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

USO DE TECNOLOGÍAS GPU PARA EL DESARROLLO DE ALGORITMOS CRIPTOGRÁFICOS

DANIEL ALEJANDRO TUREO MUÑOZ

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

Junio 2013

Pontificia Universidad Católica de Valparaíso – Chile
Facultad de Ingeniería
Escuela de Ingeniería Informática

USO DE TECNOLOGÍAS GPU PARA EL DESARROLLO DE ALGORITMOS CRIPTOGRÁFICOS

DANIEL ALEJANDRO TUREO MUÑOZ

Profesor Guía: **Jaime Briggs Luque**

Carrera: **Ingeniería Civil Informática**

Junio 2013

A mi familia por su apoyo y

permanente paciencia.

AGRADECIMIENTOS

Me gustaría agradecer a todas aquellas personas que me ayudaron en el desarrollo y me apoyaron en este trabajo, de una u otra forma.

A mi Familia, que siempre ha estado presente dándome ánimo y apoyándome para continuar.

A mi profesor guía, Jaime Briggs, por guiarme en este nuevo tema ante de mi estadía en el extranjero y presentarme este proyecto para desarrollar como trabajo final. Gracias por su paciencia, dedicación y conocimientos.

A todos ellos, muchas gracias.

Índice

Lista de Abreviaturas ..... **i**

Lista de Figuras ..... **ii**

Lista de Tablas ..... **iii**

Resumen ..... **iv**

Abstract ..... **v**

1 Presentación del tema **1**

1.1 Introducción *1*

1.2 Objetivos del proyecto *2*

1.2.1 Objetivo General 2

1.2.2 Objetivos Específicos 2

1.3 Organización del Texto *2*

2 Graphics Processing Units **3**

2.1 Introducción *3*

2.2 Unidad de Procesamiento Gráfico o GPU *4*

2.3 GPGPU o General Purpose Computation on Graphics Processor Units *4*

2.4 Aplicaciones Actuales *7*

3 CUDA **8**

3.1 Introducción *8*

3.2 Concepto de CUDA *9*

3.3 Elementos Claves de CUDA *10*

3.3.1 Kernel, Rejilla (Grid) e Hilos (Threads) 11

3.3.2 Memoria 12

3.3.3 Variables en CUDA 14

3.3.4 Funciones en CUDA 15

3.3.5 Funciones Atómicas 15

3.3.6 Ejemplo 16

3.4 Funcionamiento de CUDA en el Proyecto *17*

3.5 Aplicaciones Básicas en CUDA *17*

3.6 Buenas Prácticas en CUDA *20*

4 Criptografía **21**

4.1 Introducción *21*

4.2 Concepto de Criptografía *21*

4.3 Criptosistemas Simétricos *23*
4.3.1 Cifrado por Bloques 23
4.3.2 Cifrado por Flujo 25

4.4 Criptosistemas Asimétricos *26*

4.5 Criptografía con GPU *27*

4.6 Criptoanálisis *27*
4.6.1 Tipos de Ataques 28
4.6.2 Ataques con GPU 29

5 Uso de la tecnología GPU en la Criptografía 30

5.1 TEA o Tiny Encryption Algorithm *30*
5.1.1 Cifrado Feistel 30
5.1.2 Confusión y Difusión 32
5.1.3 Cifrado por Bloque TEA 33
5.1.4 Cifrado STEA 35
5.1.5 Ataque Dividir y Conquistar sobre STEA 36

5.2 RC4 ..... *37*
5.2.1 Descripción de RC4 37
5.2.2 Debilidad del IV en el KSA en WEP 38
5.2.3 Ataque FMS 39
5.2.4 Ataque Korek 40

6 Resultados Obtenidos 45

6.1 Desarrollo de los algoritmos *45*
6.1.1 Implementación de STEA 45
6.1.2 Implementación de RC4 50

6.2 Resultados STEA *57*
6.2.1 STEA 57
6.2.2 Ataque Divide-and-Conquer sobre STEA 58

6.3 Resultados RC4-WEP *59*
6.3.1 Ataque Korek 59

7 Conclusiones 62

7.1 Recomendaciones *63*

8 Bibliografía 64

Lista de Abreviaturas

AES: Advanced Encryption Standard

CUDA: Compute Unified Device Architecture

DES: Data Encryption Standard

FEAL: Fast data Encipherment Algorithm

GPGPU: General purpose computation on Graphics Processing Unit

GPU: Graphics Processing Unit

OpenGL: Open Graphics Library

RC4: Rivest Cipher 4

RSA: Rivest, Shamir, Adleman

TEA: Tiny Encryption Algorithm

Lista de Figuras

Figura 2.1 CPU vs GPU 4

Figura 2.2 GFLOP/s CPU (Intel) vs GPU (NVidia) 6

Figura 2.3 Ancho de Banda tarjetas NVidia v/s procesadores Intel 6

Figura 3.1 Arquitectura de CUDA 9

Figura 3.2 Ejecución aplicación en CUDA con C 10

Figura 3.3 Threads, Blocks y Grids 12

Figura 3.4 Vista lógica de CUDA 13

Figura 4.1 Criptosistema convencional 22

Figura 4.2 Cifrado por bloques 24

Figura 4.3 Esquema de un cifrado por flujo 25

Figura 5.1 Ronda Feistel 31

Figura 5.2 Red Feistel 32

Figura 5.3 Manejo de clave en un cifrado por flujo 39

Figura 6.1 Implementación de STEA 46

Figura 6.2 Implementación de RC4 50

Figura 6.3 Creación de Textos en Claro para STEA 58

Figura 6.4 Ataque Divide-and-Conquer 59

Figura 6.5 Tiempo ataque Korek CPU y GPU 60

Figura 6.6 Tiempo ataque Korek GPU 60

Lista de Tablas

Tabla 3.1 Declaración de variables 14

Tabla 3.2 Declaración de funciones 15

Tabla 5.1 Claves equivalentes en TEA 34

Tabla 5.2 Comparación TEA y XTEA para claves equivalentes 34

Tabla 5.3 Cifrado con STEA 35

Tabla 6.1 Encriptado STEA CPU v/s GPU 57

Resumen

El uso de la GPU en los últimos años se ha extendido a diferentes áreas científicas y no solamente para aplicaciones gráficas. Una de las nuevas áreas de aplicación es la criptografía. El objetivo principal del proyecto es modelar las pruebas, usando las ventajas de la GPU en procesamiento paralelo de datos para algoritmos criptográficos. Para esto se eligieron dos tipos de cifrados distintos pertenecientes a la criptografía simétrica, uno de ellos RC4 y el otro TEA, de tal manera para verificar las ventajas en dos campos diferentes.

La idea es ocupar algunas debilidades conocidas de ciertos algoritmos e implementarlas sobre la tarjeta gráfica verificando un mejor rendimiento. En el caso de TEA, la implementación del algoritmo sobre la GPU resultó más veloz y en el ataque, dependiendo de los ajustes, se obtuvieron resultados mejores que su implementación en la CPU. En RC4, el uso de la GPU se redujo al ataque tal cual se ocupa en la implementación sobre WEP y, tal como en el caso anterior, algunas de las distintas implementaciones presentaron mejores resultados.

Los resultados muestran que una correcta implementación sobre de GPU de una rutina criptográfica puede ser más eficiente que una hecha sobre la CPU. Una manera preliminar de obtener mejores resultados sería disminuir el traspaso de información entre la CPU y la GPU. Lo anterior demuestra que mejoras existe la posibilidad de hacer en futuras implementaciones de ataques criptográficos usando el binomio CPU/GPU.

Palabras-claves: Criptografía, Graphics Processing Unit, GPU, Compute Unified Device Architecture, CUDA, TEA, RC4

Abstract

GPU use in recent years has been extended to various fields of science and not only for graphic applications. One of the new fields of application is cryptography. The project's main objective is to model the tests, using the advantages of parallel processing GPU data for cryptographic algorithms. Two different kinds of algorithms were chosen, one belong to the symmetric and the other one to the asymmetric cryptographic.

The idea is to exploit some known weaknesses and implement certain algorithms on graphics card to verify a better performance. For TEA, the implementation of the algorithm on the GPU was faster and attacks, depending on the settings, the results were better than the CPU implementation. On RC4 the use of the GPU is reduced to such attack which deals with the implementation on WEP and, as in the previous case, some of the various implementations had better outcomes.

The results show that a correct GPU implementation of cryptographic routines may be more efficient than one made on the CPU. A preliminary way to get better results would be reducing the transfer of information between the CPU and GPU. This shows that improvement is possible in future implementations to cryptographic attacks using the binomial CPU / GPU.

Key Words: Cryptography, Graphics Processing Unit, GPU, Compute Unified Device Architecture, CUDA, TEA, RC4

1 Presentación del tema
1.1 Introducción

Las tarjetas gráficas eran chips que se encontraban habitualmente dentro de consolas de videojuegos y que al comenzarse a masificarse produjeron el crecimiento de esta industria, lo que llevó al aumento de la necesidad de mejores tarjetas que sean capaces de soportar altos requerimientos de procesamiento gráfico. Sin duda los requerimientos no eran simples e incluían, por ejemplo, procesamiento de gráficos en 3D, los cuales eran más complejos que procesar gráficos en 2D. Con los años, esta ventaja fue traspasada a los computadores personales y luego, este concepto evoluciona a lo que se conoce actualmente como GPU.

En sus inicios el uso de GPU se remitió específicamente a lo gráfico, debido a las restricciones como las limitadas instrucciones que proporcionaban una programación cerca del hardware o el hecho de que la programación sobre la tarjeta era una excepción, lo que limitaba su uso. Sin embargo, gracias a los avances tecnológicos en el aumento del procesamiento paralelo, en la mayor cantidad de memoria y en la alta velocidad para acceder al dispositivo, los usos de las tarjetas de video fueron aumentando. Es por esto que uno de los mayores fabricantes en la materia, NVidia, introdujo el concepto de una GPU para propósito general o General Purpose Computation on Graphics Processor Units (GPGPU) [Pankratius, 09].

