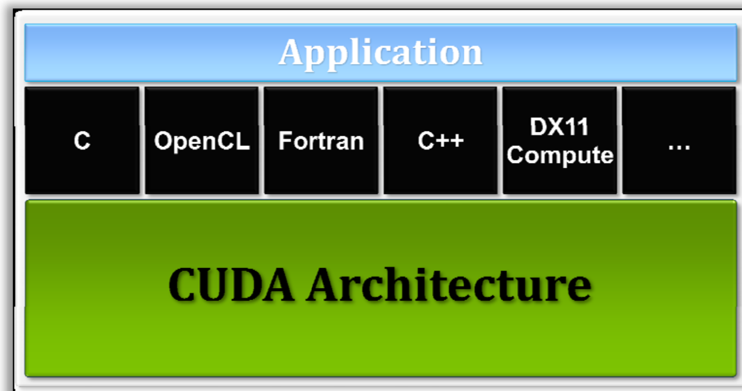

Figura 6.1: CUDA es diseñado para soportar varios lenguajes de programación.

El principal objetivo de este capítulo es presentar las características que posee el entorno CUDA, dentro de las que se encuentran su modelo de programación y su lenguaje de programación, que permitirán comprender como se compone esta plataforma, y que beneficios entrega a los programadores y sus algoritmos. Cabe destacar, que en el desarrollo del ataque de fuerza bruta no se utilizarán todas las características que entregan CUDA y las tarjetas gráficas, como sucede en todo desarrollo de una aplicación, pero de todas formas es importante presentarlas para tener un entendimiento global de la tecnología que posteriormente será de ayuda para aterrizar el conocimiento a la aplicación específica del presente proyecto.

6.1 Modelo de Programación

Esta sección introduce los conceptos principales del modelo de programación que posee CUDA. Los detalles del lenguaje de programación C para CUDA se revisan en la sección 6.2.

6.1.1 Kernels

C para CUDA extiende el lenguaje de programación C para permitir al desarrollador definir funciones, llamadas *Kernels*, que cuando son llamadas se ejecutan N veces en paralelo por N diferentes hebras (*threads*) CUDA, opuesto a lo que ocurre con las funciones regulares de C [19].

Un Kernel es definido usando la declaración específica *__global__* y el número de hebras para cada llamada es especificado usando la nueva sintaxis <<<…>>> [19]. A continuación se presenta un breve ejemplo del código fuente para una función Kernel:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
}

int main()
{
    // Kernel invocation
    VecAdd<<<1, N>>>(A, B, C);
}
```

A cada una de las hebras que ejecuta un Kernel se le asigna un único ID, llamado *thread ID*, que es accesible dentro del Kernel a través de una variable que ha sido incorporada, llamada *threadIdx* [19]. A modo de ejemplo, el siguiente código fuente suma dos vectores A y B de tamaño N y los almacena en un vector C. Cada una de las hebras que ejecutan VecAdd realiza una suma en parejas de posiciones de los vectores.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Kernel invocation
    VecAdd<<<1, N>>>(A, B, C);
}
```

6.1.2 Thread Hierarchy – Jerarquía de Hebras

Para una mayor conveniencia, *threadIdx* es un vector de 3 componentes, debido a esto las hebras se pueden identificar usando un *thread index* de una dimensión, dos dimensiones o tres dimensiones, formando a su vez un *thread block* de una dimensión, dos dimensiones o tres dimensiones. Un *thread block* es un grupo de Warps que son ejecutados juntos y que pueden compartir memoria en un único multiprocesador [19]. Estas características proveen de una forma natural para invocar cálculos a través de elementos en un dominio como un vector, una matriz o un campo. A continuación se presenta un ejemplo de código fuente donde se suman dos matrices A y B de dimensiones NxN y se almacenan los resultados en una matriz C.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],float C[N][N])
{
   int i = threadIdx.x;
   int j = threadIdx.y;
   C[i][j] = A[i][j] + B[i][j];
}

int main()
{
   // Kernel invocation
   dim3 dimBlock(N, N);
   MatAdd<<<1, dimBlock>>>(A, B, C);
}
```

El índice de una hebra y su thread ID se relacionan entre si de una manera sencilla:

- Para un bloque unidimensional, son las mismas.
- Para un bloque bidimensional de tamaño (*Dx, Dy*), el thread ID de una hebra de índice (*x, y*) es (*x + y*Dx*).
- Para un bloque tridimensional de tamaño (*Dx, Dy, Dz*), el thread ID de una hebra de índice (*x, y, z*) es (*x + y*Dx + z*Dx*Dy*).

Las hebras dentro de un bloque pueden cooperar entre sí mediante el intercambio de datos a través de la memoria compartida y sincronizar su ejecución para coordinar accesos a memoria. Mas precisamente, se pueden especificar los puntos de sincronización en el Kernel llamando a la función intrínseca *__syncthreads( )*. Esta función sincroniza todas las hebras de un bloque, por ende es usada para coordinar la comunicación entre estos. Una vez que todas las hebras alcancen este punto, la ejecución se reanuda con normalidad [19].

Para una cooperación eficiente, la memoria compartida es de baja latencia y se encuentra cerca de los núcleos de procesamiento (como la memoria L1 Cache), la función *__syncthreads( )* debe ser ligera y todas las hebras de un bloque se espera que residan en el

mismo núcleo de procesamiento. El número de hebras por bloque es, por lo tanto, restringido por los recursos limitados de memoria de un núcleo de procesamiento. En las GPUs actuales, un thread block puede contener hasta 512 hebras.

Sin embargo, un Kernel puede ser ejecutado por múltiples thread blocks de igual forma, de modo que el número total de hebras es igual al número de hebras por bloque multiplicado por el número de bloques. Estos múltiples bloques están organizados en una *grid* (red) de thread blocks de una o dos dimensiones, tal como se ilustra en la Figura6.2. Una grid es simplemente el conjunto de todos los thread blocks que ejecuta un Kernel, y su dimensión es especificada en el primer parámetro de la sintaxis <<<…>>> [19]. Cada bloque dentro de la grid puede ser identificado por un índice unidimensional o bidimensional accesible dentro del Kernel a través de la variable incorporada *blockIdx*. La dimensión del thread block es accesible dentro del Kernel a través de la variable incorporada *blockDim*. Basándose en el ejemplo anterior del código fuente, este puede ser modificado quedando de la siguiente forma:

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    // Kernel invocation
    dim3 dimBlock(16, 16);
    dim3 dimGrid((N + dimBlock.x – 1) / dimBlock.x,
    (N + dimBlock.y – 1) / dimBlock.y);
    MatAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

El tamaño del thread block es 16x16 = 256 hebras, valor elegido arbitrariamente, y una grid es creada con suficientes bloques para tener una hebra por matriz de elementos.

Los thread blocks requieren ejecutarse independientemente. Debe ser posible que se ejecuten en cualquier orden, ya sea en paralelo o en serie. Este requisito de independencia permite que los thread blocks puedan ser organizados en cualquier orden a través de cualquier número de núcleos de procesamiento, permitiendo a los programadores escribir código escalable.

El número de thread blocks en una grid es típicamente dictado por el tamaño de los datos que están siendo procesados antes que por el número de procesadores del sistema.

En resumen, la jerarquía de hebras se puede esquematizar de la siguiente forma sencilla: un Kernel es ejecutado por muchas hebras, los se dividen en bloques de igual tamaño llamados

thread blocks, y estos a su vez se agrupan en una grid. Además, en esta jerarquía, cada hebra tiene un identificador único.

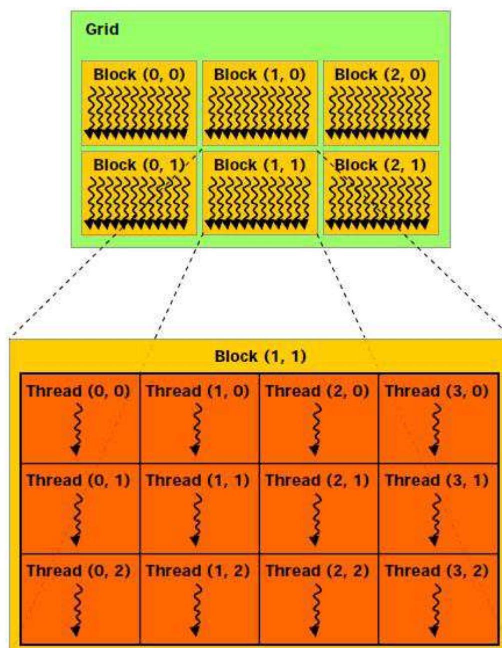

Figura 6.2: Grid de thread blocks.

6.1.3 Memory Hierarchy – Jerarquía de la Memoria

Las hebras CUDA pueden acceder a datos de múltiples espacios de memoria durante su ejecución, tal como se muestra en la Figura 6.3. Cada hebra tiene una memoria local privada. Cada thread block tiene una memoria compartida visible por todas las hebras de un bloque y que tiene una duración de tiempo igual a la vida útil de este. Finalmente, todas las hebras tienen acceso a la misma memoria global.

También existen dos espacios de memoria adicionales de solo lectura, accesibles por todas las hebras: los espacios de memoria constante y de textura. Los espacios de memoria global, constante y de textura son optimizados para diferentes usos. La memoria de textura también ofrece diferentes modos de direccionamiento, como también el filtrado de datos, para algunos formatos específicos de estos.

Los espacios de memoria global, constante y de textura son persistentes a través del Kernel ejecutado por la misma aplicación.

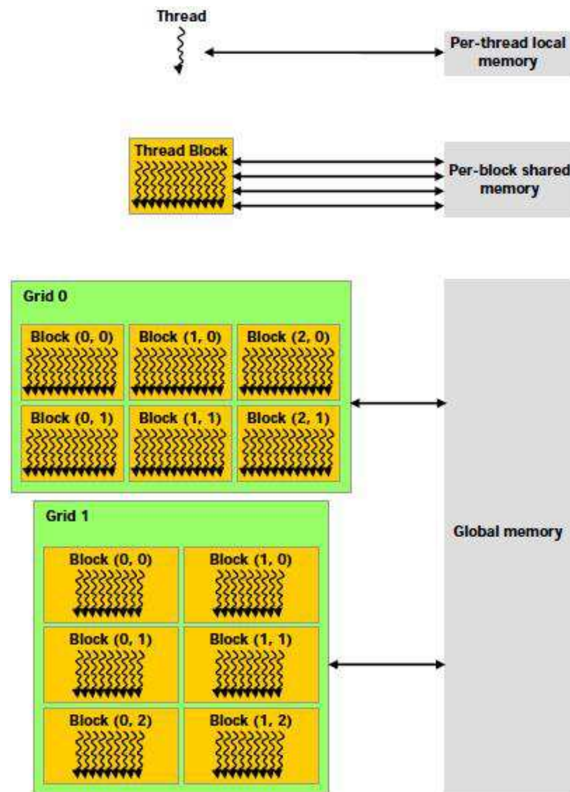

Figura 6.3: Jerarquía de la Memoria.

6.1.4 Host and Device – Anfitrión y Dispositivo

CUDA asume que las hebras pueden ejecutarse en un dispositivo (*device)* físico separado, que opera como un co-procesador para el anfitrión (*host)* que ejecuta el programa escrito en el lenguaje C, esto queda graficado en la Figura 6.4. Para este caso, por ejemplo, los Kernels son ejecutados por la GPU (código paralelo) y el resto del programa C por el CPU (código serial) [19].

El concepto de dispositivo y anfitrión es clave en CUDA, ya que estos especifican cuando una operación es llevada a cabo por la GPU, y cuando es desarrollada por el CPU, por lo tanto, el término dispositivo (*device*) hace referencia a la GPU, y el término anfitrión (*host*) hace referencia al CPU.

CUDA también asume que tanto el anfitrión como el dispositivo poseen y mantienen sus propias memorias RAM, referenciadas como la memoria del anfitrión (*host memory*) y la memoria del dispositivo (*device memory*) [19], respectivamente. Por lo tanto, un programa administra los espacios de memoria global, constante y de textura visibles por los Kernels a través de las llamadas al tiempo de ejecución (*runtime*) de CUDA. Esto incluye la asignación y desasignación de memoria del dispositivo, como también la transferencia de datos entre la memoria del anfitrión y del dispositivo.

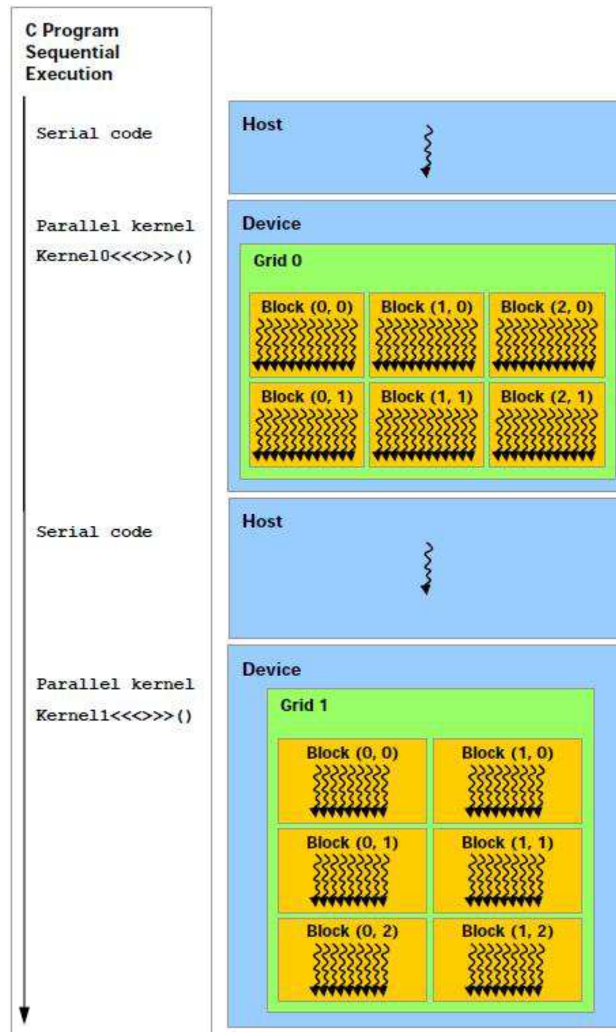

Figura 6.4: Ejecución de un programa C en el anfitrión y el dispositivo.

6.1.5 Capacidad de Cómputo

La capacidad de cómputo de un dispositivo es definida por un número mayor de revisión y un número menor de revisión.

Los dispositivos con el mismo número mayor de revisión son de la misma arquitectura base. En la actualidad todos los dispositivos de NVIDIA tienen capacidad de cómputo 1.x (el número mayor de revisión es 1). El número menor de revisión corresponde a un mejoramiento incremental de la arquitectura base, posiblemente incluyendo nuevas características. El número menor de revisión es el valor ".x" y va desde 0 a 3.

Cada capacidad de cómputo tiene características específicas, y a medida que el número menor de revisión aumenta, estas son incluidas en la capacidad de cómputo que la sigue más nuevas características. Por ejemplo, las tarjetas gráficas de la serie GeForce GTX200 tienen capacidad de cómputo 1.3 (la más alta en la actualidad), por ende, su arquitectura incluye todas las características específicas de las capacidades de cómputo previas 1.0, 1.1 y 1.2, más

la 1.3, esta última tiene como propiedad específica la capacidad de soportar números de punto flotante de doble precisión.

6.2 C para CUDA

La meta de C para CUDA es proveer de un camino simple para que usuarios familiarizados con el lenguaje de programación C puedan escribir programas que puedan ser ejecutados en la tarjeta gráfica, también denominada dispositivo.

Esto consiste de dos puntos clave, los cuales son [19], [20]:

- Un conjunto mínimo de extensiones del lenguaje C que permitan al programador determinar y escribir ciertas porciones del código fuente, para que estas sean llevadas a cabo por la GPU.

- Una librería de "tiempo de ejecución", que se divide en:

    - Un componente que funciona en el anfitrión (CPU), que proporciona funciones de control y de acceso a uno o más dispositivos (GPUs) en el computador desde este.

    - Un componente que funciona en el dispositivo, que proporciona funciones especificas de este.

    - Un componente común, que incorpora tipos de vectores y un subconjunto de la librería C estándar que es soportada tanto por el anfitrión como por el dispositivo.

Cabe señalar que las únicas funciones de la librería C estándar que son soportadas para ser ejecutadas en el dispositivo son aquellas que proporciona el componente común en tiempo de ejecución, por ende, existe un subconjunto de funciones del lenguaje de programación C que se pueden utilizar y ejecutar en el dispositivo sin necesidad de modificar su sintaxis o de agregar nuevas extensiones a las declaraciones de estas, simplificando el aprendizaje de C para CUDA, ya que ciertas funciones que en la actualidad pueden ser desarrolladas por el anfitrión, también pueden ser llevadas a cabo por el dispositivo sin modificaciones.

6.2.1 Extensiones del Lenguaje

Las extensiones del lenguaje de programación C son [19], [20]:

- Tipos de clasificadores de función, para especificar si una función es ejecutada en el anfitrión o en el dispositivo, y si es llamada por el anfitrión o el dispositivo.

- Tipos de clasificadores de variable, para especificar la ubicación de una variable en la memoria del dispositivo.

- Una nueva directiva para especificar cómo es ejecutado un Kernel en el dispositivo desde el anfitrión.

- Cuatro variables incorporadas que especifican las dimensiones de la grid y del thread block, y los índices del thread block y las hebras.

Cada código fuente que contenga estas extensiones, es compilado por el compilador CUDA llamado *nvcc*. Estas extensiones tienen algunas restricciones que se describen en las secciones a continuación. Las restricciones al ser violadas el compilador nvcc emite avisos de error o advertencia, pero algunas veces estas infracciones no pueden ser detectadas específicamente, es decir, señalando el lugar exacto del código donde se encuentra el error, similar a lo que sucede en otros lenguajes de programación y entornos de programación, en donde el desarrollador de la aplicación debe revisar el código fuente completamente para detectar errores de sintaxis o de mala implementación de una función que realiza una operación indebida, por ejemplo, en la memoria.

6.2.1.1 Tipos de Clasificadores de Función

__device__

El clasificador **__device__** declara una función que es: ejecutable en el dispositivo e invocada solo desde este.

__global__

El clasificador **__global__** declara una función como un Kernel. Esta función es: ejecutable en el dispositivo y es invocada solo desde el anfitrión.

__host__

El clasificador **__host__** declara una función que es: ejecutable en el anfitrión e invocada solo desde este.

Esto es equivalente a declarar la función solo con el clasificador **__host__**, o declararla sin ningún clasificador como **__host__**, **__device__** o **__global__**, en cualquiera de los dos casos la función es compilada solo para el anfitrión.

Sin embargo, el clasificador **__host__** puede ser usado en combinación con el clasificador **__device__**, en cuyo caso la función es compilada para el anfitrión y para el dispositivo.

Restricciones

- Las funciones **__device__** y **__global__** no soportan recursividad.

- Las funciones **__device__** y **__global__** no pueden declarar variables estáticas dentro de su cuerpo.

- Las funciones **__device__** y **__global__** no pueden tener un número variable de argumentos.

- Los punteros de función a funciones **__global__** son soportados.

- Los clasificadores **__global__** y **__host__** no pueden ser utilizados juntos.

- Las funciones **__global__** deben tener un tipo de retorno *void*.

- Cualquier llamado a una función **__global__** debe especificar su configuración de ejecución.

- Una llamada a una función **__global__** es asincrónica, lo que significa que regresa antes de que el dispositivo haya finalizado su ejecución.

- Los parámetros de la función **__global__** son comúnmente traspasados al dispositivo vía memoria compartida y se limitan a 256 bytes [19], [20].

6.2.1.2 Tipos de Clasificadores de Variables

__device__

El clasificador **__device__** declara una variable que reside en el dispositivo.

A lo más uno de los otros tipos de clasificadores de variables descritos en esta sección puede ser usado junto con **__device__** para especificar con mayor detalle a que espacio de memoria pertenece la variable. Si ninguno de ellos está presente, la variable:

- Reside en el espacio de memoria global.

- Tiene el tiempo de vida de una aplicación.

- Es accesible desde todas las hebras dentro de una grid, y desde el anfitrión a través de la librería de tiempo de ejecución.

__constant__

El clasificador **__constant__**, opcionalmente utilizado en conjunto con **__device__**, declara una variable que:

- Reside en el espacio de memoria constante.

- Tiene el tiempo de vida de una aplicación.

- Es accesible desde todas las hebras dentro de una grid, y desde el anfitrión a través de la librería de tiempo de ejecución.

__shared__

El clasificador **__shared__**, opcionalmente utilizado en conjunto con **__device__**, declara una variable que:

- Reside en el espacio de memoria compartida de un thread block.
- Tiene el tiempo de vida del bloque.
- Solo es accesible desde todas las hebras dentro de un bloque.

Cuando se declara una variable en la memoria compartida como un arreglo externo, como se muestra a continuación:

```
extern __shared__ float shared[];
```

El tamaño del arreglo es determinado en el tiempo de ejecución. Todas las variables declaradas de esta forma, comienzan con la misma dirección en la memoria, de modo que el diseño de las variables en el arreglo debe ser explícitamente manejado a través de offsets. Por ejemplo, si se necesita la equivalencia de:

```
short array0[128];
float array1[64];
int array2[256];
```

En la memoria compartida asignada dinámicamente, se puede declarar e inicializar los arreglos de la siguiente manera:

```
extern __shared__ char array[];
__device__ void func() // __device__ or __global__ function
{
    short* array0 = (short*)array;
    float* array1 = (float*)&array0[128];
    int* array2   = (int*)&array1[64];
}
```

Restricciones

- Estos clasificadores no son permitidos en miembros de estructuras *struct* o *union*, en parámetros formales y en variables locales dentro de una función que se ejecuta en el anfitrión.
- Las variables **__shared__** y **__constant__** tienen implícitamente un almacenamiento estático.

- Las variables **__device__**, **__shared__** y **__constant__** no pueden ser definidas como externas usando la palabra clave *extern*. La única excepción es para las asignaciones dinámicas de las variables **__shared__**.

- Las variables **__device__** y **__constant__** solo se permiten en el ámbito de los archivos.

- Las variables **__constant__** no pueden ser asignadas desde el dispositivo, solo desde el anfitrión a través de las funciones de tiempo de ejecución del mismo.

- Las variables **__shared__** no pueden tener una inicialización como parte de su declaración [19], [20].

Una variable automática, declarada en el código del dispositivo, sin ninguno de estos clasificadores generalmente reside en un registro. Sin embargo en algunos casos, el compilador puede elegir ponerla en la memoria local, lo cual puede traer consecuencias negativas en el rendimiento.

Los punteros en el código que son ejecutados en el dispositivo son soportados siempre que el compilador sea capaz de resolver si apuntan a cualquier espacio de la memoria compartida o de la memoria global, de lo contrario son restringidos solo a apuntar a la memoria asignada o declarada en el espacio de memoria global.

Las direcciones obtenidas de las variables **__device__**, **__shared__** o **__constant__** pueden solo ser usadas en el código del dispositivo. Las direcciones de una variable **__device__** o **__constant__** obtenidas a través de la función *cudaGetSymbolAddress( )* pueden solo ser usadas por el código del anfitrión.

6.2.1.3 Configuración de la Ejecución

Cualquier llamada a una función **__global__** debe especificar la *configuración de la ejecución* para esa llamada.

La configuración de la ejecución define las dimensiones de la grid y de los bloques que se utilizarán para desarrollar la función en el dispositivo, así como también sus Stream asociados. Esto se especifica insertando una expresión de la forma <<< *Dg, Db, Ns, S* >>> entre el nombre de la función y su lista de argumentos entre paréntesis, donde:

- *Dg* es del tipo *dim3*, y especifica la dimensión y el tamaño de una grid, tal que Dg.x * Dg.y es igual al número de bloques que es lanzado y Dg.z debe ser igual a 1.

- *Db* es del tipo *dim3*, y especifica la dimensión y el tamaño de cada bloque, tal que Db.x * Db.y * Db.z es igual al número de hebras por bloque.

- *Ns* es del tipo *size_t*, y especifica el número de bytes en la memoria compartida que son asignados dinámicamente por bloque para la presente llamada, además de la memoria asignada estáticamente. Esta memoria asignada dinámicamente es usada por cualquiera

de las variables declaradas como un arreglo externo mencionados en el detalle del clasificador **__shared__**. Ns es un argumento opcional, y su valor por defecto es 0.

- *S* es del tipo *cudaStream_t*, y especifica los Stream asociados. S es un argumento opcional, y su valor por defecto es 0 [19], [20].

Por ejemplo, si una función es declarada como:

__global__ void Func(float* parameter);

Debe ser llamada de la siguiente forma:

Func<<< Dg, Db, Ns >>>(parameter);

Los argumentos para la configuración de la ejecución son evaluados antes de los actuales argumentos de la función y similares a estos, son traspasados al dispositivo vía memoria compartida.

La llamada a la función falla si Dg o Db exceden el tamaño máximo permitido por las capacidades del dispositivo, o si Ns es mayor que la cantidad máxima de la memoria compartida disponible en el dispositivo, menos la cantidad de memoria compartida necesaria para la asignación estática, los argumentos de la función y la configuración de la ejecución.

6.2.1.4 Variables Incorporadas

gridDim

Esta variable es del tipo *dim3* y contiene las dimensiones de la grid.

blockIdx

Esta variable es del tipo *uint3* y contiene el índice del bloque dentro de la grid.

blockDim

Esta variable es del tipo *dim3* y contiene las dimensiones del bloque.

threadIdx

Esta variable es del tipo *uint3* y contiene el índice de hebras dentro del bloque.

warpSize

Esta variable es del tipo *int* y contiene el tamaño del Warp en hebras.

Restricciones

- No se permite tomar la dirección de cualquiera de las variables incorporadas.
- No se permite asignar valores a cualquiera de las variables incorporadas [19], [20].

6.2.2 Compilación con NVCC

El compilador *nvcc* es un controlador (driver) que simplifica el proceso de compilación de un código fuente CUDA. Este proporciona opciones de línea de comandos simples y familiares, y las ejecuta invocando a la colección de herramientas que implementan las diferentes etapas de compilación.

El flujo de trabajo básico de nvcc consiste en separar el código del dispositivo del código del anfitrión, y compilar el primero en una forma de ensamblador (*código ptx*) o en una forma binaria (*objeto cubin*). El código del anfitrión es generado como una salida, ya sea en código C que puede ser compilado por otra herramienta o en código objeto directamente invocando al compilador del anfitrión en la última etapa de compilación.

Las aplicaciones pueden ignorar el código del anfitrión generado, y cargar y ejecutar el código ptx o el objeto cubin en el dispositivo empleando el *CUDA driver API*, o pueden vincular el código del anfitrión generado, que incluye el objeto cubin como un arreglo de datos inicializado global y contiene una traducción de la sintaxis de configuración de la ejecución (descrita en la sección 6.2.1.3) en el arranque del tiempo de ejecución CUDA necesario para cargar y lanzar cada Kernel compilado.

El final de la compilación procesa los archivos fuentes CUDA según las reglas sintácticas de C++. La totalidad de C++ es soportado por el código anfitrión. Sin embargo, solo el subconjunto C de C++ es totalmente soportado por el código del dispositivo. C++ especifica las características como las clases, herencia, o declaración de variables dentro de bloques básicos. Como consecuencia de emplear las reglas sintácticas de C++, los punteros void (por ejemplo, los retornados por malloc( )) no pueden ser asignados a punteros no void sin un encasillamiento.

Algunas instrucciones ptx solo son soportadas en dispositivos con capacidades de cómputo mayores. Por ejemplo, instrucciones atómicas en la memoria global solo son soportadas en dispositivos con capacidad de cómputo 1.1 o superior, instrucciones de doble precisión solo son soportadas en dispositivos con capacidad de cómputo 1.3. La opción de compilación *–arch* determina la capacidad de cómputo que es asumida cuando se compila el código ptx [19], [20]. Así, un código que contiene aritmética de doble precisión, por ejemplo, debe ser compilado con "*-arch sm_13*", de lo contrario la aritmética de doble precisión será degradada a una aritmética de simple precisión. El proceso de compilación CUDA se muestra en la Figura 6.5.

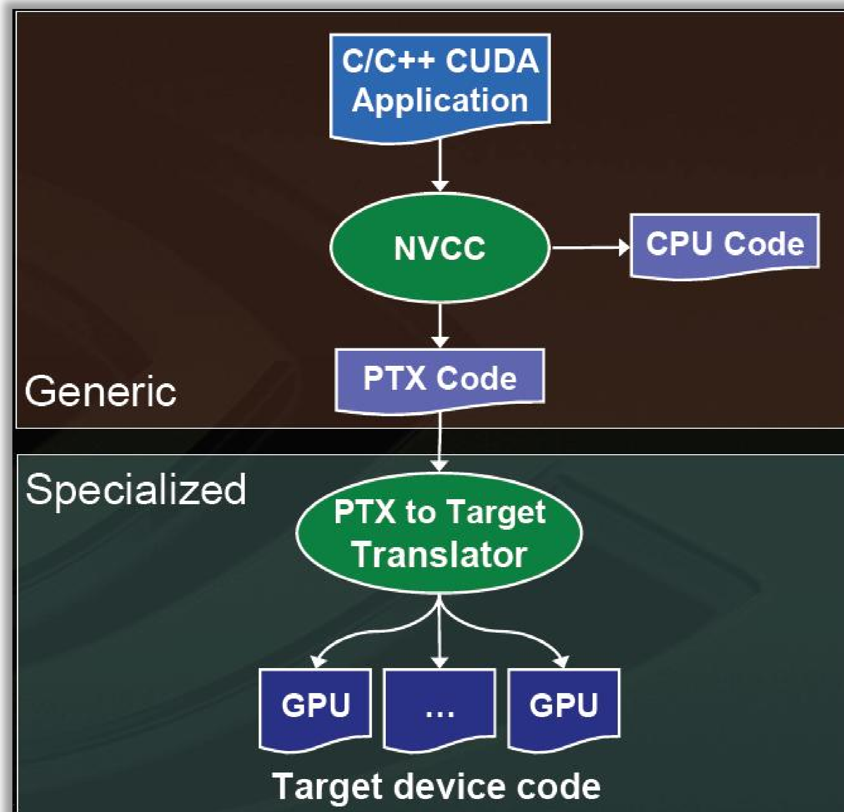

Figura 6.5: Proceso de compilación CUDA.

6.2.3 Componente Común en Tiempo de Ejecución

El componente común en tiempo de ejecución puede ser usado tanto por funciones del anfitrión como del dispositivo.

6.2.3.1 Tipos de Vectores Incorporados

char1, **uchar1**, **char2**, **uchar2**, **char3**, **uchar3**, **char4**, **uchar4**, **short1**, **ushort1**, **short2**, **ushort2**, **short3**, **ushort3**, **short4**, **ushort4**, **int1**, **uint1**, **int2**, **uint2**, **int3**, **uint3**, **int4**, **uint4**, **long1**, **ulong1**, **long2**, **ulong2**, **long3**, **ulong3**, **long4**, **ulong4**, **float1**, **float2**, **float3**, **float4**, **double2**

Son tipos de vectores derivados de los tipos básicos de enteros y punto flotante. Son estructuras y sus componentes 1^{ero}, 2do, 3^{ero} y 4to son accesibles a través de los campos *x, y, z,* y *w*. Los acompaña una función "constructor" de la forma *make_<type name>*, por ejemplo:

```
int2 make_int2(int x, int y);
```

Que crea un vector de tipo *int2* con valores (*x*, *y*).

6.2.3.2 Tipo dim3

El tipo *dim3*, es un tipo de vector entero basado en *uint3*, que es usado para especificar dimensiones. Cuando se define una variable de tipo *dim3*, cualquier componente no especificado es inicializado en 1 [19], [20].

6.2.4 Ejemplo: Multiplicación de Matrices

La tarea de calcular el producto C de dos matrices *A* y *B* de dimensiones *(wA, hA) y (wB, wA)* respectivamente, se divide entre varias hebras de la siguiente manera [19], [20]:

- Cada thread block es responsable de calcular una sub-matriz cuadrada C_{sub} de *C*.

- Cada hebra dentro del bloque es responsable de calcular un elemento de C_{sub}.

La dimensión *block_size* de C_{sub} es elegida igual a 16, de modo que el número de hebras por bloque es un múltiplo del tamaño de un Warp, y se mantiene por debajo del máximo número de hebras por bloque.

Como se ilustra en la Figura 6.6, C_{sub} es igual al producto de dos matrices rectangulares: la sub-matriz de *A* de dimensiones *(wA, block_size)* que tiene los mismos índices de fila como C_{sub} y la sub-matriz de *B* de dimensiones *(block_size, wA)* que tiene los mismos índices de columna como C_{sub}. Con el fin de encajar dentro de los recursos del dispositivo, estas dos matrices rectangulares son divididas en muchas matrices cuadradas de dimensión *block_size* según sea necesario, y C_{sub} es calculada como la suma de los productos de esas matrices cuadradas. Cada uno de estos productos son llevados a cabo por una primera carga de las dos matrices cuadradas correspondientes desde la memoria global hacia la memoria compartida con una hebra cargando un elemento de cada matriz, y luego cada hebra calcula un elemento del producto. Cada hebra acumula los resultados de cada uno de estos productos en un registro y una vez terminado escribe el resultado en la memoria global [19], [20].

Al realizar el cálculo de esta forma, se aprovecha la ventaja de una memoria compartida más rápida y se ahorra una gran cantidad de ancho de banda de la memoria global, ya que *A* y *B* son leídas desde esta solo *(wA / block_size)* veces [19], [20].

Sin embargo, este ejemplo se presenta para exponer una mayor claridad, ilustrando varios principios de programación de CUDA, por ende su finalidad no es proporcionar un Kernel de alto desempeño para una multiplicación de matrices genérica, y por lo tanto no debe interpretarse como tal.

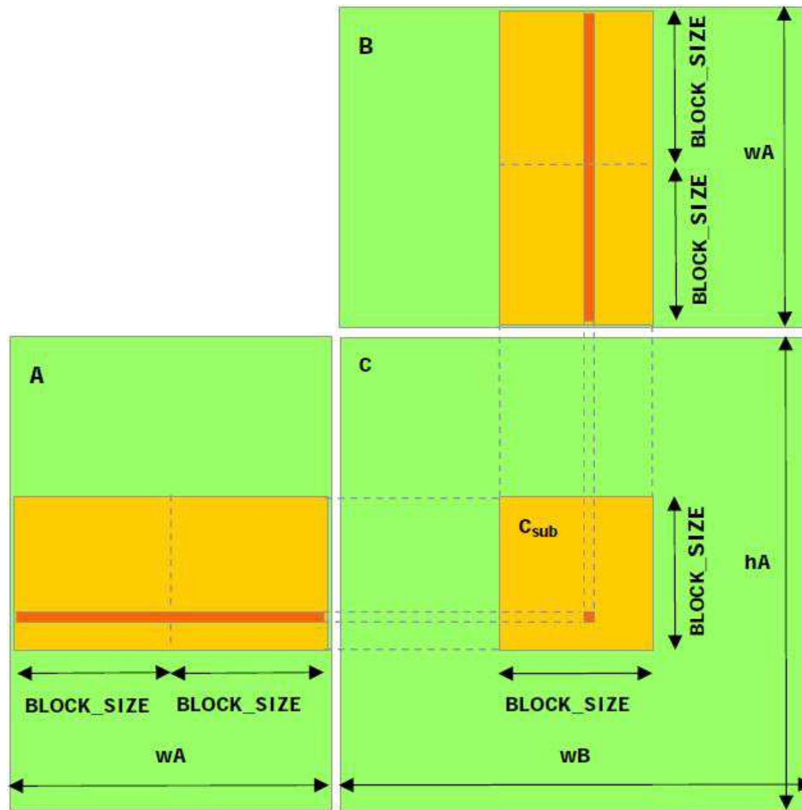

Figura 6.6: Multiplicación de matrices.

El código fuente de este ejemplo se presenta a continuación [19], [20]:

```
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication function
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C)
{
    int size;

    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd;
```

```
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    float* Cd;
    size = hA * wB * sizeof(float);
    cudaMalloc((void**)&Cd, size);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}

// Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
```

```
    int aStep = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;a <= aEnd;a += aStep, b += bStep)
    {
        // Shared memory for the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Shared memory for the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += As[ty][k] * Bs[k][tx];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

El código fuente tiene dos funciones, estas son [19], [20]:

- **Mul( )**, una función del anfitrión que sirve como "envoltorio" para **Muld( )**. Esta función toma como entrada:

  - Dos punteros a la memoria del anfitrión que apuntan a los elementos de *A* y *B*.

  - La altura y el ancho de *A*, y el ancho de *B*.

  - Un puntero a la memoria del anfitrión que apunta donde *C* debe ser escrita.

  Además, lleva a cabo las siguientes operaciones:

  - Asigna suficiente memoria global para almacenar *A*, *B* y *C* usando *cudaMalloc( )*.

  - Copia *A* y *B* desde la memoria del anfitrión a la memoria global usando *cudaMemcpy( )*.

  - Llama a **Muld( )** para calcular *C* en el dispositivo.

  - Copia *C* desde la memoria global a la memoria del anfitrión usando *cudaMemcpy( )*.

  - Libera la memoria global asignada para *A*, *B* y *C* usando *cudaFree( )*.

- **Muld( )** tiene la misma entrada de **Mul( )**, excepto que los punteros apuntan a la memoria del dispositivo en lugar de la memoria del anfitrión. Para cada bloque, esta función itera a través de todas las sub-matrices de *A* y *B* necesarias para calcular C_{sub}. En cada iteración:

  - Carga una sub-matriz de *A* y una sub-matriz de *B* desde la memoria global hacia la memoria compartida.

  - Sincroniza para asegurarse que ambas sub-matrices están totalmente cargadas por todas las hebras dentro del bloque.

  - Calcula el producto de las dos sub-matrices y lo añade al producto obtenido en la iteración anterior.

  - Sincroniza nuevamente para asegurarse que el producto de las dos sub-matrices se realizó antes de iniciar la siguiente iteración.

Una vez que todas las sub-matrices han sido manipuladas, C_{sub} está completamente calculada y **Muld( )** la escribe en la memoria global. Además, esta función permite maximizar el desempeño de la memoria, ya que por ejemplo, no existen conflictos en el banco de memoria compartida debido a que cada hebra accede a diferentes zonas de esta.

7 Criptografía y Criptoanálisis

La criptografía es la otra gran área que abarca el presente proyecto, y que a su vez es tan importante como la referente al conocimiento de la arquitectura, funcionamiento y programación de las tarjetas gráficas. Por lo tanto es de extrema importancia exponer y entender los conceptos básicos asociados con la criptografía que permitirán darle forma a la solución al problema expuesto en el presente proyecto.

El presente capítulo busca definir algunos conceptos importantes de criptografía para entender de qué se trata cada uno de ellos y así proponer de forma más clara aquel algoritmo que será puesto a prueba posteriormente. El capítulo se compone de la definición de criptografía y los tipos de criptografía existentes en la actualidad, pasando por las diferentes clases de ataques que se pueden llevar a cabo sobre sistemas criptográficos, para finalizar con el enfoque y la postulación del algoritmo que será puesto a prueba en la etapa correspondiente del presente proyecto.

7.1 Definición de Criptografía y Criptoanálisis

Según el Diccionario de la Real Academia Española, la palabra *Criptografía* se define como: “Arte de escribir con clave secreta o de un modo enigmático”. Desde un punto de vista más técnico la criptografía se puede definir como el conjunto de técnicas que permiten proteger la información de observadores no autorizados [21]. Por el contrario, el Diccionario de la Real Academia Española define la palabra *Criptoanálisis* como: “Arte de descifrar criptogramas”, por ende el criptoanálisis es lo opuesto a la criptografía, y técnicamente se puede definir como el conjunto de técnicas que permiten “romper” los códigos utilizados por la criptografía para proteger la información [21]. El intento de criptoanálisis de un algoritmo criptográfico es conocido con el nombre de *Ataque*.

Cabe destacar que ambas disciplinas están íntimamente ligadas, ya que al momento de diseñar un sistema criptográfico se debe tener en cuenta su posible criptoanálisis para así fortalecer de forma correcta el sistema evitando posibles ataques de usuarios maliciosos que puedan obtener información que solo debe ser vista por los usuarios autorizados. Ambas ramas, criptografía y criptoanálisis, a su vez se engloban en un concepto único llamado *Criptología*.