A pesar de la introducción del nuevo concepto siguieron existiendo dificultades para utilizar la GPU, puesto que los conocimientos técnicos seguían siendo gráficos, como en el uso de OpenGL. Eso limitaba la utilización de las GPU a otras áreas científicas. Nuevamente es NVidia quien introdujo una solución, CUDA o Compute Unified Device Architecture, una plataforma que inicialmente era una extensión del lenguaje de programación C. Sufoco principal era el de fomentar el uso de las tarjetas NVidia para el desarrollo de aplicaciones con un propósito general, además de ocupar la GPU como un potente coprocesador de la CPU.

Existen actualmente una gama amplia de aplicaciones de CUDA [Schive *et al.*, 07], [Manavski y Valle, 08], [Preis *et al.*, 09], que son transversales a las distintas áreas científicas. Una de esas área es la criptografía [Cook *et al.*, 05], perteneciente a la seguridad informática que es muy sensible para distintos tipos de organizaciones. Uno de los desafíos y un área muy estudiada es lograr que los sistemas criptográficos no sean vulnerados que gracias a la publicación libre de códigos de algoritmos criptográficos permite explotar de mejor manera los diferentes tipos de vulnerabilidades. Es por ello que con las nuevas capacidades de procesamiento que entregan las GPU, las técnicas de quiebre de algoritmos – que en la teoría son efectivas contra sistemas criptográficos, pero que en la práctica encontraban trabas por el altísimo tiempo necesario para su ejecución – se han comenzado a implementar.

Bajo este contexto, el objetivo central de la presente investigación es implementar pruebas para el quiebre de algoritmos criptográficos sobre las GPU, ocupando técnicas conocidas que exploten las debilidades de los cifrados que se ocuparán.

La primera parte de esta investigación apuntó a conocer el estado actual de los conceptos de GPU y CUDA que servirán para el trabajo y, también, el de algunos conceptos generales en el área de la criptografía y de las técnicas para el quiebre de estos algoritmos. Esta parte fue

completada correctamente y se creó un marco conceptual que ayuda a comprender mejor la segunda parte en la que se implementaron los algoritmos criptográficos y sus diferentes quiebres.

En la segunda parte de la investigación, se definieron los algoritmos de cifrado que se ocuparían y, de la misma forma, se definieron las debilidades que se explotarían. Con ello, se delimitó el trabajo a realizar y en la última parte de este informe se explicarán las diferentes implementaciones realizadas y se mostrarán los diferentes resultados obtenidos.

1.2 Objetivos del proyecto

1.2.1 Objetivo General

Modelar pruebas de algoritmos criptográficos usando ventajas de las tecnologías GPU en el procesamiento paralelo de datos.

1.2.2 Objetivos Específicos

- Conocer el estado del arte de las GPU, CUDA y los algoritmos criptográficos.
- Definir las técnicas criptoanalíticas para el quiebre de los algoritmos elegidos.
- Verificar las ventajas de utilizar la GPU en la prueba de debilidades de algoritmos criptográficos.

1.3 Organización del Texto

Este documento está conformado por 6 capítulos. En la primera parte se presenta una introducción al proyecto presentado en el informe, se definen el objetivo general, los objetivos específicos y se muestra el plan de trabajo tentativo. Posteriormente, en el capítulo 2, se presenta la Unidad de Procesamiento Gráfico o GPU, su nacimiento como concepto al igual que GPU para propósitos generales. Luego, en el siguiente capítulo, se aborda de manera detallada la arquitectura presentada por NVidia para el manejo de sus GPU, cómo facilita el trabajo en las distintas áreas y cómo ha sido el trabajo realizado hasta ahora. El capítulo 4 presenta conceptos generales de criptografía y criptoanálisis, con el fin de mostrar dónde puede ser factible ocupar las nuevas tecnologías de procesamiento paralelo y también algunas aplicaciones de CUDA en el desarrollo de mejores algoritmos. El capítulo 5, se definen los algoritmos que se ocuparán y se detallan de forma amplia las debilidades que se explotarán. En el capítulo 6, se muestran los resultados que se han obtenido hasta el momento de la entrega del presente informe. Finalmente, se muestran las conclusiones tras el desarrollo de este documento.

2 Graphics Processing Units
2.1 Introducción

Graphics Processing Unit (GPU) o unidad de procesamiento gráfico es lo que comúnmente se conoce como tarjeta de video. Una GPU es un procesador dedicado que se preocupa exclusivamente del procesamiento de datos gráficos, lo que libera de trabajo a la CPU. Normalmente se ha utilizado para trabajar con videojuegos o en aplicaciones que requieren un alto procesamiento gráfico. Muchas compañías se han dedicado a su desarrollo, dentro de las más conocidas se encuentran Intel, NVidia y ATI.

Su aparición data de los años 70', sin embargo, hasta fines de los 80'solo una pequeña porción de computadores poseía una tarjeta aceleradora de video. Esto debido a que la demanda por una mejor gráfica todavía no tenía su auge. Es en los años 90' y con la demanda de tecnología que soportará 3D que aparecen las primeras tarjetas capaces de soportar el alto nivel de procesamiento. La primera aparición del concepto GPU fue presentada a finales de los años 90' por NVidia con la presentación de la GeForce 256. Esto da inicio a una tecnología que avanza año tras año de manera enorme, especialmente en los últimos años, donde los avances tecnológicos de una GPU superan a los que se han obtenido con las CPU. Una ventaja dada especialmente por la independencia que ha tenido una de otra. Por lo mismo, se encuentran actualmente tarjetas gráficas en una gran variedad de aparatos tales como: computadores portátiles, computadores personales, consolas de juego, servidores, etc.

La gran capacidad de procesamiento que tienen las tarjetas gráficas en la actualidad se debe, por una parte, a la gran especialización que ha alcanzado su desarrollo y, por otra parte, a la gran capacidad de procesamiento paralelo que poseen. El rendimiento se mide principalmente en las operaciones de punto flotante por segundo (o FLOP/s por las siglas en inglés) capaces de procesar. Otra característica que ha sobresalido en su evolución, se puede ver en la cantidad de procesadores que son capaces de contener actualmente. Por ejemplo, la GeForce GTX 275 posee una total de 240 procesadores, agrupados en 30 multiprocesadores, y con una capacidad de procesamiento de sobre 1000 GFLOP/s (o Giga FLOP/s) [NVidia, 09]. Sin duda ventajas que han marcado la diferencia.

Un tema de investigación por varios años ha sido aprovechar al máximo el potencial que tienen las tarjetas de video, pero como se dijo anteriormente: las tarjetas gráficas se ocupaban en un principio únicamente en el aceleramiento de gráficos. Sin embargo, con el pasar de los años la necesidad de ocupar las GPU para aplicaciones especiales fue creciendo y que finalmente dio paso al nacimiento de un nuevo concepto conocido como General Purpose computation on Graphics Processor Units (GPGPU), lo que en español se puede comprender como unidad de procesamiento gráfico para un propósito general. Es decir, ampliar el ámbito de trabajo de un GPU para procesamiento de aplicaciones no solamente de gráficos. Esto se debe básicamente a: el alto trabajo en paralelo que pueden realizar, al mayor espacio de memoria, al alto ancho de banda (Memory bandwidth) y a su bajo costo en comparación con una CPU. De ahí que nuevos modelos de programación han nacido bajo este el concepto de GPGPU, lo que ha permitido ampliar el posible uso de esta tecnología.

2.2 Unidad de Procesamiento Gráfico o GPU

Técnicamente la GPU es un procesador especializado que posee, al igual que una CPU, una DRAM (*Dynamic random access memory* o memoria de acceso aleatoria dinámica) propia. Pero, a diferencia de una CPU, poseen varias áreas de Cache, creadas para acelerar filtros de textura, que a su vez contienen sus propias áreas de Control de flujo y muchas ALU (*Arithmetic Logic Units* o unidades lógicas aritméticas) respectivamente. La Figura 2.1 muestra gráficamente cómo se pueden visualizar las diferencias [NVidia, 09]. Lo que diferencia a simple vista la arquitectura que se ocupa sobre la CPU y sobre la GPU.

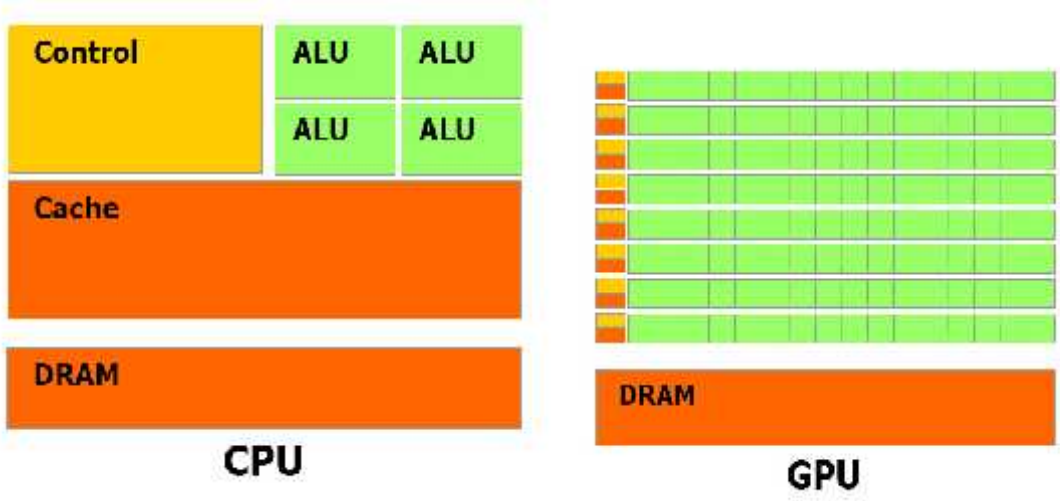


Figura 2.1 CPU vs GPU

Con la alta demanda de los últimos años de mejores GPU que trabajen a la par con la CPU, se ha llegado a niveles de tecnologías no pensados en sus inicios. Es así que,por ejemplo, la GeForce GTX 275 tiene un núcleo que posee una frecuencia de reloj de 633MHz, comparable con antiguas CPU. Otras características importantes son: una memoria DDR3 de 896 MB con un ancho de banda de 127 GB/seg y 240 procesadores. Estás características han convertido a la GPU en un coprocesador de la CPU, sobre el cual se pueden delegar ciertas funciones, en una gran variedad de aplicaciones.