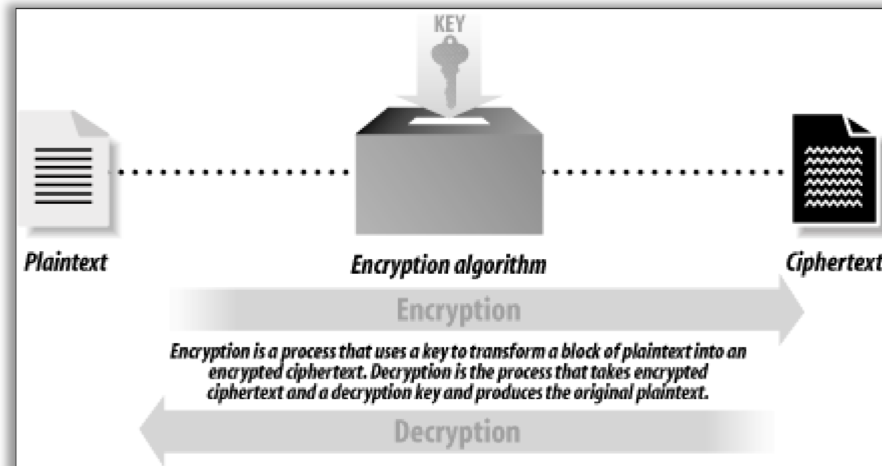

Figura 7.1: Proceso simplificado de encriptación y desencriptación.

7.2 Tipos de Criptografía

7.2.1 Criptografía Simétrica

La criptografía simétrica engloba a todos aquellos algoritmos criptográficos que utilizan la misma clave para encriptar (o cifrar) y desencriptar (o descifrar) uno o más mensajes [22]. Por ejemplo, el emisor encripta un mensaje con una clave K, este lo envía al receptor, el cual para desencriptar el mensaje debe poseer la misma clave K que utilizó el emisor para encriptar el mensaje, de caso contrario no podrá obtener la información contenida en dicho mensaje.

Por lo general, este tipo de algoritmos utilizan las claves que generan en un determinado orden para encriptar y para desencriptar simplemente invierte el orden de las claves utilizadas en el proceso de cifrado. Un par de características importantes es que los algoritmos simétricos son extremadamente rápidos a la hora de cifrar datos y poseen un gran número de claves posibles [22].

Los algoritmos criptográficos simétricos se dividen en dos categorías: los algoritmos de cifrado por bloques (*Block Ciphers*) y los algoritmos de cifrados de flujo (*Stream Ciphers*). En los algoritmos de cifrado por bloques se encriptan los datos en bloques de tamaño fijo de bits mientras que en los algoritmos de cifrados de flujo los datos se encriptan byte a byte (o incluso bit a bit) [21], [22]. Dentro de los algoritmos de cifrado por bloques se encuentran los siguientes algoritmos: DES, CAST, Blowfish, IDEA, AES, etc., y por su parte, dentro de los algoritmos de cifrados de flujo se encuentran: RC4, SEAL, WAKE, etc.

La capacidad de los algoritmos para resistir a diferentes tipos de ataques es llamada *fuerza* (*strength*). La fuerza de los algoritmos se basa en varios factores, pero quizás uno de los más importantes es el largo de la clave utilizada y como se mantiene en secreto dicha clave.

El largo de clave es importante a la hora de realizar un ataque de fuerza bruta, ya que a mayor cantidad de bits, el ataque se vuelve cada vez más inviable debido a que se debe invertir cada vez más tiempo en él para probar todas las posibles claves utilizadas en el proceso de

cifrado. Cada vez que la clave aumenta en un bit, el espacio de claves posibles que acepta el algoritmo aumenta al doble, es decir que el número de claves es igual a $2^{(\text{número de bits})}$. Por ejemplo, si se intenta llevar a cabo un ataque de fuerza bruta con una tecnología que permita probar 1 billón de claves por segundo sobre dos algoritmos, uno que utilice 40 bits de clave y otro que utilice 128 bits de clave, se necesitarían aproximadamente 18 minutos y 10^{22} años, respectivamente, para probar todas las claves posibles utilizadas por cada uno de los algoritmos criptográficos y quizás llegar a dar con la clave correcta empleada en el proceso de cifrado. Con este ejemplo queda en manifiesto la robustez que entrega el largo de clave y el efecto provocado por la adición de más bits a esta, es por esto que se utilizan otras técnicas más efectivas y viables para intentar probar la fuerza de un sistema criptográfico, las cuales se detallarán más adelante.

7.2.2 Criptografía Asimétrica

La criptografía asimétrica engloba a todos aquellos algoritmos que utilizan una clave para encriptar un mensaje y otra para desencriptarlo. Generalmente, la clave que se utiliza para cifrar el mensaje es denominada *clave pública*, mientras que la que se emplea para descifrar el mensaje se denomina *clave privada* [22], por lo tanto, ambas claves en conjunto forman un par de claves. Muchas veces la criptografía asimétrica también es denominada criptografía de clave pública. Algunos algoritmos asimétricos son: RSA, ElGamal, Diffie-Hellman, etc.

Los algoritmos criptográficos asimétricos generalmente emplean longitudes de claves mucho mayores que los algoritmos criptográficos simétricos. Por ejemplo, para algoritmos simétricos se considera segura una clave de 128 bits, mientras que para un algoritmo asimétrico se recomiendan claves de 2048 bits. Por otro lado, los algoritmos asimétricos son mucho más lentos que los algoritmos simétricos, es por esto que en la práctica se utilizan únicamente para cifrar la clave de sesión simétrica que se empleará en una determinada transacción o para un intercambio de mensajes [21].

Una aplicación importante de la criptografía de clave pública es la llamada *firma digital*, la cual sirve para autenticar mensajes. Para esto, necesita de la ayuda de las funciones *Hash* (que serán explicadas en el siguiente punto), las cuales sirven para generar un archivo considerablemente más pequeño que el archivo original que se desea enviar. El proceso de generación y comprobación de la firma digital funciona de la siguiente manera: A desea enviar un mensaje autenticado a B, entonces A genera un archivo Hash del mensaje original, luego A cifra el archivo Hash con su clave privada y envía este junto al mensaje original a B. Cuando B recibe los dos archivos, descifra el archivo Hash con la clave pública de A, y a su vez genera un archivo Hash del mensaje original que recibió. Si ambos archivos Hash coinciden en su valor, quiere decir que el mensaje recibido es de A y no puede ser de otro emisor, en caso contrario, B puede asumir que no conoce el emisor del mensaje y descartarlo, o incluso que ha sido cambiado por un atacante. El proceso de cifrado utilizando algoritmos asimétricos, como el de firma digital, se muestran en la Figura 7.2.

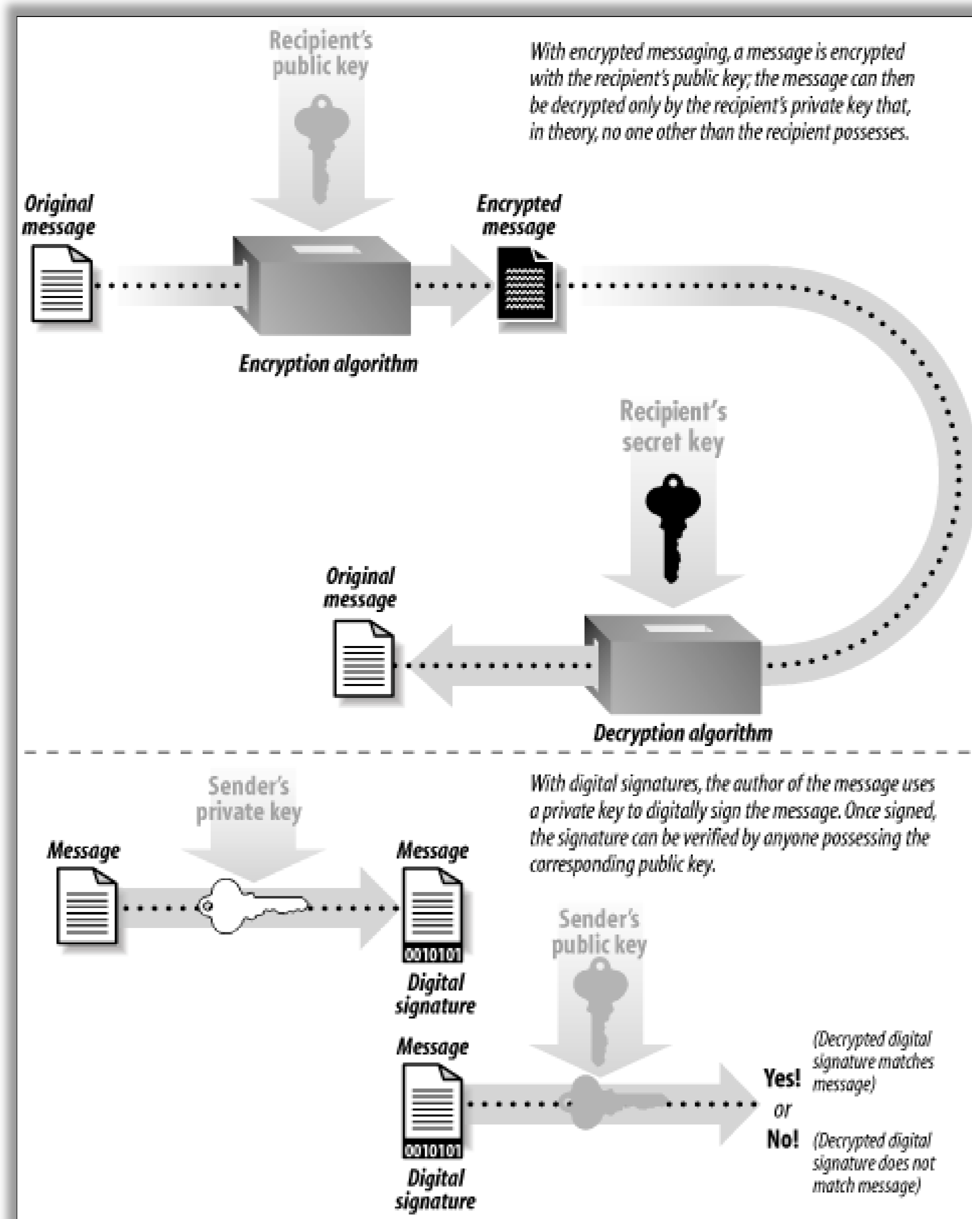

Figura 7.2: Criptografía asimétrica utilizada para cifrar mensajes o para firma digital.

7.2.3 Funciones Hash

Las funciones *Hash* (o *Message Digest Functions*) operan sobre archivos de tamaño arbitrario, devolviendo como resultado de su operación un valor hash de largo fijo, típicamente entre 128 y 256 bits [22], [23]. Su funcionamiento de forma abstracta se ilustra en la Figura 7.3. Si bien las funciones Hash no son algoritmos criptográficos, sirven para apoyar las aplicaciones de algunos de ellos, como se expuso anteriormente en el tema referente a firma digital.

Estas funciones tienen otras propiedades, que combinadas permiten obtener una función Hash segura. Estas son:

- Dado un archivo *M*, es fácil calcular el valor Hash *h*.

- Dado *h*, es computacionalmente intratable recuperar *M*, tal que el valor Hash de *M* sea igual a *h*.

- Dado *M*, es computacionalmente intratable obtener *M'*, tal que el valor Hash de *M* sea igual al valor Hash de *M'*.

Existen funciones Hash que emplean en sus cálculos una clave adicional, las cuales son denominadas *MAC* (*Message Authentication Code*) y otras que no necesitan de una clave, denominadas genéricamente *MDC* (*Modification Detection Codes*) [21].

Dentro de las funciones Hash más conocidas se encuentran: MD4, MD5, SHA-0, SHA-1, RIPE-MD, HAVAL, etc.

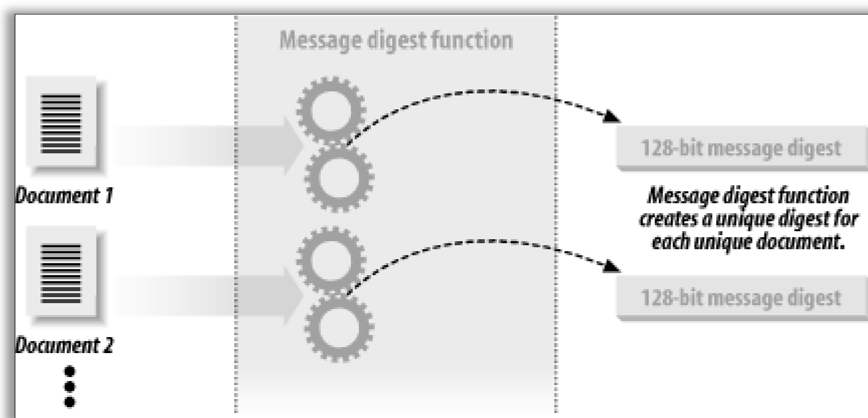

Figura 7.3: Funcionamiento de forma abstracta de una función Hash o Message Digest Function.

7.3 Tipos de Ataques

7.3.1 Brute Force Attack – Ataque de Fuerza Bruta

La manera más sencilla de atacar un mensaje encriptado es simplemente tratar de desencriptar el mensaje con todas las posibles claves utilizadas. A este tipo de ataque se le conoce como *ataque de fuerza bruta* o *ataque de búsqueda de claves* [22]. No existe forma de poder defender un sistema criptográfico de este tipo de ataques, debido a que el atacante siempre tiene la posibilidad de intentar descifrar un mensaje probando todas las claves posibles.

En este tipo de ataques, la mayoría de los intentos por dar con la clave utilizada en el proceso de cifrado fallan, pero eventualmente uno de los intentos podría tener éxito, permitiendo al atacante descifrar uno o más mensajes o ingresar al sistema criptográfico.

Los ataques de búsqueda de claves no son muy eficientes, ya que como se expuso en el punto 7.2.1, son altamente dependientes del largo de clave empleada en el proceso de encriptación, por ende mientras la clave sea más larga, es decir su longitud en bits sea mayor,

el ataque tomará más tiempo en llevarse a cabo completamente o quizás dar con la clave correcta. Por ejemplo, si el largo de clave del algoritmo criptográfico es de 128 bits, se necesita probar 2^{128} posibles claves, lo que medido en tiempo se traduce en un ataque inviable y poco eficiente.

Cabe destacar que como objetivo del presente proyecto, se implementará un ataque de este tipo utilizando la tecnología de las tarjetas gráficas, sobre un algoritmo criptográfico que se propondrá más adelante en este capítulo.

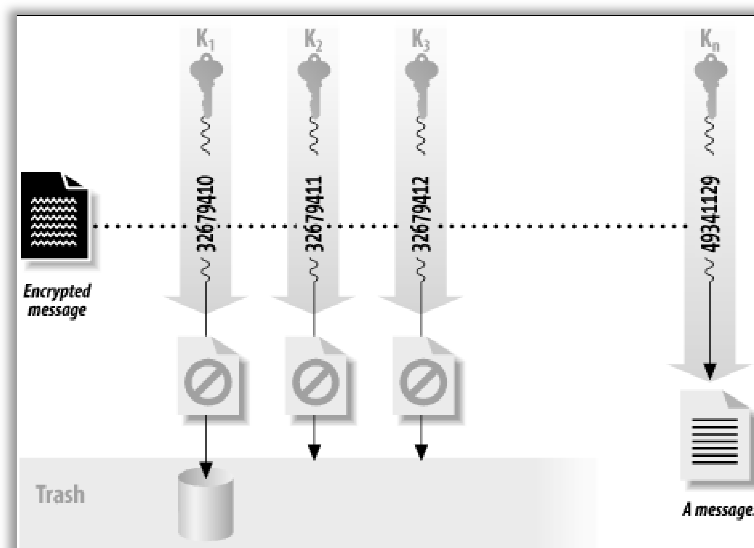

Figura 7.4: Un ataque de fuerza bruta o ataque de búsqueda de claves.

7.3.2 Ciphertext-only Attack – Ataque de Sólo Texto Cifrado

El criptoanalista tiene los textos cifrados de varios mensajes, siendo todos ellos encriptados con el mismo algoritmo criptográfico y la misma clave. El trabajo de este es recuperar el texto claro de la mayor cantidad de mensajes como sea posible, o mejor incluso encontrar la clave empleada para cifrar dichos mensajes, en función de poder descifrar futuros mensajes cifrados con esa clave [23].

7.3.3 Known-plaintext Attack – Ataque de Texto Claro Conocido

El criptoanalista tiene acceso no solo al texto cifrado de varios mensajes, sino también a los textos claros de aquellos mensajes. Esto se da comúnmente cuando se encriptan ciertos mensajes que tienen un mismo inicio o cabecera, o un mismo final, como por ejemplo: e-mails, formularios estándar, etc. El trabajo del criptoanalista es deducir la clave (o claves) utilizadas en el proceso de cifrado de los mensajes o un algoritmo para desencriptar cualquier nuevo mensaje cifrado con la misma clave (o claves) [22], [23].

7.3.4 Chosen-ciphertext Attack – Ataque de Texto Cifrado Elegido

El criptoanalista puede escoger diferentes textos cifrados para ser desencriptados y tiene acceso a varios textos claros descifrados. Este tipo de ataque es principalmente aplicable sobre algoritmos criptográficos asimétricos pero algunas veces también es efectivo sobre algoritmos simétricos. La finalidad de este tipo de ataque es deducir la clave empleada en el algoritmo criptográfico utilizado [23].

Algunas veces, cuando se realiza un ataque de texto claro elegido en conjunto con un ataque de texto cifrado elegido, se le denomina *Chosen-text Attack* (*Ataque de Texto Elegido*).

7.3.5 Chosen-plaintext Attack – Ataque de Texto Claro Elegido

En este tipo de ataque, el criptoanalista no solo tiene acceso al texto cifrado y a sus textos claros asociados de varios mensajes, sino que también elige el texto claro que se cifrará. Este tipo de ataques es mucho más poderoso que el ataque de texto claro conocido, debido a que el criptoanalista puede elegir bloques específicos del texto claro para ser cifrados, siendo escogidos aquellos que le entreguen mayor información sobre la clave. La finalidad de este ataque es deducir la clave empleada para cifrar los mensajes o un algoritmo para descifrar futuros mensajes encriptados con la misma clave [23].

Existe una variante de este tipo de ataque, llamada *Adaptative-chosen-plaintext Attack* (*Ataque de Texto Claro Elegido Adaptativo*), que sigue la misma filosofía pero que su variante se encuentra en que el criptoanalista, aparte de tener la capacidad de elegir el texto claro que se cifrará, también es capaz de modificar sus elecciones basándose en los resultados de encriptaciones previas. Al criptoanalista se le permite elegir trozos más pequeños de los textos claros que se cifrarán y según los resultados entregados por estos, puede decidir a continuación, elegir otros trozos de textos claros que le puedan entregar más información sobre la clave, y así sucesivamente [23].

Cabe destacar que para el ataque de texto cifrado elegido también existe su respectivo ataque adaptativo, llamado *Adaptative-chosen-ciphertext Attack* (*Ataque de Texto Cifrado Elegido Adaptativo*), que sigue la misma idea que el ataque revisado en el párrafo anterior, pero que en vez de utilizar textos claros para ser cifrados y según sus resultados tomar decisiones, emplea textos cifrados escogidos para ser desencriptados y así obtener resultados que permitan deducir la clave o dar paso a la elección de otros textos cifrados para repetir el proceso. De esta forma se entrega información gradualmente al criptoanalista sobre la clave empleada.

7.3.6 Criptoanálisis Diferencial

En 1990, Eli Biham y Adi Shamir introducen el criptoanálisis diferencial. Usando este método, Biham y Shamir encontraron un ataque de texto claro elegido contra DES que era más eficiente que el ataque de fuerza bruta [23].

El criptoanálisis diferencial se centra específicamente en pares de textos cifrados. Para generar estos textos cifrados, se generan pares de textos claros idénticos que difieren en una serie de bits fijados de antemano. Después se calcula la diferencia entre los dos textos cifrados asociados, con la esperanza de detectar patrones estadísticos. Para el caso especifico de DES, se analizan las diferencias de los textos claros que se propagan a través de las rondas del algoritmo cuando están cifrados con la misma clave [21], [23].

La diferencia entre dos textos claros está dada, por ejemplo, por la operación XOR, tal que si se tienen dos textos claros M_1 y M_2, y su diferencia se define como ΔM, la ecuación queda de la siguiente forma:

$$\Delta M = M_1 \oplus M_2$$

Entonces, si se cifran los textos claros M_1 y M_2 (con una clave k), se puede obtener la diferencia en bits (ΔC) de ambos textos cifrados, por lo tanto la ecuación tiene la siguiente forma:

$$\Delta C = C_k\,(M_1) \oplus C_k\,(M_2)$$

El par calculado (ΔM, ΔC) es denominado *diferencial*, y es el que entrega información para su posterior análisis. Una de las ventajas de este tipo de criptoanálisis, es que permite al atacante realizarlo incluso sin conocer detalles del algoritmo criptográfico. Claramente, si se conocen detalles del algoritmo a atacar, el análisis puede ser más preciso ya que puede estudiarse cómo se propagan las diferencias a lo largo de las distintas fases del mismo [21].

7.3.7 Criptoanálisis Lineal

El criptoanálisis lineal, descubierto por Mitsuru Matsui en 1992, funciona de la siguiente manera: se toman algunos bits del texto claro y se efectúa una operación XOR entre ellos, luego se toman algunos bits del texto cifrado y se realiza la misma operación, para finalmente hacer un XOR entre los dos resultados anteriores, obteniendo un único bit [21]. Efectuando esta operación a una gran cantidad de pares de texto claro y texto cifrado diferentes, el resultado debería tener una apariencia aleatoria, por lo que la probabilidad de obtener un 1 o un 0 debería estar próxima a 1/2.

Principalmente, este ataque se lleva a cabo para tener una aproximación a la deducción de la clave que está siendo utilizada por el algoritmo. Si el algoritmo utilizado es vulnerable frente a ataques de este tipo, existirían combinaciones de bits que, bien escogidas, den lugar a un sesgo significativo en la medida anteriormente definida, es decir, que el número de ceros (o

unos) es apreciablemente superior. Esta propiedad permite que se pueda asignar mayor probabilidad a unas claves sobre otras y de esta forma descubrir la clave que se busca [21].

7.4 Propuesta del Algoritmo a Probar

El enfoque para determinar el algoritmo que será puesto a prueba bajo un ataque de fuerza bruta implementado tanto en la tarjeta gráfica como en el procesador es bastante sencillo. La elección de un algoritmo por sobre otros existentes en la actualidad se fundamenta básicamente en parámetros abordables, es decir, que se ha investigado las características del algoritmo a grandes rasgos (específicamente el largo de clave) o cuan masivo es su uso actualmente, o si previamente ha sido vulnerada su seguridad, y a partir de uno o varios de estos parámetros se ha realizado la elección de un algoritmo que será sometido al ataque antes mencionado. Cabe destacar que las razones para seleccionar el algoritmo no son rebuscadas ni complejas debido a que el presente proyecto no busca infringir la seguridad de los algoritmos criptográficos propuestos mediante un ataque, sino que pretende medir la mejora en estos procesos de validación mediante la inclusión de la tecnología de las tarjetas gráficas.

En la sección 7.4.1 se profundizará más en la elección y sus características relevantes para así tener una visión más acabada de este. El algoritmo elegido es un algoritmo simétrico de cifrado de flujo, dejando fuera de la elección a los algoritmos asimétricos y a los algoritmos simétricos de cifrado de bloque, debido a que los primeros son utilizados en procesos como las negociaciones de claves en un sistema criptográfico simétrico, y no en cifrados de grandes volúmenes de datos y los segundos por temas de complejidad de implementación ya que la mayoría de los algoritmos de esta clase utilizan funciones complejas que pueden ser bastante difíciles de entender y codificar, tanto en el procesador como en la tarjeta gráfica. Por otra parte, se dejan fuera las funciones Hash, ya que a pesar de que apoyan la validación de autenticidad e integridad de archivos (en el caso de la firma digital), estas no son algoritmos criptográficos propiamente tales.

7.4.1 Algoritmo Simétrico de Cifrado de Flujo: RC4

RC4 es un algoritmo de cifrado de flujo con clave de largo variable desarrollado en 1987 por Ron Rivest para RSA Data Security, Inc. Es un algoritmo propietario, es decir, que cada persona que quiera incluirlo en una aplicación debe pagar ciertos derechos por el uso del algoritmo. Por muchos años, RC4 se mantuvo en secreto pero en septiembre de 1994 alguien colocó en internet los detalles del algoritmo, y ciertos lectores que poseían copias legales de RC4 confirmaron la compatibilidad de este [23].

Su implementación es extremadamente sencilla y rápida, y está orientado a generar secuencias en unidades de un byte, además de permitir claves de diferentes longitudes (entre 1-2048 bits) [21], [22]. Consta de una *S-Box* de 8x8: $S_0, S_1, \ldots, S_{255}$. Las entradas son una permutación de números del 0 al 255, y la permutación es una función de la clave de largo variable. También posee dos contadores, i y j, que se ponen en 0. Para generar un byte aleatorio, se utiliza el siguiente código:

```
i= (i + 1) mod 256
j= (j + S_i) mod 256
swap S_i and S_j
t= (S_i + S_j) mod 256
K= S_t
```

Para producir el texto cifrado, se toma el byte resultante *K* junto con el texto claro y se le aplica una operación XOR, y para producir el texto claro se desarrolla la misma operación, pero en vez de utilizar el texto claro se emplea el texto cifrado. La encriptación es rapidísima, sobre 10 veces más rápida que DES [23].

La inicialización de la S-Box también es sencilla, primero se llena con $S_0 = 0$, $S_1 = 1$,..., $S_{255} = 255$. Entonces se llena otro vector de 256 bytes con la clave, repitiendo la clave cuantas veces sea necesario para llenar todo el vector: K_0, K_1,..., K_{255} [23]. El contador *j* se configura en 0. Entonces para estas operaciones, se tiene el siguiente código:

```
for i= 0 to 255:
j= (j + S_i + K_i) mod 256
swap S_i and S_j
```

RSA Data Security, Inc. dice que el algoritmo es inmune al criptoanálisis diferencial y lineal. El algoritmo genera secuencias en que los ciclos son bastante grandes y el resultado es altamente no lineal. La S-Box evoluciona lentamente con su uso: *i* asegura que cada elemento cambia y *j* asegura que los elementos cambian aleatoriamente.

Las razones de la elección de este algoritmo para el presente proyecto son bastante sencillas, primero el algoritmo acepta diferentes largos de clave, por lo tanto se podría utilizar para las pruebas una clave de muy pocos bits, por ejemplo 40 bits o incluso menos, y segundo, es que RC4 es un algoritmo bastante difundido y se utiliza en variadas aplicaciones, como SSL, redes wireless WEP, el protocolo de encriptación de BitTorrent, etc., lo que hace interesante someterlo a pruebas de ataques de fuerza bruta en pos de obtener algún resultado positivo que dé cuenta del poder de las tarjetas gráficas.

7.4.2 Resumen de la Propuesta

La idea general de este capítulo y en especial de las secciones 7.3 y 7.4 es comprender que en la actualidad existen avanzados métodos para hacer criptoanálisis del algoritmo propuesto como de otros tantos que solo se nombran en este proyecto. Lo más importante y a lo que se avoca el presente proyecto, sin perder el foco de lo que se desea, son dos temas fundamentales, por un lado se encuentra la incorporación de la tecnología de las tarjetas gráficas a los procesos de validación de algoritmos criptográficos y como estas pueden beneficiar ciertas pruebas especificas, buscando desarrollarlas en menores periodos de tiempo, y por otro lado se encuentra el poder efectuar con esta tecnología, mediciones a los niveles de fortaleza del algoritmo anteriormente propuesto. El objetivo es determinar si existe o no una mejora o beneficio que pueda entregar la tecnología de las tarjetas gráficas en estos procesos de validación. Es por esta razón que se ha propuesto un tipo particular de algoritmo,

proveniente de una de las sub-ramas de la ciencia de la criptografía, para poder probar cómo se comporta frente a una prueba específica.

8 Diseño de la Solución al Problema

Toda aplicación, sin importar su envergadura, necesita de un diseño previo a la codificación que permita asegurar que esta cubrirá todas las necesidades del problema en cuestión. Este diseño puede ir desde la simpleza de un diagrama desarrollado con un lápiz en una hoja de papel, hasta diagramas más complejos desarrollados con herramientas CASE.

La idea principal de este capítulo es definir de manera sencilla la aplicación que intentará solucionar el problema expuesto en el presente proyecto. Si bien la aplicación es bastante especifica y hace una sola tarea (un ataque de fuerza bruta), es importante definir su diseño en palabras y utilizar algún diagrama UML que permita visualizar de mejor manera los componentes de esta.

8.1 Diseño Conceptual

El propósito del siguiente diseño conceptual es definir en palabras lo que se espera que incluya la aplicación. Es importante dejar en claro desde un principio que la aplicación no poseerá una interfaz gráfica avanzada, ya que como se revisó en el Capítulo 6, CUDA es una extensión del lenguaje C y este no da la posibilidad de desarrollar aplicaciones con una interfaz avanzada como la que entregan otros lenguajes, es por esto que la aplicación se concentra en su funcionalidad, y en cumplir su tarea de forma efectiva y eficiente más que en la calidad gráfica de esta.

La aplicación, si se mira de una forma abstracta, deberá entregar la posibilidad de elegir, al usuario, un archivo cifrado con el algoritmo propuesto en el Capítulo 7 (Sección 7.4), luego la aplicación realizará el ataque de fuerza bruta correspondiente al tipo de cifrado utilizado, y finalmente desplegará el resultado por pantalla donde se podrá visualizar el tiempo empleado en el ataque y si este tuvo éxito o no. Por otro lado, la aplicación busca verificar un ataque de fuerza bruta en la tarjeta grafica comparándolo con el tiempo que demora este mismo ataque en un procesador de PC, de esta forma se busca validar el uso de esta tecnología en esta área del conocimiento.

Si se profundiza en el diseño, la aplicación contendrá un conjunto de detalles que se especifican a continuación:

- La aplicación será sencilla, es decir, se concentrará en el ataque de fuerza bruta sobre los archivos cifrados más que en otros aspectos de la misma.

- La aplicación no poseerá una interfaz avanzada, más que la que ofrece el lenguaje de programación C, ya que su objetivo no es ser *usable*, sino que entregue resultados concretos.

- La aplicación funcionará tanto en la CPU como en la GPU, para poder comparar los resultados de las pruebas en ambos dispositivos, y determinar cual tiene un rendimiento superior.

- La aplicación será optimizada incrementalmente para la tarjeta gráfica, es decir, que a medida que pase el tiempo de desarrollo, esta deberá ajustarse cada vez más a las características de la GPU para mejorar el desempeño de la misma.

- La aplicación implementará un ataque de fuerza bruta para el algoritmo criptográfico RC4.

- La aplicación tomará el tiempo que transcurre desde que se inició el ataque hasta cuando finaliza este, con la finalidad de obtener resultados que puedan entregar información sobre el desempeño del ataque y su implementación en la tarjeta gráfica y en el procesador.

- La aplicación desplegará por pantalla el tiempo consumido por el ataque de fuerza bruta y si la clave fue vulnerada. En el caso de que el ataque sea exitoso, se mostrará por pantalla la clave utilizada en el cifrado.

- Para esta aplicación, se determinarán las simplificaciones o reducciones que sean aplicables a los algoritmos para efectuar con un mayor margen de éxito las pruebas.

8.2 Diseño en UML

La aplicación desde un punto de vista general es bastante sencilla en cuanto a la cantidad de funcionalidades que debe poseer, sin embargo es bastante compleja en la tarea específica que realiza. Debido a que la aplicación no presenta diferentes tareas, es que sus diagramas UML son bastante reducidos y entendibles a simple vista.

8.2.1 Diagramas de Casos de Uso

El primer diagrama presenta las funcionalidades que debe tener la aplicación. Principalmente esta debe ser capaz de llevar a cabo un ataque de fuerza bruta y entregar resultados al usuario para que este pueda estudiarlos y manipularlos para así vislumbrar cual de los dos dispositivos, la tarjeta gráfica o el procesador, tiene un mejor desempeño frente a la solución del problema.

El *Diagrama de Caso de Uso de Alto Nivel* en su forma gráfica es el siguiente:

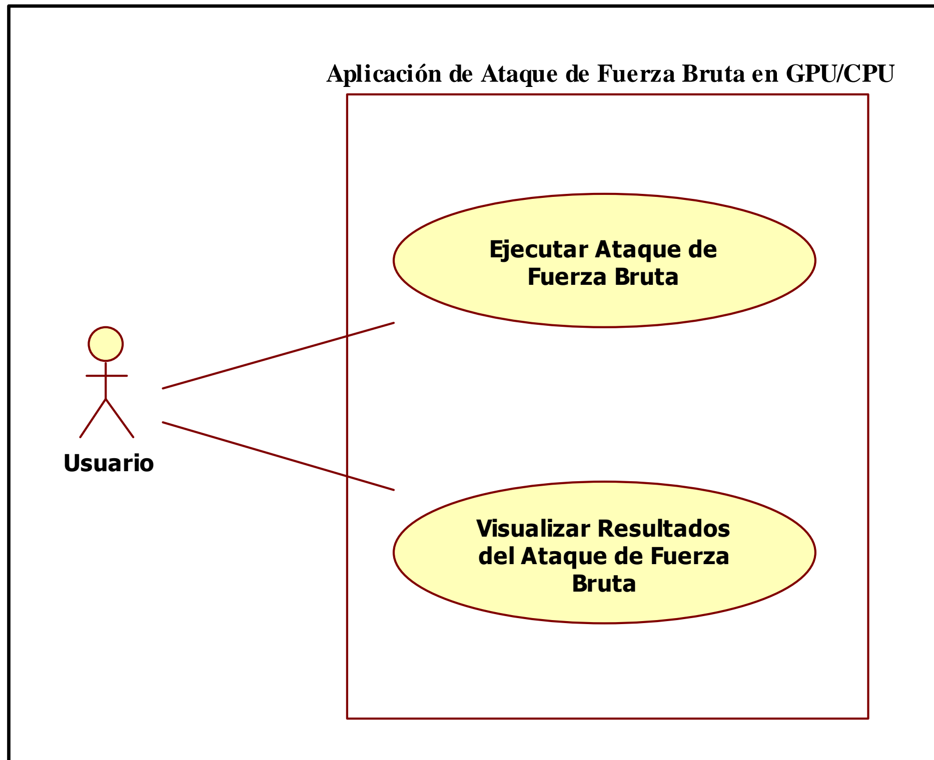

Figura 8.1: Diagrama de Caso de Uso de Alto Nivel (Forma Gráfica).

Si se expanden los dos casos de uso contenidos en el diagrama de caso de uso de alto nivel se tiene dos nuevos diagramas de casos de uso extendidos, los que se presentan a continuación:

- El *Diagrama de Caso de Uso "Realizar Ataque de Fuerza Bruta"* (Figura 8.2) explica las diferentes opciones que brindará la aplicación al usuario a la hora de llevar a cabo el ataque de fuerza bruta. En el diagrama aparecen las diferentes opciones para elegir dos diferentes archivos cifrados con dos tipos de algoritmos, propuestos en el capítulo anterior. Según el archivo cifrado que se ingrese, gatillará el ataque de fuerza bruta correspondiente al algoritmo criptográfico con que se cifró el archivo. Por otra parte, el usario estará facultado para abortar en cualquier momento el ataque por la razón que el estime conveniente, por lo tanto la aplicación tendrá una opcion especial para esta tarea.

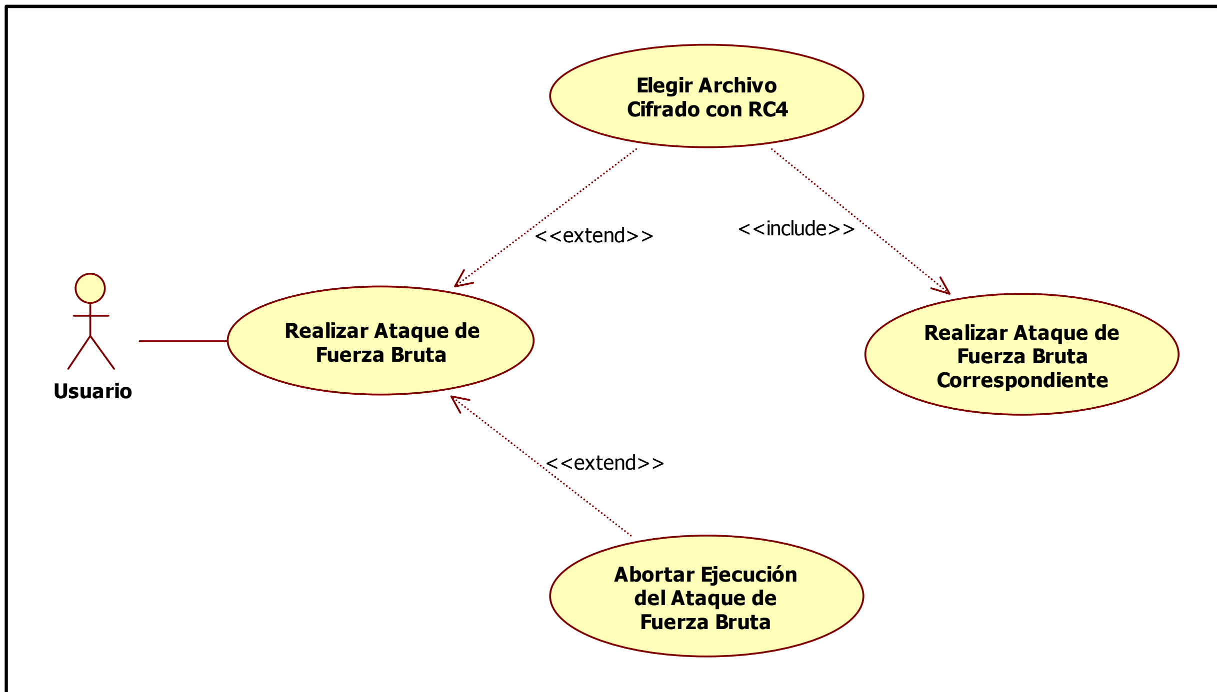

Figura 8.2: Diagrama de Caso de Uso “Realizar Ataque de Fuerza Bruta” (Forma Gráfica).

- El *Diagrama de Caso de Uso “Mostrar Resultados del Ataque de Fuerza Bruta”* (Figura 8.3) explica los diferentes resultados que el usuario podrá obtener luego de que el ataque se haya completado de forma exitosa o no. Principalmente, la aplicación desplegará por pantalla el tiempo en que incurrió para completar el ataque de fuerza bruta y cuantos bits de la clave fueron vulnerados por el ataque, pudiendo ser todos o ninguno, pero es importante que esta información quede detallada para que el usuario pueda utilizarla posteriormente. En caso de que el ataque fuese exitoso completamente, es decir, logré obtener toda la clave desde el archivo cifrado, esta se desplegará por pantalla como resultado del ataque.

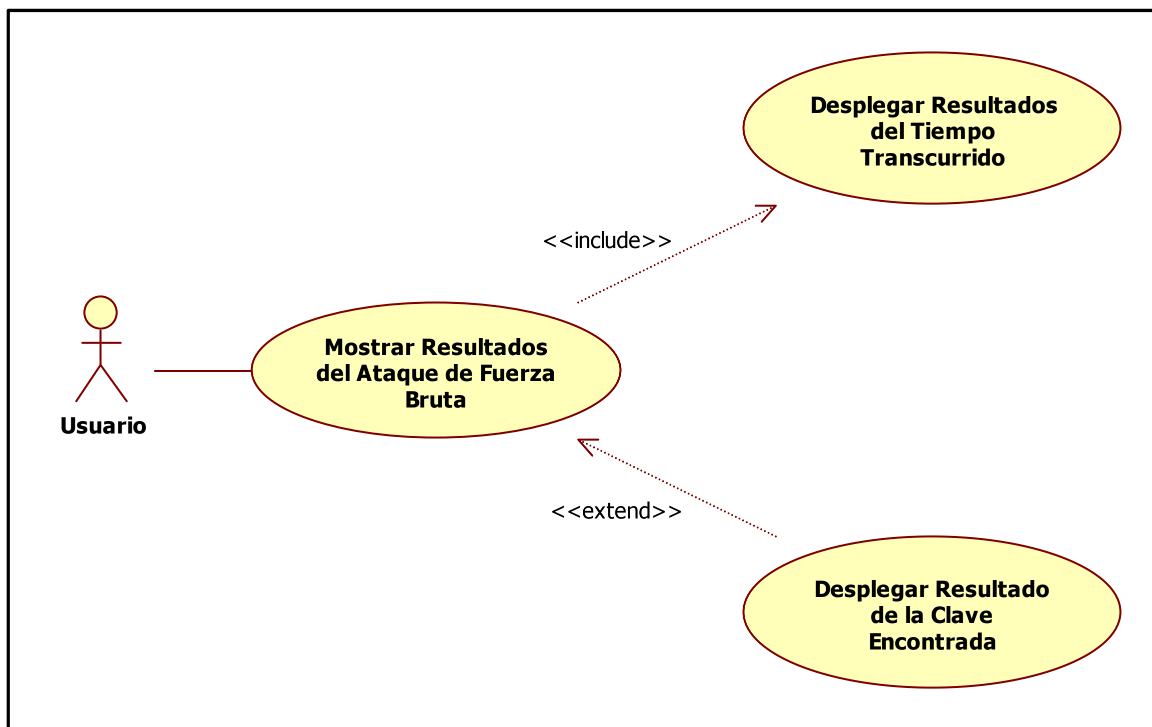

Figura 8.3: Diagrama de Caso de Uso "Mostrar Resultados del Ataque de Fuerza Bruta" (Forma Gráfica).