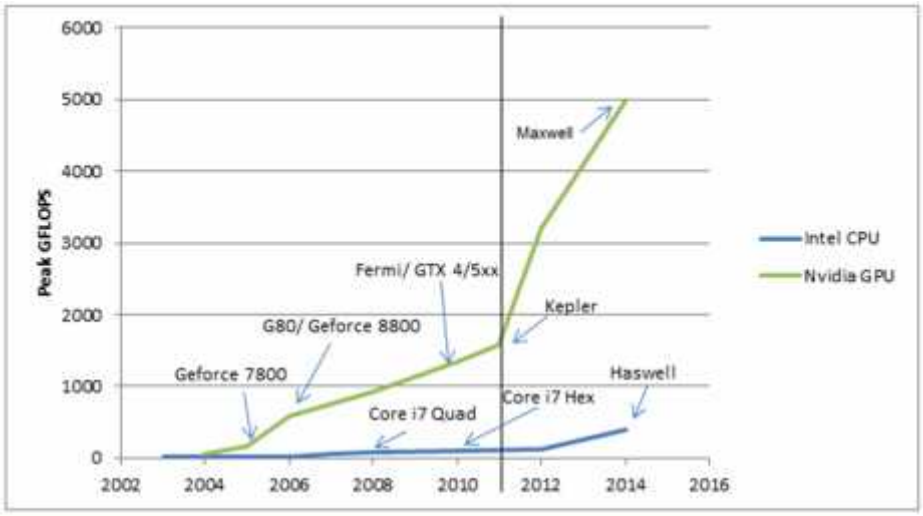
2.3 GPGPU o General Purpose Computation on Graphics Processor Units

El crecimiento del mercado, que demanda por mejores gráficos en 3D, procesamiento en tiempo real y la capacidad de programar aplicaciones personalizadas sobre la GPU, ha transformado el antiguo concepto de las tarjetas gráficas solo para el aceleramiento de gráficos que se utilizaban sobre videojuegos y programas a fines. Este nuevo concepto de utilizar la tarjeta gráfica para propósitos generales ha aparecido debido a que la GPU posee un alto paralelismo en el procesamiento, la posibilidad de tener multi-threads, la gran cantidad de

procesadores y el alto ancho de banda de la memoria [NVidia, 09]. Lo que también ha llevado a pensar que el concepto de tener una supercomputadora de escritorio no es algo tan lejano.

La GPU, o conocida también como tarjeta gráfica aceleradora, fue pensada desde un principio cómo un apoyo a la CPU en el procesamiento de información, específicamente para procesamiento de gráficos en 2D y más adelante en 3D. Es por eso, que la utilización de esta tecnología estuvo siempre pensada en videojuegos y aplicaciones gráficas profesionales. Los usos comunes en sus inicios eran el de mejorar la textura que tenía una imagen para generar una imagen desde un modelo o *rendering* y, más tarde, para acelerar la geometría de las gráficas, por último, para la rotación y traslación de los vértices en coordenadas de 2 ó 3 dimensiones. En las versiones modernas de GPU, se permite el manejo del sombreado de una imagen, que anteriormente se realizaban dentro de la CPU. Otra característica de las actuales tarjetas, es la capacidad de *framebuffer*, es decir, que los píxeles que se despliegan se encuentran en alguna zona de memoria.

Las características anteriores han convertido a las GPU en un coprocesador de las CPU muy potente y los avances que se han hecho sobre la primera van creciendo año tras año. En la Figura 2.2 se observa cómo ha avanzado, en los últimos años, la capacidad de procesamiento de alguna de las tarjetas NVidia en comparación al aumento de procesamiento de los procesadores de Intel. En la Figura 2.3 se aprecia cómo el ancho de banda es mucho mayor en las tarjetas NVidia [NVidia, 09].



	Peak GFLOPS	Year		Peak GFLOPS	Year
Intel Pentium 4	7	2003	Nvidia Geforce 6800	54	2004
Intel Pentium D	13	2005	Nvidia Geforce 7800	165	2005
Intel Core 2 Duo	23	2006	Nvidia Geforce 8800	576	2006
Intel Core 2 Quad	51	2007	Nvidia Geforce GTX 280	933	2008
Intel Core i7 Quad	70	2008	Nvidia Geforce GTX 480	1,350	2010
Intel Core i7 Hex	109	2010	Nvidia Geforce GTX 580	1,580	2011
Intel Ivy Bridge	130	2012	Nvidia Kepler	3,200	2012
Intel Haswell	400	2014	Nvidia Maxwell	5,000	2014

Figura 2.2 GFLOP/s CPU (Intel) vs GPU (NVidia)

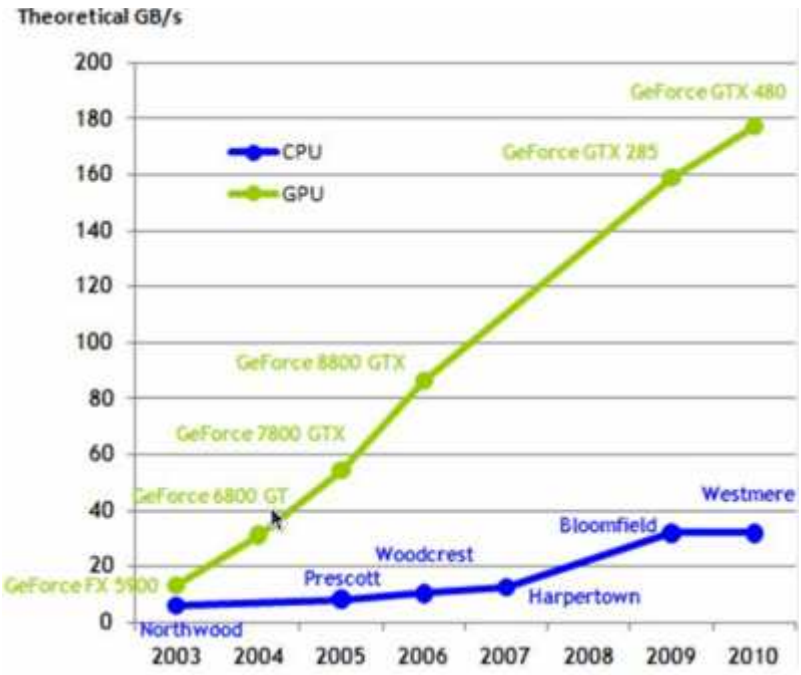


Figura 2.3 Ancho de Banda tarjetas NVidia v/s procesadores Intel

Los bajos requerimientos de control de flujo, la gran velocidad en realizar cálculos aritméticos y la baja latencia han hecho que las GPU se transformen en excelentes procesadores de datos paralelos, al aprovechar la arquitectura con que trabajan. Por ejemplo, un uso gráfico conocido es el *Rendering* en 3D, que necesita del procesamiento de una gran cantidad de pixeles y vértices de forma paralela. En este caso el mapeo de los datos que se procesan en paralelo se hace por medio de hilos (threads) y es a través de los hilos que se manejan los datos, primero de forma separada y, finalmente, se unen los resultados. La técnica de ocupar threads ha permitido ampliar la forma en que los algoritmos pueden trabajar con la CPU y la GPU al mismo tiempo. Actualmente, su uso se ha extendido a por ejemplo: el procesamiento de señales (eléctrica y electrónica), realización de simulaciones (física y química) o el mejoramiento de técnicas tanto criptográfico como de criptoanálisis.

2.4 Aplicaciones Actuales

La GPU usada para propósitos generales ha sido materia de estudio durante los últimos años y, los resultados han llevado su aplicación de manera exitosa a una gran variedad de temas tales como: análisis financieros, simulaciones biológicas y físicas, y también algoritmos criptográficos. A continuación se listan algunos ejemplos de nuevas aplicaciones no-gráficas que se han estudiado ampliamente en el último tiempo:

- Análisis financieros**:** Utilizado, por ejemplo, en el aceleramiento de análisis de fluctuación en conjunto con la formación de patrones. La necesidad de un alto poder de procesamiento fue el argumento perfecto para el uso de la GPU [Preis *et al.*, 09].
- Simulaciones biológicas**:** La búsqueda de similitudes en proteínas y en bases de datos de ADN es un tema recurrente en la biología molecular. Nuevamente, el alto costo de procesamiento debido al alto número de operaciones proporcionales al producto del largo de 2 secuencias, hizo que la utilización del algoritmo de Smith-Watermann tuviera que ser implementado en una GPU [Manavski y Valle, 08].
- Simulaciones físicas: La gran cantidad de datos que se tiene que procesar, suele ser un problema en cualquier rama científica. Dentro de la astrofísica, la simulación de la gravedad existente entre N-cuerpos varía su complejidad entre O(10) y O(10^{10}). Es por eso que se planteó la mejora del sistema GraCCA, mediante la creación de un clúster de GPUs [Schive *et al.*, 07].
- Algoritmos criptográficos: Implementaciones recientes de algoritmos basados en cifrado por bloques han sido exitosamente implementados en la GPU. En [Manavski, 07], se presenta cómo AES puede ser implementado efectivamente ocupando el modelo de programación CUDA. El principio del algoritmo sigue intacto, lo único que ha cambiado es la forma en que se ha implementado y se ha tratado de obtener una mayor eficiencia sobre la GPU. También en [Boeing, 08] se ocupa un algoritmo usando en la encriptación de comunicaciones llamado RC4 para desarrollar una implementación sobre una GPU. Los resultados difieren dependiendo del tipo de implementación que se haga, pero alguno de ellos resultan ser mejores. Al igual que los ejemplos anteriores, muchos otros algoritmos han sido implementados con éxito sobre la tarjeta gráfica y los resultados en torno a un mejor rendimiento, en muchos casos, han sido exitosos.