8.2.2 Diagramas de Actividad

El *Diagrama de Actividad* que se presenta a continuación modela el flujo de procesamiento [24] que seguirá la aplicación al momento de ser implementada. La idea principal de este diagrama es dar a entender la secuencia de pasos que sucederán uno tras otro para llegar a un resultado que permita vislumbrar la capacidad de cómputo de las tarjetas graficas actuales y que a su vez permita determinar la fortaleza de cada uno de los algoritmos propuestos en el capítulo anterior. En la Figura 8.4 quedan detalladas gráficamente las acciones que se llevarán a cabo paso a paso, desde la elección del archivo cifrado a probar hasta el resultado exitoso o no de la ejecución de la aplicación.

La aplicación inicialmente permitirá al usuario elegir entre dos tipos de archivos, que estarán cifrados con alguno de los dos algoritmos propuestos. A partir de eso la aplicación verificará que el archivo ingresado corresponde a uno de los dos tipos de archivos cifrados que puede aceptar, sino dará un aviso al usuario para que este ingrese un archivo correcto. Si el archivo ingresado es correcto, la aplicación iniciará su correspondiente ataque de fuerza bruta. Cuando esta tarea se completa, se iniciarán las tareas que corresponden al ataque en sí, es decir se gatilla el funcionamiento de un contador de tiempo y por otro lado se prueba clave a clave el archivo cifrado, intentando obtener la que ha sido utilizada en el proceso, luego cuando se completa esta acción, si existen bits que han sido vulnerados por el ataque, se almacenarán ya que posteriormente servirán para ser parte del resultado del ataque, sino simplemente continuará el flujo de procesamiento hacia una siguiente acción. Al terminar estas acciones que se ejecutan en paralelo, se iniciará la etapa de entrega de resultados al usuario, en donde se realizarán tres acciones importantes, en primer lugar, la aplicación desplegará por pantalla el tiempo ocupado en realizar el ataque, en segundo lugar, se mostrarán la cantidad de bits

vulnerados pudiendo ser estos entre 0 (ni un bit vulnerado) y el máximo de clave que corresponda al algoritmo utilizado para cifrar el archivo (todos los bits vulnerados), y finalmente si el ataque es exitoso se desplegará la clave utilizada en el proceso de cifrado, en caso contrario no se desplegará la clave ya que el ataque no logro obtener resultados positivos sobre este objetivo. Luego de la entrega de resultados, el flujo de procesamiento terminará para así dar al usuario la opción de llevar a cabo una nueva prueba.

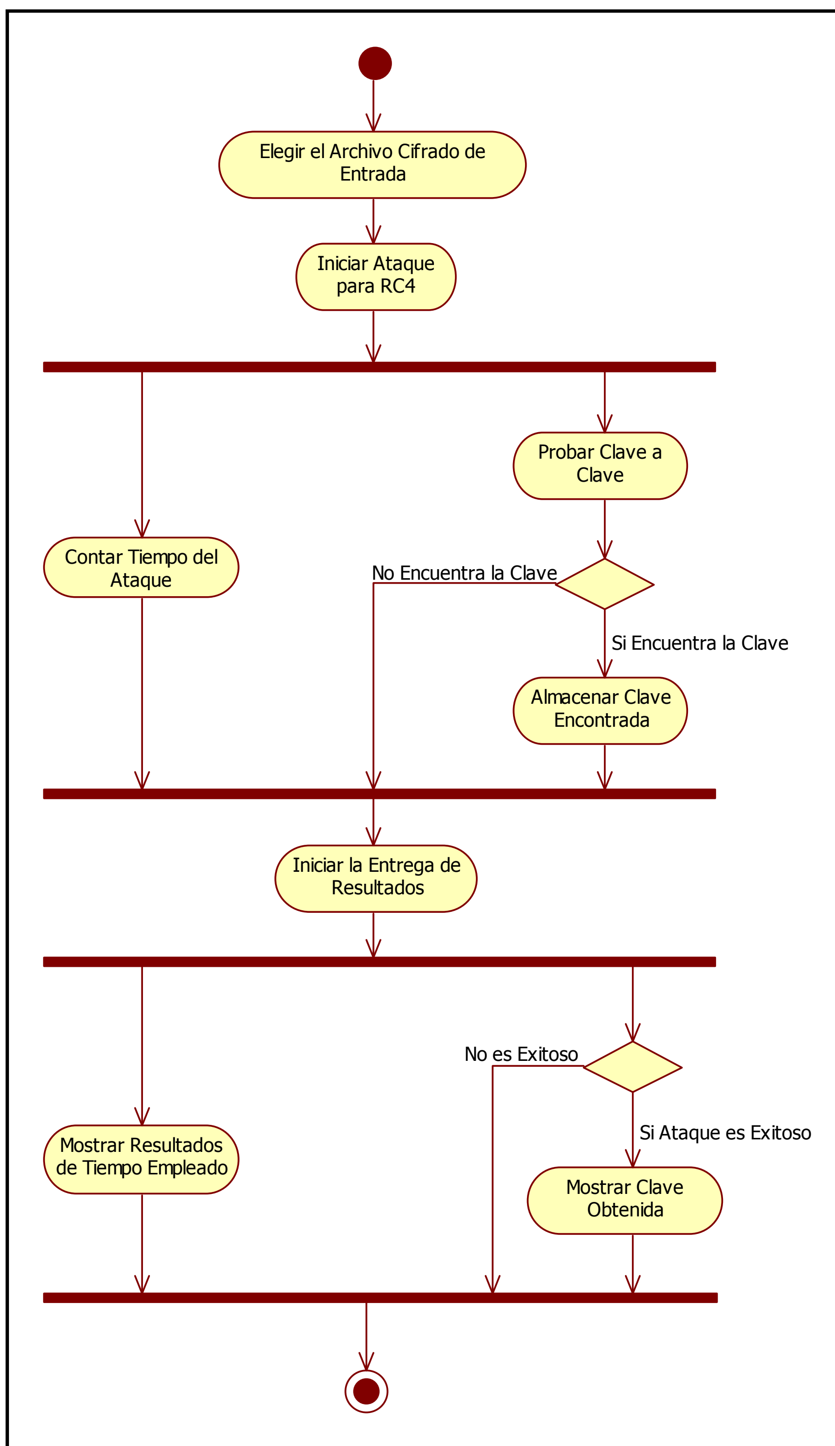

Figura 8.4: Diagrama de Actividad de la aplicación.

9 Configuración de las Pruebas

Todo proyecto de este tipo y envergadura debe tener definidos a priori las características de las pruebas que se realizarán en virtud de obtener resultados tangibles que permitan, a su vez, llegar a conclusiones concretas sobre las tecnologías que se emplean para resolver el problema.

El presente capítulo presenta los detalles de los elementos que se emplearán en las pruebas que serán realizadas en el procesador del PC, como también en la tarjeta gráfica, en busca de poder concluir que tecnología presenta una mejor performance a la hora de resolver el problema planteado en el presente proyecto.

Las siguientes secciones detallan las características de los componentes de las pruebas, y como estas se llevarán a cabo. Lo más importante del enfoque de las pruebas es recalcar que estas se ejecutarán tanto en el procesador como en la tarjeta gráfica por separado, para posteriormente realizar comparaciones de los resultados que revelen la eficiencia y performance de estos dos dispositivos en la ejecución de un ataque de fuerza bruta sobre los archivos cifrados con el algoritmo criptográfico propuesto.

9.1 Características de los Archivos y Claves

El proceso de cifrado de un algoritmo criptográfico involucra archivos que son encriptados añadiéndoles una clave, que permite que estos archivos no sean legibles por cualquier usuario. Sin embargo, cuando un atacante quiere obtener la información que se transmite en un mensaje, este lo captura e intenta obtener dicha información mediante alguna técnica de criptoanálisis presentadas en el Capítulo 7. Si el atacante logra tener éxito, podría obtener la información contenida en el mensaje o mejor aún, podría obtener la clave utilizada en el proceso de cifrado. Es por esto que, basándose en la idea anterior de cómo realizar un ataque, independiente del tipo que sea este, es que en el presente proyecto se someterán a pruebas cierto tipo de archivos con determinadas características en común.

La idea principal es simplificar las pruebas, ya que como se ha revisado hasta este punto, el presente proyecto no busca hacer criptoanálisis ni probar la seguridad de un algoritmo criptográfico, sino que su meta es determinar que para este tipo de ataque cuál de los dos dispositivos, si el procesador o la tarjeta gráfica, pueden entregar una mayor velocidad de ejecución del ataque y por ende un menor tiempo de desarrollo de este.

Basándose en las ideas expuestas, es que las pruebas se llevarán a cabo sobre archivos de texto (archivos con extensión .txt) que comparten ciertas características claves, estas son:

- Los archivos de texto poseen diferentes cantidades de caracteres. Solamente se probarán tres archivos de diferentes tamaños, en los cuales se encuentran solo caracteres imprimibles. Estos caracteres abarcan las letras minúsculas de la $a - z$ y letras mayúsculas de la $A - Z$, en ambos casos se ha eliminado la letra *ñ* ya que esta presenta problemas a la hora de ser impresa por pantalla en el sistema operativo Linux. Además

se encuentran ciertos caracteres de puntuación (solo la coma y el punto). Por motivos de simplicidad se han reemplazado todas las letras con tilde por las mismas pero sin tilde.

- Los archivos de texto tienen diferentes tamaños. El archivo *textoplano.txt* posee un tamaño de 1022 bytes, el archivo *textoplano2.txt* posee un tamaño de 499 bytes y el último archivo llamado *textoplano3.txt* tiene un tamaño de 102 bytes. Estas diferencias buscan obtener relaciones en las pruebas, para lograr dilucidar si un archivo de mayor o menor tamaño influye en los tiempos del ataque de fuerza bruta.

El motivo de trabajar con archivos que posean estas características es que permiten que las pruebas se simplifiquen, ya que en teoría, al tener un menor tamaño, es más rápido y fácil su descifrado lo cual permite probar de forma más eficiente las claves en el ataque de fuerza bruta. Si los archivos fueran más grandes, por ejemplo, de tamaños en gigas, al ataque le tomaría más tiempo probar una clave completamente antes de determinar si esta es la correcta o si se desecha y se intenta el ataque con una nueva. Los caracteres que incluyen cada archivo principalmente son para saber cuando un archivo es descifrado o no, como también para darle el peso determinado en bytes.

Todos los archivos de texto son cifrados utilizando el algoritmo propuesto en el Capítulo 7 y se utilizan las mismas claves para dicho proceso, ya que según su contenido, el cifrado es diferente sin importar que se ocupe la misma clave para cifrar. Por ejemplo: los tres archivos son cifrados utilizando el algoritmo RC4 con la misma primera clave, obteniendo así un archivo cifrado que varía según su tamaño, por lo tanto se obtienen tres archivos diferentes siendo todos cifrados con la misma clave, y así sucesivamente. Cabe destacar que las claves son de un largo de 40 bits (5 caracteres), para así obtener un campo de claves posibles más reducido.

La premisa para establecer las claves para cifrar los archivos de texto es que estas sean simples, por ende, se ha determinado que estas solo pueden incluir números del *0 – 9*, por lo tanto quedan fuera las letras minúsculas y mayúsculas, y los caracteres de puntuación, paréntesis, etc. Estas características buscan acortar el espacio de claves, permitiendo que las claves sean más sencillas y a su vez las pruebas se puedan realizar en intervalos de tiempo menores. Con esta combinación de números se genera un campo total de 100.000 claves, siendo la primera 00000, la segunda 00001, y así sucesivamente hasta la última clave la cual es 99999. Todas las claves fueron generadas con una sencilla aplicación obtenida desde Internet, y los archivos se cifran con claves obtenidas de forma selectiva y aleatoria desde este mismo espacio de claves.

Cabe señalar que todas las medidas determinadas y expuestas anteriormente tienen por objetivo la simplificación de las pruebas, ya que como se ha presentado en el presente proyecto y en la literatura referenciada, estas pruebas con tamaños de claves superiores y una amplia combinación de caracteres pueden tomar días, semanas, meses, incluso hasta años en desarrollarse completamente.

9.2 Características de las Pruebas

Las pruebas son bastantes sencillas de entender y explicar. Como ya se ha expuesto en el Capítulo 7 (Sección 7.3.1), un ataque de fuerza bruta es simplemente intentar descifrar un mensaje con todas las claves posibles contenidas en un determinado espacio de claves para un algoritmo en particular. Es por esto que las pruebas se basan en tomar uno de los tres archivos cifrados e intentar desencriptarlo probando una batería de claves definidas con anterioridad. Este conjunto de claves es definido antes de ejecutar todas las pruebas, y contiene todas las combinaciones de caracteres que se puedan dar para el espacio de claves, por lo tanto, se puede ver a este conjunto de claves como un *diccionario de claves*.

Cuando se da inicio a una prueba, se pondrá en funcionamiento un reloj que permita cronometrar el tiempo que tarda en realizarse la prueba completamente, es decir, el tiempo que se demora en probar todas las claves obtenidas desde el diccionario o hasta que dé con la clave correcta que se utilizó en el proceso de cifrado. Entonces, la prueba consiste en tomar un archivo cifrado, iniciar el reloj, y probar todas las claves posibles hasta que no quede ni una clave o se llegue a la correcta y en ese mismo instante se detiene el reloj. El valor entregado por el reloj es almacenado ya que servirá para confeccionar gráficos, obtener conclusiones y determinar si el procesador o la tarjeta gráfica tiene un mejor desempeño frente a esta tarea. Cuando se terminan todos estos pasos (que se ejecutan para un solo archivo), se inicia una nueva prueba con el siguiente archivo que se encuentra en el repositorio y se repiten los escenarios explicados anteriormente, hasta llegar a procesar los tres archivos cifrados.

Cabe señalar que las pruebas, en primera instancia, se llevarán a cabo en el procesador del computador para tener un primer acercamiento. Cuando se completen todas las pruebas en este hardware, se pasará a realizar las pruebas en la tarjeta gráfica, en donde se ejecutará una modificación de las implementaciones de los algoritmos que les permitan funcionar a estos sin inconvenientes y que a su vez puedan sacar provecho de las características propias de las GPUs. Claramente, los algoritmos que se ejecutarán en el procesador difieren en algunos detalles estructurales de los que se ejecutarán en la tarjeta gráfica, ya que los primeros deben ser adaptados en su modo de operación a las características de las GPUs, para poder explotar de forma correcta el poder de procesamiento de estas.

Finalizando las pruebas se espera que la cantidad de resultados obtenidos permitan extraer conclusiones claras de que tecnología es más eficiente para la resolución del problema.

9.3 Plataforma para Pruebas

La plataforma para llevar a cabo las pruebas es básicamente la enumeración del hardware que se utilizará para realizar los ataques de fuerza bruta sobre los archivos cifrados con el algoritmo RC4, poniendo énfasis en detallar las características del procesador y la tarjeta gráfica a emplear.

Esta plataforma es la que posee de antemano el proyectista en su hogar para la realización del presente proyecto.

Los componentes hardware y sus respectivos detalles principales se listan a continuación:

- Placa Base: ABIT KN9 SLI, la que cuenta con un Chipset NVIDIA NFORCE 570 SLI para controlar el resto de los dispositivos adjuntos a ella.

- Procesador: AMD Athlon 64 X2 modelo 3800+, el cual opera a una frecuencia de reloj de 2.0 GHz y posee doble núcleo de procesamiento.

- Tarjeta Gráfica: NVIDIA GTX 460 marca EVGA. El Chip cuenta con una frecuencia de reloj de 726 MHz, 768 MB de memorias GDD5 de frecuencia de reloj de 900 MHz, interfaz de memoria de 192-bits y 336 Stream Processors o CUDA Cores (336 núcleos de procesamiento).

- Memorias RAM: 2GB de memorias RAM de 667 MHz de frecuencia de reloj.

- HDD: Disco Duro marca HITACHI, SATA2 de 250GB, el cual opera a 7200RPM.

- Fuente de Poder: Marca Antec modelo TruePower 750 Watts.

- Sistema Operativo: Ubuntu Linux, versión 10.04.

10 Ataque de Fuerza Bruta en el Procesador

El presente capítulo entrega los resultados obtenidos de las pruebas del ataque de fuerza bruta en el procesador para el algoritmo de cifrado de flujo RC4, y un análisis de estos.

Las pruebas miden el tiempo que toma en encontrar una determinada clave, contenida en el espacio de claves propuesto anteriormente, que permita descifrar el algoritmo y entregar el texto plano.

A continuación se explican y exponen los resultados de las pruebas en el procesador, como fueron determinadas las claves para el cifrado de los archivos, los tiempos empleados en cada uno de los ataques por clave, y las tablas y gráficos para presentar de forma clara los resultados del ataque. Para completitud y un mayor entendimiento, se analizan y se realizan observaciones de estos.

Cabe destacar, que por motivos de presentación, los resultados expuestos en el presente capítulo corresponden a las pruebas realizadas sobre el archivo *textoplano.txt* debido a que este es el de mayor tamaño y se emplea en cada prueba un tiempo mayor para encontrar cada clave. Por consiguiente las tablas y gráficos con los resultados de los ataques de fuerza bruta desarrollados para los archivos *textoplano2.txt* y *textoplano3.txt* pueden ser revisadas en el *Anexo C*.

10.1 Porción de Código Fuente que Realiza el Ataque de Fuerza Bruta

En todo código fuente que realiza una tarea específica, existe o existen ciertas funciones que realizan el cálculo preciso para el resultado final de este.

A continuación se presenta la porción de código fuente que desempeña el ataque de fuerza bruta en el procesador o también llamado Host en el entorno de programación CUDA. La totalidad del código fuente puede ser inspeccionado en el *Anexo A*.

```
void brute_force(FILE *txt, FILE *rc4, int lt, int cifrar, unsigned char *texto) {
    unsigned char clave[257];
    unsigned char descifrado[1025];
    unsigned char c; // Variable para guardar cada carácter de la clave

    unsigned char alfabeto[56] = {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u',
'v',w','x','y','z','A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','
Z',' ',',','.','\0'};
    int i, y, c1, c2, lc = 5, flag = 0, ii, gen_key = 0, valor;
    time_t Comienzo_Seg, Final_Seg; // Variables para capturar la hora de inicio y
término del ataque
    struct tm *TiempoComienzoPtr, *TiempoFinalPtr; // Punteros para tomar el tiempo
inicial y el tiempo final del ataque
```

```
        clock_t comienzo;  // Variable para tomar el tiempo con CLOCKS_PER_SEC

        Comienzo_Seg = time(NULL); // Obtiene la hora inicial en segundos cuando se inicia
el ataque de fuerza bruta
        TiempoComienzoPtr = gmtime(&Comienzo_Seg); // Convierte los segundos a la hora
GMT inicial, por lo tanto tiene un desfase de 4 horas (GMT-4 da la hora local)
        Comienzo = clock(); // Obtiene el "clock inicial" cuando se inicia el ataque de fuerza
bruta

        printf("\n\n|----------> INICIO DEL ATAQUE DE FUERZA BRUTA SOBRE RC4
<----------|\n");
        printf("\n> Hora de comienzo del ataque: %s", asctime(TiempoComienzoPtr));

        while (flag == 0) {
            valor = gen_key; // La asignación permite no perder el valor de la clave que es
dividida hasta llegar a un residuo igual a 0

            for (i = 0; i < 5; i++) { // En el bucle se genera una nueva clave en cada iteración
                clave[4-i] = '0' + (valor % 10);
                valor = valor / 10;
            }
            clave[lc] = '\0'; // Con \0 se marca el final de la clave (string)

            rc4_init(clave, lc); // Inicializa el algoritmo para el descifrado

            for (y = 0; y < lt; y++) { // Se ejecuta el descifrado
                descifrado[y] = texto[y] ^ rc4_output(); // El texto cifrado XOR la salida de la
función PRGA da el descifrado (texto plano)
            }

            for (c1 = 0; c1 < lt; c1++) { // Se compara el descifrado, carácter por carácter, con
el alfabeto para saber si se logró encontrar la clave
                for (c2 = 0; c2 < 56; c2++) {
                    if (descifrado[c1] == alfabeto[c2]) {
                        break;
                    }
                }
                if (c2 == 56) {
                    break;
                }
            }

            if (c1 == (lt-1)) { // Condición para escribir el texto plano en el archivo .txt y
determinar la clave que descifra el archivo .rc4
```

```
                    for (y = 0; y < lt; y++) {
                        fprintf(rc4, "%c", descifrado[y]);
                    }

                    printf("\n> La CLAVE para descifrar el Archivo RC4 es: \"%s\"", clave);

                    Final_Seg=time(NULL);
                    TiempoFinalPtr=gmtime(&Final_Seg); // Convierte los segundos en la hora
GMT final, por lo tanto tiene un desfase de 4 horas (GMT - 4 da la hora local)

                    printf("\n\n> Hora de término del ataque: %s", asctime(TiempoFinalPtr));
                    printf("\n> Número de segundos transcurridos desde el comienzo hasta el final:
\"%.6f\" segundos.\n", difftime(Final_Seg, Comienzo_Seg));
                    printf("\n> Número de segundos transcurridos desde el comienzo hasta el final
(clocks): \"%.6f\" segundos.\n\n", (clock()-comienzo)/(double)CLOCKS_PER_SEC);

                    printf("|----------> FIN DEL ATAQUE DE FUERZA BRUTA SOBRE RC4 <--
--------|\n\n\n");
                    flag = 1; // Con la variable en 1 finaliza el bucle while
                }
                gen_key++;  // Aumenta en 1 la variable generadora de claves para la siguiente
iteración
            }
        }
```

10.2 Ataque de Fuerza Bruta para Claves Seleccionadas

La explicación de esta selección de claves, y del ataque es bastante sencilla. A priori se ha determinado que se tiene un campo de 100.000 claves, la idea inicial es probar claves con saltos de 5.000 en 5.000 y ver cómo se comporta el ataque. Se realizan tres pruebas por clave, para poder determinar el promedio de los tiempos que se demoró el ataque en encontrar la clave. Los ataques se inician con la clave 05000, luego la 10000 y así sucesivamente hasta llegar a la última clave que es 99999.

El ataque es serial, es decir va tomando en forma ascendente las claves del dominio e intenta descifrar el archivo hasta dar con la clave utilizada en el proceso de cifrado. En teoría, inicialmente, se puede suponer que si con la clave 05000, el ataque toma un tiempo de *X* segundos, para la clave 10000 debería ser *2X* segundos, para la clave 15000 debería ser *3X* segundos y así sucesivamente. Como se verá en los resultados esta ley se cumple casi de forma exacta, y el ataque de fuerza bruta es extremadamente rápido debido a los supuestos y reducciones que se presentaron en el capítulo anterior.

La tabla a continuación expone los tiempos que se emplearon en encontrar cada una de las claves, siendo cada prueba ejecutada tres veces para obtener un valor más exacto de los segundos transcurridos en dar con la clave utilizada en el proceso de cifrado. Tal como se ve en la tabla, prácticamente se cumple la teoría inicial de que cada clave tomaría un cierto

tiempo *X* multiplicado por la *posición de la clave* dentro del campo de claves. Por ejemplo, la clave 05000 tiene un promedio de 0,22 segundos, por lo tanto, la clave 10000 debería tardarse un tiempo de 2 por 0,22, lo cual sucede de forma casi exacta. A continuación se expone la tabla con los resultados de tiempo para el ataque de fuerza bruta sobre el archivo *textoplano.txt*.

Claves	Prueba Nº 1 (seg.)	Prueba Nº 2 (seg.)	Prueba Nº 3 (seg.)	Promedio (seg.)
05000	0,22	0,22	0,22	0,22
10000	0,45	0,45	0,45	0,45
15000	0,68	0,68	0,68	0,68
20000	0,91	0,90	0,90	0,90
25000	1,14	1,14	1,14	1,14
30000	1,36	1,36	1,36	1,36
35000	1,59	1,59	1,59	1,59
40000	1,82	1,81	1,82	1,82
45000	2,04	2,04	2,04	2,04
50000	2,27	2,28	2,27	2,27
55000	2,50	2,50	2,50	2,50
60000	2,72	2,72	2,73	2,72
65000	2,95	2,96	2,95	2,95
70000	3,17	3,18	3,18	3,18
75000	3,41	3,41	3,40	3,41
80000	3,64	3,63	3,64	3,64
85000	3,88	3,86	3,86	3,87
90000	4,10	4,10	4,08	4,09
95000	4,31	4,32	4,31	4,31
99999	4,54	4,54	4,55	4,54

Tabla 10.1: Tiempos empleados en cada ataque de fuerza bruta para claves seleccionadas para el archivo textoplano.txt.

Asociado a la tabla anterior, se presenta el siguiente gráfico:

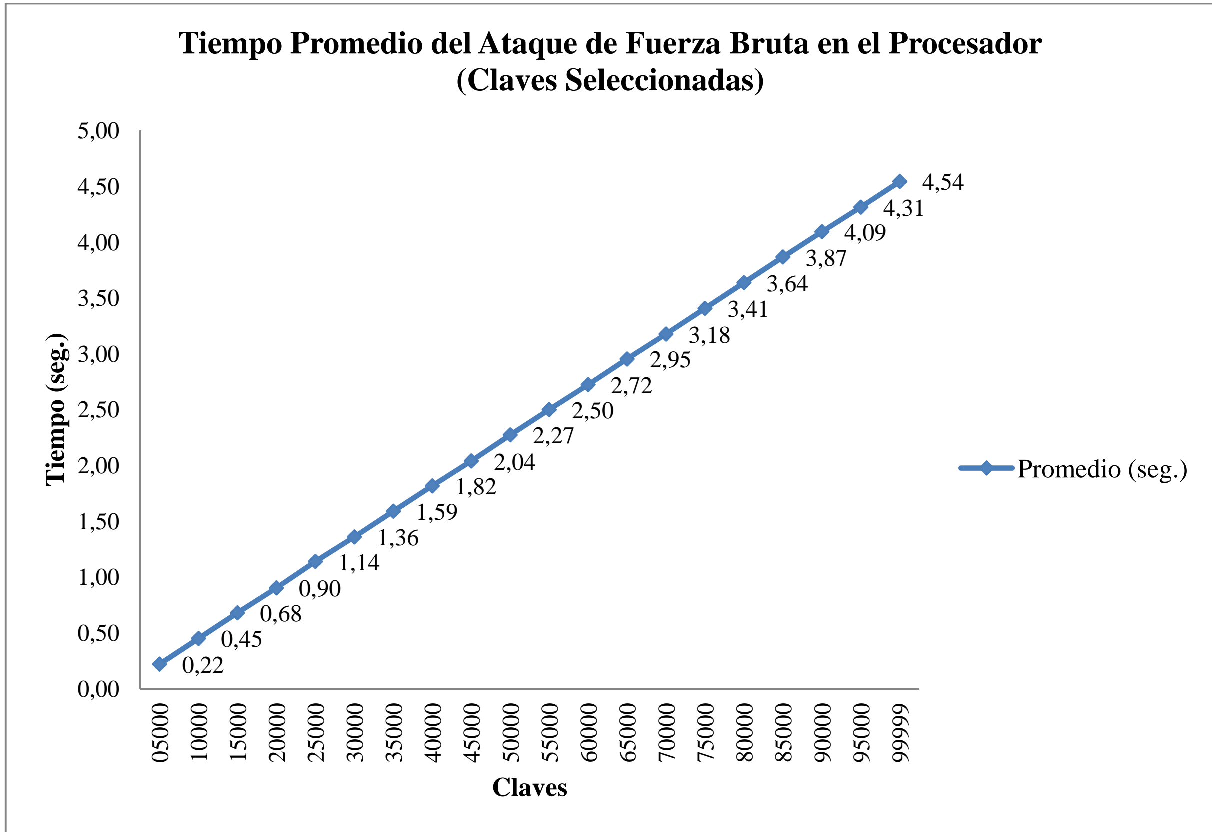

Figura 10.1: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta para claves seleccionadas para el archivo textoplano.txt.

En el gráfico anterior, se aprecia claramente que la curva va en dirección diagonal hacia arriba, y no posee cambios notorios, lo que viene a recalcar la teoría inicial de cómo se relaciona el tiempo y la posición de las claves, obteniéndose una variación de 0,22 a 0,23 segundos cada 5.000 claves.

Como es un ataque el cual se lleva a cabo de forma secuencial, es normal que mientras más lejos se posicione la clave buscada de la clave 00000, más tiempo transcurrirá para encontrarla, pero a pesar de esto, los tiempos son bajos para un ataque de esta naturaleza. La explicación radica en las reducciones que se han realizado en el campo de claves, el cual posee una cantidad abordable de combinaciones de claves. Por otro lado la velocidad del procesador, y la eficiencia y rapidez propia del algoritmo RC4, ayudan a que el ataque de fuerza bruta se pueda desarrollar en tiempos razonables y con un comportamiento estable, lo que se refleja en las variaciones de tiempo antes mencionadas.

10.3 Ataque de Fuerza Bruta para Claves Aleatorias

Las claves utilizadas en estas pruebas fueron elegidas de forma aleatoria, es decir, no tienen relación entre ellas como las claves propuestas en las pruebas anteriores. Simplemente se utilizó una pequeña aplicación que genera claves de forma aleatoria, y se seleccionaron las que eran más atractivas para ser probadas.

Al igual que las pruebas anteriores, para este ataque se probaron veinte claves sobre un solo archivo, siendo elegido nuevamente el que presenta un mayor tamaño. Todo archivo más pequeño que el utilizado en las pruebas presenta tiempos menores para encontrar cada clave, algo que ha quedado destacado en las pruebas para claves seleccionadas.

A continuación se expone la tabla con las claves seleccionadas al azar, y los respectivos tiempos que se emplearon en desarrollar el ataque de fuerza bruta sobre el archivo elegido.

Claves	Prueba Nº 1 (seg.)	Prueba Nº 2 (seg.)	Prueba Nº 3 (seg.)	Promedio (seg.)
02200	0,10	0,10	0,10	0,10
07556	0,34	0,34	0,34	0,34
11226	0,51	0,51	0,50	0,51
16389	0,74	0,74	0,74	0,74
25999	1,18	1,17	1,18	1,18
29779	1,35	1,35	1,35	1,35
32693	1,48	1,48	1,48	1,48
39665	1,81	1,80	1,80	1,80
44499	2,02	2,02	2,02	2,02
47586	2,16	2,16	2,16	2,16
53551	2,44	2,43	2,42	2,43
59999	2,72	2,72	2,72	2,72
61843	2,81	2,81	2,82	2,81
68429	3,10	3,11	3,10	3,10
76311	3,46	3,47	3,47	3,47
79378	3,61	3,60	3,60	3,60
82010	3,73	3,73	3,72	3,73
86375	3,92	3,92	3,91	3,92
92017	4,17	4,18	4,17	4,17
98797	4,49	4,49	4,48	4,49

Tabla 10.2: Tiempos empleados en cada ataque de fuerza bruta para claves aleatorias para el archivo textoplano.txt.

El gráfico asociado a la tabla anterior se presenta a continuación:

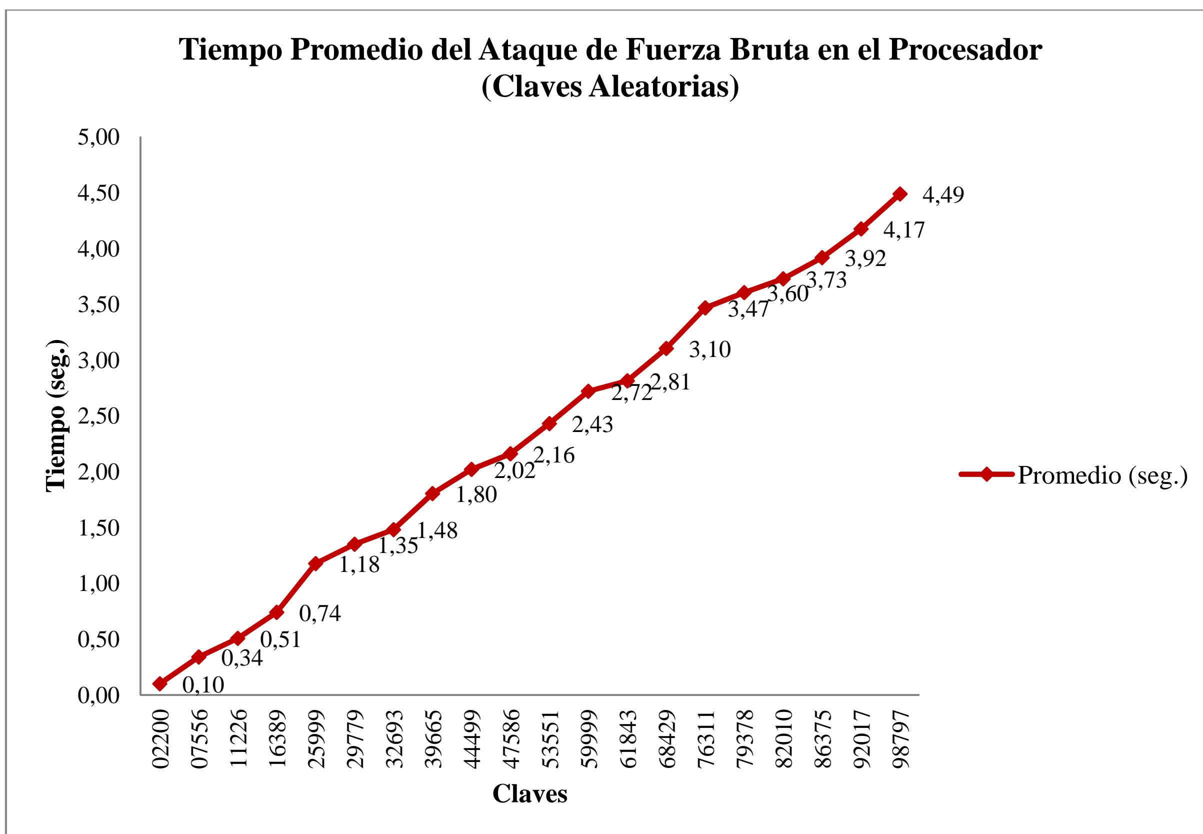

Figura 10.2: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta para claves aleatorias para el archivo textoplano.txt.

Como se ve en el gráfico, las claves y el tiempo empleado para encontrarlas en el campo de claves van siempre en orden ascendente, como se estima a priori y tal como sucede con las pruebas para claves seleccionadas, ya que el ataque es llevado a cabo en forma serial, probando una a una cada clave, hasta hallar la correcta que descifra el texto encriptado con el algoritmo RC4. En resumen, a pesar de utilizar claves aleatorias, se mantienen las características y resultados de los ataques de fuerza bruta realizados con claves seleccionadas.

11 Ataque de Fuerza Bruta en la Tarjeta Gráfica

El presente capítulo, al igual que el anterior, entrega los resultados obtenidos de las pruebas del ataque de fuerza bruta en la tarjeta gráfica para el algoritmo de cifrado de flujo RC4, y un análisis de estos.

Las pruebas son exactamente iguales a las realizadas en el procesador, con la salvedad de que el código fuente se ha ajustado para que funcione correctamente en la GPU. A su vez, la determinación de las claves, la obtención de los tiempos finales, etc. sigue con la misma propuesta de las pruebas desempeñadas en el procesador.

Previo a la presentación de los resultados de las pruebas, es necesario señalar que se han desarrollado diferentes variantes del mismo código fuente que realiza el ataque de fuerza bruta en la GPU, los cuales pueden ser revisados en el *Anexo B*. La primera solución a la problemática es un ataque implementado prácticamente igual al del procesador, es decir, sin utilizar hebras (CUDA Threads) para lograr un paralelo entre la CPU y la GPU. En los códigos fuentes desarrollados posteriormente de a poco se ha ido incluyendo la característica del manejo de múltiples hebras, utilizando inicialmente diez hebras hasta llegar a doscientas hebras en la última configuración del código fuente, todo esto con el afán de determinar cuánto influían en los resultados del ataque de fuerza bruta.

A continuación se exponen los resultados de las pruebas en la tarjeta gráfica con tablas y figuras para presentar de forma clara los resultados de cada uno de los ataques y sus variantes. Para completitud y un mayor entendimiento, se analizan y se realizan observaciones de estos.

Es importante señalar, que por motivos de presentación, los resultados expuestos en el presente capítulo corresponden a las pruebas realizadas sobre el archivo *textoplano.txt* debido a que este es el de mayor tamaño y se emplea en cada prueba un tiempo mayor para encontrar cada clave. Por consiguiente las tablas y gráficos con los resultados de los ataques de fuerza bruta desarrollados para los archivos *textoplano2.txt* y *textoplano3.txt* pueden ser revisadas en el *Anexo D.*

11.1 Porción de Código Fuente que Realiza el Ataque de Fuerza Bruta

Al igual que en el ataque de fuerza bruta en el CPU, se tiene para el código fuente implementado en la tarjeta gráfica, una porción de este que ejecuta y obtiene los resultados finales para observar la efectividad y/o eficiencia de la GPU o también denominada Device en el ambiente de programación CUDA.

A continuación se presenta la función de ataque de fuerza bruta implementada en la GPU. El código fuente, en toda su extensión, puede ser inspeccionado en el *Anexo B*.

```
    __global__ void brute_force(unsigned char *texto_dev, int lt, unsigned char *clave_dev,
unsigned char *descifrado_dev, unsigned char *tid_dev) { // El Kernel CUDA - Función
principal que realiza el ataque de fuerza bruta, y que es invocada por el host

        unsigned char alfabeto[56] = {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u',
'v','w','x','y','z','A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','
Z',' ',',','.','\0'};
        int y, c1, c2, lc = 5, ii, gen_key = 0, valor;
        int id = threadIdx.x;
        int N_combinaciones = 500; // Con esta variable se maneja la cantidad de claves que
puede manejar cada hebra, la cual debe ser acorde con la cantidad de hebras definidas
inicialmente
        int key_inicial = id * N_combinaciones;
        int key_final = key_inicial + N_combinaciones;

        for (gen_key = key_inicial; gen_key < key_final && tid_dev[1] == 0; gen_key++) {
// Cuando la clave de la i-ésima iteración logra descifrar el texto, gen_key toma el valor
100.000, finalizando el bucle for

            valor = gen_key; // La asignación permite no perder el valor de la clave que es
dividida hasta llegar a un residuo igual a 0
            for (ii = 0; ii < 5; ii++) { // En el bucle se genera una nueva clave en cada iteración
                clave_dev[257 * id + 4 - ii] = '0' + (valor % 10);
                valor = valor / 10;
            }
            clave_dev[257 * id + lc] = '\0'; // Con \0 se marca el final de la clave (string)

            rc4_init(id, &clave_dev[257 * id], lc); // Inicializa el algoritmo para el descifrado

            for (y = 0; y < lt; y++) { // Se ejecuta el descifrado, y si tiene éxito el ataque
escribe el texto plano en un archivo .txt
                descifrado_dev[1025 * id + y] = texto_dev[y] ^ rc4_output(id); // XOR que
entrega el descifrado
            }

            for (c1 = 0; c1 < lt; c1++) { // Se compara el descifrado con el alfabeto para saber
si se logró encontrar la clave
                for (c2 = 0; c2 < 56; c2++) {
                    if (descifrado_dev[1025 * id + c1] == alfabeto[c2]) {
                        break;
                    }
                }
                if (c2 == 56) { // Condición de término si es que el descifrado arroja "caracteres
basura" (ilegibles)
                  break;
                }
            }
```

```
            if (c1 == (lt-1)) { // Condición para escribir el texto plano en el archivo .txt y
determinar la clave que descifra el archivo .rc4
                tid_dev[0] = (unsigned char) ('0' + id);
                tid_dev[1] = 1;
            }
        }
    }
```

11.2 Ataque de Fuerza Bruta para Claves Seleccionadas

La selección de claves y la filosofía del ataque es exactamente igual al realizado en el procesador, por lo cual se procede a omitir toda explicación redundante y se entregan directamente los resultados de los ataques de fuerza bruta.

El código fuente fue ajustado para que este empleara 1 hebra (exactamente igual que para el ataque en el procesador), 10 hebras, 50 hebras, 100 hebras y 200 hebras, con el fin de determinar cómo se comporta cada configuración y como se podían disminuir los tiempos en cada ejecución.

Las tablas y gráficos que se presentan a continuación, muestran los tiempos que se emplearon en encontrar cada una de las claves, siendo cada prueba ejecutada tres veces para obtener un valor más exacto de los segundos transcurridos en dar con la clave empleada en el proceso de cifrado.

11.2.1 Ataque de Fuerza Bruta con Una Hebra

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
05000	3,77	3,78	3,78	3,78
10000	7,55	7,56	7,55	7,55
15000	11,33	11,34	11,33	11,33
20000	15,12	15,11	15,12	15,12
25000	18,90	18,90	18,89	18,90
30000	22,67	22,68	22,67	22,67
35000	26,46	26,45	26,45	26,45
40000	30,23	30,22	30,23	30,23
45000	34,01	34,01	34,01	34,01
50000	37,78	37,79	37,79	37,79
55000	41,57	41,57	41,57	41,57
60000	45,36	45,35	45,35	45,35
65000	49,13	49,13	49,13	49,13
70000	52,91	52,91	52,91	52,91
75000	56,70	56,69	56,70	56,70
80000	60,47	60,47	60,47	60,47

85000	64,25	64,25	64,25	64,25
90000	68,03	68,03	68,04	68,03
95000	71,81	71,81	71,80	71,81
99999	75,60	75,59	75,60	75,60

Tabla 11.1: Tiempos empleados en cada ataque de fuerza bruta con una hebra para claves seleccionadas para el archivo textoplano.txt.

Asociado a la tabla anterior, se presenta el siguiente gráfico:

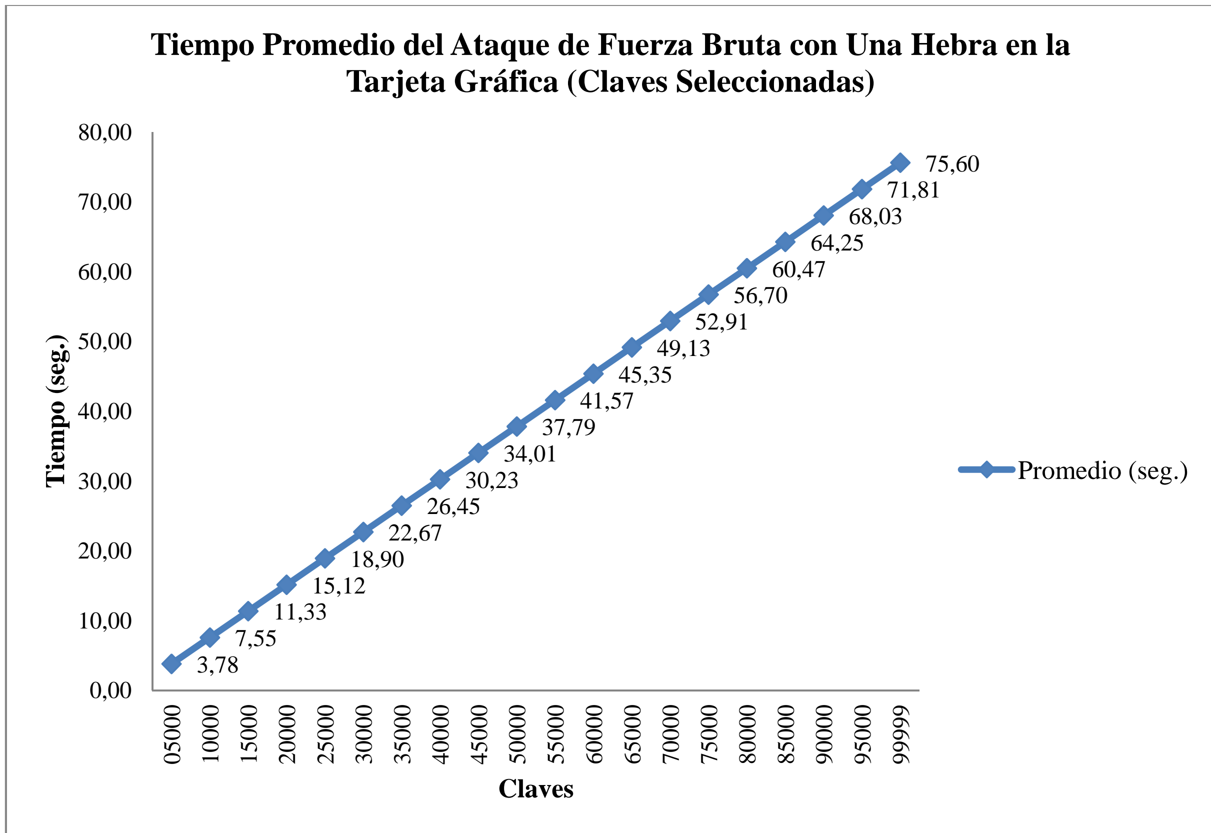

Figura 11.1: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con una hebra para claves seleccionadas para el archivo textoplano.txt.

Similar a la Figura 10.1, se observa un gráfico con valores de tiempo ascendentes, determinado por la posición en que se encuentre la clave dentro del campo de claves, pero muy por el contrario los tiempos son altos en comparación con los arrojados por las pruebas en la CPU. Comparándolas de forma directa, se puede visualizar que el procesador obtiene más rápidamente las claves en el ataque de fuerza bruta, no así la tarjeta gráfica, esto se debe a varias razones: la CPU posee una velocidad de más del doble de potencia que la GPU, por otro lado el procesador accede directamente a los datos desde la memoria RAM, mientras que la tarjeta gráfica debe copiar los datos desde la memoria RAM a su memoria propia, manipularlos y luego enviarlos de vuelta a la memoria RAM, y por último se desaprovecha la característica principal de la GPU, la cual es la utilización de múltiples hebras que pueden ser ejecutadas por sus múltiples núcleos.

Las características desaprovechadas de la GPU, su baja velocidad en comparación con el procesador, y la copia de datos entre el anfitrión y el dispositivo, hacen que este ataque sea inviable. El escenario se torna aun más complejo si se configuran pruebas con archivos de mayor envergadura y claves más complejas, compuestas por diferentes caracteres y de un largo superior a las utilizadas en estas pruebas. Si se emplean archivos de mayor tamaño, la copia de datos entre la memoria RAM y la memoria propia de la tarjeta gráfica, aumenta el tiempo de efectividad del ataque de fuerza bruta, y si a esto se le suma la velocidad de procesamiento que posee la GPU, el ataque se torna difícil o imposible de realizar en un tiempo abordable.

11.2.2 Ataque de Fuerza Bruta con 10 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
05000	16,50	16,50	16,50	16,50
10000	0,01	0,00	0,01	0,01
15000	16,50	16,50	16,50	16,50
20000	0,00	0,01	0,00	0,00
25000	16,50	16,49	16,50	16,50
30000	0,01	0,01	0,00	0,01
35000	16,50	16,49	16,49	16,49
40000	0,01	0,01	0,01	0,01
45000	16,49	16,50	16,49	16,49
50000	0,00	0,01	0,01	0,01
55000	16,49	16,50	16,50	16,50
60000	0,00	0,00	0,01	0,00
65000	16,50	16,50	16,50	16,50
70000	0,01	0,01	0,01	0,01
75000	16,50	16,50	16,50	16,50
80000	0,00	0,01	0,00	0,00
85000	16,50	16,49	16,50	16,50
90000	0,01	0,01	0,01	0,01
95000	16,50	16,49	16,49	16,49
99999	32,99	32,99	32,98	32,99

Tabla 11.2: Tiempos empleados en cada ataque de fuerza bruta con 10 hebras para claves seleccionadas para el archivo textoplano.txt.

Asociado a la tabla anterior, se presenta el siguiente gráfico:

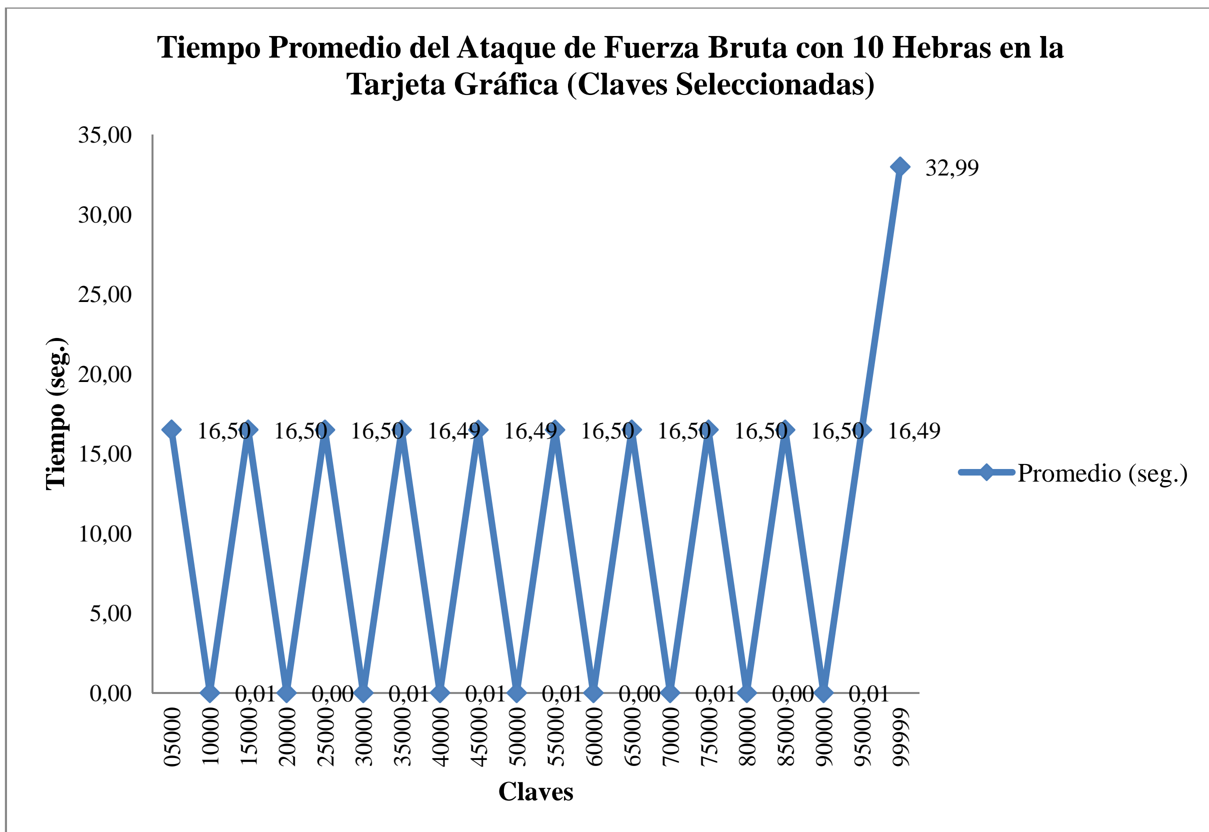

Figura 11.2: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 10 hebras para claves seleccionadas para el archivo textoplano.txt.

Al comenzar a utilizar más de una hebra, en este caso diez hebras, ya se comienza a notar una mejora en la reducción del tiempo empleado para encontrar una clave determinada dentro del campo de claves. La Figura 11.2 luce un aspecto de líneas ascendentes y descendentes en puntos claves, la explicación a este fenómeno es simple: al desarrollar un código fuente que emplea diez hebras, cada una de estas divide el campo de claves en subconjuntos de 10.000 claves. Cada hebra tiene la misión de probar al mismo tiempo 10.000 claves del dominio de claves, es decir, la hebra 0 prueba desde la clave 00000 hasta la 09999, la hebra 1 prueba desde la clave 10000 hasta la 19999, y así sucesivamente cubriendo todas el campo de claves. Cada hebra se ejecuta al mismo tiempo, por lo cual van a la par probando cada segmento asignado hasta que una de ellas encuentra la clave que descifra el archivo, con esto se detiene el ataque ya que las demás hebras son informadas del hallazgo.

Lo explicado en el párrafo anterior permite entender porque con algunas claves se obtienen tiempos de prácticamente 0 segundos mientras que con otras aumenta considerablemente, esto se debe a que cada segmento de claves probado por una hebra comienza en la clave 10000, 20000, y así sucesivamente hasta la clave 90000, en todas aquellas claves se obtienen tiempos despreciables para un ataque de esta naturaleza, mientras que en las claves 05000, 15000, y así sucesivamente hasta la clave 95000 se obtiene un tiempo medio de ataque de 16,5 segundos, debido a que estas claves se encuentran en la mitad de cada

subconjunto de claves, por consiguiente, la clave 99999, que es la última clave de un grupo de claves obtiene el mayor tiempo para ser determinada. A partir de este tiempo se puede determinar que todas las claves finales de cada segmento poseen el mismo tiempo para ser encontradas, teniéndose en el peor de los casos un tiempo de ataque de 32,99 segundos. A pesar de utilizar 10 hebras, no se logra mejorar ostensiblemente el tiempo del ataque de fuerza bruta en la tarjeta gráfica versus a la del procesador, pero comparándolo con el caso en que se ocupa una hebra en el código fuente, este mejora en más de un 50%, obteniendo tiempos de menos de la mitad en el peor de los casos. Derivado de lo anterior, no importa en qué punto se encuentre la clave, el tiempo de ataque no supera los 32,99 segundos, lo cual es una mejora en la búsqueda de claves que se encuentran más allá de la clave 45000 en un ataque de fuerza bruta con una hebra.

11.2.3 Ataque de Fuerza Bruta con 50 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
05000	8,31	8,31	8,32	8,31
10000	0,01	0,01	0,01	0,01
15000	8,31	8,31	8,30	8,31
20000	0,02	0,01	0,01	0,01
25000	8,31	8,31	8,30	8,31
30000	0,01	0,02	0,01	0,01
35000	8,30	8,31	8,31	8,31
40000	0,02	0,01	0,01	0,01
45000	8,32	8,31	8,30	8,31
50000	0,02	0,00	0,01	0,01
55000	8,32	8,31	8,31	8,31
60000	0,01	0,01	0,01	0,01
65000	5,13	5,14	5,12	5,13
70000	0,01	0,01	0,01	0,01
75000	5,12	5,12	5,12	5,12
80000	0,01	0,00	0,01	0,01
85000	5,12	5,13	5,13	5,13
90000	0,00	0,01	0,01	0,01
95000	5,13	5,12	5,13	5,13
99999	10,24	10,23	10,24	10,24

Tabla 11.3: Tiempos empleados en cada ataque de fuerza bruta con 50 hebras para claves seleccionadas para el archivo textoplano.txt.

Asociado a la tabla anterior, se presenta el siguiente gráfico:

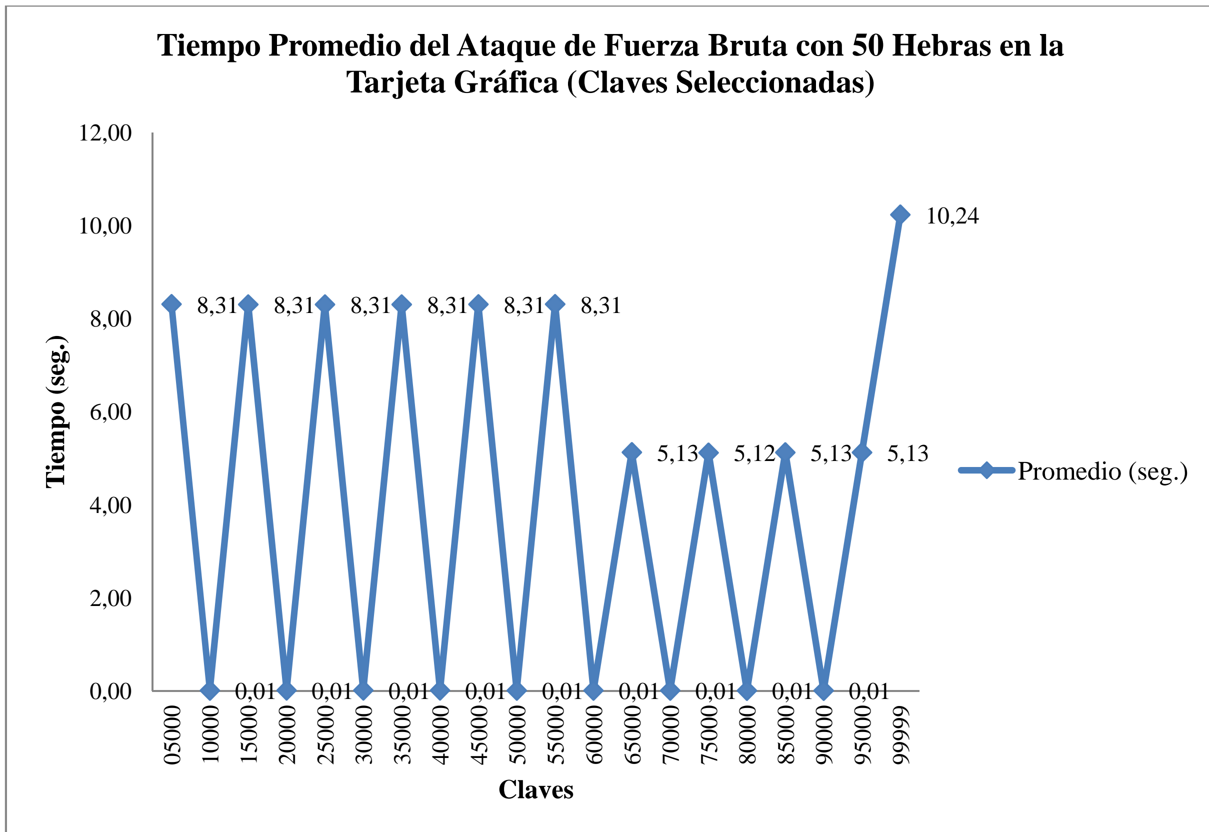

Figura 11.3: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 50 hebras para claves seleccionadas para el archivo textoplano.txt.

Las pruebas realizadas con el código fuente que incorpora 50 hebras, disminuye los tiempos del ataque de fuerza bruta, es así como al ir incorporando una mayor cantidad de hebras, el ataque se vuelve cada vez más eficiente. Al emplear 50 hebras, cada una de estas debe procesar grupos de 2.000 claves, lo cual permite que por ejecución simultánea de cada hebra, se reduzcan los tiempos comparados con las pruebas con una hebra y 10 hebras. Para este caso, las claves 10000, 20000, etc. son las que se ven beneficiadas y son las que obtienen tiempos prácticamente de 0 segundos, mientras que las claves restantes de las pruebas obtienen tiempos de aproximadamente 8,31 segundos, por lo cual se puede calcular que en el peor de los casos, se debe tener un tiempo aproximado de 16,62, lo cual no sucede siempre, debido a un fenómeno extraño que comienza a ocurrir desde esta variante del código fuente en adelante. Desde la clave 65000, hasta la clave final, las pruebas arrojan tiempos menores, cayendo hasta los 5,13 segundos aproximadamente, lo cual repercute en la clave final, 99999, entregando un valor aproximado de 10,24 segundos, tal como se puede apreciar en la Figura 11.3. El comportamiento de la tarjeta gráfica simplemente mejora luego de la clave 65000, es decir, específicamente desde la hebra 32 la cual gestiona el segmento entre la clave 64000 y 66000. Luego de pruebas y lecturas de documentación asociada, no se ha podido determinar el porqué de este comportamiento al emplear más hebras que en las pruebas anteriores, en donde la GPU y sus resultados han sido estables y parejos. Cabe destacar que este comportamiento se

mantiene en las siguientes pruebas, con más hebras, y siempre desde la misma clave antes mencionada.

11.2.4 Ataque de Fuerza Bruta con 100 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
05000	0,02	0,01	0,02	0,02
10000	0,01	0,01	0,02	0,01
15000	0,01	0,01	0,02	0,01
20000	0,02	0,02	0,02	0,02
25000	0,02	0,02	0,02	0,02
30000	0,02	0,02	0,01	0,02
35000	0,02	0,01	0,02	0,02
40000	0,02	0,01	0,02	0,02
45000	0,02	0,02	0,03	0,02
50000	0,02	0,01	0,01	0,01
55000	0,01	0,02	0,02	0,02
60000	0,01	0,02	0,02	0,02
65000	0,01	0,01	0,02	0,01
70000	0,02	0,01	0,02	0,02
75000	0,02	0,02	0,02	0,02
80000	0,02	0,02	0,02	0,02
85000	0,01	0,02	0,02	0,02
90000	0,02	0,02	0,01	0,02
95000	0,01	0,02	0,01	0,01
99999	2,09	2,08	2,08	2,08

Tabla 11.4: Tiempos empleados en cada ataque de fuerza bruta con 100 hebras para claves seleccionadas para el archivo textoplano.txt.

Asociado a la tabla anterior, se presenta el siguiente gráfico:

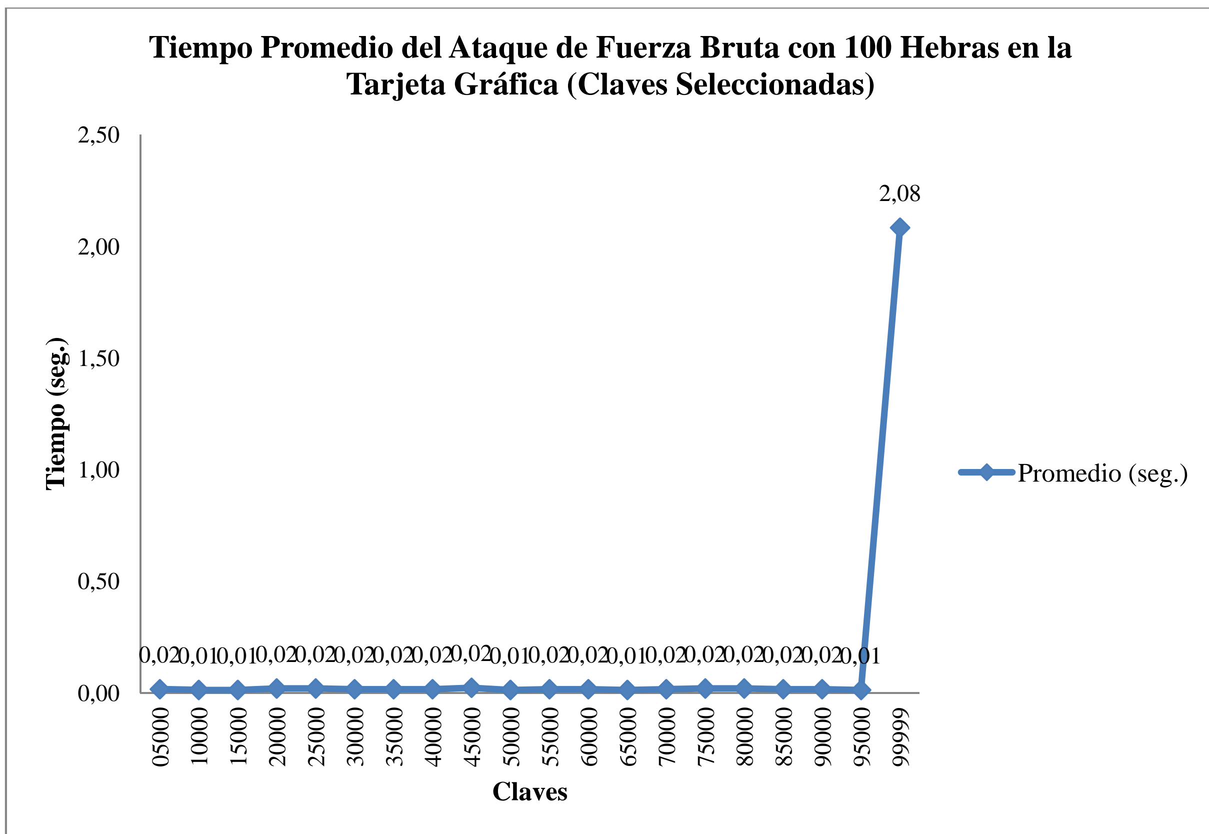

Figura 11.4: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 100 hebras para claves seleccionadas para el archivo textoplano.txt.

Empleando 100 hebras en el código fuente, se obtiene un gráfico como se ve en la Figura 11.4, el cual es plano y aumenta en la clave final del dominio de claves. La explicación a este comportamiento es sencilla: con 100 hebras, el campo de claves se divide en segmentos de 1.000 claves que deben ser probadas por cada hebra, por lo cual, en el ataque de fuerza bruta para claves seleccionadas, cada hebra inicia su subconjunto con las claves XX000, siendo X un valor que varía entre 0 y 9, por ende, se obtienen tiempos despreciables en este tipo de ataque. Con la clave 99999, se tiene un tiempo de 2,08 segundos, el cual nuevamente se encuentra bajo el efecto del fenómeno que se da al pasar la clave 65000, lo cual queda más claro en las pruebas posteriores con claves aleatorias. Nuevamente, el aumento de hebras repercute en la disminución general de los tiempos de prueba, obteniendo de forma más eficiente las claves utilizadas para desencriptar el archivo.

11.2.5 Ataque de Fuerza Bruta con 200 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
05000	0,02	0,01	0,02	0,02
10000	0,02	0,02	0,02	0,02
15000	0,02	0,02	0,02	0,02
20000	0,02	0,02	0,02	0,02
25000	0,03	0,03	0,02	0,03
30000	0,02	0,03	0,02	0,02
35000	0,02	0,01	0,01	0,01
40000	0,02	0,02	0,02	0,02
45000	0,02	0,01	0,02	0,02
50000	0,01	0,02	0,02	0,02
55000	0,02	0,02	0,01	0,02
60000	0,03	0,02	0,03	0,03
65000	0,02	0,02	0,01	0,02
70000	0,03	0,02	0,02	0,02
75000	0,02	0,02	0,03	0,02
80000	0,02	0,01	0,03	0,02
85000	0,02	0,01	0,02	0,02
90000	0,02	0,02	0,02	0,02
95000	0,02	0,02	0,02	0,02
99999	1,66	1,67	1,67	1,67

Tabla 11.5: Tiempos empleados en cada ataque de fuerza bruta con 200 hebras para claves seleccionadas para el archivo textoplano.txt.

Asociado a la tabla anterior, se presenta el siguiente gráfico:

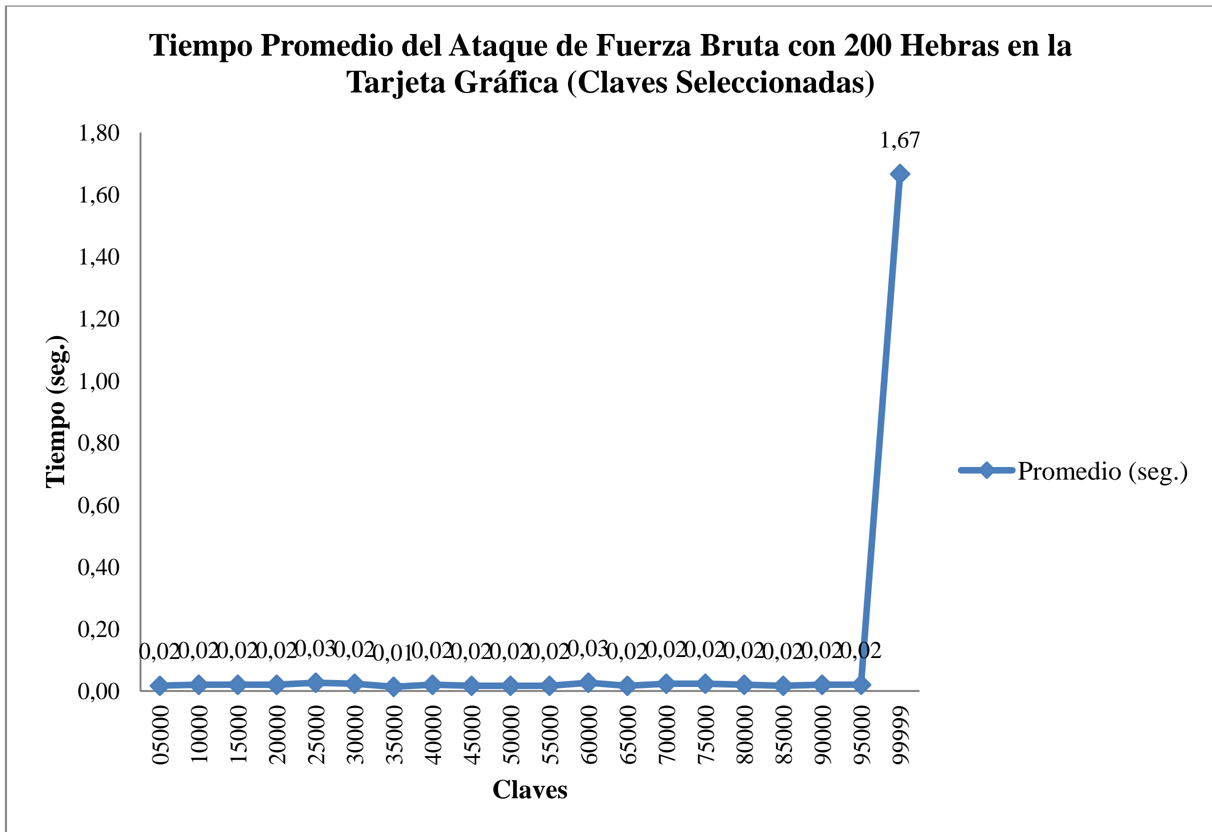

Figura 11.5: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 200 hebras para claves seleccionadas para el archivo textoplano.txt.

Al igual que en las pruebas con un código fuente que emplea 100 hebras, los tiempos que se obtienen son similares, tal como se aprecia en la Figura 11.5. Nuevamente los resultados son de tiempos despreciables, debido a que con 200 hebras, los grupos de claves a probar por cada hebra poseen 500 claves, lo cual deriva en un ataque de fuerza bruta mucho más rápido que con una menor cantidad de hebras.

Los grupos de claves comienzan en claves como 00000, 00500, 01000, y así sucesivamente cubriendo todo el dominio de estas. Tal como ha sucedido en las pruebas con códigos fuente que utilizan 50 y 100 hebras, se da el fenómeno de la clave 65000 que a partir de ella se disminuyen los tiempos, entregando un resultado que en el peor de los casos es de 1,67 segundos, lo cual no es totalmente cierto ya que como se expondrá en las pruebas para claves aleatorias, el tiempo real para una última clave de un segmento de 500 claves es mayor.

Cabe destacar que se han realizado pruebas hasta con 200 hebras, debido a que la tarjeta gráfica posee una cantidad de 336 CUDA Cores (núcleos CUDA), los que procesan cada hebra de forma simultánea. Al tratar de dividir el campo de 100.000 claves se obtenían segmentos de claves con un valor no entero, lo cual incidía en que algunas hebras no procesaran la misma cantidad de claves, provocando problemas como la detención temprana de la ejecución de cada prueba a pesar de que ciertas hebras no terminaran de procesar su

subconjunto de claves. Además, se ha excluido el caso de un código fuente con 250 hebras, lo que si bien al dividir el campo de claves se obtiene un numero entero para cada segmento de claves, no genera una mejora considerable en la reducción de tiempo del ataque de fuerza bruta, por lo cual se ha optado con exponer los casos más representativos, los que se han presentado hasta este punto con tablas y gráficos.

11.3 Ataque de Fuerza Bruta para Claves Aleatorias

Las claves aleatorias utilizadas en estas pruebas son las mismas que se emplearon en los ataques de fuerza bruta en el procesador, puesto que se necesita comparar el desempeño de ambas tecnologías en escenarios iguales.

A continuación se exponen las tablas y gráficos con los resultados de las pruebas realizadas con las claves seleccionadas al azar.

11.3.1 Ataque de Fuerza Bruta con Una Hebra

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
02200	1,66	1,67	1,67	1,67
07556	5,71	5,71	5,72	5,71
11226	8,48	8,48	8,49	8,48
16389	12,39	12,39	12,39	12,39
25999	19,67	19,65	19,65	19,66
29779	22,51	22,51	22,51	22,51
32693	24,71	24,71	24,70	24,71
39665	29,97	29,98	29,98	29,98
44499	33,64	33,64	33,64	33,64
47586	35,97	35,97	35,97	35,97
53551	40,48	40,47	40,47	40,47
59999	45,35	45,35	45,35	45,35
61843	46,74	46,74	46,75	46,74
68429	51,73	51,73	51,73	51,73
76311	57,69	57,68	57,68	57,68
79378	60,01	60,01	60,02	60,01
82010	62,00	61,99	61,99	61,99
86375	65,29	65,29	65,29	65,29
92017	69,56	69,56	69,57	69,56
98797	74,69	74,69	74,69	74,69

Tabla 11.6: Tiempos empleados en cada ataque de fuerza bruta con una hebra para claves aleatorias para el archivo textoplano.txt.

El gráfico asociado con la tabla anterior se presenta a continuación:

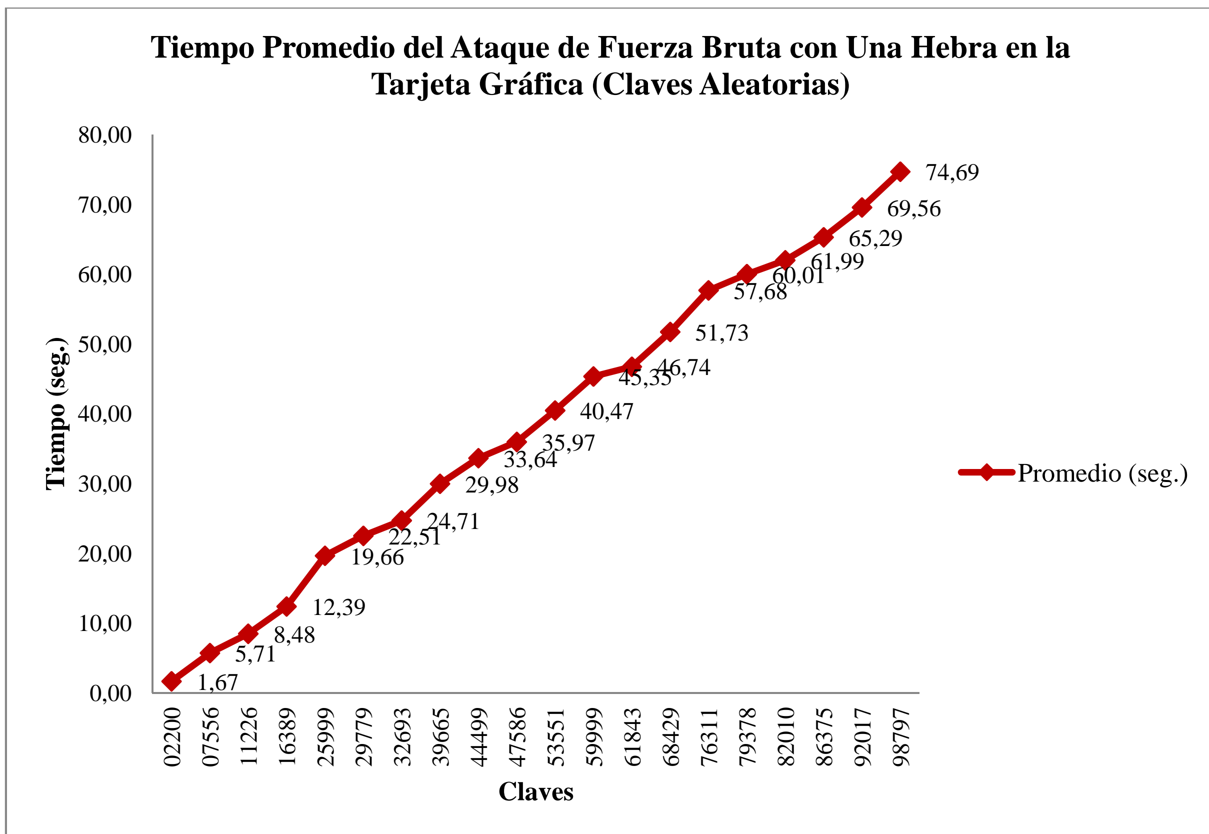

Figura 11.6: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con una hebra para claves aleatorias para el archivo textoplano.txt.

La utilización de claves elegidas aleatoriamente, claramente no incide en los resultados de las pruebas en comparación con aquellas en donde se ha utilizado claves que se han seleccionado específicamente. En la Figura 11.6 se aprecia que la curva tiene un comportamiento similar a la de la Figura 11.1. Nuevamente se obtienen tiempos crecientes a lo largo de las pruebas, y se mantiene la premisa que en esta variante del ataque, en donde las claves se van probando secuencialmente, transcurren más segundos para determinar la clave mientras esta se encuentre más alejada de la clave 00000 en el dominio, lo que viene a corroborar los resultados obtenidos en las pruebas para claves seleccionadas.

11.3.2 Ataque de Fuerza Bruta con 10 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
02200	7,27	7,26	7,27	7,27
07556	24,93	24,94	24,93	24,93
11226	4,05	4,06	4,04	4,05
16389	21,08	21,08	21,07	21,08
25999	19,79	19,79	19,79	19,79
29779	32,26	32,26	32,27	32,26
32693	8,89	8,89	8,89	8,89
39665	31,89	31,88	31,89	31,89
44499	14,84	14,85	14,84	14,84
47586	25,03	25,03	25,03	25,03
53551	11,72	11,72	11,72	11,72
59999	32,99	32,99	32,98	32,99
61843	6,08	6,08	6,08	6,08
68429	27,80	27,80	27,81	27,80
76311	20,83	20,82	20,81	20,82
79378	30,95	30,94	30,94	30,94
82010	6,64	6,63	6,64	6,64
86375	21,03	21,03	21,03	21,03
92017	6,65	6,66	6,66	6,66
98797	29,03	29,03	29,02	29,03

Tabla 11.7: Tiempos empleados en cada ataque de fuerza bruta con 10 hebras para claves aleatorias para el archivo textoplano.txt.

El gráfico asociado con la tabla anterior se presenta a continuación:

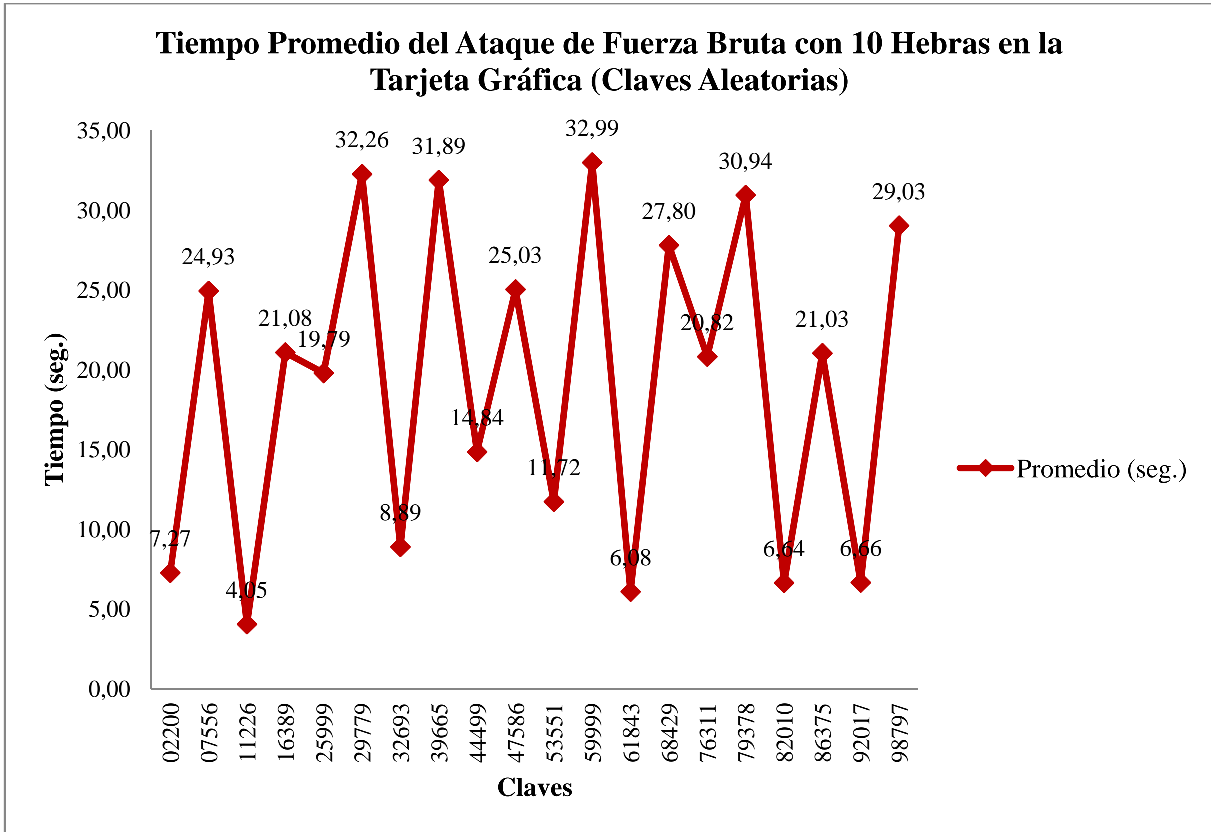

Figura 11.7: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 10 hebras para claves aleatorias para el archivo textoplano.txt.