3 CUDA
3.1 Introducción

Originalmente las primeras tarjetas aceleradoras gráficas tenían como propósito ser un hardware especializado para el tratamiento de gráficos, habiendo algunas API disponibles para trabajar sobre ellas como OpenGL y DirectX [DirectX]. En el caso de esta última, había un problema con la dependencia de la plataforma con que tenía que trabajar (Windows). Es por eso que OpenGL se convirtió por años en la API más utilizada debido a la capacidad de adaptarse a diferentes plataformas. Entre ellas: Linux, Mac OS X, Microsoft Windows y consolas de video juegos como PlayStation 3 [OpenGL]. Y debido a su modalidad multiplataforma es que bajo el proyecto de OpenGL se agruparon compañías importantes como: AMD, Apple, NVidia, Microsoft, Intel, entre otras más.

Otra de las ventajas que presentaban estas librerías era la posibilidad de trabajar con diferentes lenguajes de programación, dándole mayor flexibilidad. Su uso tradicional se relacionaba con aplicaciones que trabajaban con gráficos en 2D y 3D. La interfaz existente proporcionaba sobre 250 funciones, que eran implementadas directamente en la tarjeta gráfica [Rosenberg, 07]. Sin embargo, la mayor limitante que presentaba OpenGL era la necesidad que había de tener conocimientos en gráfica de computadores, los cuales no eran transversales a todas las áreas. Es por ello que el principio básico que se seguía para crear una aplicación, con propósitos no gráficos, era ocupar las texturas como datos de entrada para *rendering* que era finalmente la que realizaba los cálculos. Los resultados eran finalmente escritos sobre el *framebuffer* [Amorim *et al.*, 08]. En el fondo, lo que se hacía era utilizar los conceptos gráficos y adaptarlos a las distintas problemáticas que se querían resolver.

En la utilización de las tarjetas gráficas para otros propósitos [Manavski, 07] influía mucho el principio seguido anteriormente, pero también existían otras razones más técnicas que son las que a continuación se muestran:

- La falta de representación de operaciones a nivel de bit o bit a bit.
- Los modelos limitados de programación de las API no permitían ciertas operaciones.
- Limitación en el acceso a la memoria de la GPU.
- Overhead del pipeline en gráficos fijos.

A pesar de que la necesidad existía hace años, un nuevo modelo de programación de GPU es introducido por NVidia recién en noviembre del 2006. Este modelo llamado CUDA o *Compute Unified Device Architecture* tenía como idea principal entregar las herramientas necesarias para que el uso de la GPU se extendiera a aplicaciones que no necesariamente fuesen del tipo gráfico. Esto principalmente para aprovechar las ventajas que se podían obtener utilizando el procesamiento paralelo en la aplicación en diferentes áreas de estudio.

3.2 Concepto de CUDA

El objetivo por el cual CUDA fue desarrollado era ser una arquitectura para el desarrollo de aplicaciones de propósitos generales, aprovechando las ventajas del procesamiento en paralelo que permite la GPU. Fue creado por NVidia y, por lo mismo, sólo se puede utilizar con tarjetas gráficas de esa empresa. Inicialmente, CUDA nació como una extensión del lenguaje C que funcionaba como un lenguaje de alto nivel por el cual se hacían llamado a funciones que accedían directamente a la GPU. Bajo este nuevo concepto, la GPU actúa como un coprocesador de la CPU para el procesamiento paralelo. Es decir, la parte del programa que necesita de procesamiento paralelo será ejecutada en la GPU, mientras el resto será ejecutado normalmente sobre la CPU.

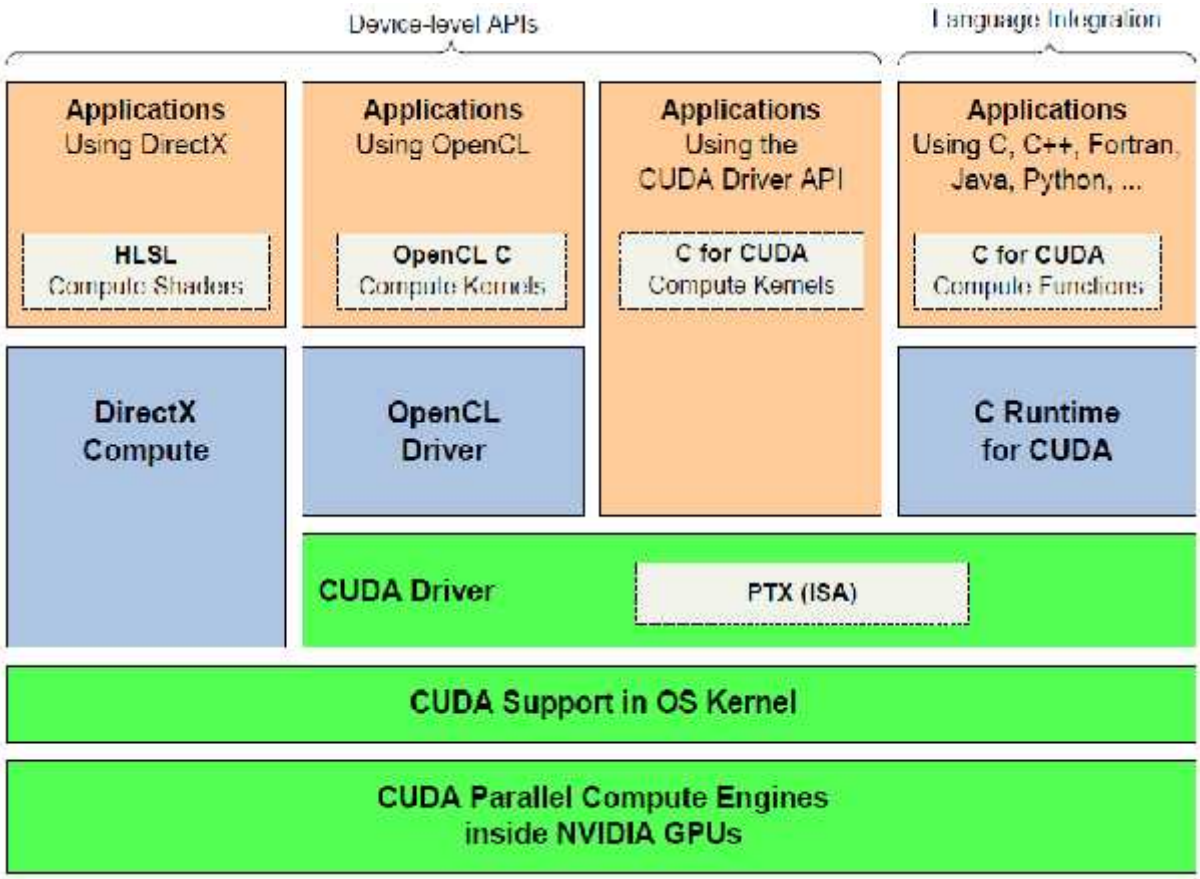


Figura 3.1 Arquitectura de CUDA

La Figura 3.1 muestra una vista general de las capas que conforman a CUDA [NVidia, 09]. Son tres los componentes principales que soportan esta arquitectura. El primer componente representa es soporte de la GPU para CUDA (no todas las tarjetas de NVidia tienen núcleos CUDA). El segundo nivel representa el soporte del sistema operativo para trabajar con CUDA. En sus inicios sólo estaba soportado por algunas distribuciones de Linux, pero actualmente Mac Os X, Windows, Solaris y otros sistemas operativos son capaces de soportarla. El tercer nivel está representado por el driver que es una API de bajo nivel para C. Esta entrega funciones a los desarrolladores para poder acceder a la GPU. PTX (*Parallel*

Thread Execution) es un conjunto de instrucciones para procesamiento paralelo de las funciones (o también llamadas *kernels*), muchas veces parecido al código *assembler*.

Las otras partes de la arquitectura se podrían dividir básicamente en dos: las API a nivel de dispositivo y los lenguajes de integración. La primera, formada por las interface de programación a nivel de dispositivo, está compuesta por DirectX, OpenGL o el driver de CUDA propiamente tal. La otra forma, que será ocupada más adelante para el desarrollo de algunas aplicaciones, es la llamada interface de programación de alto nivel. Con ella se pueden escribir funciones en C, las que pueden correr directamente en la GPU.

En la actualidad existen varios lenguajes de alto nivel que están siendo soportados tales como: C, C++, Python, Java y otros. La ventaja que presentan estos lenguajes es que permiten el uso tipos estándares como también definidos por el usuario y, que además permiten una integración de código entre el que será ejecutado por la CPU y el que será ejecutado por la GPU. El compilador para programas escritos en C, que también forma parte de CUDA, es llamada NVCC. El principio básico para la programación de una aplicación que tenga parte del código que se ejecuta en la GPU se puede ver en la Figura 3.2.