Tal como se ha expuesto en las pruebas con claves seleccionadas, y comparando la Figura 11.7 con lo que se ha graficado en la Figura 11.2, los resultados de las pruebas arrojan una curva con líneas ascendentes y descendentes, destacando en esta ocasión el peor caso, el cual se presenta cuando se realiza un ataque de fuerza bruta que busca la clave 59999 con un tiempo de 32,99 segundos, el cual ratifica el peor caso en las pruebas para claves seleccionadas, el cual se da con la clave 99999.

11.3.3 Ataque de Fuerza Bruta con 50 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
02200	1,67	1,67	1,67	1,67
07556	12,91	12,91	12,91	12,91
11226	10,18	10,19	10,18	10,18
16389	3,24	3,24	3,24	3,24
25999	16,59	16,59	16,58	16,59
29779	14,76	14,77	14,76	14,76
32693	5,76	5,76	5,76	5,76
39665	13,81	13,82	13,82	13,82
44499	4,15	4,15	4,15	4,15
47586	13,16	13,16	13,16	13,16
53551	12,88	12,88	12,87	12,88
59999	16,58	16,57	16,58	16,58
61843	15,29	15,29	15,28	15,29
68429	2,21	2,20	2,21	2,21
76311	1,62	1,61	1,60	1,61
79378	7,06	7,06	7,07	7,06
82010	0,06	0,06	0,06	0,06
86375	1,93	1,93	1,94	1,93
92017	0,11	0,10	0,10	0,10
98797	4,09	4,09	4,08	4,09

Tabla 11.8: Tiempos empleados en cada ataque de fuerza bruta con 50 hebras para claves aleatorias para el archivo textoplano.txt.

El gráfico asociado con la tabla anterior se presenta a continuación:

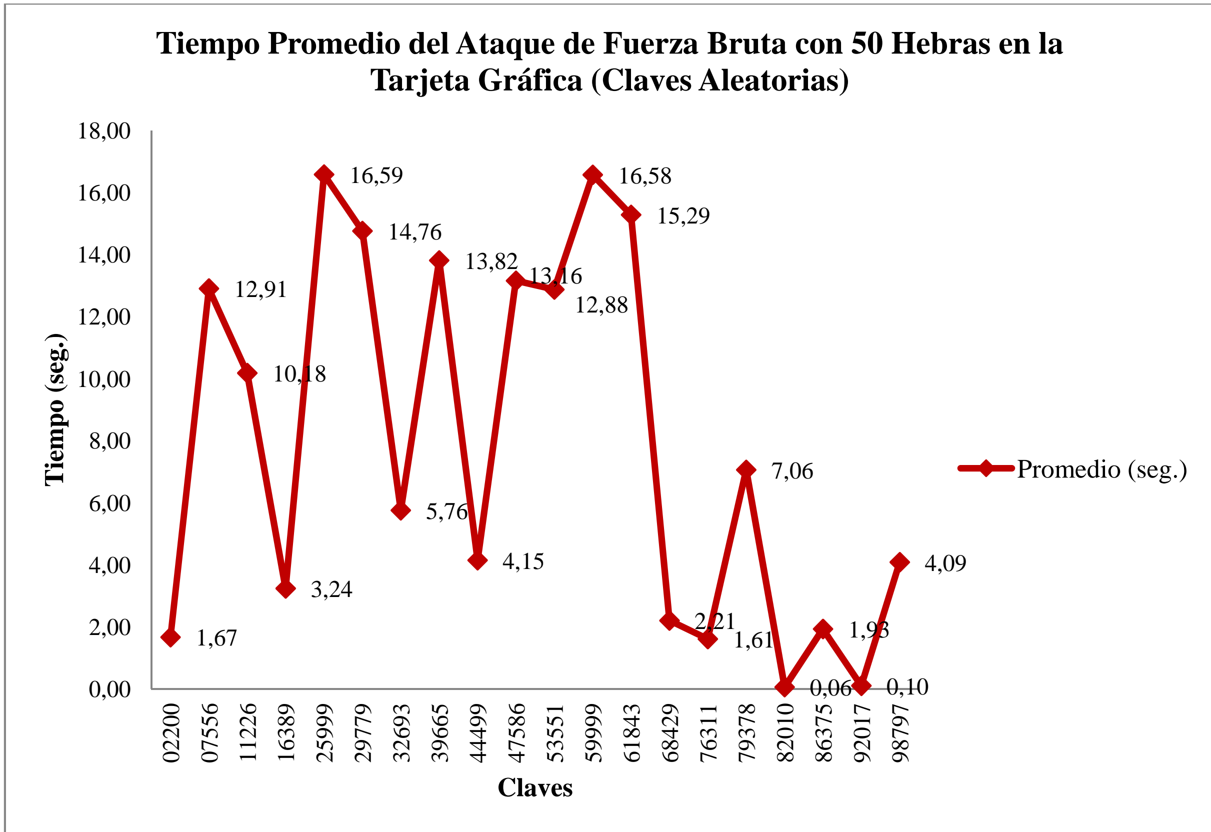

Figura 11.8: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 50 hebras para claves aleatorias para el archivo textoplano.txt.

El ataque de fuerza bruta que emplea 50 hebras, tal como se pudo apreciar en las pruebas para claves seleccionadas, disminuye en un porcentaje importante el tiempo de ejecución de este en comparación con los ataques anteriores.

Los resultados de tiempo de las claves 25999 y 59999 corroboran, y exponen el tiempo real empleado en el peor de los escenarios, debido a que con esta variante del ataque de fuerza bruta, su ejecución sufre comportamientos extraños luego de probar claves más allá de la clave 65000, fenómeno que ya se ha explicado en las pruebas para claves seleccionadas en donde se utilizaron variantes del código fuente que incorporan 50, 100 y 200 hebras. Por lo anteriormente explicado, se debe tomar en cuenta que el mayor tiempo que se puede emplear en determinar una clave en cualquier punto dentro del dominio de claves es de 16,59 segundos aproximadamente y no los 10,24 segundos que entregan las pruebas con claves seleccionadas. La diferencia entre los valores de tiempo adecuados y aquellos que se ven afectados por el fenómeno luego de la clave 65000 es de un poco más de 6 segundos, lo que no es relevante para las pruebas realizadas pero puede ser perjudicial a la hora de emplear archivos cifrados de una mayor envergadura o un campo de claves de mayor complejidad y/o mayor tamaño.

11.3.4 Ataque de Fuerza Bruta con 100 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
02200	1,74	1,74	1,74	1,74
07556	4,79	4,79	4,79	4,79
11226	1,96	1,97	1,97	1,97
16389	3,36	3,37	3,36	3,36
25999	8,59	8,58	8,57	8,58
29779	6,71	6,71	6,70	6,71
32693	5,97	5,96	5,97	5,97
39665	5,73	5,72	5,73	5,73
44499	4,29	4,30	4,29	4,29
47586	5,04	5,04	5,04	5,04
53551	4,74	4,73	4,74	4,74
59999	8,58	8,58	8,58	8,58
61843	7,24	7,24	7,25	7,24
68429	3,71	3,72	3,71	3,71
76311	2,69	2,70	2,69	2,69
79378	3,27	3,27	3,27	3,27
82010	0,11	0,11	0,10	0,11
86375	3,24	3,24	3,24	3,24
92017	0,16	0,16	0,16	0,16
98797	1,67	1,67	1,66	1,67

Tabla 11.9: Tiempos empleados en cada ataque de fuerza bruta con 100 hebras para claves aleatorias para el archivo textoplano.txt.

El gráfico asociado con la tabla anterior se presenta a continuación:

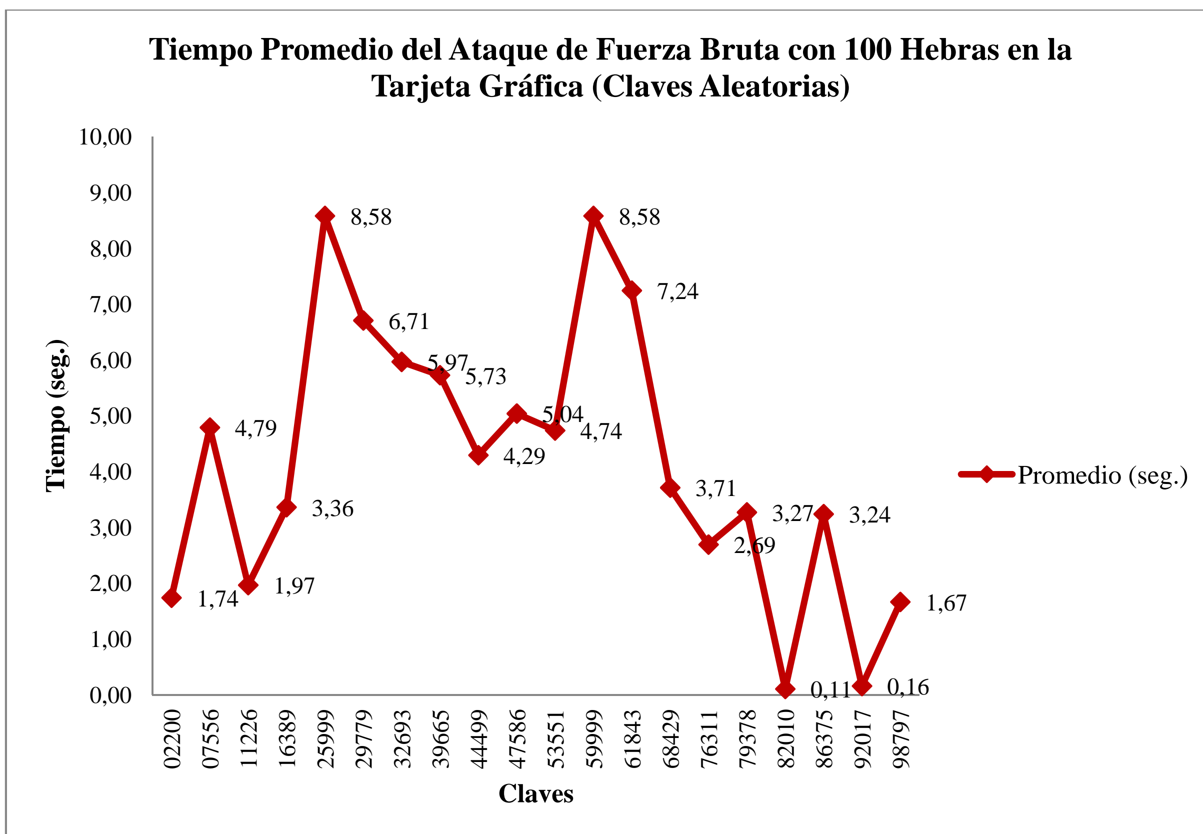

Figura 11.9: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 100 hebras para claves aleatorias para el archivo textoplano.txt.

Al igual que en las pruebas con la variante del código fuente que emplea 50 hebras, las claves 25999 y 59999 revelan la verdadera utilización de segundos en que el ataque de fuerza bruta incurre para poder obtener la clave que descifra el archivo. Comparando los resultados de tiempo con los obtenidos en las pruebas para claves seleccionadas, se aprecia el tiempo máximo real que es necesario para encontrar una clave en el dominio siendo este de 8,58 segundos aproximadamente, superior a los 2,08 segundos determinados, el que es afectado por el comportamiento extraño después de la clave 65000. Nuevamente el resultado real, supera en aproximadamente 6 segundos al resultado entregado por las pruebas para claves seleccionadas.

11.3.5 Ataque de Fuerza Bruta con 200 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
02200	1,83	1,83	1,83	1,83
07556	0,53	0,53	0,53	0,53
11226	2,06	2,05	2,06	2,06
16389	3,50	3,51	3,51	3,51
25999	4,50	4,50	4,49	4,50
29779	2,55	2,53	2,54	2,54
32693	1,76	1,77	1,77	1,77
39665	1,51	1,52	1,51	1,51
44499	4,49	4,50	4,49	4,49
47586	0,80	0,81	0,80	0,80
53551	0,48	0,48	0,49	0,48
59999	4,49	4,48	4,48	4,48
61843	3,10	3,11	3,11	3,11
68429	3,88	3,87	3,88	3,88
76311	2,82	2,83	2,82	2,82
79378	3,42	3,42	3,42	3,42
82010	0,11	0,12	0,11	0,11
86375	3,40	3,40	3,40	3,40
92017	0,18	0,17	0,18	0,18
98797	0,99	1,00	1,00	1,00

Tabla 11.10: Tiempos empleados en cada ataque de fuerza bruta con 200 hebras para claves aleatorias para el archivo textoplano.txt.

El gráfico asociado con la tabla anterior se presenta a continuación:

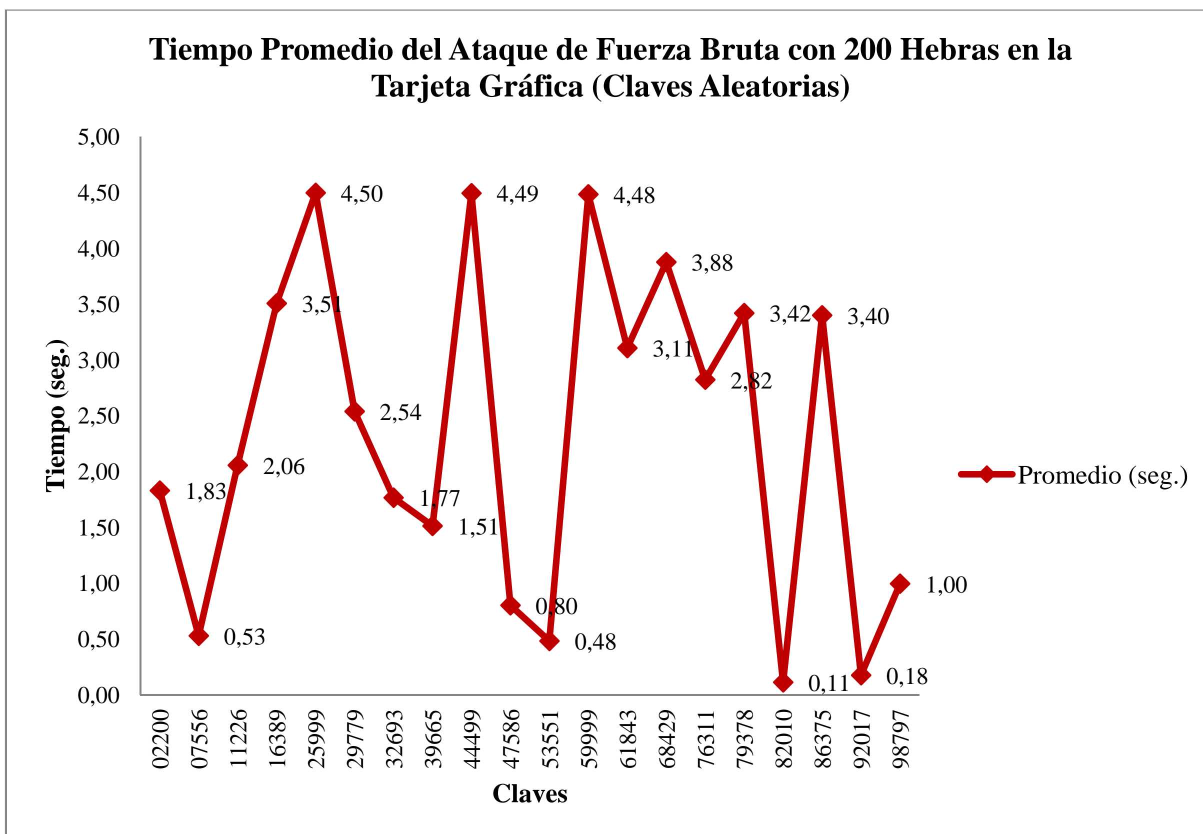

Figura 11.10: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 200 hebras para claves aleatorias para el archivo textoplano.txt.

A lo largo de las pruebas, se ha podido corroborar resultados y comportamientos poco usuales en las variantes del ataque de fuerza bruta. Para el presente caso, que emplea 200 hebras, no ha sido excepción de que más de una clave revele el tiempo real necesario para que la prueba pueda determinarla. Las claves 25999, 44499 y 59999, demuestran que la cantidad de segundos que transcurren desde que se inicia el ataque, hasta que este encuentra la clave que permite descifrar el archivo, es de 4,50 segundos aproximadamente, el cual es mayor a los 1,67 segundos obtenidos en las pruebas para claves seleccionadas, siendo el primero, el tiempo en que se debe incurrir para tener un ataque de fuerza bruta exitoso. Por consiguiente, se tiene una diferencia de tiempo de aproximadamente 3 segundos entre el valor real y el valor afectado por el fenómeno de la clave 65000.

12 Conclusiones

La utilización de tarjetas gráficas para el procesamiento de datos de gran envergadura es un enfoque que en la actualidad se ha demostrado eficiente en varias áreas de conocimiento, es por esto que es aconsejable que los profesionales relacionados con la informática y otros de áreas afines a esta, se interioricen en esta tecnología desarrollando investigaciones para conocerla, entenderla y aplicarla al universo de problemas que pueden ser resueltos de una forma más eficiente con su uso. Tecnologías como esta ayudan a resolver problemas complejos en diferentes áreas, lo cual se ve claramente reflejado en los proyectos desarrollados en donde, en la mayoría de los casos, se ve un aumento en el rendimiento de las implementaciones de diferentes aplicaciones, llevando a estas a desarrollar sus tareas en menores periodos de tiempo.

El desarrollo exitoso de proyectos y aplicaciones comerciales, ha generado que esta tecnología se vuelva atractiva para ser empleada en diferentes problemas, que en muchos casos ya están resueltos por aplicaciones que si bien funcionan correctamente, no son del todo óptimas simplemente porque se encuentran limitadas por la capacidad del hardware existente.

La tecnología de las GPUs, desde el punto de vista de su arquitectura lógica y física, se encuentra en una constante evolución, similar a lo que sucede en el presente con los demás componentes hardware de un computador. Por otro lado el avance en la programación de estas ha sido considerable con la llegada de CUDA. La documentación es bastante adecuada y actualizada, constantemente NVIDIA se preocupa de ir corrigiendo errores de su entorno de programación, liberando nuevas versiones y entregando la documentación asociada a estos, para que los programadores se encuentren al tanto de las novedades y correcciones realizadas.

Tal como propone NVIDIA, y como se ha podido corroborar a lo largo de la presente proyecto, CUDA permite ser abordado de una manera un tanto más sencilla al utilizar el lenguaje de programación C, cumpliendo con la premisa de una baja curva de aprendizaje para los programadores, ya que se puede afirmar que prácticamente cualquier desarrollador está familiarizado con el lenguaje C. Además, parte de esta premisa se cumple gracias a la inclusión de nuevas funcionalidades que utilizan la misma sintaxis de C. Por otro lado, la alta utilización de hebras que se puede conseguir en los algoritmos es un plus de la tecnología, permitiendo descomponer problemas complejos en sub-problemas o grandes volúmenes de datos en fragmentos más pequeños, para ser ejecutados y procesados de una manera más eficiente y rápida, ventajas que en la actualidad son requeridas ya que día a día se presentan problemas más complejos que deben ser resueltos, o se deben procesar mayores volúmenes de datos con la tecnología existente, que muchas veces no es suficiente para lograr de buena forma estas tareas.

El área de conocimiento y aplicación de la tecnología en el presente proyecto, en la actualidad expone una gran variedad de algoritmos criptográficos, cada uno con características propias que los hacen ser más o menos fuertes a la hora de resistir diferentes ataques de criptoanálisis. Obviamente un algoritmo que no resiste a algún tipo de ataque es desechado por la comunidad debido a que ya no presenta seguridad alguna para resguardar la información de una organización. Tal como existe una gran variedad de algoritmos de diferentes clases,

también existe una gran cantidad de tipos de ataques de criptoanálisis que permiten probar la robustez de los algoritmos que un profesional puede elegir a la hora de proteger la información sensible de una empresa, por ende es importante comprender que la criptografía y el criptoanálisis no son áreas separadas, sino que en conjunto permiten dar solución a los problemas de protección de información y a su vez probar si los sistemas criptográficos instaurados son lo suficientemente eficaces para repeler a futuros atacantes, por lo tanto, un profesional del área de seguridad informática debe comprender esta relación para así fortalecer los posibles puntos débiles de un sistema criptográfico que se pueda adoptar en una determinada organización.

Dentro de la gran variedad de ataques de criptoanálisis que se encuentran en la actualidad, el ataque de fuerza bruta claramente es el más ineficiente debido a su costo en recursos y tiempo de ejecución, y muchas veces es el menos fructífero a la hora de probar determinados algoritmos criptográficos, ya que cuando estos son diseñados se espera que resistan a ataques de este tipo. Por otro lado, muchas veces estos ataques son inviables debido al largo de clave que utiliza un determinado algoritmo, razón por la cual muchos criptoanalistas prefieren aplicar otras técnicas que permiten reducir los tiempos empleados en un ataque contra un algoritmo criptográfico, y que a su vez entreguen resultados exitosos.

Las pruebas del ataque de fuerza bruta en el procesador confirmaron variadas características del algoritmo de cifrado de flujo RC4, principalmente su rapidez de trabajo a la hora de cifrar y descifrar archivos, como también su reducido tamaño lo que hace que se cumpla a cabalidad la premisa de ser un cifrador de flujo de alta velocidad. Por otra parte, las reducciones en varias características del algoritmo, como lo es el largo de clave, el que fue solamente de 40 bits (5 caracteres) y el uso de números solamente en las claves, permitieron que los ataques fueran exitosos con un consumo de tiempo reducido, de otra forma las pruebas hubiesen tomado horas, días o mucho más que eso, siendo infructíferas y no aplicables. Las pruebas se realizaron en un computador común, que puede ser encontrado en cualquier hogar, por lo tanto, eso justifica que no se necesita un "super-computador" para realizar un ataque de fuerza bruta, siempre y cuando sea en un ambiente controlado, y con los supuestos y reducciones necesarias para que las pruebas sean exitosas y se lleven a cabo rápidamente. Por consiguiente, a la hora de implementar un sistema criptográfico robusto, se debe tener en cuenta que con un largo de clave pequeño, es fácilmente quebrantable el sistema y el atacante puede obtener la clave en un par de segundos o minutos. Es importante tomar el peso y responsabilidad de esta observación, ya que no es una tarea fácil implementar un sistema criptográfico robusto, pero a la hora de incluir uno o más algoritmos criptográficos en un determinado sistema, debe hacerse con un máximo de cuidado, pensando en los posibles ataques que se puedan efectuar sobre él y por sobre todas las cosas, nunca se deben usar largos de claves pequeños. Hace algunos años atrás, 128 bits de clave era una cantidad considerable para mantener seguro un sistema, en la actualidad, con los componentes hardware que existen, claramente se pone en duda la capacidad de protección de una clave de 128 bits, por lo tanto se recomiendan claves mucho mayores, de 192 bits o 256 bits a lo menos para mantener un sistema seguro por algún buen tiempo.

Profundizando en los resultados de las pruebas, y sus conclusiones asociadas, se puede inferir variadas ideas, que hacen inclinar la balanza hacia el procesador o la tarjeta gráfica, dependiendo del escenario en el cual se vean envueltas las pruebas.

El primer escenario postula que si el procesador y la GPU compiten directamente por la obtención de claves en un menor tiempo de ejecución mediante un ataque de fuerza bruta, es decir, sin que la tarjeta gráfica utilice hebras ni sus múltiples núcleos, se puede determinar que el CPU sobrepasa ampliamente en la tarea a la GPU, debido a que como se ha podido apreciar en los gráficos y tablas, los tiempos obtenidos por el procesador son aproximadamente 16 veces más bajos que los entregados por la tarjeta gráfica. En pruebas posteriores, en donde la GPU realiza sus cálculos utilizando 10, 50 y 100 hebras, esta sigue sin poder competir contra el procesador, ya que este la sigue sobrepasando y obteniendo mejores resultados de tiempo, aunque las diferencias entre ellos ya no son tan notorias como en el caso inicial. Cuando la tarjeta gráfica emplea 100 hebras, esta se acerca al procesador en desempeño, donde este último obtiene tiempos que solo son 2 veces más bajos que los de la GPU.

El segundo escenario se encuentra al comparar las pruebas ejecutadas en el procesador y aquellas realizadas en la tarjeta gráfica con el código fuente que emplea 200 hebras. En esta comparación la GPU supera por solo unas décimas de segundo a la CPU, permitiendo observar que la tarjeta gráfica si es más veloz que el procesador en este tipo de ataques y demuestra que la utilización de las características claves de la GPU, como lo son, las hebras y sus múltiples núcleos, pueden obtener resultados de mayor eficiencia que los que se pueden determinar con las pruebas en el procesador. Dentro de esta conclusión, es importante destacar un punto clave: es claro que la tarjeta gráfica supera al procesador en las pruebas, pero para todas las claves comprendidas dentro del dominio y segmentadas en rangos de 500 de estas, el tiempo mayor para encontrarlas es el mismo, es decir, que el tiempo de búsqueda de claves en la GPU cercanas a las claves finales de cada intervalo de 500 de estas, son similares a los tiempos de búsqueda en el procesador de las claves cercanas a la clave final de todo el campo de claves. Para clarificar la idea, se propone el siguiente ejemplo: al realizar el ataque de fuerza bruta en la tarjeta gráfica y buscar la clave 00499, el tiempo resultante es de 4,50 segundos. Si se busca la clave 99999 en el ataque de fuerza bruta en el procesador, se obtiene un tiempo de 4,54 segundos, por lo tanto, es posible concluir que la GPU es más eficiente al momento de buscar claves que se encuentran más lejanas a la clave inicial del dominio total de claves, ya que no superará un tiempo determinado, mientras que el CPU es más eficiente a la hora de buscar claves cercanas a la clave inicial del dominio de claves. Lo explicado anteriormente se debe a que las hebras en la tarjeta gráfica se ejecutan simultáneamente y de forma sincronizada, por lo cual cada una de las 200 hebras toman el mismo tiempo, entre 0 y 4,50 segundos, en encontrar una clave posicionada en cualquier punto del dominio.

A partir de lo explicado en el párrafo anterior, es posible concluir que al utilizar hebras que se ejecutan en forma simultánea y sincronizada, es posible obtener rangos de tiempos en las pruebas que permiten saber de antemano cuantos segundos, minutos, horas, etc. se deben invertir para determinar una clave. Basta, en este caso, con realizar las pruebas en las primeras 500 claves para saber cuanto tiempo se debe invertir en buscar una clave posicionada en cualquier punto del dominio de claves. Muy por el contrario, en las pruebas ejecutadas en el procesador, no es posible determinar a priori cuanto tiempo tomará la búsqueda de una clave en el campo de claves, siendo estos tiempos crecientes cada vez que este aumenta en cantidad de claves y/o estas son más complejas.

Derivado de las últimas conclusiones, es posible determinar que para ciertas pruebas, o en ciertas situaciones el procesador es más eficiente que la tarjeta gráfica, y en otras, la GPU

supera en capacidad de cálculo a la CPU. En resumen, si en el procesador y en la tarjeta gráfica se ocupan algoritmos similares en sus características de diseño y codificación, claramente el CPU superará en los ataques de fuerza bruta a la GPU puesto que el primero posee una mayor frecuencia de reloj y no debe realizar una copia de datos entre memorias, lo que le permite realizar más cálculos por segundo. Es claro que si se utiliza un procesador de mayor velocidad, estas diferencias en las pruebas aumentarán. Por otro lado, si los algoritmos utilizados difieren en su diseño y codificación, principalmente porque el código fuente que opera en la tarjeta gráfica incorpora una gran cantidad de hebras y utiliza mejor sus recursos, esta superará en eficiencia y capacidad de cálculo al procesador, obteniendo tiempos menores en las pruebas y permitiendo saber con acciones iniciales, cuales serán los posibles tiempos que se esperarán a lo largo de todos los ataques de fuerza bruta. Si en este tipo de pruebas, se utilizan GPUs con una mayor cantidad de CUDA cores, es posible reducir aún más los tiempos de búsqueda de claves, debido a que con una mayor cantidad de núcleos en la tarjeta gráfica es posible dividir el dominio total de claves en rangos más pequeños y acotados, tal como se aprecia cuando se emplean 200 hebras.

En el futuro se espera que el trabajo realizado en el presente proyecto permita seguir investigando y profundizando sobre la tecnología de las tarjetas gráficas, realizando pruebas sobre algoritmos criptográficos simétricos de cifrado de bloques y algoritmos criptográficos asimétricos, como también funciones Hash, para tener una mayor variedad de resultados y se pueda concluir de una forma más exacta en que campo del conocimiento de la criptografía se pueden desempeñar de forma óptima las GPUs. Por otro lado, existen otros tipos de ataques que es posible implementar tanto en el procesador como en la tarjeta gráfica, que pueden permitir determinar cual es la tecnología que mejor se ajusta para un determinado tipo de prueba sobre un algoritmo criptográfico. Se espera que el presente proyecto sea el punto de partida para que otros alumnos se interesen en desarrollar temas afines con las tarjetas gráficas, no solo en el campo de la criptografía o la seguridad informática, sino también en temas de procesamiento de datos u otros, siempre mirado desde el punto de vista que las GPUs son, al igual que el CPU, una fuente tecnológica que permite manipular, procesar y entregar resultados de diferentes tipos de datos, siempre y cuando se utilicen las características propias que provee la tecnología, como lo es el manejo de una gran cantidad hebras de forma simultánea y sincronizada, y la utilización de múltiples núcleos, propios del chip de la tarjeta gráfica.

Anexo A. Código Fuente del Ataque de Fuerza Bruta en el Procesador

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/*************** Prototipos de las funciones ***************/
void swap(unsigned char *s, int i, int j);
void rc4_init(unsigned char *key, int key_length);
unsigned char rc4_output();
void brute_force(FILE *txt, FILE *rc4, int lt, int cifrar, unsigned char *texto);
/*************** Fin Prototipos de las funciones ***************/

/*************** Main ***************/
int main(int argc, unsigned char *argv[]) {
    FILE *txt, *rc4;
    int verbose = 0, cifrar = 1;
    int p, y, x = 0, lc = 5, lt = 0, ii;
    unsigned char clave[257]; // Tamaño máximo de clave = 256 + el carácter nulo \0
    unsigned char texto[1025]; // Tamaño máximo de texto = 1024 + el carácter nulo \0
    unsigned char c; // Variable que guarda el carácter cifrado del XOR

    for (p = 0; p < argc; p++) {
        if (!strcmp(argv[p], "-v")) {
            verbose++;
        } else if (!strcmp(argv[p], "-c")) {
            cifrar = 1;
        } else if (!strcmp(argv[p], "-d")) {
            cifrar = 0;
        } else if (!strcmp(argv[p], "-h")) {
            printf( "Uso : %s [-c] [-d] [-v] [-h]\n"
                    "    -c Cifrar archivo txt/*.txt -> rc4/*.rc4\n"
                    "    -d Descifrar archivo rc4/*.rc4 -> txt/d*.txt\n"
                    "    -v muestra detalles\n"
                    "    -h muestra ayuda\n"
                    );
        }
    }

    if (cifrar == 1) {
        printf("\n- Ingrese la clave de 5 caracteres numéricos [0-9] para cifrar:  ");
        scanf("%s", clave);
        clave[lc] = '\0';
    }
```

```
	if (cifrar == 1) { // Se define previamente que el formato del nombre del archivo a cifrar es txt/textoplano.txt
		txt = fopen("txt/textoplano.txt", "r");
		rc4 = fopen("rc4/textocifrado.rc4", "w");
	} else { // Se define previamente que el formato del nombre del archivo a descifrar es rc4/textocifrado.rc4
		txt = fopen("rc4/textocifrado.rc4", "r");
		rc4 = fopen("rc4d/textoplano.txt", "w");
	}

	while (!feof(txt)) { // Se almacena en el arreglo texto[1025] el contenido del archivo a cifrar/descifrar
		texto[lt++] = (unsigned char) fgetc(txt);
	}
	texto[--lt] = '\0';
	fclose(txt);

	if (cifrar == 1) { // Muestra por pantalla la clave a utilizar en el cifrado/descifrado y el largo de esta
		printf("\nCIFRAR: \"%s\" [%d] 'txt/textoplano.txt' -> 'rc4/textocifrado.rc4'\n\n", clave, lc);
	}
	if (verbose > 0) { // Permite mostrar detalles del cifrado/descifrado del archivo
		if (cifrar == 1) {
			printf ("texto [%d] : %s\n", lt, texto);
		} else {
			printf ("texto [%d] : ", lt);
			for (ii=0; ii < lt; ii++) {
				printf("%02x", texto[ii]);
			}
			printf("\n");
		}
	}

	if (cifrar == 1) { // Inicializa el algoritmo para el cifrado
		rc4_init(clave, 5);

		if (verbose > 1) {
			if (cifrar == 1) {
				printf ("texto cifrado : ");
			} else {
				printf ("texto descifrado : ");
			}
		}
```

```
        for (y = 0; y < lt; y++) {  // Se ejecuta el cifrado del texto y lo graba a un archivo .rc4
            c = texto[y] ^ rc4_output();
            fprintf(rc4, "%c", c);
            if (verbose > 1) {
                if (cifrar == 1) {
                    printf ("%02x", c);
                } else {
                    printf ("%c", c);
                }
            }
        }
    } else {  // Se ejecuta el descifrado con el ataque de fuerza bruta incluido
        brute_force(txt, rc4, lt, cifrar, texto);  // Se llama a la función para ejecutar el descifrado con el ataque de fuerza bruta
    }
    if (verbose > 1) {
        printf("\n");
    }
    if (verbose > 0) {
        printf ("\n");
    }
    fclose(rc4);
    return 0;
}
/*************** Fin Main ***************/

/*************** Funciones del algoritmo RC4 ***************/
unsigned char S[256];
int i, j;
/*** Swap ***/
void swap(unsigned char *s, int i, int j) {
    unsigned char temp = s[i];
    s[i] = s[j];
    s[j] = temp;
}

/*** KSA - Key-Scheduling Algorithm ***/
void rc4_init(unsigned char *key, int key_length) {
    for (i = 0; i < 256; i++)
        S[i] = i;
     for (i = j = 0; i < 256; i++) {
        j = (j + key[i % key_length] + S[i]) & 255;
        swap(S, i, j);
    }
    i = j = 0;
}
```

```
/*** PRGA - Pseudo-Random Generation Algorithm ***/
unsigned char rc4_output() {
    i = (i + 1) & 255;
    j = (j + S[i]) & 255;
    swap(S, i, j);
    return S[(S[i] + S[j]) & 255];
}
/*************** Fin de las funciones del algoritmo RC4 ***************/

/*************** Ataque de Fuerza Bruta sobre el archivo cifrado con el algoritmo RC4 ***************/
void brute_force(FILE *txt, FILE *rc4, int lt, int cifrar, unsigned char *texto) {
    unsigned char clave[257];
    unsigned char descifrado[1025];
    unsigned char c; // Variable para guardar cada carácter de la clave
    unsigned char alfabeto[56] = {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u',
'v','w','x','y','z','A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','
Z',' ',',','.','\0'};
    int i, y, c1, c2, lc = 5, flag = 0, ii, gen_key = 0, valor;
    time_t Comienzo_Seg, Final_Seg; // Variables para capturar la hora de inicio y término del ataque
    struct tm *TiempoComienzoPtr, *TiempoFinalPtr; // Punteros para tomar el tiempo inicial y el tiempo final del ataque
    clock_t comienzo; // Variable para tomar el tiempo con CLOCKS_PER_SEC

    Comienzo_Seg = time(NULL); // Obtiene la hora inicial en segundos cuando se inicia el ataque de fuerza bruta
    TiempoComienzoPtr = gmtime(&Comienzo_Seg); // Convierte los segundos a la hora GMT inicial, por lo tanto tiene un desfase de 4 horas (GMT-4 da la hora local)
    Comienzo = clock(); // Obtiene el "clock inicial" cuando se inicia el ataque de fuerza bruta

    printf("\n\n|----------> INICIO DEL ATAQUE DE FUERZA BRUTA SOBRE RC4 <----------|\n");
    printf("\n> Hora de comienzo del ataque: %s", asctime(TiempoComienzoPtr));

    while (flag == 0) {
        valor = gen_key; // La asignación permite no perder el valor de la clave que es dividida hasta llegar a un residuo igual a 0
        for (i = 0; i < 5; i++) { // En el bucle se genera una nueva clave en cada iteración
            clave[4-i] = '0' + (valor % 10);
            valor = valor / 10;
        }
        clave[lc] = '\0'; // Con \0 se marca el final de la clave (string)

        rc4_init(clave, lc); // Inicializa el algoritmo para el descifrado
```

```
for (y = 0; y < lt; y++) { // Se ejecuta el descifrado
    descifrado[y] = texto[y] ^ rc4_output(); // El texto cifrado XOR la salida de la
función PRGA da el descifrado (texto plano)
}

for (c1 = 0; c1 < lt; c1++) { // Se compara el descifrado, carácter por carácter, con
el alfabeto para saber si se logró encontrar la clave
    for (c2 = 0; c2 < 56; c2++) {
        if (descifrado[c1] == alfabeto[c2]) {
            break;
        }
    }
    if (c2 == 56) {
        break;
    }
}

if (c1 == (lt-1)) { // Condición para escribir el texto plano en el archivo .txt y
determinar la clave que descifra el archivo .rc4
    for (y = 0; y < lt; y++) {
        fprintf(rc4, "%c", descifrado[y]);
    }

    printf("\n> La CLAVE para descifrar el Archivo RC4 es: \"%s\"", clave);

    Final_Seg = time(NULL);
    TiempoFinalPtr = gmtime(&Final_Seg); // Convierte los segundos en la hora
GMT final, por lo tanto tiene un desfase de 4 horas (GMT - 4 da la hora local)

    printf("\n\n> Hora de término del ataque: %s", asctime(TiempoFinalPtr));
    printf("\n> Número de segundos transcurridos desde el comienzo hasta el final:
\"%.6f\" segundos.\n", difftime(Final_Seg, Comienzo_Seg));
    printf("\n> Número de segundos transcurridos desde el comienzo hasta el final
(clocks): \"%.6f\" segundos.\n\n", (clock()-comienzo)/(double)CLOCKS_PER_SEC);

    printf("|----------> FIN DEL ATAQUE DE FUERZA BRUTA SOBRE RC4 <--
--------|\n\n\n");
    flag = 1; // Con la variable en 1 finaliza el bucle while
}
gen_key++;  // Aumenta en 1 la variable generadora de claves para la siguiente
iteración
        }
    }
    /************** Fin de la función de ataque de fuerza bruta sobre el archivo cifrado
con RC4 ***************/
```

Anexo B. Código Fuente del Ataque de Fuerza Bruta en la Tarjeta Gráfica

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <time.h>
#include <cuda.h>

#define MAX_NTHREADS 200 // Se define inicialmente que se utilizarán 200 hebras. Cambiando este valor se puede utilizar una menor o mayor cantidad de hebras

/*************** Prototipos de las funciones ***************/
__device__ void rc4_init(int id, unsigned char *key, int key_length);
__device__ unsigned char rc4_output(int id);
__global__ void brute_force(unsigned char *texto_dev, int lt, unsigned char *clave_dev, unsigned char *descifrado_dev, unsigned char *tid_dev);
/*************** Fin Prototipos de las funciones ***************/

/*************** Main **************/
int main(int argc, char *argv[]) {
    FILE *txt, *rc4; // Punteros para la apertura de archivos
    int y, lt = 0, tid;
    size_t long_clave = sizeof(unsigned char)*257; // Tamaño máximo de clave = 256 + el carácter nulo \0
    size_t long_texto = sizeof(unsigned char)*1025; // Tamaño máximo de texto = 1024 + el carácter nulo \0

    unsigned char *clave_host; // Puntero de la Clave en el host (CPU)
    unsigned char *texto_host; // Puntero del Texto a cifrar/descifrar en el host (CPU)
    unsigned char *descifrado_host; //  Puntero del Texto Descifrado por el ataque en el host (CPU)
    unsigned char *tid_host; // Puntero de la ID de la hebra en el host (CPU)

    unsigned char *clave_dev; // Puntero de la Clave que logra descifrar, mediante el ataque de fuerza bruta, el texto cifrado en el device (GPU)
    unsigned char *texto_dev; // Puntero del Texto a descifrar en el device (GPU)
    unsigned char *descifrado_dev; // Puntero del Texto Descifrado por el ataque de fuerza bruta en el device (GPU)
    unsigned char *tid_dev; // Puntero de la ID de la hebra en el device (GPU)

    time_t Comienzo_Seg, Final_Seg; // Variables para capturar la hora de inicio y término del ataque
    struct tm *TiempoComienzoPtr, *TiempoFinalPtr; // Punteros para tomar el tiempo inicial y el tiempo final del ataque
    clock_t comienzo; // Variable para tomar el tiempo con CLOCKS_PER_SEC
```

```
	clave_host = (unsigned char *)malloc(MAX_NTHREADS * long_clave); // Reservar
memoria en el host (CPU) para todos los punteros que se utilizarán
	texto_host = (unsigned char *)malloc(long_texto);
	descifrado_host = (unsigned char *)malloc(MAX_NTHREADS * long_texto);
	tid_host = (unsigned char *)malloc(2);
	tid_host[0] = 255;
	tid_host[1] = 0; // Inicialización de la variable "encontrado"

	cudaMalloc((void **)&clave_dev, MAX_NTHREADS * long_clave); // Reservar
memoria en el device (GPU) para todos los punteros que se utilizarán
	cudaMalloc((void **)&texto_dev, long_texto);
	cudaMalloc((void **)&descifrado_dev, MAX_NTHREADS * long_texto);
	cudaMalloc((void **)&tid_dev, 2);

	rc4 = fopen("rc4/textocifrado.rc4", "r"); // Se asume que el formato del nombre del
archivo a descifrar es rc4/textocifrado.rc4
	txt = fopen("rc4d/textoplano.txt", "w");

	while (!feof(rc4)) { // Se almacena en el puntero texto_host el contenido del archivo a
descifrar hasta que se encuentre un EOF
	    texto_host[lt++] = (unsigned char)fgetc(rc4);
	}
	texto_host[--lt] = '\0'; // Marca el final del texto a descifrar obtenido desde el archivo
.rc4
	fclose(rc4);

	Comienzo_Seg = time(NULL);
	TiempoComienzoPtr = gmtime(&Comienzo_Seg); // Convierte los segundos a la hora
GMT inicial, por lo tanto tiene un desfase de 4 horas (GMT-4 da la hora local)
	comienzo = clock();

	printf("\n\n|----------> INICIO DEL ATAQUE DE FUERZA BRUTA SOBRE RC4
<----------|\n");

	printf("\n> Hora de comienzo del ataque: %s", asctime(TiempoComienzoPtr));

	cudaMemcpy((void*)texto_dev,(void*)texto_host,long_texto,
cudaMemcpyHostToDevice); // Transfiere el contenido de texto_host desde el host (CPU)
hacia el device (GPU)

	cudaMemcpy((void*)tid_dev,(void*)tid_host,2,cudaMemcpyHostToDevice);
// Transfiere el contenido de tid_host desde el host (CPU) hacia el device (GPU)

	brute_force<<<1,MAX_NTHREADS>>>(texto_dev, lt, clave_dev, descifrado_dev,
tid_dev); // Se llama a la función (KERNEL CUDA) para ejecutar el descifrado con el ataque
de fuerza bruta
```

```
cudaMemcpy((void*)clave_host,(void*)clave_dev,MAX_NTHREADS * long_clave, cudaMemcpyDeviceToHost); // Transfiere el resultado de clave_dev desde el device hacia el host, donde clave_dev es la clave encontrada por el ataque de fuerza bruta

cudaMemcpy((void*)descifrado_host, (void*)descifrado_dev, MAX_NTHREADS * long_texto,cudaMemcpyDeviceToHost); // Transfiere el resultado de descifrado_dev desde el device hacia el host, donde descifrado_dev es el texto claro encontrado por el ataque de fuerza bruta

cudaMemcpy((void *)tid_host, (void *)tid_dev, 2, cudaMemcpyDeviceToHost);

tid = tid_host[0] - '0';

Final_Seg=time(NULL);
TiempoFinalPtr=gmtime(&Final_Seg); // Convierte los segundos en la hora GMT final, por lo tanto tiene un desfase de 4 horas (GMT - 4 da la hora local)

printf("\n> La CLAVE para descifrar el Archivo RC4 es: \"%.5s\"", &clave_host[257 * tid]); // Inicio del reporte final

printf("\n\n> Hora de término del ataque: %s", asctime(TiempoFinalPtr));

printf("\n> Número de segundos transcurridos desde el comienzo hasta el final: \"%.6f\" segundos.\n", difftime(Final_Seg, Comienzo_Seg));

printf("\n> Número de segundos transcurridos desde el comienzo hasta el final (clocks): \"%.6f\" segundos.\n\n", (clock()-comienzo)/(double)CLOCKS_PER_SEC);

printf("|----------> FIN DEL ATAQUE DE FUERZA BRUTA SOBRE RC4 <----------|\n\n\n");

for (y = 0; y < lt; y++) { // Se escribe el texto descifrado en el archivo de texto de salida
    fprintf(txt, "%c", descifrado_host[1025 * tid + y]);
}
fclose(txt);

free(clave_host);  // Libera la memoria del host y del device
free(texto_host);
free(descifrado_host);
free(tid_host);

cudaFree(clave_dev);
cudaFree(texto_dev);
cudaFree(descifrado_dev);
cudaFree(tid_dev);
```

```
    return 0;
}
/*************** FIN MAIN ***************/

/*************** Funciones del algoritmo RC4 ***************/
__device__ unsigned char S[MAX_NTHREADS][256];
__device__ unsigned char temp[MAX_NTHREADS];
__device__ int i[MAX_NTHREADS];
__device__ int j[MAX_NTHREADS];

/*** KSA - Key-Scheduling Algorithm ***/
__device__ void rc4_init(int id, unsigned char *key, int key_length) {
    for (i[id] = 0; i[id] < 256; i[id]++) {
        S[id][i[id]] = i[id];
    }

    for (i[id] = j[id] = 0; i[id] < 256; i[id]++) {
        j[id] = (j[id] + key[i[id] % key_length] + S[id][i[id]]) & 255;
        temp[id] = S[id][i[id]];      /************/
        S[id][i[id]] = S[id][j[id]];  /*** SWAP ***/
        S[id][j[id]] = temp[id];      /************/
    }
    i[id] = j[id] = 0;
}

/*** PRGA - Pseudo-Random Generation Algorithm ***/
__device__ unsigned char rc4_output(int id) {
    i[id] = (i[id] + 1) & 255;
    j[id] = (j[id] + S[id][i[id]]) & 255;
    temp[id] = S[id][i[id]];      /************/
    S[id][i[id]] = S[id][j[id]];  /*** SWAP ***/
    S[id][j[id]] = temp[id];      /************/

    return S[id][(S[id][i[id]] + S[id][j[id]]) & 255];
}
/*************** Fin de las funciones del algoritmo RC4 ***************/

/*************** Ataque de Fuerza Bruta sobre el archivo cifrado con el algoritmo
RC4 ***************/
__global__ void brute_force(unsigned char *texto_dev, int lt, unsigned char *clave_dev,
unsigned char *descifrado_dev, unsigned char *tid_dev) { // El Kernel CUDA - Función
principal que realiza el ataque de fuerza bruta, y que es invocada por el host

    unsigned char alfabeto[56] = {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u',
'v','w','x','y','z','A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X','Y','
Z',' ',',','.','\0'};
    int y, c1, c2, lc = 5, ii, gen_key = 0, valor;
```

```
            int id = threadIdx.x;
            int N_combinaciones = 500; // Con esta variable se maneja la cantidad de claves que
puede manejar cada hebra, la cual debe ser acorde con la cantidad de hebras definidas
inicialmente
            int key_inicial = id * N_combinaciones;
            int key_final = key_inicial + N_combinaciones;

            for (gen_key = key_inicial; gen_key < key_final && tid_dev[1] == 0; gen_key++) {
// Cuando la clave de la i-ésima iteración logra descifrar el texto, gen_key toma el valor
100.000, finalizando el bucle for

                valor = gen_key;  // La asignación permite no perder el valor de la clave que es
dividida hasta llegar a un residuo igual a 0
                for (ii = 0; ii < 5; ii++) { // En el bucle se genera una nueva clave en cada iteración
                    clave_dev[257 * id + 4 - ii] = '0' + (valor % 10);
                    valor = valor / 10;
                }
                clave_dev[257 * id + lc] = '\0'; // Con \0 se marca el final de la clave (string)

                rc4_init(id, &clave_dev[257 * id], lc); // Inicializa el algoritmo para el descifrado

                for (y = 0; y < lt; y++) { // Se ejecuta el descifrado, y si tiene éxito el ataque
escribe el texto plano en un archivo .txt
                    descifrado_dev[1025 * id + y] = texto_dev[y] ^ rc4_output(id); // XOR que
entrega el descifrado
                }
                for (c1 = 0; c1 < lt; c1++) { // Se compara el descifrado con el alfabeto para saber
si se logró encontrar la clave
                    for (c2 = 0; c2 < 56; c2++) {
                        if (descifrado_dev[1025 * id + c1] == alfabeto[c2]) {
                            break;
                        }
                    }
                    if (c2 == 56) { // Condición de término si es que el descifrado arroja "caracteres
basura" (ilegibles)
                        break;
                    }
                }
                if (c1 == (lt-1)) { // Condición para escribir el texto plano en el archivo .txt y
determinar la clave que descifra el archivo .rc4
                    tid_dev[0] = (unsigned char) ('0' + id);
                    tid_dev[1] = 1;
                }
            }
        }
        /************** Fin de la función de ataque de fuerza bruta sobre el archivo cifrado
con RC4 **************/
```

Anexo C. Resultados del Ataque de Fuerza Bruta en el Procesador

C.1 Resultados del Ataque de Fuerza Bruta en el Procesador para el Archivo: textoplano2.txt

C.1.1 Ataque de Fuerza Bruta para Claves Seleccionadas

Claves	Prueba Nº 1 (seg.)	Prueba Nº 2 (seg.)	Prueba Nº 3 (seg.)	Promedio (seg.)
05000	0,14	0,14	0,14	0,14
10000	0,28	0,28	0,28	0,28
15000	0,43	0,42	0,45	0,43
20000	0,57	0,57	0,57	0,57
25000	0,71	0,71	0,71	0,71
30000	0,86	0,86	0,86	0,86
35000	0,99	1,00	1,00	1,00
40000	1,14	1,14	1,14	1,14
45000	1,28	1,29	1,28	1,28
50000	1,42	1,43	1,42	1,42
55000	1,57	1,58	1,58	1,58
60000	1,71	1,72	1,72	1,72
65000	1,85	1,86	1,86	1,86
70000	2,00	2,00	2,00	2,00
75000	2,14	2,14	2,15	2,14
80000	2,28	2,29	2,28	2,28
85000	2,44	2,43	2,44	2,44
90000	2,57	2,58	2,57	2,57
95000	2,71	2,71	2,71	2,71
99999	2,86	2,86	2,85	2,86

Tabla C.1: Tiempos empleados en cada ataque de fuerza bruta para claves seleccionadas para el archivo textoplano2.txt.

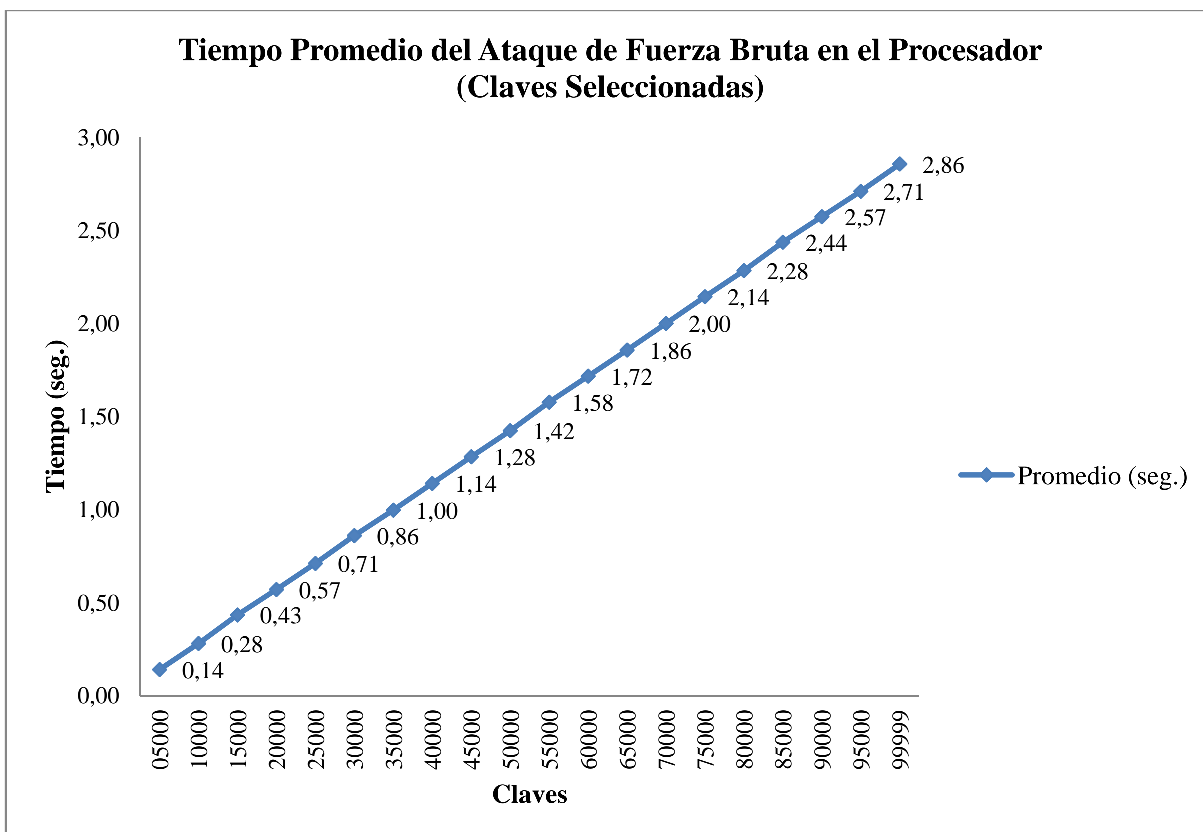

Figura C.1: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta para claves seleccionadas para el archivo textoplano2.txt.

C.1.2 Ataque de Fuerza Bruta para Claves Aleatorias

Claves	Prueba Nº 1 (seg.)	Prueba Nº 2 (seg.)	Prueba Nº 3 (seg.)	Promedio (seg.)
02200	0,06	0,06	0,06	0,06
07556	0,22	0,21	0,22	0,22
11226	0,32	0,32	0,32	0,32
16389	0,46	0,47	0,47	0,47
25999	0,74	0,74	0,74	0,74
29779	0,84	0,85	0,85	0,85
32693	0,94	0,93	0,94	0,94
39665	1,14	1,13	1,13	1,13
44499	1,26	1,27	1,26	1,26
47586	1,36	1,36	1,36	1,36
53551	1,53	1,53	1,53	1,53
59999	1,72	1,71	1,71	1,71
61843	1,77	1,77	1,76	1,77
68429	1,96	1,95	1,96	1,96
76311	2,18	2,18	2,18	2,18

79378	2,27	2,28	2,26	2,27
82010	2,35	2,34	2,35	2,35
86375	2,48	2,48	2,47	2,48
92017	2,63	2,64	2,64	2,64
98797	2,82	2,83	2,83	2,83

Tabla C.2: Tiempos empleados en cada ataque de fuerza bruta para claves aleatorias para el archivo textoplano2.txt.

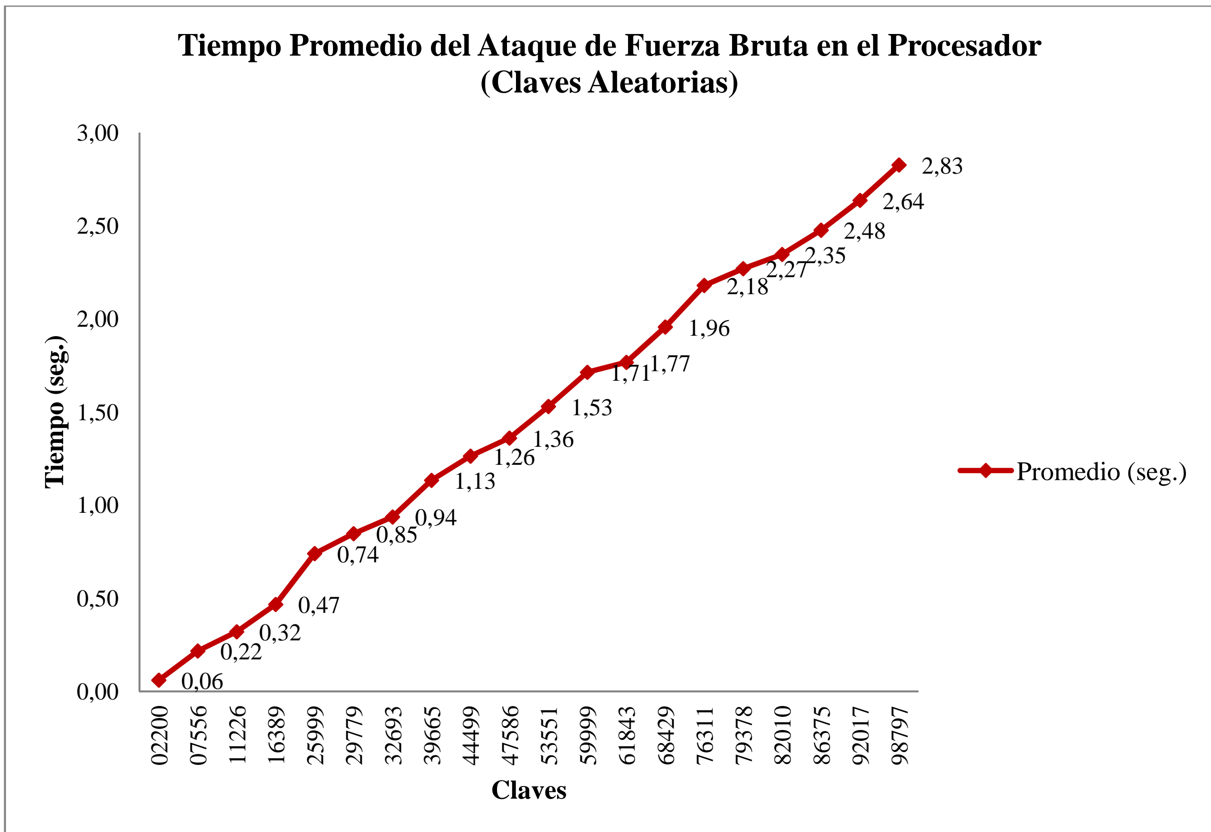

Figura C.2: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta para claves aleatorias para el archivo textoplano2.txt.

C.2 Resultados del Ataque de Fuerza Bruta en el Procesador para el Archivo: textoplano3.txt

C.2.1 Ataque de Fuerza Bruta para Claves Seleccionadas

Claves	Prueba Nº 1 (seg.)	Prueba Nº 2 (seg.)	Prueba Nº 3 (seg.)	Promedio (seg.)
05000	0,08	0,08	0,08	0,08
10000	0,16	0,16	0,16	0,16
15000	0,24	0,24	0,24	0,24
20000	0,32	0,32	0,32	0,32
25000	0,40	0,40	0,40	0,40
30000	0,48	0,48	0,48	0,48
35000	0,55	0,55	0,55	0,55
40000	0,63	0,64	0,63	0,63
45000	0,71	0,71	0,70	0,71
50000	0,78	0,79	0,79	0,79
55000	0,86	0,87	0,87	0,87
60000	0,94	0,95	0,95	0,95
65000	1,03	1,03	1,02	1,03
70000	1,10	1,11	1,11	1,11
75000	1,18	1,18	1,18	1,18
80000	1,27	1,27	1,27	1,27
85000	1,34	1,35	1,34	1,34
90000	1,42	1,43	1,41	1,42
95000	1,50	1,50	1,49	1,50
99999	1,58	1,58	1,57	1,58

Tabla C.3: Tiempos empleados en cada ataque de fuerza bruta para claves seleccionadas para el archivo textoplano3.txt.

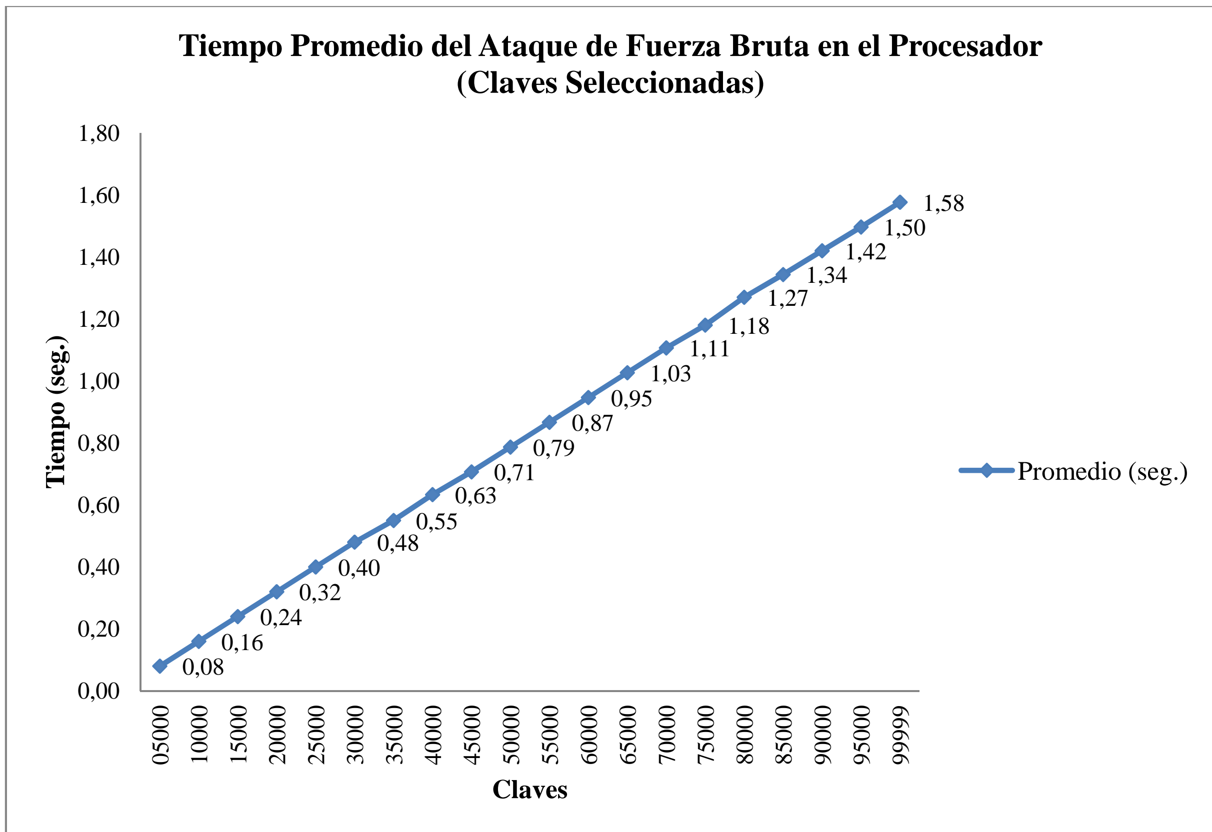

Figura C.3: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta para claves seleccionadas para el archivo textoplano3.txt.

C.2.2 Ataque de Fuerza Bruta para Claves Aleatorias

Claves	Prueba Nº 1 (seg.)	Prueba Nº 2 (seg.)	Prueba Nº 3 (seg.)	Promedio (seg.)
02200	0,03	0,03	0,03	0,03
07556	0,12	0,12	0,12	0,12
11226	0,17	0,17	0,18	0,17
16389	0,26	0,26	0,26	0,26
25999	0,40	0,40	0,41	0,40
29779	0,46	0,47	0,46	0,46
32693	0,52	0,52	0,52	0,52
39665	0,63	0,63	0,63	0,63
44499	0,70	0,70	0,70	0,70
47586	0,74	0,75	0,75	0,75
53551	0,84	0,84	0,84	0,84
59999	0,94	0,95	0,95	0,95
61843	0,97	0,97	0,97	0,97
68429	1,07	1,07	1,08	1,07
76311	1,21	1,21	1,20	1,21

79378	1,24	1,25	1,25	1,25
82010	1,30	1,29	1,30	1,30
86375	1,36	1,36	1,36	1,36
92017	1,46	1,45	1,46	1,46
98797	1,55	1,56	1,56	1,56

Tabla C.4: Tiempos empleados en cada ataque de fuerza bruta para claves aleatorias para el archivo textoplano3.txt.

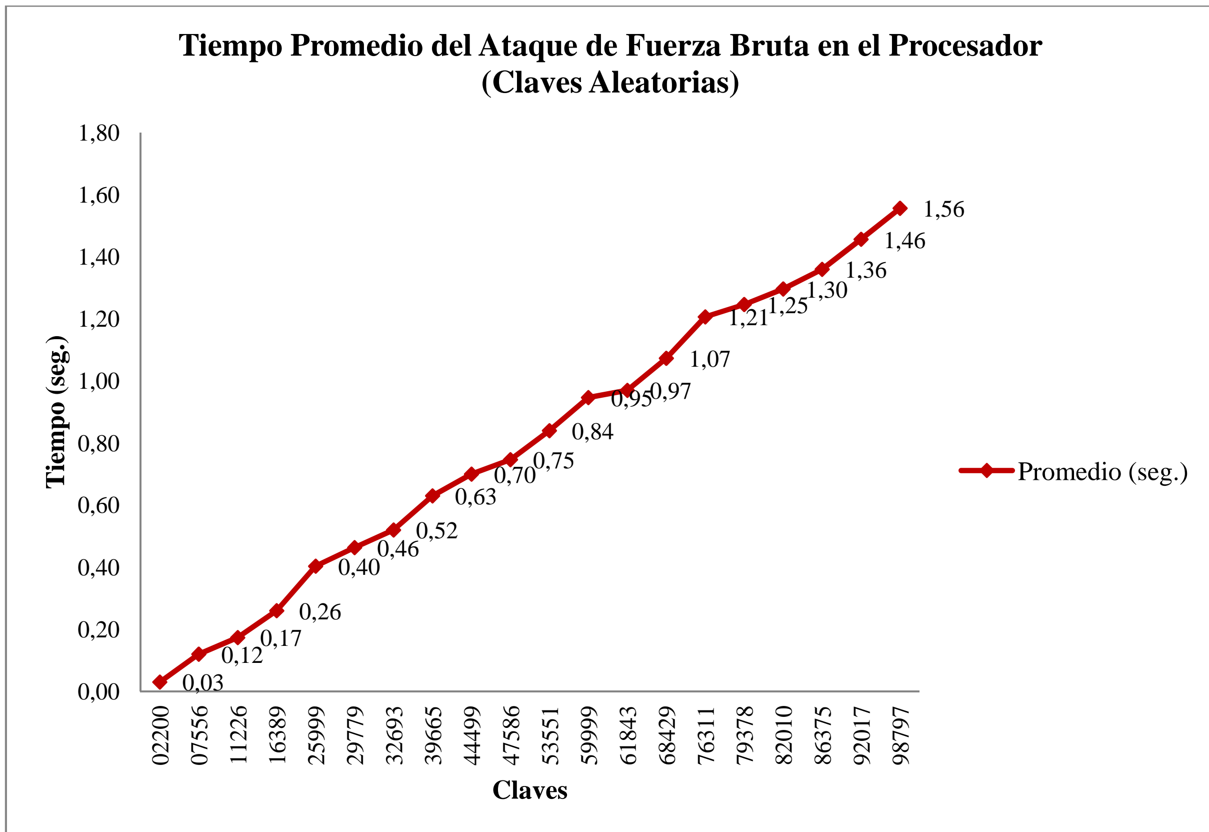

Figura C.4: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta para claves aleatorias para el archivo textoplano3.txt.

Anexo D. Resultados del Ataque de Fuerza Bruta en la Tarjeta Gráfica

D.1 Resultados del Ataque de Fuerza Bruta en la Tarjeta Gráfica para el Archivo: textoplano2.txt

D.1.1 Ataque de Fuerza Bruta para Claves Seleccionadas

D.1.1.1 Ataque de Fuerza Bruta con Una Hebra

Claves	Prueba Nº 1 (seg.)	Prueba Nº 2 (seg.)	Prueba Nº 3 (seg.)	Promedio (seg.)
05000	2,16	2,15	2,16	2,16
10000	4,31	4,30	4,31	4,31
15000	6,46	6,45	6,47	6,46
20000	8,61	8,60	8,62	8,61
25000	10,77	10,76	10,77	10,77
30000	12,91	12,92	12,91	12,91
35000	15,07	15,07	15,07	15,07
40000	17,22	17,21	17,22	17,22
45000	19,37	19,37	19,37	19,37
50000	21,52	21,53	21,53	21,53
55000	23,68	23,68	23,69	23,68
60000	25,83	25,85	25,84	25,84
65000	27,98	27,99	27,98	27,98
70000	30,14	30,13	30,14	30,14
75000	32,29	32,28	32,28	32,28
80000	34,45	34,45	34,44	34,45
85000	36,61	36,60	36,60	36,60
90000	38,76	38,75	38,75	38,75
95000	40,92	40,90	40,90	40,91
99999	43,06	43,06	43,06	43,06

Tabla D.1: Tiempos empleados en cada ataque de fuerza bruta con una hebra para claves seleccionadas para el archivo textoplano2.txt.

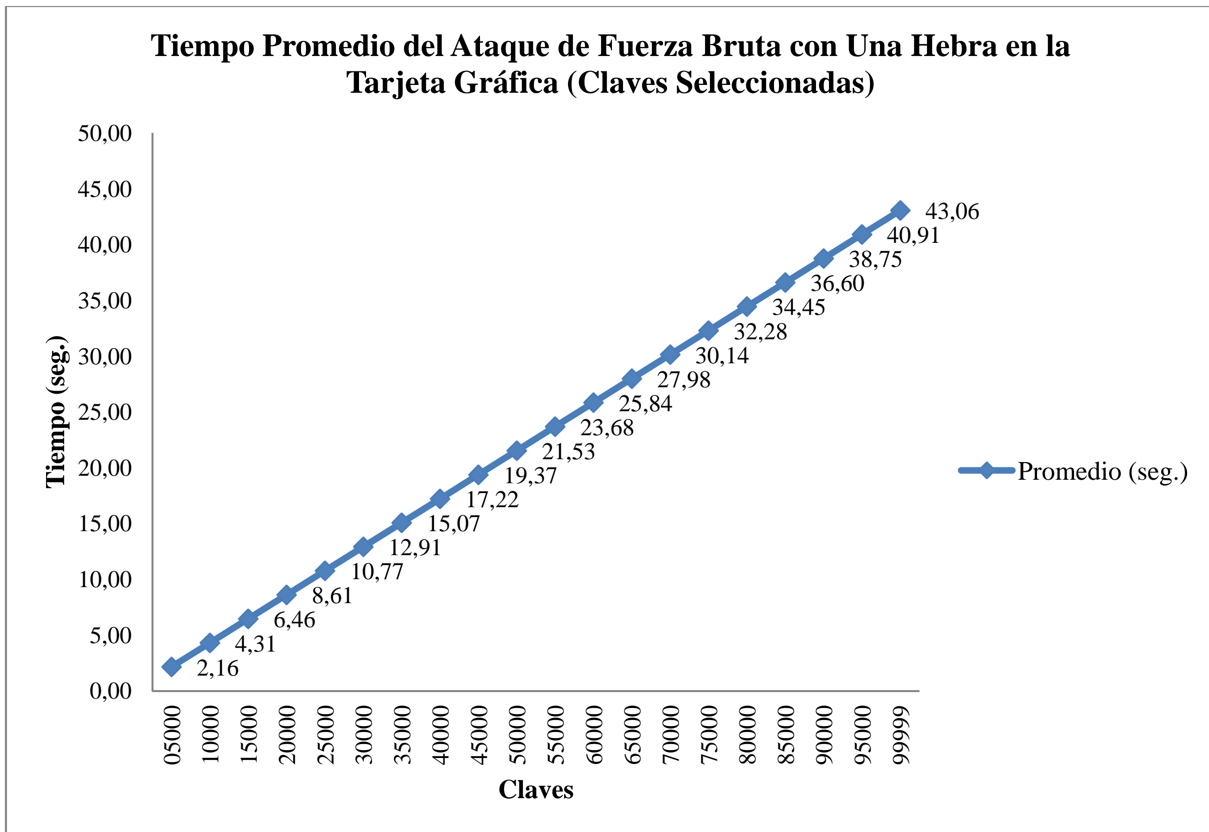

Figura D.1: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con una hebra con claves seleccionadas para el archivo textoplano2.txt.

D.1.1.2 Ataque de Fuerza Bruta con 10 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
05000	9,45	9,45	9,45	9,45
10000	0,00	0,00	0,01	0,00
15000	9,45	9,45	9,45	9,45
20000	0,00	0,00	0,00	0,00
25000	9,44	9,45	9,45	9,45
30000	0,00	0,00	0,00	0,00
35000	9,45	9,45	9,45	9,45
40000	0,00	0,00	0,01	0,00
45000	9,45	9,45	9,44	9,45
50000	0,00	0,01	0,00	0,00
55000	9,44	9,44	9,44	9,44
60000	0,01	0,01	0,01	0,01
65000	9,45	9,45	9,45	9,45
70000	0,00	0,01	0,01	0,01
75000	9,45	9,45	9,45	9,45

80000	0,01	0,01	0,01	0,01
85000	9,45	9,45	9,44	9,45
90000	0,01	0,00	0,01	0,01
95000	9,43	9,44	9,45	9,44
99999	18,89	18,88	18,90	18,89

Tabla D.2: Tiempos empleados en cada ataque de fuerza bruta con 10 hebras para claves seleccionadas para el archivo textoplano2.txt.

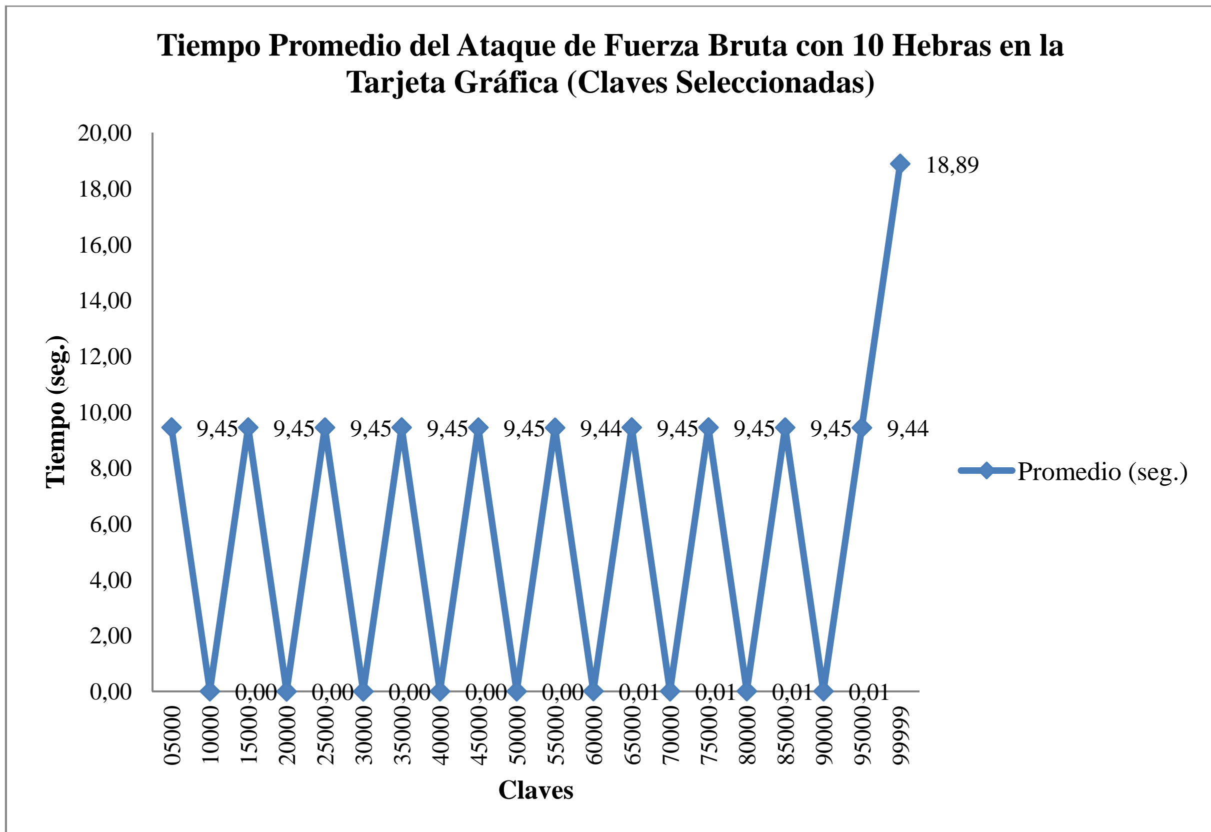

Figura D.2: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 10 hebras para claves seleccionadas para el archivo textoplano2.txt.

D.1.1.3 Ataque de Fuerza Bruta con 50 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
05000	4,73	4,75	4,74	4,74
10000	0,01	0,01	0,01	0,01
15000	4,73	4,74	4,74	4,74
20000	0,02	0,01	0,01	0,01
25000	4,74	4,74	4,73	4,74
30000	0,01	0,00	0,01	0,01
35000	4,73	4,74	4,73	4,73
40000	0,01	0,00	0,00	0,00
45000	4,74	4,73	4,73	4,73
50000	0,01	0,01	0,01	0,01
55000	4,73	4,73	4,73	4,73
60000	0,00	0,01	0,01	0,01
65000	2,93	2,93	2,93	2,93
70000	0,01	0,00	0,01	0,01
75000	2,92	2,93	2,92	2,92
80000	0,00	0,01	0,01	0,01
85000	2,93	2,93	2,94	2,93
90000	0,01	0,00	0,00	0,00
95000	2,92	2,93	2,93	2,93
99999	5,84	5,84	5,84	5,84

Tabla D.3: Tiempos empleados en cada ataque de fuerza bruta con 50 hebras para claves seleccionadas para el archivo textoplano2.txt.