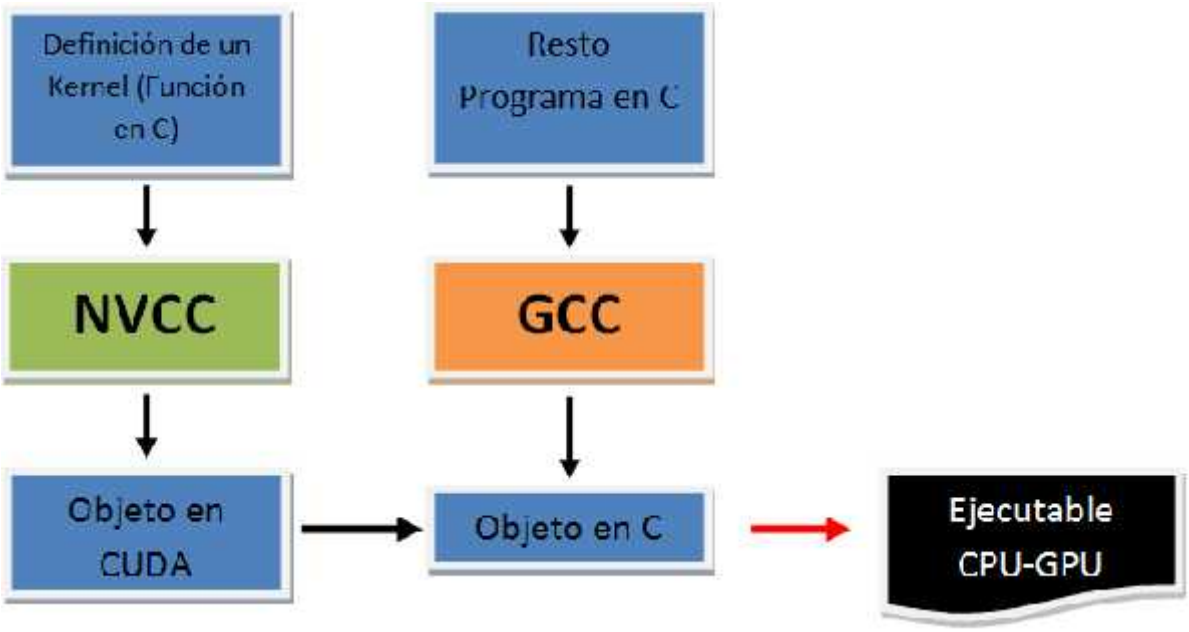


Figura 3.2 Ejecución aplicación en CUDA con C

3.3 Elementos Claves de CUDA

CUDA representa una nueva forma de programación que trabaja exclusivamente con las tarjetas de video NVidia y con el que además nacen nuevos conceptos que no aparecen al trabajar directamente con la CPU. Es por eso que el entendimiento de estos conceptos es la base para el desarrollo de futuras aplicaciones y, además, para que el rendimiento que se obtenga sobre la GPU sea mejor el que se obtiene al trabajar sobre la CPU.

3.3.1 Kernel, Rejilla (Grid) e Hilos (Threads)

Uno de los conceptos centrales, dentro de C en CUDA, es la posibilidad de definir funciones. Estas funciones son llamadas *kernels* [NVidia, 09] y cuando un kernel es llamado se ejecuta N veces en paralelo N diferentes threads de CUDA. Codificado, un kernel es definido la función mediante *__global__* y la cantidad de threads que se ocuparán usando <<<**…**>>>. El código siguiente es un ejemplo básico de una implementación:

```
// Definición del Kernel
__global__ void VecAdd(float* A, float* B, float* C)
{
 int i = threadIdx.x;
 C[i] = A[i] + B[i];
}
int main()
{
 ...
 // Llamado del Kernel
 VecAdd<<<1, N>>>(A, B, C);
}
```

Existe una serie de elementos que son propios del modelo que entrega CUDA para la programación y que a su vez ayudan a entender cómo se debería trabajar al momento de programar. Estos conceptos se encuentran divididos en niveles y nombrados en forma descendiente son: la jerarquía threads (hilos), la jerarquía de memoria y, la barrera de sincronización.

A su vez, los threads se pueden identificar mediante tres dimensiones. Para definir un thread dentro del kernel, se ocupa la etiqueta *threadIdx* y se pueden ocupar una, dos, o tres dimensiones para el índice del thread, lo que forma bloques de threads de uno, dos, o tres dimensiones. El índice de un thread y el Id de un thread se ven de esta forma: Para un bloque de una dimensión, ambos son el mismo número; para un bloque de dos dimensiones de tamaño (B_x, B_y) con (x, y) como índice del thread, el thread ID sería (x + y B_x); para un bloque de tres dimensiones de tamaño (B_x, B_y, B_z), con (x, y, z) de índice del thread, el thread ID sería (x + y B_x+ z B_xB_y). De la misma forma que para un thread, pero con una dimensión menos, la forma de identificar un bloque puede hacerse por medio de una ó dos dimensiones y para los hilos puede ser de una, dos ó tres dimensiones. Este esquema queda ejemplificado de mejor manera en la Figura 3.3.

Cada rejilla, o *Grid*, está formado por un conjunto de bloques de hilos y cada uno de estos bloques tiene un número limitado de hilos, cuyos hilos son capaces de cooperar entre sí. Esto significa que hilos de un mismo bloque pueden intercambiar datos de manera rápida, usando la memoria compartida, y que también pueden sincronizar su acceso a los datos. Esto se limita solo a hilos de un mismo bloque, ya que hilos de diferentes bloques no pueden cooperar entre ellos. Esta independencia que existe en los bloques permite que se puedan ejecutar de cualquier forma en que se planifique, puesto que se puede delegar trabajar a los diferentes bloques. Esto último es una ventaja al momento de escribir código que escala dependiendo del número de núcleos, por ejemplo.

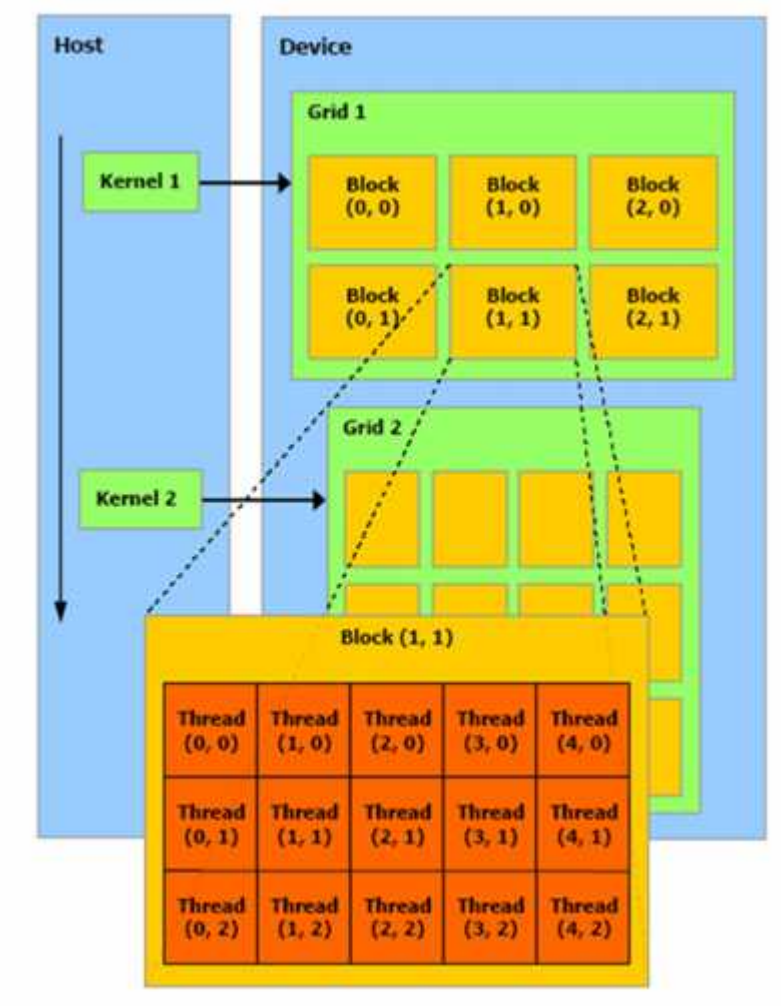


Figura 3.3 Threads, Blocks y Grids

Las actuales GPU son capaces de soportar bloques con más de 512 threads. En base a esto, se puede hacer un cálculo del número total de threads siendo igual al número de threads por bloque por la cantidad de bloques del dispositivo. Los bloques dentro de una rejilla pueden ser accedidos desde un kernel por medio de la variable *blockIdx*. Mientras que la dimensión de un bloque puede ser accedido por medio de la variable *blockDim*.

3.3.2 Memoria

Una de las ventajas más relevantes en CUDA, que nace de la forma en que se encuentra estructurada la GPU, es la capacidad de acceder a los datos desde múltiples espacios de memoria. Cada thread tiene su propio espacio de memoria llamado espacio local. De la misma forma cada bloque tiene una memoria compartida para todos su threads, con tiempos de vida idénticos y esta memoria actúa con una latencia baja, muy cerca de cada núcleo. De manera complementaria, todos los threads de cada bloque pueden acceder a una memoria llamada memoria global. Además, existen otros dos espacios de memoria, de sólo lectura, accesibles para todos los threads: La memoria constante y la memoria de textura.

La Figura 3.4, es una vista lógica de lo que sucede en cada rejilla [Pankratius, 09]. La figura permite visualizar cómo interactúan las memorias. También cada hilo está asociado a un registro. Cada hilo puede leer y escribir un registro, el espacio de memoria local, el espacio de memoria común para el bloque y el espacio de memoria global para una rejilla. Sobre el espacio de la memoria de datos constantes y de la memoria de textura sólo es posible hacer lecturas, puesto que es el host el que las modifica. Esta es la única forma en que se produce comunicación entre host y GPU.