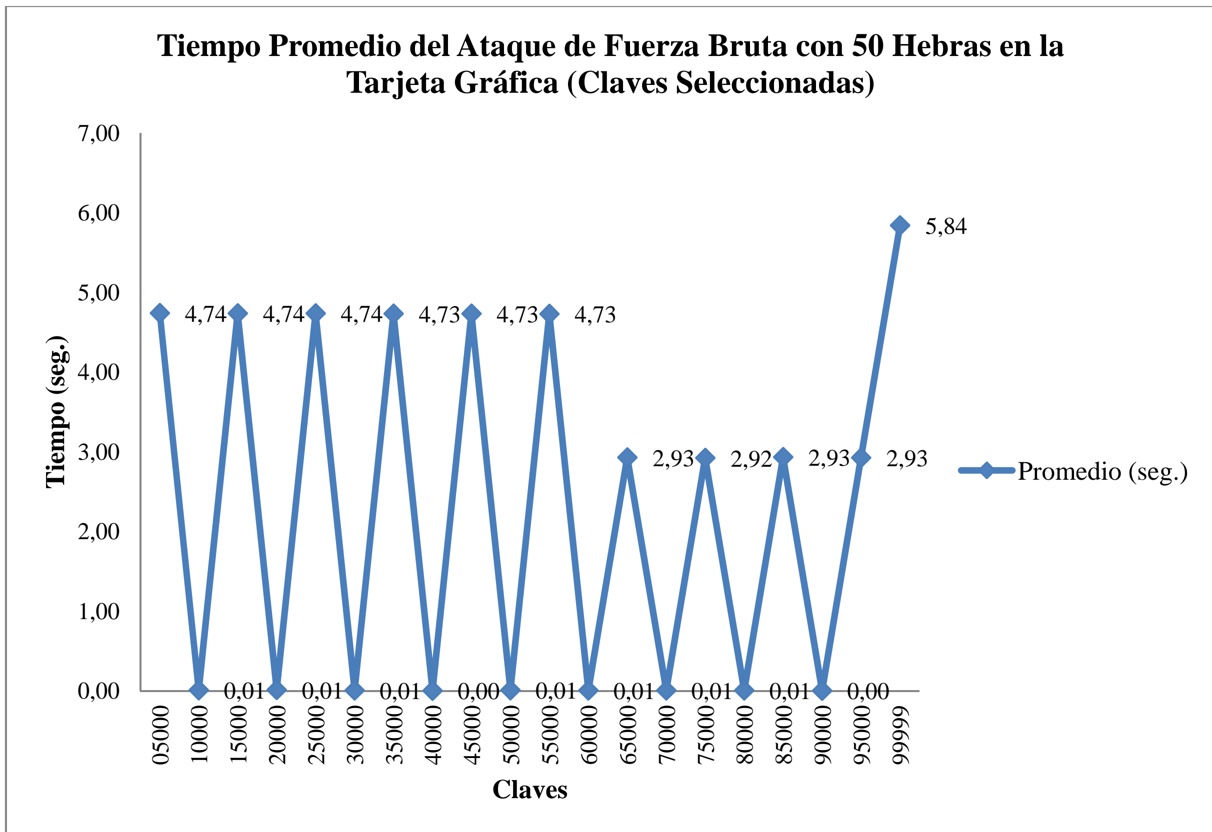

Figura D.3: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 50 hebras para claves seleccionadas para el archivo textoplano2.txt.

D.1.1.4 Ataque de Fuerza Bruta con 100 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
05000	0,02	0,01	0,01	0,01
10000	0,01	0,00	0,01	0,01
15000	0,01	0,02	0,01	0,01
20000	0,01	0,01	0,01	0,01
25000	0,02	0,01	0,01	0,01
30000	0,00	0,00	0,01	0,00
35000	0,01	0,01	0,01	0,01
40000	0,02	0,00	0,01	0,01
45000	0,01	0,01	0,01	0,01
50000	0,01	0,01	0,01	0,01
55000	0,01	0,01	0,01	0,01
60000	0,01	0,02	0,01	0,01
65000	0,02	0,01	0,00	0,01
70000	0,01	0,02	0,01	0,01
75000	0,01	0,00	0,01	0,01

80000	0,01	0,01	0,01	0,01
85000	0,01	0,01	0,01	0,01
90000	0,02	0,01	0,01	0,01
95000	0,01	0,02	0,01	0,01
99999	1,20	1,20	1,21	1,20

Tabla D.4: Tiempos empleados en cada ataque de fuerza bruta con 100 hebras para claves seleccionadas para el archivo textoplano2.txt.

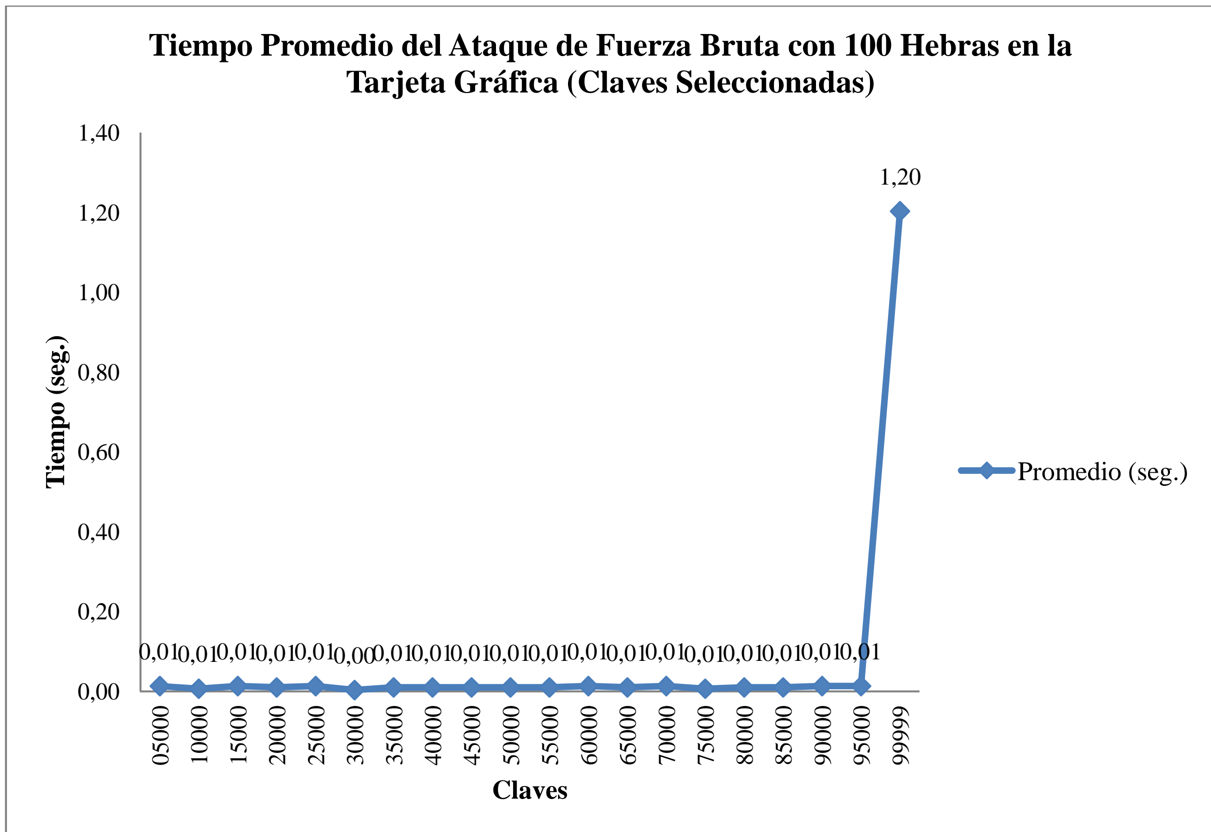

Figura D.4: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 100 hebras para claves seleccionadas para el archivo textoplano2.txt.

D.1.1.5 Ataque de Fuerza Bruta con 200 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
05000	0,01	0,01	0,01	0,01
10000	0,00	0,01	0,01	0,01
15000	0,02	0,02	0,01	0,02
20000	0,02	0,02	0,00	0,01
25000	0,02	0,01	0,02	0,02
30000	0,02	0,02	0,01	0,02
35000	0,01	0,01	0,01	0,01
40000	0,00	0,01	0,01	0,01
45000	0,01	0,02	0,02	0,02
50000	0,01	0,02	0,02	0,02
55000	0,02	0,01	0,02	0,02
60000	0,01	0,02	0,02	0,02
65000	0,01	0,00	0,01	0,01
70000	0,01	0,01	0,01	0,01
75000	0,01	0,02	0,01	0,01
80000	0,02	0,00	0,01	0,01
85000	0,01	0,01	0,02	0,01
90000	0,01	0,01	0,02	0,01
95000	0,01	0,01	0,01	0,01
99999	0,95	0,95	0,95	0,95

Tabla D.5: Tiempos empleados en cada ataque de fuerza bruta con 200 hebras para claves seleccionadas para el archivo textoplano2.txt.

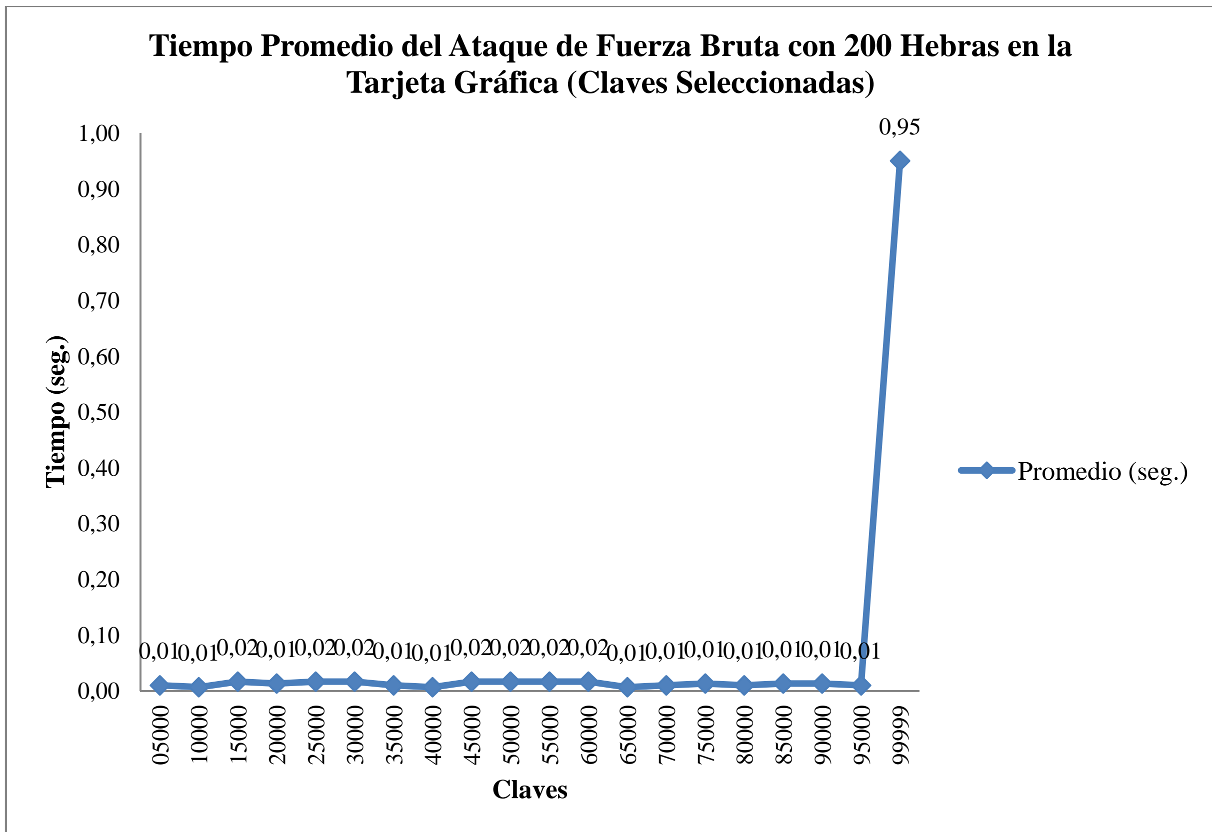

Figura D.5: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 200 hebras para claves seleccionadas para el archivo textoplano2.txt.

D.1.2 Ataque de Fuerza Bruta para Claves Aleatorias

D.1.2.1 Ataque de Fuerza Bruta con Una Hebra

Claves	Prueba Nº 1 (seg.)	Prueba Nº 2 (seg.)	Prueba Nº 3 (seg.)	Promedio (seg.)
02200	0,96	0,95	0,95	0,95
07556	3,25	3,26	3,25	3,25
11226	4,83	4,83	4,84	4,83
16389	7,05	7,06	7,05	7,05
25999	11,20	11,19	11,20	11,20
29779	12,82	12,82	12,82	12,82
32693	14,08	14,07	14,07	14,07
39665	17,07	17,07	17,07	17,07
44499	19,17	19,16	19,16	19,16
47586	20,48	20,48	20,49	20,48
53551	23,05	23,06	23,05	23,05
59999	25,83	25,83	25,84	25,83
61843	26,64	26,62	26,63	26,63
68429	29,46	29,46	29,46	29,46

76311	32,86	32,85	32,85	32,85
79378	34,17	34,18	34,19	34,18
82010	35,31	35,31	35,31	35,31
86375	37,20	37,20	37,19	37,20
92017	39,62	39,62	39,62	39,62
98797	42,55	42,54	42,54	42,54

Tabla D.6: Tiempos empleados en cada ataque de fuerza bruta con una hebra para claves aleatorias para el archivo textoplano2.txt.

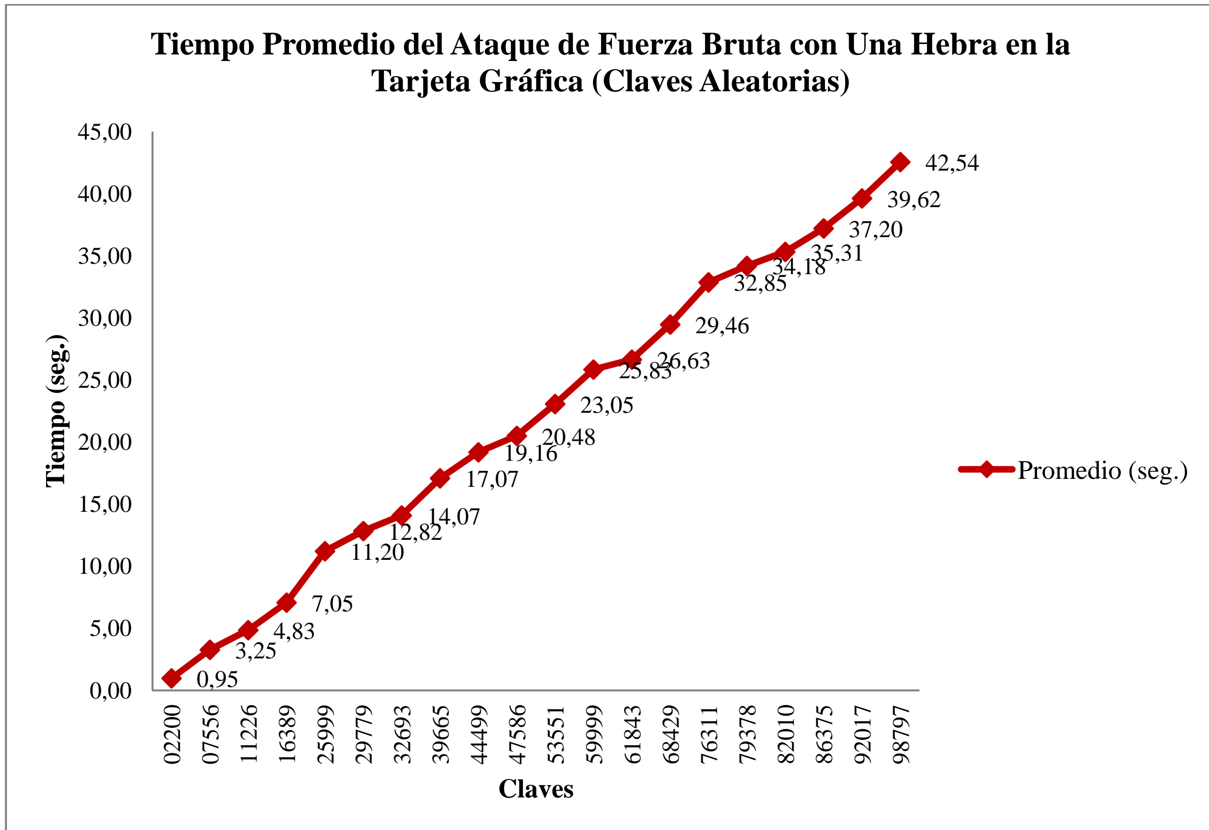

Figura D.6: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con una hebra para claves aleatorias para el archivo textoplano2.txt.

D.1.2.2 Ataque de Fuerza Bruta con 10 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
02200	4,14	4,13	4,13	4,13
07556	14,19	14,19	14,20	14,19
11226	2,30	2,31	2,30	2,30
16389	11,99	11,99	11,99	11,99
25999	11,27	11,26	11,27	11,27
29779	18,35	18,36	18,36	18,36
32693	5,04	5,05	5,05	5,05
39665	18,14	18,15	18,14	18,14
44499	8,44	8,44	8,45	8,44
47586	14,23	14,24	14,24	14,24
53551	6,67	6,67	6,67	6,67
59999	18,78	18,78	18,78	18,78
61843	3,45	3,46	3,46	3,46
68429	15,84	15,83	15,83	15,83
76311	11,85	11,85	11,85	11,85
79378	17,61	17,60	17,60	17,60
82010	3,78	3,78	3,78	3,78
86375	11,96	11,97	11,97	11,97
92017	3,80	3,80	3,80	3,80
98797	16,51	16,52	16,52	16,52

Tabla D.7: Tiempos empleados en cada ataque de fuerza bruta con 10 hebras para claves aleatorias para el archivo textoplano2.txt.

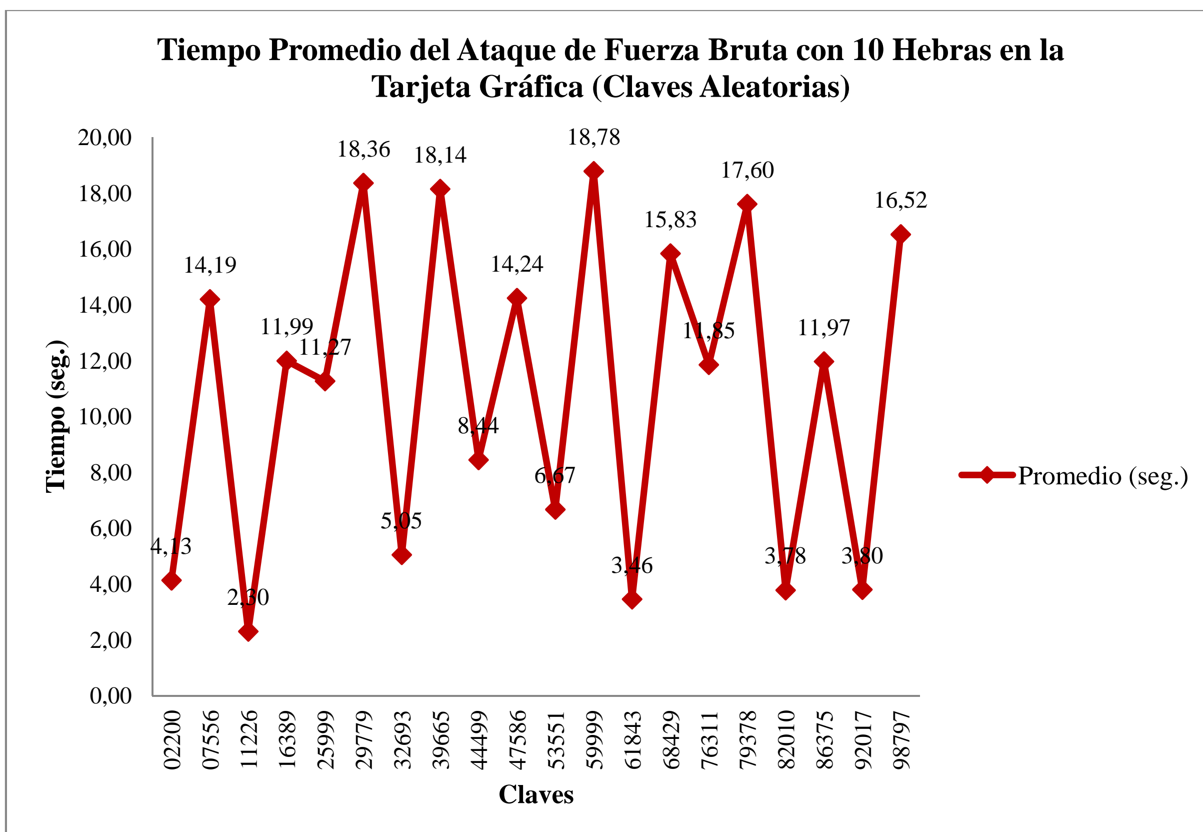

Figura D.7: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 10 hebras para claves aleatorias para el archivo textoplano2.txt.

D.1.2.3 Ataque de Fuerza Bruta con 50 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
02200	0,94	0,95	0,95	0,95
07556	7,35	7,34	7,35	7,35
11226	5,80	5,80	5,81	5,80
16389	1,85	1,84	1,85	1,85
25999	9,43	9,44	9,43	9,43
29779	8,40	8,40	8,40	8,40
32693	3,28	3,28	3,28	3,28
39665	7,87	7,86	7,87	7,87
44499	2,36	2,36	2,37	2,36
47586	7,49	7,48	7,49	7,49
53551	7,32	7,32	7,33	7,32
59999	9,43	9,43	9,43	9,43
61843	8,70	8,70	8,70	8,70
68429	1,25	1,25	1,26	1,25
76311	0,92	0,91	0,91	0,91

79378	4,02	4,02	4,03	4,02
82010	0,03	0,03	0,03	0,03
86375	1,10	1,10	1,11	1,10
92017	0,06	0,05	0,06	0,06
98797	2,33	2,33	2,33	2,33

Tabla D.8: Tiempos empleados en cada ataque de fuerza bruta con 50 hebras para claves aleatorias para el archivo textoplano2.txt.

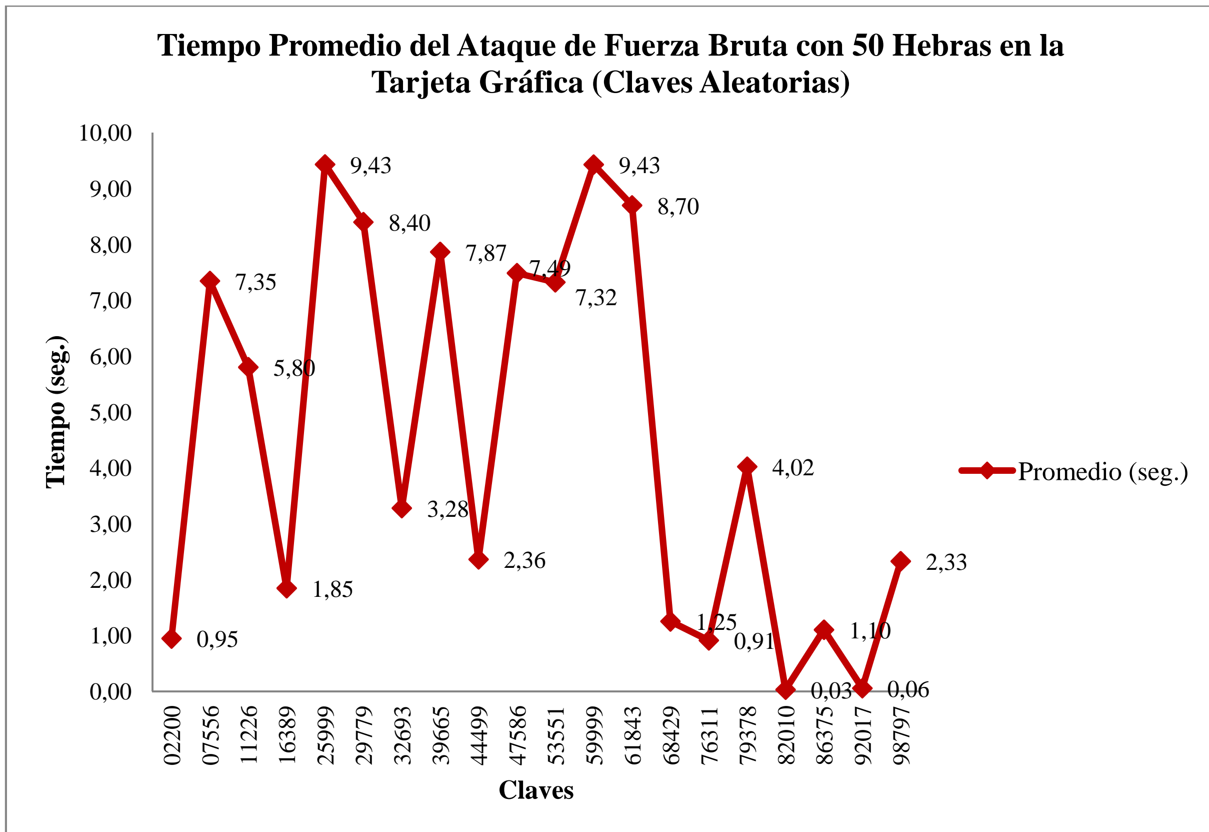

Figura D.8: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 50 hebras para claves aleatorias para el archivo textoplano2.txt.

D.1.2.4 Ataque de Fuerza Bruta con 100 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
02200	0,99	0,98	0,99	0,99
07556	2,74	2,73	2,73	2,73
11226	1,12	1,12	1,12	1,12
16389	1,91	1,92	1,91	1,91
25999	4,88	4,88	4,87	4,88
29779	3,82	3,81	3,81	3,81
32693	3,40	3,40	3,40	3,40
39665	3,26	3,25	3,26	3,26
44499	2,45	2,45	2,44	2,45
47586	2,87	2,88	2,88	2,88
53551	2,69	2,70	2,69	2,69
59999	4,87	4,88	4,88	4,88
61843	4,12	4,13	4,12	4,12
68429	2,11	2,11	2,11	2,11
76311	1,52	1,53	1,54	1,53
79378	1,86	1,86	1,86	1,86
82010	0,06	0,06	0,05	0,06
86375	1,84	1,83	1,84	1,84
92017	0,09	0,09	0,10	0,09
98797	0,97	0,96	0,96	0,96

Tabla D.9: Tiempos empleados en cada ataque de fuerza bruta con 100 hebras para claves aleatorias para el archivo textoplano2.txt.

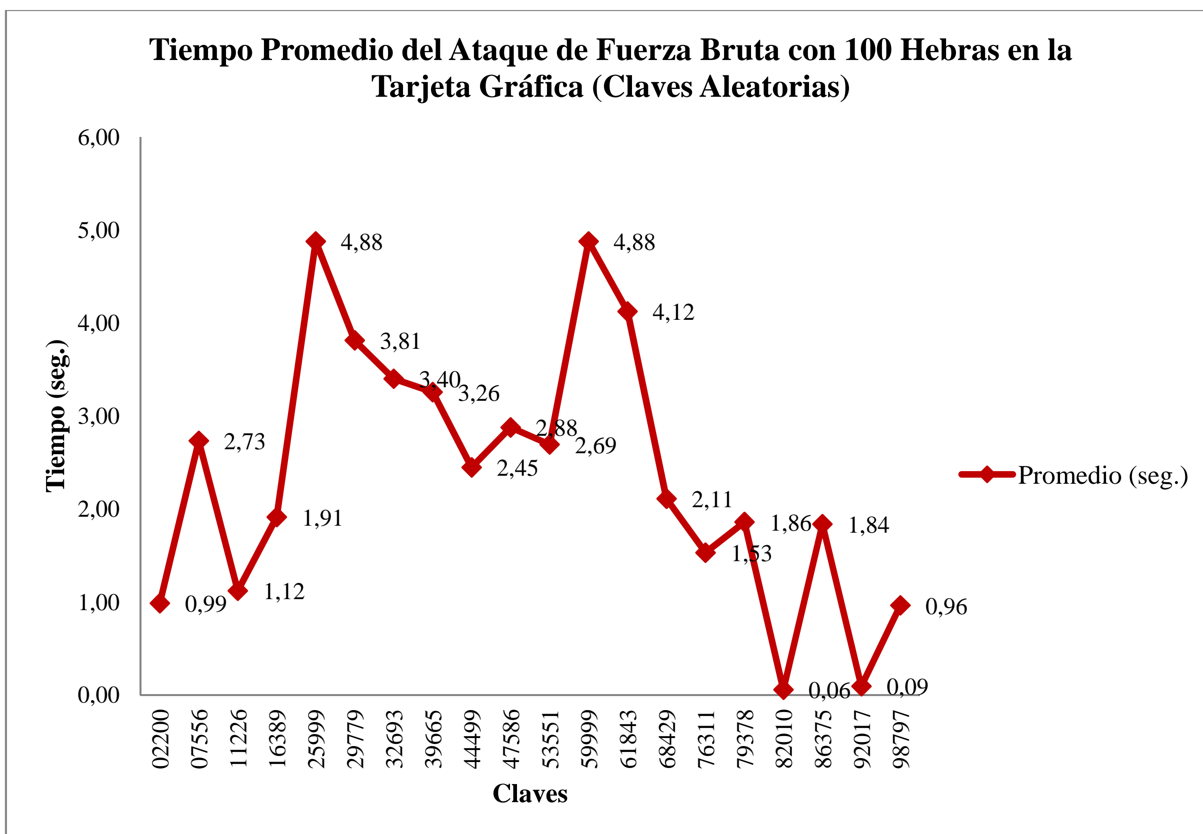

Figura D.9: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 100 hebras para claves aleatorias para el archivo textoplano2.txt.

D.1.2.5 Ataque de Fuerza Bruta con 200 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
02200	1,05	1,04	1,04	1,04
07556	0,31	0,31	0,31	0,31
11226	1,17	1,17	1,17	1,17
16389	2,00	2,01	2,00	2,00
25999	2,55	2,56	2,56	2,56
29779	1,45	1,44	1,45	1,45
32693	1,01	1,01	1,01	1,01
39665	0,87	0,86	0,86	0,86
44499	2,55	2,56	2,55	2,55
47586	0,47	0,46	0,45	0,46
53551	0,28	0,29	0,28	0,28
59999	2,56	2,55	2,56	2,56
61843	1,77	1,77	1,77	1,77
68429	2,22	2,22	2,20	2,21
76311	1,61	1,61	1,60	1,61

79378	1,95	1,95	1,95	1,95
82010	0,06	0,07	0,06	0,06
86375	1,94	1,95	1,94	1,94
92017	0,11	0,10	0,10	0,10
98797	0,57	0,57	0,58	0,57

Tabla D.10: Tiempos empleados en cada ataque de fuerza bruta con 200 hebras para claves aleatorias para el archivo textoplano2.txt.

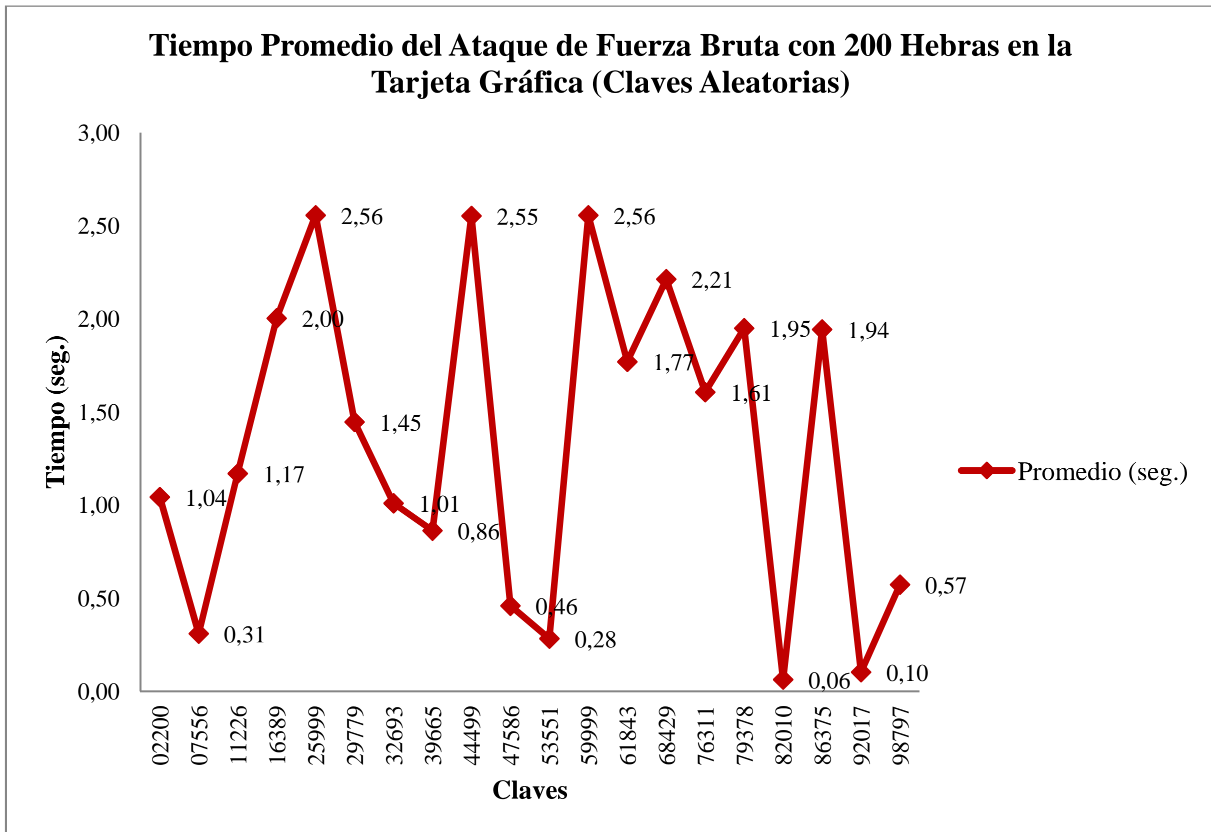

Figura D.10: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 200 hebras para claves aleatorias para el archivo textoplano2.txt.

D.2 Resultados del Ataque de Fuerza Bruta en la Tarjeta Gráfica para el Archivo: textoplano3.txt

D.2.1 Ataque de Fuerza Bruta para Claves Seleccionadas

D.2.1.1 Ataque de Fuerza Bruta con Una Hebra

Claves	Prueba Nº 1 (seg.)	Prueba Nº 2 (seg.)	Prueba Nº 3 (seg.)	Promedio (seg.)
05000	0,97	0,97	0,98	0,97
10000	1,96	1,96	1,96	1,96
15000	2,94	2,94	2,93	2,94
20000	3,91	3,92	3,92	3,92
25000	4,88	4,89	4,89	4,89
30000	5,89	5,87	5,87	5,88
35000	6,84	6,86	6,84	6,85
40000	7,84	7,83	7,83	7,83
45000	8,81	8,80	8,81	8,81
50000	9,78	9,80	9,79	9,79
55000	10,76	10,76	10,76	10,76
60000	11,74	11,74	11,75	11,74
65000	12,72	12,72	12,71	12,72
70000	13,71	13,71	13,71	13,71
75000	14,68	14,68	14,68	14,68
80000	15,66	15,66	15,65	15,66
85000	16,64	16,64	16,65	16,64
90000	17,62	17,62	17,62	17,62
95000	18,59	18,61	18,60	18,60
99999	19,59	19,58	19,58	19,58

Tabla D.11: Tiempos empleados en cada ataque de fuerza bruta con una hebra para claves seleccionadas para el archivo textoplano3.txt.

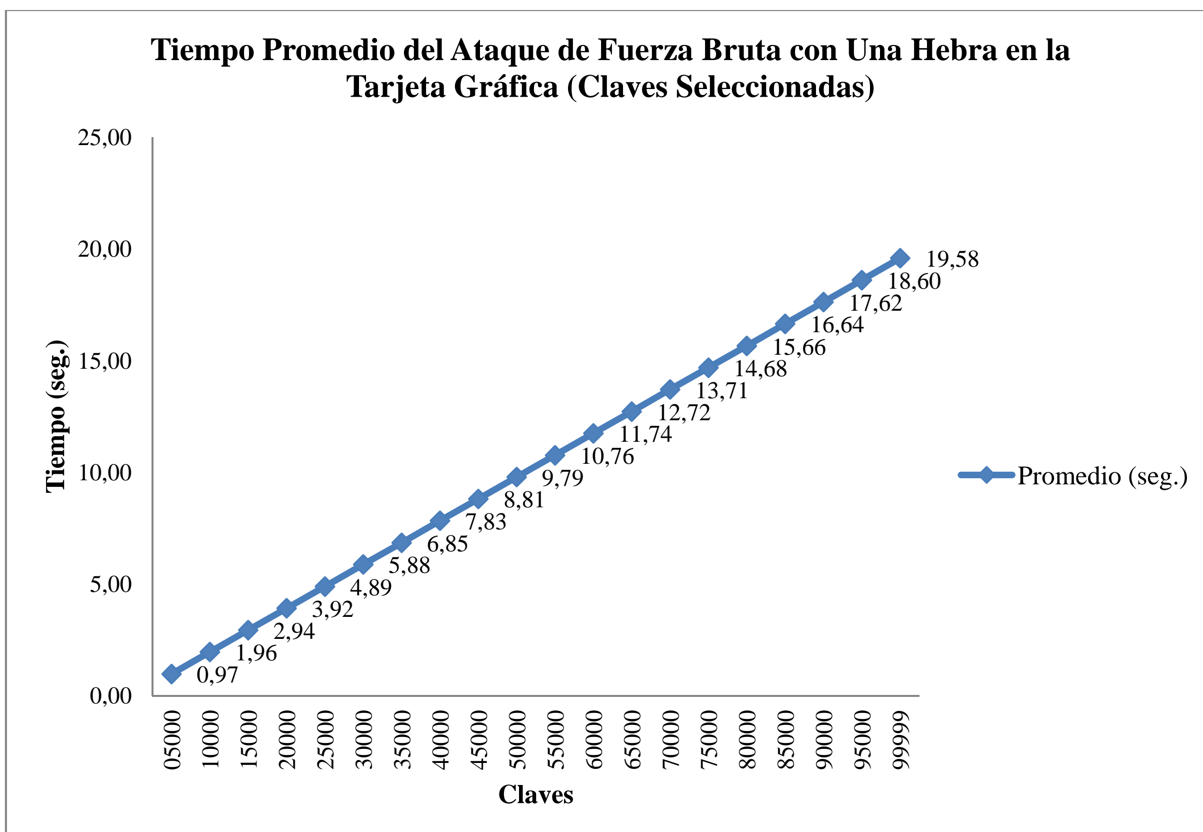

Figura D.11: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con una hebra para claves seleccionadas para el archivo textoplano3.txt.

D.2.1.2 Ataque de Fuerza Bruta con 10 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
05000	4,10	4,09	4,09	4,09
10000	0,00	0,00	0,01	0,00
15000	4,10	4,09	4,10	4,10
20000	0,00	0,01	0,00	0,00
25000	4,08	4,10	4,09	4,09
30000	0,00	0,01	0,01	0,01
35000	4,09	4,09	4,10	4,09
40000	0,00	0,00	0,01	0,00
45000	4,09	4,10	4,10	4,10
50000	0,00	0,01	0,00	0,00
55000	4,10	4,10	4,09	4,10
60000	0,01	0,01	0,00	0,01
65000	4,09	4,10	4,10	4,10
70000	0,00	0,00	0,00	0,00
75000	4,10	4,10	4,09	4,10

80000	0,00	0,01	0,00	0,00
85000	4,10	4,09	4,10	4,10
90000	0,01	0,00	0,01	0,01
95000	4,10	4,09	4,10	4,10
99999	8,18	8,19	8,19	8,19

Tabla D.12: Tiempos empleados en cada ataque de fuerza bruta con 10 hebras para claves seleccionadas para el archivo textoplano3.txt.

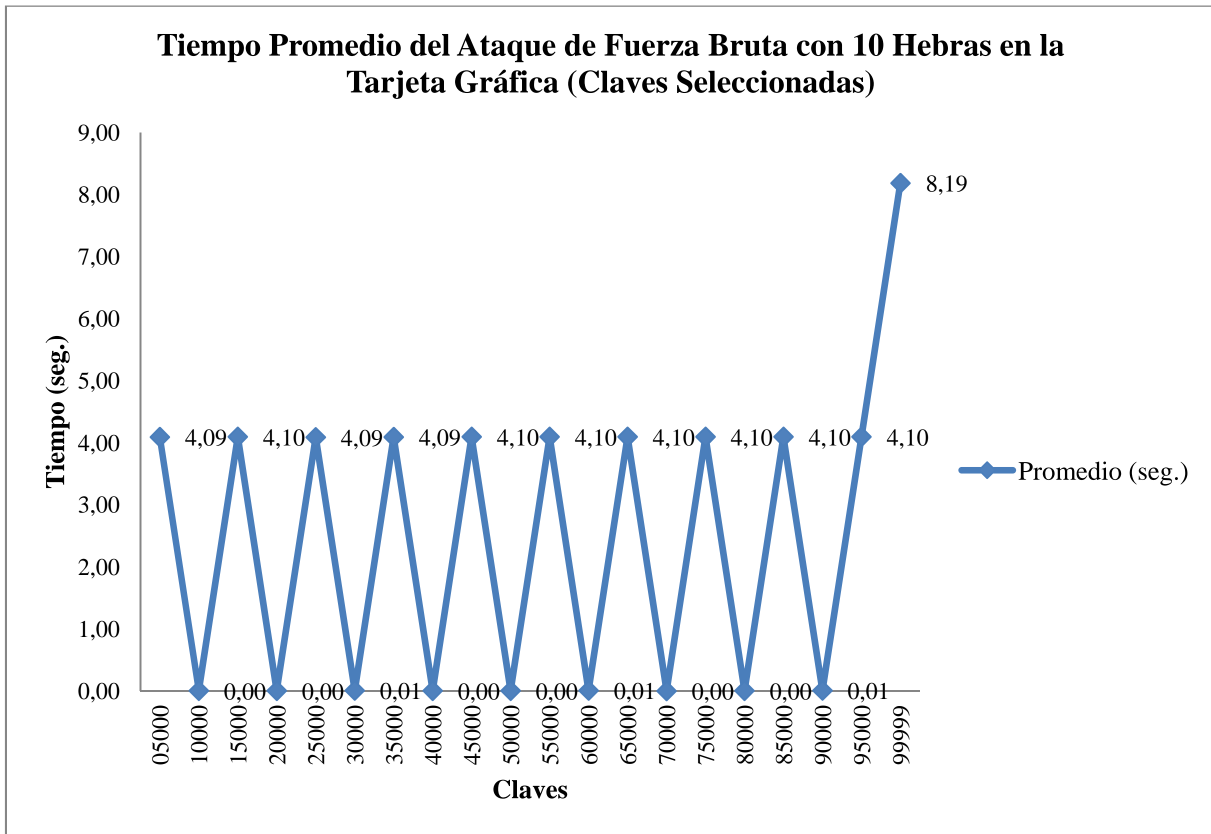

Figura D.12: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 10 hebras para claves seleccionadas para el archivo textoplano3.txt.