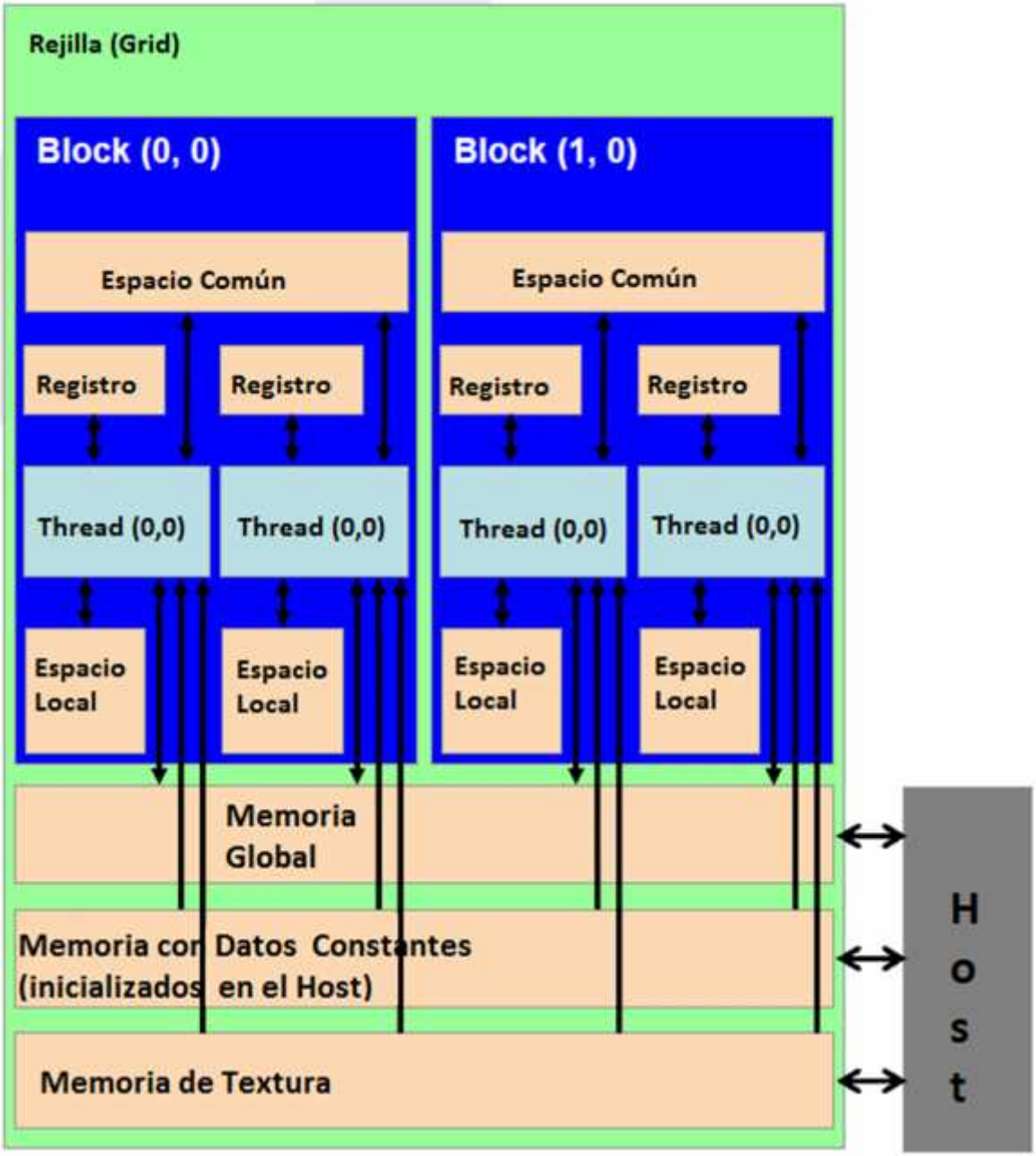


Figura 3.4 Vista lógica de CUDA

Para ejemplificarlo a algo más conocido como lo que sucede entre el software y el hardware, dentro del software se encuentran los threads que son manejados por el procesador

de threads en el hardware. Un bloque de threads es manejado por el multiprocesador en el hardware y una rejilla es controlada por el dispositivo. Esto es lo que hace que al ejecutar una aplicación, solo un kernel pueda ser llamado al mismo tiempo por la GPU.

Como se señaló anteriormente, los threads que pertenecen a un mismo bloque pueden compartir datos y además se puede hacer que trabajen de manera ordenada. Para sincronizar esto se hace un llamado a *__syncthreads()*. Esta función sirve para ordenar a las threads de un bloque para esperar a acceder a los datos mientras otro thread se está ejecutando, es decir, actúa como una barrera de sincronización y es la única forma de sincronizar a los threads en ejecución.

Otras llamadas importantes para el manejo de la memoria se pueden resumir en:

- *cudaMalloc()* para la creación de variables globales cuyos argumentos necesarios son la dirección del puntero al inicio de la localización del dato y el tamaño del dato.
- *cudaFree()* para liberar la memoria donde es necesario únicamente el puntero.
- *cudaMemcpy()* para transferir datos entre memorias. Se deben especificar 4 parámetros: el puntero de la fuente y del destino, el número de bytes a copiar y el tipo de transferencia que puede ser Host a Host, Host a GPU, GPU a Host o GPU a GPU.

3.3.3 Variables en CUDA

La estructura de CUDA permite a su vez definir diferentes tipos de variables. Cada una tiene un propósito específico y la correcta elección del tipo de variable depende de dónde se requiera ocupar. La Tabla 3.1 muestra cómo se realiza la declaración de las variables disponibles y, a su vez, de la memoria en donde se alojan, la visibilidad (rejilla o hilos) que tiene y la vida de utilización.

Tabla 3.1 Declaración de variables

Declaración de variables	Memoria	Visibilidad	Vida
__device__ __local__ int localVar;	Local	Hilos	Hilos
__device__ __shared__ int sharedVar;	Común	Bloque	Bloque
__device__ int globVar;	Global	Rejilla	Aplicación
__device__ __constant__ int constVar;	Datos Constantes	Rejilla	Aplicación

3.3.4 Funciones en CUDA

De manera similar a la definición de variables, CUDA permite la definición de funciones que puedan ser ejecutadas en diferentes lugares para los distintos propósitos. Existen básicamente tres tipos de funciones: *device*, *global* y *host*. Las primeras dos tienen la virtual de que solo se ejecutan en la GPU, pero una función de tipo *global* es llamada desde el host mientras que la *device* solo desde la GPU. Adicionalmente, las funciones *device* y *global* tienen problemas al no soportar recursión, declaración estática de variables dentro de la función y cantidad de variables en el argumento. Por otro lado, las funciones *host* se llaman y se ejecutan en la CPU similar a una función común en C. La Tabla 3.2 muestra en detalle las características de cada tipo de función permitida.

Tabla 3.2 Declaración de funciones

Declaración de funciones	Sólo ejecutada en	Sólo llamada desde
__device__ float gpuFunction()	GPU	GPU
__global__ void kernelFunction()	GPU	Host
__host__ float hostFunction()	Host	Host

Una función que será ejecutada sobre la GPU tiene que contener una serie de parámetros específicos, de otra forma no será posible su correcta ejecución. Se tiene que definir el número de bloques que hay en la rejilla, el número de hilos en el bloque y el tamaño total de la memoria que se asignará dinámicamente. Los parámetros serán analizados por la función y luego de eso pasados de la GPU a la memoria común. Una forma muy simple de ejemplificar esto se puede ver en el siguiente ejemplo:

```
__global__ void kernelFunktion(params);

// Llamada al kernel
kernelFunktion<<<GridDim, BlockDim, BytesSharedMem>>> (params)
```

3.3.5 Funciones Atómicas

Existen algunas funciones atómicas que tienen la finalidad de realizar una operación en una palabra de 32 ó 64 bit. Estas operaciones pueden ser del tipo lectura, escritura o modificación. Algunos ejemplos de estas operaciones son: *atomicAdd*(), *atomicSub*(), *atomicMin*(), *atomicMax*(), *atomicInc*(), *atomicDec*(), *atomicAnd*(), *atomicOr*(), *atomicXor*(), *atomicExch*(), *atomicCAS*(). Las 2 últimas sólo pueden ser utilizadas en la memoria global. Un ejemplo de funciones atómicas se puede ver a continuación:

```
int atomicSub(int* address, int val);
unsigned int atomicSub(unsigned int* address,unsigned int val);
```

Se lee una valor de 32 ó 64 bit (*antiguo*) que es colocado en memoria (global o compartida), luego se calcula *antiguo – val* y se almacena el resultado en el mismo espacio de memoria. Estás 2 operaciones, la de asignación de memoria y la de cálculo, son realizadas por una sola transacción atómica.

3.3.6 Ejemplo

Para ejemplificar varios de los conceptos anteriormente vistos y agruparlos en un ejemplo concreto, se realizará el cálculo de C = A + B. Se dividirá el problema en la parte que se calculará en la GPU y la parte que se ejecuta en el host.

```
// Función en la GPU
global__ void vecAdd(float* A, float *B, float *C){
      int i = threadIdx.x + blockDim.x * blockId.x;
      C[i] = A[i] + B[i];
      }
// Llamada en el Host
int main(){
// memoria para A, B y C en el host
float *h_A = (float *)malloc(N*sizeof(float));
float *h_B = (float *)malloc(N*sizeof(float));
float *h_C = (float *)malloc(N*sizeof(float));

// memoria para A, B, C en la GPU
float *d_A, d_B, *d_C;
cudaMalloc( (void**)&d_A, N*sizeof(float));
cudaMalloc( (void**)&d_B, N*sizeof(float));
cudaMalloc( (void**)&d_C, N*sizeof(float));

// Copiar el contenido desde el host a la GPU
cudaMemcpy(d_A, h_A, N*sizeof(float), cudaMemcpyHostToDevice) );
cudaMemcpy(d_B, h_B, N*sizeof(float), cudaMemcpyHostToDevice) );

// Suma de vectores de largo N, con N/256 bloques y con 256 hilos por
Bloque
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);

// Copiar resultado a la CPU en h_C
cudaMemcpy((void *) h_C, (void *) d_C, N*sizeof(float),
cudaMemcpyHostToDevice) );
// Liberar memoria
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

La primera parte del ejemplo es la función que se define para el cálculo del vector, que será la que se ejecuta sobre la GPU. Los parámetros que recibe la función son tres punteros de tipo flotante, se define un valor entero *i*, que está representado por la cantidad de hilos disponibles y, finalmente se hace el cálculo para cada posición del arreglo. La segunda parte del ejemplo, es la que se ejecuta en la CPU y en ella se definen dos tipos de variables: una que guardará los resultados en el host (*h_A*, *h_B*) y otra que se ocupará dentro de la GPU (d_*A*, *d_B*, *d_C*). También se les da memoria, acorde con lo que se ocupará, a todas la variables, se hace el traspaso de los a datos a la GPU y, luego se hace el llamado a la función para el

cálculo. Finalmente, es necesario copiar el valor desde la GPU a la CPU y liberar la memoria de las variables ocupadas.

NVidia ha provisto, como ayuda a los programadores, una serie de manuales entre los cuales se encuentra el manual de programación de CUDA, la guía de buenas prácticas, el manual de referencia de CUDA y un pequeño manual para empezar. Además de eso, hay una página para desarrolladores (http://developer.download.NVidia.com), en donde se puede encontrar más ayuda relacionada con el tema.

3.4 Funcionamiento de CUDA en el Proyecto

Para el desarrollo del proyecto, se ocupó una tarjeta de video NVidia modelo GTX 275. Algunas de las principales características que posee la tarjeta son: 240 CUDA cores, 8 procesadores, 30 multiprocesadores (la misma cantidad que la GTX 280 y GTX 285), 896 Mb de RAM y una velocidad de transferencia máxima de 127 GB/seg.

El driver de CUDA que se ocupó para el desarrollo es el 2.3 para Linux y es la misma versión para el SDK y el Toolkit. En principio, si la instalación de las herramientas necesarias resulta exitosa, los ejemplos que vienen integrados deberían funcionar correctamente. Sin embargo, mucho de los ejemplos que se intentaron compilar no funcionaron y se tuvo que proceder con la compilación de forma manual para cada uno. De los 17 que finalmente no se compilaron se llegó a la conclusión de que el problema radicaba en la librería *GLU*. A pesar de eso, la prueba de otros programas no presentó problemas y al finalizar las implementaciones realizadas no se presentaron más problemas con la librería

Todos los programas que se nombran en el trabajo fueron desarrollados para C. Dentro de los ejemplos que vienen en el SDK existen algunos que están hechos en C++, lo que representa la versatilidad al momento de trabajar sobre la GPU. También se trató de hacer desarrollos sobre Java mediante jcuda, pero se optó por desechar la opción puesto que no era uno de los objetivos del proyecto el probar las diferentes plataformas de desarrollo. Sin duda, que trabajar con diferentes lenguajes en CUDA puede ser un buen punto de comparación para decir cuáles presentan un mejor rendimiento. También puede ampliar los futuros trabajos que se desarrollen en esta plataforma, pues incluso ya existe la posibilidad de trabajar en otros sistemas operativos diferentes a Linux.

Todos los problemas surgidos después de pasar esta etapa inicial del proyecto estaban relacionados con conocimientos de programación, en espacial con temas relacionados directamente con el trabajo sobre la GPU. La mayoría de ellos se resolvían con el pasar del tiempo, después de realizar pruebas a distintas alternativas de códigos.

3.5 Aplicaciones Básicas en CUDA

Antes de comenzar el desarrollo de algoritmos más avanzados, por ejemplo, el de algoritmos criptográficos, fue necesario desarrollar pequeñas aplicaciones que ayudarán a entender qué hay que implementar sobre la tarjeta gráfica y qué hay que dejar trabajando sobre la CPU. También, cómo se debe definir una función para que funcione correctamente

sobre la GPU, es decir, para aprovechar todas las ventajas de la tarjeta no transformándose en una implementación inútil.

Los siguientes ejemplos realizará la suma lineal de dos arrays de largo N. Para ver las diferencias en las implementaciones, se ocuparán dos formas de ocupar la GPU para su cálculo.

Primera forma:

```
// Función en la GPU
__global__ void vecAdd(float* A, float *B, float *C){
      int n;
      for(n=0; n < N; n++)
            C[n] = A[n] + B[n];
      }
```

Esta primera forma de implementación tiene la particularidad de que es realizada sobre la GPU, pero no aprovecha las ventajas del procesamiento paralelo que se puede hacer con los hilos y bloques. Esto se debe principalmente a que el contador *n* aumenta linealmente, sin tener una relación directa con la cantidad de bloques e hilos que se definan en la llamada a la función. Sin duda esta forma de implementación no presenta mayores ventajas y no es el camino correcto a seguir.

Segunda forma:

```
#define N 10000000
// Función en la GPU
__global__ void vecAdd(float* A, float *B, float *C){
      int i =  threadIdx.x + blockDim.x * blockId.x;
      C[i] = A[i] + B[i];
}
int main(){
…
// Suma de vectores de largo N, con N/256 bloques y con 256 hilos por
Bloque
      vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
 …
      }
```

En la segunda forma, vista en la sección 3.3.6, la función hace uso de las ventajas de la GPU por medio del uso de los hilos disponibles. En ella, la definición del kernel se hace para un total de $N/256$ bloques con 256 diferentes threads por bloque. Esto se puede realizar de la misma forma para la primera forma mostrada, pero los resultados no cambiarían. El cambio más importante es que la función *vecAdd* define un entero *i* por medio de la identificación del bloque dentro de un grilla (*blockId.x*) y la dimensión del hilo en el bloque (*blockDim.x*). Además de eso, se utiliza un thread por medio de una dimensión (*threadIdx.x*). Esto hace que el procesamiento para cada valor del vector la realice un hilo a la vez, lo cual es mucho más eficiente que recorrer todo el vector.

Sin duda, el ejemplo anterior no representa ninguna dificultad desde un punto de vista de la implementación, incluso se puede realizar con una definición más simple del kernel. Sin embargo, las diferentes pruebas realizadas sobre la implementación base indican que al hacer

cambios en los valores del kernel, el rendimiento que se obtiene difiere. Por ejemplo, al cambiar <<<N/256, 256>>> por <<<N, 256>>> el tiempo de procesamiento sube en un 25%. Por lo tanto, es importante tener en cuenta y definir de buena forma los parámetros que serán entregados al kernel, de tal manera de optimizar el rendimiento que se obtenga.

Otras consideraciones que se pueden tomar con relación al manejo de threads es ocupar más de una dimensión. En los ejemplos anteriores la única dimensión ocupada fue x, que se relacionaba con un array. Si queremos trabajar de una forma más avanzada ocupando matrices [Gratton, 07], una opción utilizada se presenta a continuación:

```
 //kernel para la descomposición del bloque superior izquierdo de la
 matriz
 __global__ void d_choldc_topleft(float (*mat)[MAT_SIZE],int boffset){
 // Declaración de 2 variables para 2 dimensiones en un bloque
   int tx = threadIdx.x;
   int ty = threadIdx.y;

    // Definición en la memoria compartida
    __shared__ float topleft[BLOCK_SIZE][BLOCK_SIZE+1];

   topleft[ty][tx]=mat[ty+BLOCK_SIZE*boffset][tx+BLOCK_SIZE*boffset];

 // Hace posible la cooperación entre threads
   __syncthreads();