D.2.1.3 Ataque de Fuerza Bruta con 50 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
05000	2,02	2,01	2,01	2,01
10000	0,01	0,00	0,01	0,01
15000	2,02	2,02	2,02	2,02
20000	0,00	0,00	0,01	0,00
25000	2,02	2,02	2,02	2,02
30000	0,00	0,01	0,00	0,00
35000	2,02	2,02	2,03	2,02
40000	0,00	0,01	0,00	0,00
45000	2,02	2,02	2,02	2,02
50000	0,01	0,01	0,00	0,01
55000	2,02	2,02	2,02	2,02
60000	0,01	0,01	0,01	0,01
65000	1,26	1,25	1,26	1,26
70000	0,00	0,01	0,00	0,00
75000	1,26	1,26	1,26	1,26
80000	0,00	0,00	0,00	0,00
85000	1,26	1,25	1,26	1,26
90000	0,00	0,00	0,00	0,00
95000	1,26	1,25	1,26	1,26
99999	2,51	2,50	2,50	2,50

Tabla D.13: Tiempos empleados en cada ataque de fuerza bruta con 50 hebras para claves seleccionadas para el archivo textoplano3.txt.

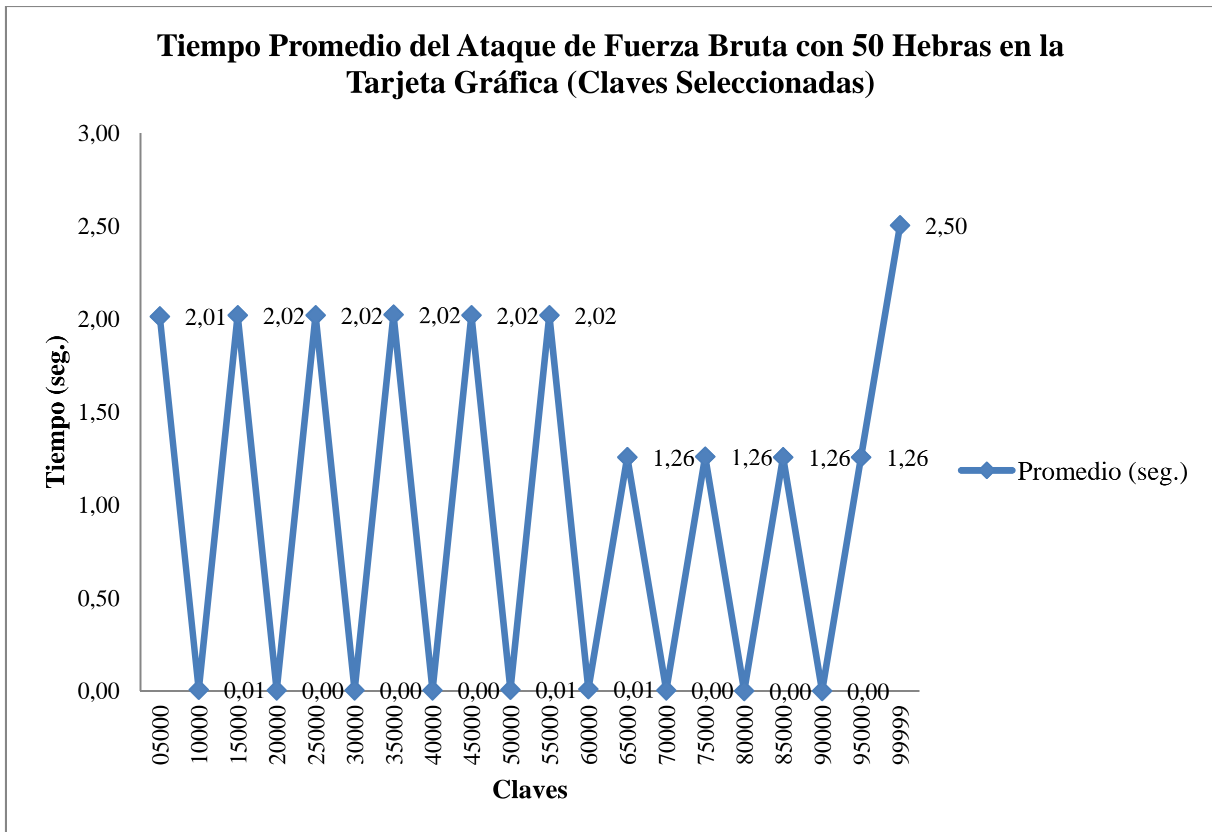

Figura D.13: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 50 hebras para claves seleccionadas para el archivo textoplano3.txt.

D.2.1.4 Ataque de Fuerza Bruta con 100 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
05000	0,01	0,00	0,00	0,00
10000	0,00	0,00	0,00	0,00
15000	0,00	0,01	0,01	0,01
20000	0,01	0,01	0,01	0,01
25000	0,01	0,00	0,01	0,01
30000	0,01	0,01	0,01	0,01
35000	0,00	0,01	0,01	0,01
40000	0,01	0,00	0,01	0,01
45000	0,00	0,01	0,00	0,00
50000	0,01	0,01	0,00	0,01
55000	0,01	0,01	0,01	0,01
60000	0,01	0,01	0,00	0,01
65000	0,00	0,00	0,01	0,00
70000	0,01	0,01	0,00	0,01
75000	0,00	0,00	0,00	0,00

80000	0,00	0,00	0,00	0,00
85000	0,00	0,01	0,00	0,00
90000	0,00	0,00	0,01	0,00
95000	0,01	0,00	0,01	0,01
99999	0,52	0,52	0,52	0,52

Tabla D.14: Tiempos empleados en cada ataque de fuerza bruta con 100 hebras para claves seleccionadas para el archivo textoplano3.txt.

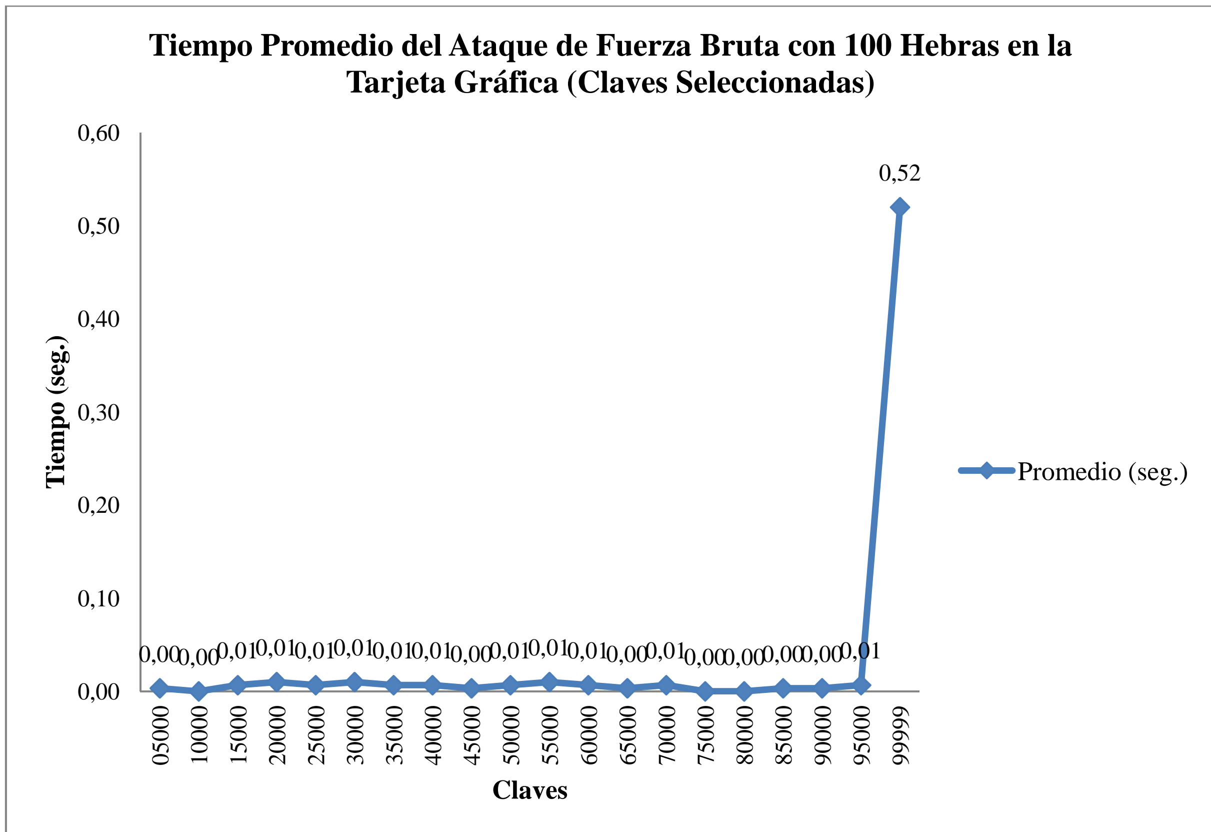

Figura D.14: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 100 hebras para claves seleccionadas para el archivo textoplano3.txt.

D.2.1.5 Ataque de Fuerza Bruta con 200 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
05000	0,01	0,00	0,00	0,00
10000	0,00	0,00	0,00	0,00
15000	0,01	0,02	0,01	0,01
20000	0,01	0,01	0,01	0,01
25000	0,00	0,01	0,02	0,01
30000	0,00	0,01	0,00	0,00
35000	0,00	0,00	0,00	0,00
40000	0,00	0,01	0,01	0,01
45000	0,01	0,00	0,00	0,00
50000	0,00	0,01	0,00	0,00
55000	0,00	0,02	0,01	0,01
60000	0,01	0,00	0,01	0,01
65000	0,01	0,01	0,01	0,01
70000	0,00	0,00	0,00	0,00
75000	0,01	0,01	0,01	0,01
80000	0,00	0,01	0,01	0,01
85000	0,00	0,01	0,01	0,01
90000	0,01	0,00	0,00	0,00
95000	0,01	0,00	0,00	0,00
99999	0,40	0,42	0,41	0,41

Tabla D.15: Tiempos empleados en cada ataque de fuerza bruta con 200 hebras para claves seleccionadas para el archivo textoplano3.txt.

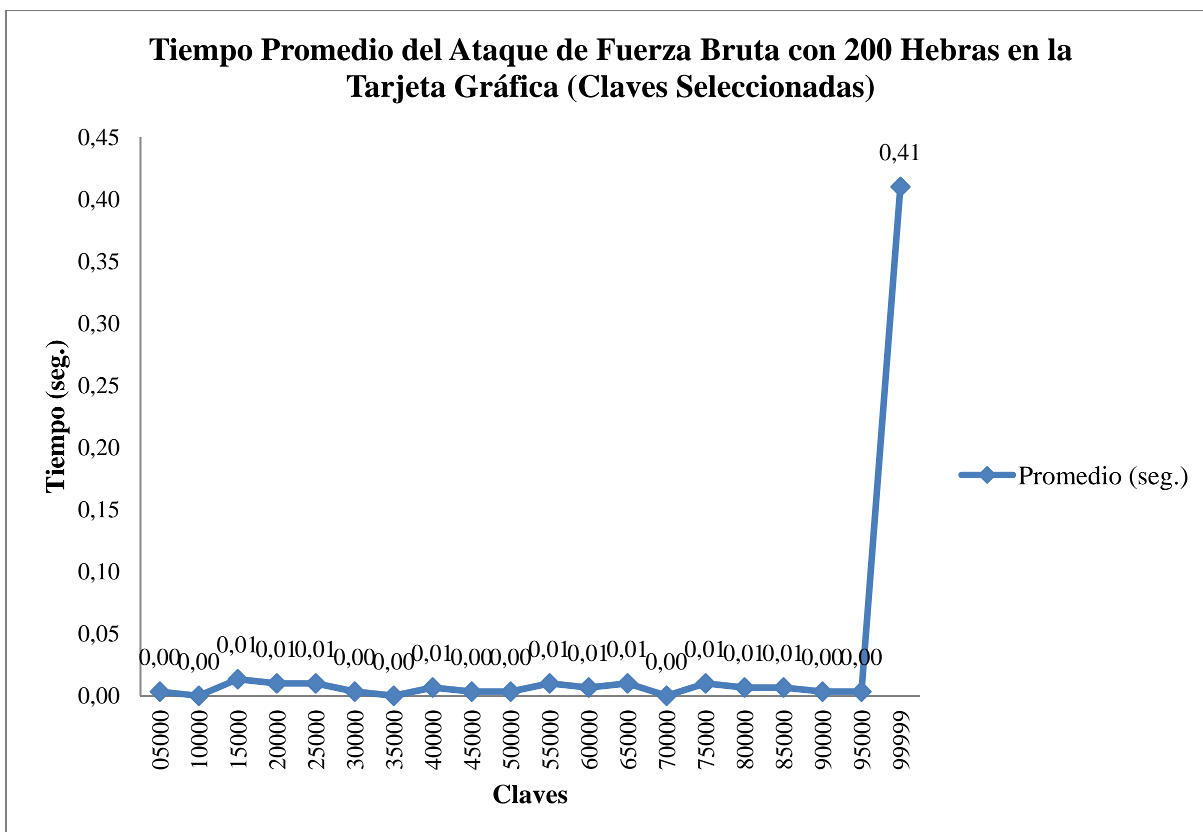

Figura D.15: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 200 hebras para claves seleccionadas para el archivo textoplano3.txt.

D.2.2 Ataque de Fuerza Bruta para Claves Aleatorias

D.2.2.1 Ataque de Fuerza Bruta con Una Hebra

Claves	Prueba Nº 1 (seg.)	Prueba Nº 2 (seg.)	Prueba Nº 3 (seg.)	Promedio (seg.)
02200	0,43	0,42	0,42	0,42
07556	1,45	1,46	1,45	1,45
11226	2,15	2,16	2,15	2,15
16389	3,15	3,15	3,16	3,15
25999	4,98	4,99	4,99	4,99
29779	5,71	5,72	5,72	5,72
32693	6,27	6,28	6,28	6,28
39665	7,61	7,60	7,61	7,61
44499	8,55	8,54	8,55	8,55
47586	9,14	9,14	9,13	9,14
53551	10,28	10,29	10,29	10,29
59999	11,52	11,52	11,52	11,52
61843	11,87	11,87	11,88	11,87
68429	13,13	13,14	13,14	13,14

76311	14,65	14,66	14,65	14,65
79378	15,25	15,24	15,24	15,24
82010	15,76	15,76	15,75	15,76
86375	16,58	16,58	16,59	16,58
92017	17,67	17,68	17,68	17,68
98797	18,98	18,99	18,98	18,98

Tabla D.16: Tiempos empleados en cada ataque de fuerza bruta con una hebra para claves aleatorias para el archivo textoplano3.txt.

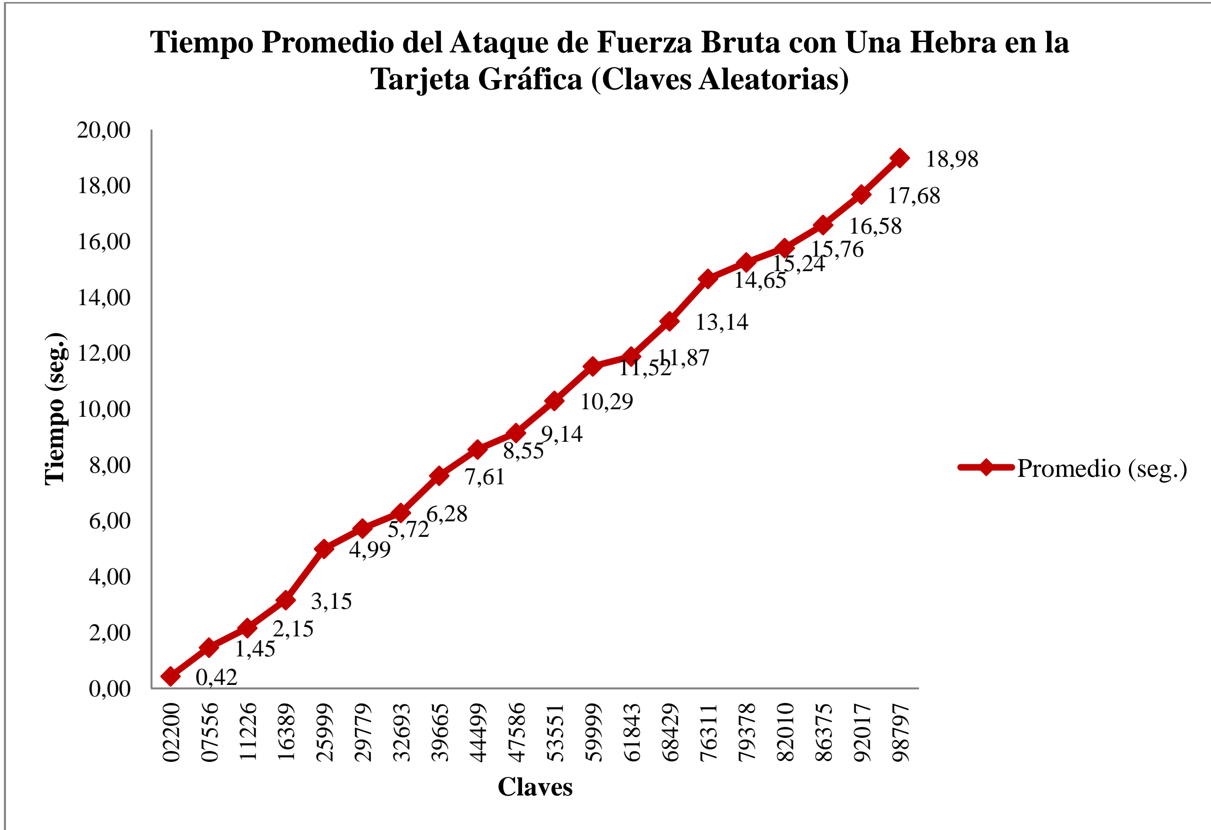

Figura D.16: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con una hebra para claves aleatorias para el archivo textoplano3.txt.

D.2.2.2 Ataque de Fuerza Bruta con 10 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
02200	1,79	1,79	1,79	1,79
07556	6,14	6,14	6,15	6,14
11226	1,00	1,01	1,01	1,01
16389	5,21	5,20	5,20	5,20
25999	4,88	4,87	4,88	4,88
29779	7,96	7,95	7,96	7,96
32693	2,20	2,20	2,19	2,20
39665	7,86	7,87	7,87	7,87
44499	3,67	3,67	3,67	3,67
47586	6,18	6,18	6,18	6,18
53551	2,90	2,89	2,89	2,89
59999	8,15	8,14	8,15	8,15
61843	1,49	1,50	1,50	1,50
68429	6,86	6,87	6,86	6,86
76311	5,15	5,14	5,14	5,14
79378	7,63	7,63	7,63	7,63
82010	1,63	1,64	1,64	1,64
86375	5,19	5,19	5,19	5,19
92017	1,65	1,65	1,64	1,65
98797	7,16	7,16	7,16	7,16

Tabla D.17: Tiempos empleados en cada ataque de fuerza bruta con 10 hebras para claves aleatorias para el archivo textoplano3.txt.

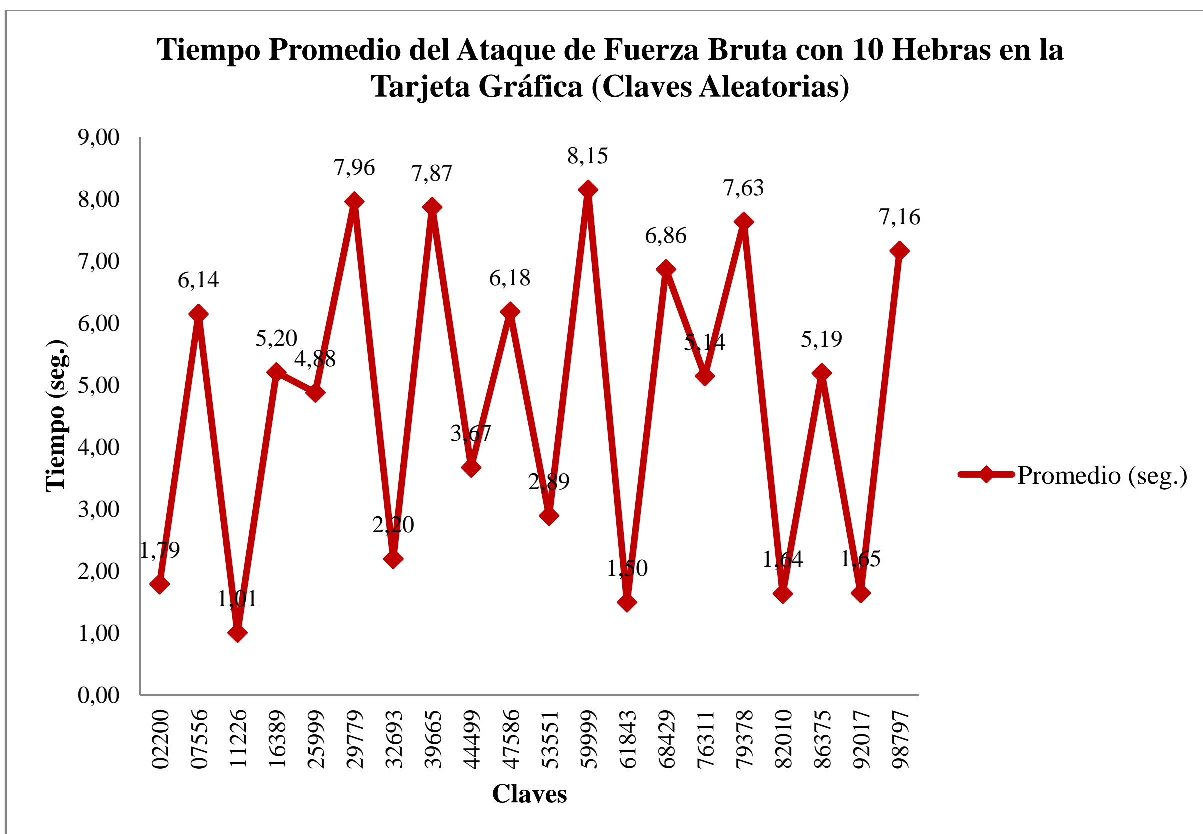

Figura D.17: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 10 hebras para claves aleatorias para el archivo textoplano3.txt.

D.2.2.3 Ataque de Fuerza Bruta con 50 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
02200	0,41	0,40	0,41	0,41
07556	3,13	3,14	3,13	3,13
11226	2,48	2,47	2,48	2,48
16389	0,79	0,79	0,79	0,79
25999	4,03	4,03	4,02	4,03
29779	3,58	3,58	3,58	3,58
32693	1,39	1,40	1,40	1,40
39665	3,36	3,35	3,35	3,35
44499	1,01	1,01	1,00	1,01
47586	3,19	3,19	3,19	3,19
53551	3,13	3,12	3,12	3,12
59999	4,02	4,01	4,02	4,02
61843	3,71	3,71	3,71	3,71
68429	0,54	0,53	0,54	0,54
76311	0,39	0,40	0,39	0,39

79378	1,73	1,74	1,73	1,73
82010	0,01	0,02	0,01	0,01
86375	0,48	0,48	0,48	0,48
92017	0,03	0,02	0,03	0,03
98797	1,00	1,00	1,00	1,00

Tabla D.18: Tiempos empleados en cada ataque de fuerza bruta con 50 hebras para claves aleatorias para el archivo textoplano3.txt.

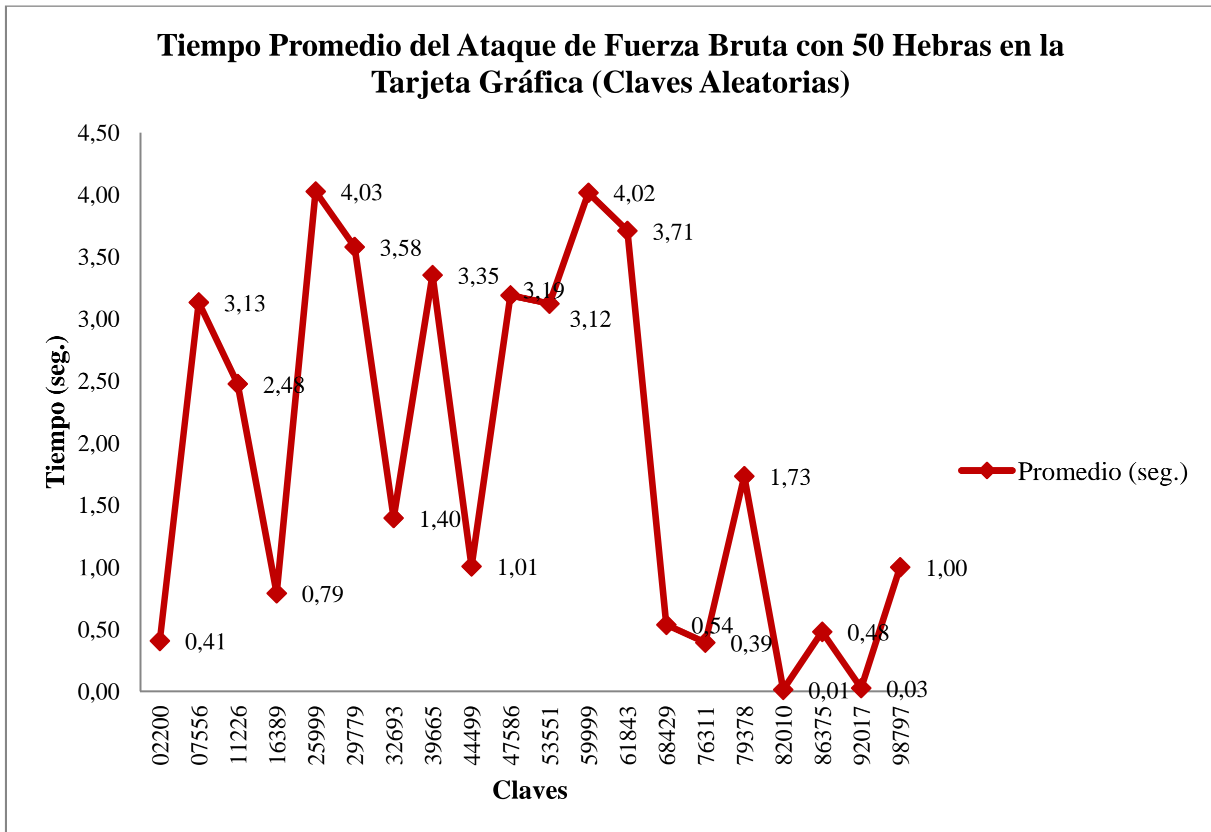

Figura D.18: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 50 hebras para claves aleatorias para el archivo textoplano3.txt.

D.2.2.4 Ataque de Fuerza Bruta con 100 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
02200	0,42	0,42	0,43	0,42
07556	1,16	1,16	1,16	1,16
11226	0,47	0,47	0,48	0,47
16389	0,82	0,82	0,81	0,82
25999	2,08	2,07	2,07	2,07
29779	1,61	1,62	1,62	1,62
32693	1,45	1,44	1,44	1,44
39665	1,38	1,38	1,39	1,38
44499	1,04	1,04	1,05	1,04
47586	1,23	1,22	1,22	1,22
53551	1,14	1,15	1,15	1,15
59999	2,08	2,07	2,08	2,08
61843	1,75	1,75	1,75	1,75
68429	0,90	0,89	0,90	0,90
76311	0,65	0,66	0,65	0,65
79378	0,79	0,80	0,79	0,79
82010	0,03	0,03	0,03	0,03
86375	0,79	0,78	0,79	0,79
92017	0,04	0,05	0,04	0,04
98797	0,42	0,42	0,43	0,42

Tabla D.19: Tiempos empleados en cada ataque de fuerza bruta con 100 hebras para claves aleatorias para el archivo textoplano3.txt.

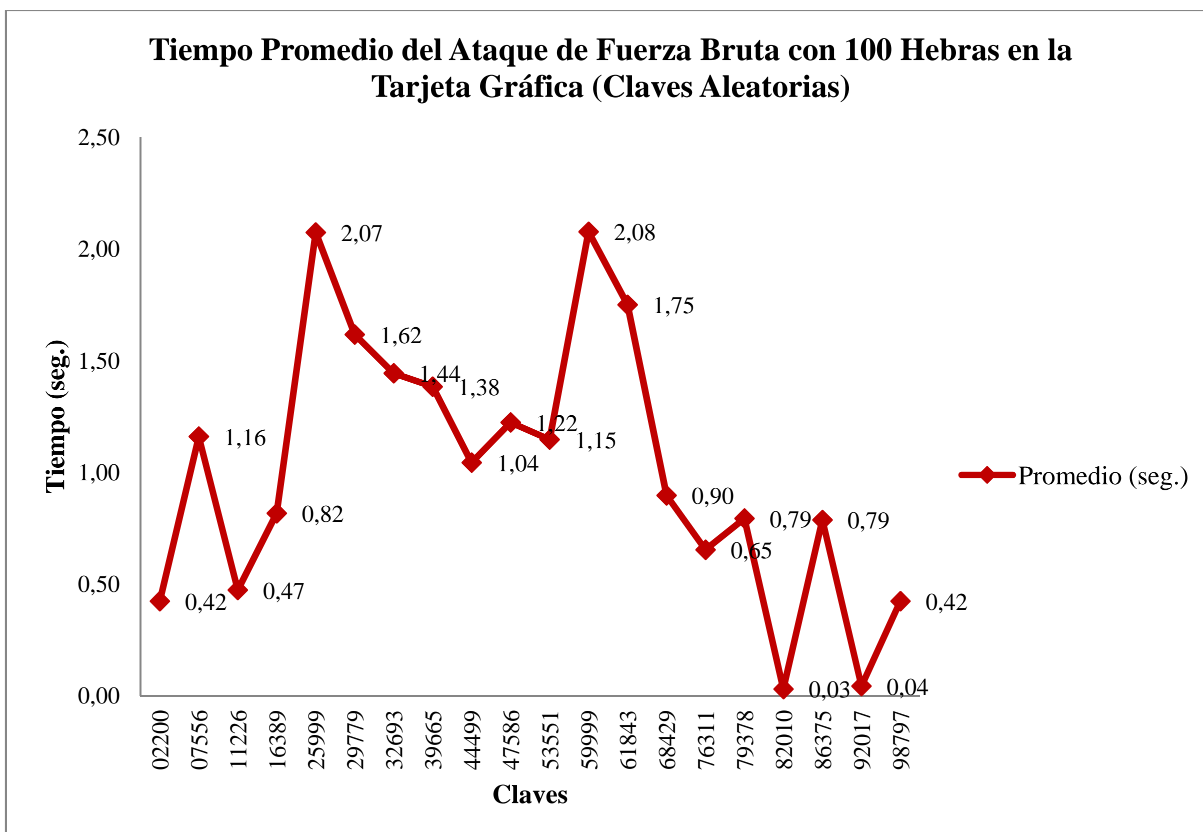

Figura D.19: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 100 hebras para claves aleatorias para el archivo textoplano3.txt.

D.2.2.5 Ataque de Fuerza Bruta con 200 Hebras

Claves	Prueba N° 1 (seg.)	Prueba N° 2 (seg.)	Prueba N° 3 (seg.)	Promedio (seg.)
02200	0,45	0,45	0,45	0,45
07556	0,13	0,14	0,12	0,13
11226	0,52	0,51	0,51	0,51
16389	0,86	0,87	0,86	0,86
25999	1,11	1,11	1,11	1,11
29779	0,63	0,63	0,62	0,63
32693	0,44	0,44	0,44	0,44
39665	0,38	0,37	0,38	0,38
44499	1,11	1,11	1,10	1,11
47586	0,20	0,20	0,20	0,20
53551	0,12	0,13	0,12	0,12
59999	1,11	1,11	1,11	1,11
61843	0,76	0,77	0,76	0,76
68429	0,96	0,96	0,96	0,96
76311	0,69	0,69	0,69	0,69

79378	0,83	0,84	0,85	0,84
82010	0,03	0,03	0,03	0,03
86375	0,83	0,83	0,84	0,83
92017	0,04	0,05	0,05	0,05
98797	0,24	0,24	0,25	0,24

Tabla D.20: Tiempos empleados en cada ataque de fuerza bruta con 200 hebras para claves aleatorias para el archivo textoplano3.txt.

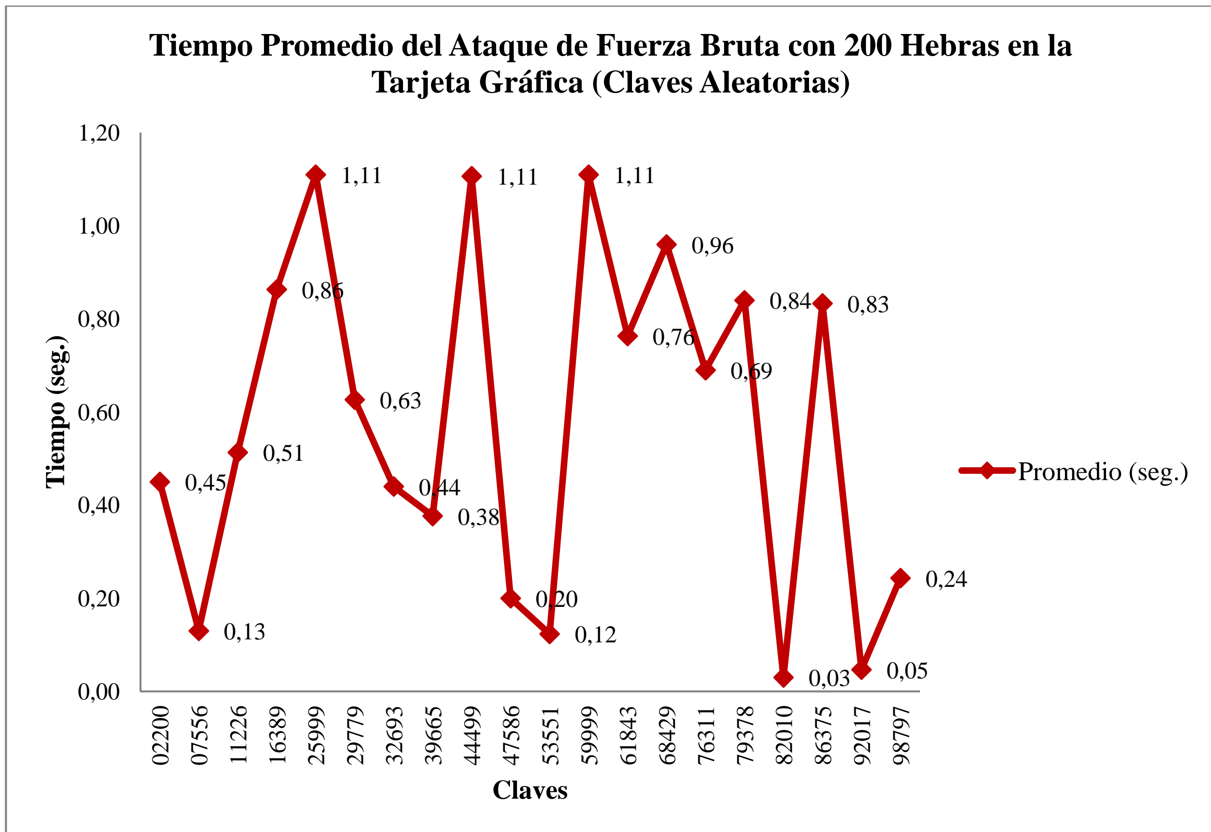

Figura D.20: Gráfico de comparación entre claves y tiempo empleado en cada ataque de fuerza bruta con 200 hebras para claves aleatorias para el archivo textoplano3.txt.

13 Referencias

[1] D. Vanden Boer, "*General-purpose computing on gpus,"* M.S. thesis, Transnationale Universiteit Limburg, Diepenbeek, Bélgica, 2005.

[2] M. Guzmán, "*Uso de tecnologías de hardware gráfico en el apoyo al realismo en entornos virtuales arquitectónicos*," M.S thesis, Universidad de Colima, Colima, México, 2004.

[3] T. Tamasi, "The evolution of computer graphics," in Nvision 08, San Jose, U.S.A., 2008.

[4] T. Monk, "7 years of graphics". [Online]. Available: http://accelenation.com/?ac.id.123.6

[5] I. Buck et al., "Brook for gpus: stream computing on graphics hardware," in *International Conference on Computer Graphics and Interactive Techniques ACM SIGGRAPH, 2004.*

[6] T. Jansen, "*GPU++ an embedded gpu development system for general-purpose computations*," Ph. D. thesis, Technische Universität München,München, Alemania, 2007.

[7] M. Harris, "Introduction to cuda," in *SIGGRAPH2007 Conference*, 2007.

[8] Pande Lab Stanford University, "Folding@home". [Online]. Available: http://folding.stanford.edu/

[9] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo and E. S. Quintana-Ortí, "Solving dense linear systems on graphics processors," Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaime I, Castellón, España, Informe Técnico ICC 02-02-2008, 2008.

[10] Elemental Technologies Inc., "Badaboom media converter". [Online]. Available: http://www.badaboomit.com/

[11] Billconan and Kavinguy, "A neural network on gpu,". [Online]. Available: http://www.codeproject.com/KB/graphics/GPUNN.aspx.

[12] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P.Markatos and S. Ioannidis, "Gnort: high performance network intrusiondetection using graphics processors," in *11th International Symposium, RAID 2008*, Cambridge, MA, U.S.A., 2008.

[13] S. A. Manavski, "CUDA compatible gpu as an efficient hardware accelerator for aes cryptography," in *IEEE International Conference on Signal Processing and Communications (ICSPC 2007)*, Dubai, United Arab Emirates, 2007.

[14] Elcomsoft Co. Ltd., "Elcomsoft distributed password recovery". [Online]. Available: http://www.elcomsoft.com/edpr.html

[15] NVIDIA, “NVIDIA GeForce GTX 200 GPU Architectural Overview,” NVIDIA Corporation, Santa Clara, CA, U.S.A., Tech. Brief TB-04044-001_v01, 2008.

[16] Madboxpc, “Review NVIDIA geforce gtx 280”. [Online]. Available: http://www.madboxpc.com/review-nvidia-geforce-gtx-280/

[17] D. Luebke, “NVIDIA gpu architecture and implications” in *Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08),* Seattle, WA, U.S.A., 2008.

[18] M. Chiappetta, “NVIDIA geforce gtx 285 unveiled”. [Online]. Available: http://hothardware.com/Articles/NVIDIA-GeForce-GTX-285-Unveiled/

[19] NVIDIA, “NVIDIA CUDA programming guide versión 3.2”. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf

[20] NVIDIA, “NVIDIA CUDA programming guide versión 3.1”. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf

[21] M. Lucena, *Criptografía y Seguridad en Computadores*. España: Creative Commons, 2010.

[22] S. Garfinkel, G. Spafford, and A. Schwartz, *Practical UNIX And Internet Security*, Third Edition. U.S.A.: O’Reilly, 2003.

[23] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, Second Edition. U.S.A.: Wiley, 1996.

[24] G. Booch, J. Rumbaugh, and I. Jacobson, *El Lenguaje Unificado de Modelado*, España: Addison-Wesley, 2000.