float diagelem,fac;
 // ty hace de fila y tx de columna
   for(int k=0;k<BLOCK_SIZE;k++){
           __syncthreads();
           fac=rsqrtf(topleft[k][k]);
           __syncthreads();
           if ((ty==k)&&(tx>=k)){
                   topleft[tx][ty]=(topleft[tx][ty])*fac;
           }
           __syncthreads();
           if ((ty>=tx)&&(tx>k))
   topleft[ty][tx]=topleft[ty][tx]-topleft[tx][k]*topleft[ty][k];
   }
   __syncthreads();
…
}
```

Lo más importante de las implementaciones anteriores es que existen recomendaciones para utilizar las ventajas del procesamiento paralelo de manera óptima. En la sección siguiente se mostrarán algunas buenas prácticas de programación que ayudan a obtener un mejor rendimiento. Principalmente, lo que se busca es ocupar de mejor manera las ventajas de la GPU. En capítulos posteriores y teniendo conceptos nuevos, se irán utilizando algunas recomendaciones y comprobando si están acordes a lo que se requiere.

3.6 Buenas Prácticas en CUDA

Como parte de la ayuda que entrega NVidia a los desarrolladores, existe un manual de buenas prácticas para la programación en C con CUDA [NVidia, 09a]. Algunos puntos importantes para tener en cuenta se señalan a continuación:

- Los datos nunca tienen que ser dependientes de otros: significa que los datos que pueden ser modificados nunca tienen que depender de otros.
- Maximización de la intensidad aritmética: la recolección de datos desde la memoria es un trabajo pesado. Para un mayor aprovechamiento de la velocidad de transmisión, es recomendable trabajar sobre datos que ya se encuentren cargados en memoria.
- Trabajar sobre la GPU: minimizar la transferencia de datos entre la CPU y la GPU. Las estructuras cargadas en la GPU tienen que ser ocupadas lo más rápido posible, de manera de eliminarlas justo después de su uso.
- Distribuir el trabajo ocupando toda la capacidad de la GPU: ocupar la mayor cantidad de bloques disponibles en el dispositivo. Entre más hilos por bloques mejor, pero a su vez se tendrán menos registros disponibles. Teniendo en cuenta el 100% de la capacidad de un multiprocesador se pueden manejar hasta 768 hilos. Lo que significa que 2 bloques manejan 384 hilos, 3 bloques 256 threads, 4 bloques 192 hilos, 6 bloques 128 hilos y 8 bloques 96hilos. En un grilla existen al menos 100 bloques disponibles.

Como el interés de este proyecto no es ser un manual para la programación de aplicaciones generales sobre CUDA, sino más bien utilizarlo para aplicaciones específicas no se hará mayor hincapié en las repercusiones que puedan tener estos consejos y tampoco en cómo solucionar algunos problemas asociados. Sin embargo, con el avance del proyecto aparecerán algunas recomendaciones específicas que solucionaron las implementaciones realizadas.

4 Criptografía
4.1 Introducción

La necesidad que existe por transmitir mensajes, que únicamente un emisor y un receptor de confianza puedan entender, es tan antigua como la necesidad de comunicarse [Beth y Geiselmann, 07]. El concepto de criptografía ha sido utilizado en sus inicios para el intercambio de información confidencial entre Estados y sus áreas diplomáticas, pero en la actualidad, las aplicaciones se han ampliado a todo tipo de organizaciones e incluso para personas comunes. Un ejemplo muy claro es lo que sucede en bancos o instituciones financieras donde la confidencialidad de los clientes y sus cuentas es un tema importante, siendo sensible la filtración de los datos a personas no autorizadas.

4.2 Concepto de Criptografía

La palabra criptografía viene del griego y se encuentra compuesto por cripto que significa *oculto* y grafía que significa *escritura*. Según la Real Academia Española, la criptografía es el *Arte de escribir con clave secreta o de un modo enigmático*. Como ciencia, se considera tanto parte de las matemáticas como de la informática [Lucena, 03].

Las matemáticas son las que permiten crear los algoritmos de codificación y de decodificación. Los primeros tipos de algoritmos ocupaban matemáticas básicas haciendo el entendimiento fácil, pero los algoritmos más avanzados ocupan conceptos y métodos matemáticos como la teoría de números que hacen más difícil su entendimiento.

Son dentro de los computadores donde los algoritmos criptográficos se utilizan comúnmente y la criptografía moderna hace uso de teorías como la de información y la de complejidad algorítmica.

En la actualidad, los métodos criptográficos permiten proteger los datos o la información de la manipulación indebida por terceros, por medio de estoque la trasmisión entre un emisor y un receptor se busca la protección ante:

- Del acceso no permitido.
- De cambios en los datos.
- De cambios en el emisor o en el receptor original.
- De la negación de un mensaje enviado o recibido.

El concepto básico que se busca tras la criptografía es que al tener un mensaje en claro, se encripta utilizando una clave, trasmitirlo al receptor y, este para poder leerlo tiene que ocupar la clave correcta.

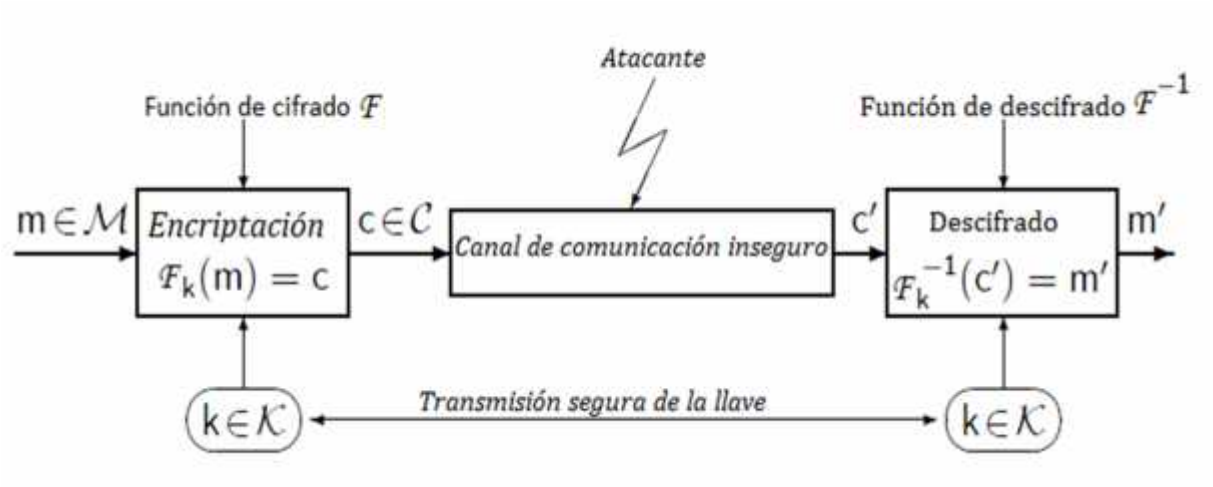


Figura 4.1 Criptosistema convencional

La Figura 4.1 muestra la estructura gráfica de cómo está compuesto un sistema criptográfico convencional [Lucena, 03]. El sistema, denotado por $S = \{M, C, \varepsilon, K\}$, está compuesto de: M que representa un conjunto no vacío de texto en claro, C que representa un conjunto no vacío de texto cifrado y la encriptación, denotada por $\varepsilon = \{F_k : M \to C \mid k \in K\}$, es un conjunto no vacío de un algoritmo de cifrado con una llave $k \in K$. El emisor encripta el texto en claro m ocupando el algoritmo de cifrado Fcon la clave $k(F_k)$, obteniendo un mensaje cifrado c. El texto cifrado c será enviado por el canal de comunicación inseguro al receptor. El receptor autorizado desencripta el texto cifrado c′ usando el algoritmo F_k^{-1}. El receptor tiene la clave k. Para estar seguro de que el mensaje desencriptado es el correcto es necesario comparar si $c = c'$. Con esto se asegura de que el mensaje no fue alterado por un atacante.

Existen tareas principales que tiene que satisfacer un sistema criptográfico y estás se pueden agrupar dentro de cuatro conceptos:

- Confidencialidad: mantenimiento del secreto del mensaje.
- Integridad: el receptor tiene que verificar que el mensaje no fue alterado.
- Autentificación: el origen del mensaje es realmente del emisor.
- Disponibilidad: el mensaje tiene que estar disponible cuando se requiere.

En la Figura 4.1 se hizo referencia a un Criptosistema que sirve para ejemplificar cómo es el funcionamiento a nivel general. Tal como se explicó, se ocupa la misma clave tanto para cifrar como para descifrar el mensaje y de ahí que recibe el nombre de algoritmo simétrico. Una nueva forma que amplía el uso de la criptografía ocupa una clave para cifrar el mensaje y otra clave distinta para descifrar el mensaje. Esto se conoce comúnmente como cifrado de llave pública. Pese a que el fuerte del trabajo se basó en el primer tipo de sistemas, en la sección 4.4 se explica más en detalle cómo funciona el segundo tipo de algoritmos. En definitiva se pueden dividir los sistemas criptográficos en: Simétricos y Asimétricos.

4.3 Criptosistemas Simétricos

El principio básico de estos algoritmos es el uso de una misma clave tanto para cifrar como para descifrar un mensaje. Este es el caso mostrado anteriormente, en donde las dos partes interesadas deben poseer la clave k y es sobre ella donde cabe toda la seguridad del algoritmo. Una de las ventajas más importantes es su rapidez al momento de la ejecución y es por ello que son ampliamente usados en computadores o dispositivos de bajos recursos.

Dentro de la criptografía simétrica se pueden distinguir dos tipos cifrados:

- Cifrado por Bloques
- Cifrado por Flujo

Cada uno de estos cifrados será explicado en detalle, puesto que los algoritmos ocupados para desarrollar las pruebas pertenecen a cada una de las categorías anteriores.

4.3.1 Cifrado por Bloques

En este tipo de cifrado simétrico se divide el texto plano antes de ser procesado en una secuencia de bloques de largo fijo, de tal manera que se aplica la función de encriptación a cada uno de los bloques y se obtiene un texto cifrado del mismo largo que el mensaje ingresado.

De forma matemática, sean $\mathcal{A}$ y $\mathcal{B}$ dos alfabetos infinitos, $m, n \in \mathbb{N}$, un cifrado por bloques es una función:

$$f : A^n \rightarrow B^m$$

Siendo f una función biyectiva. Además, f es independiente del mensaje m del tiempo t. Una suposición que se toma siempre es que $A = B$ y $n = m$. La Figura 4.2 es una representación del cifrado.

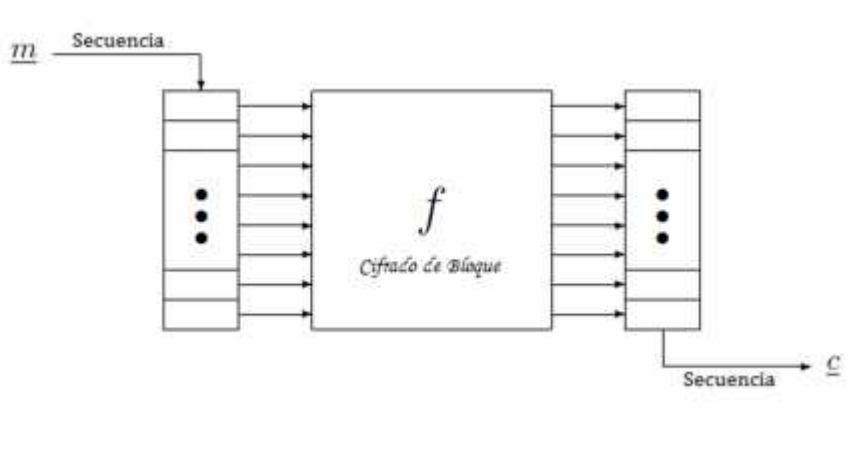


Figura 4.2 Cifrado por bloques

Una de las ventajas del cifrado por bloques es que sirve para la seguridad de discos, puesto que estos se encuentran organizados en bloques y estos bloques pueden ser cifrados con la misma clave e incluso se pueden agrupar varios bloques. Otra ventaja es la utilidad que se puede obtener en el cifrado de transmisiones punto a punto (*end-to-end transmission*).

Sin embargo, algunas desventajas que presenta este cifrado es que no existe algoritmo que sea absolutamente seguro, puesto que los algoritmos conocidos actualmente son teóricamente seguros. La velocidad de procesamiento de los bloques es rápida, y más si se realiza en hardware especializado. No son útiles para procesamiento serial de bits o para secuencias de transmisiones punto a punto, ya que producen un retardo directamente proporcional al largo del mensaje, denotado por n. Esto último se debe básicamente a que n es de largo desconocido en un principio y lo mismo sucede con la cantidad de bloques que se van a utilizar para la encriptación.

Uno de los algoritmos más conocidos en esta área es DES o *Data Encryption Standard* y se comenzó ocupar a finales de los años 70’ por el gobierno de los Estados Unidos. DES utiliza un conjunto de cifrados de bloque y productos, por medio de permutaciones y sustituciones, que corre en una serie de 16 iteraciones (rondas) de la clave. El texto en claro es separado en bloques de 64 bits. La clave se forma igualmente de 64 bits de los cuales 56 son realmente la clave, mientras que los demás 8 bits son de la función de paridad que se forma tras cada conjunto de 7 bits [Fip64, 93]. Actualmente el algoritmo se considera inseguro y se ha publicado una versión mejorada llamada 3DES (Triple DES).

FEAL o Fast *data Encipherment ALgorithm* es otro algoritmo de cifrado por bloques, que fue propuesto como una alternativa a DES. El texto en claro se separa en bloques idénticos que tienen un tamaño de 64 bits y la clave de cifrado tiene un tamaño de 64 bits. La función f opera sobre 32 bits y entrega 32 bits como resultado. El bloque inicial es dividido en 4 bloques de 8 bits. Cada bloque será procesado una tras otro, con una clave de 16 bits a través de XOR y S-Box o tablas de sustitución.

AES o *Advanced Encryption Standard* es un estándar adoptado por el gobierno de los Estados Unidos después de desechar el uso de DES. El algoritmo original es el de *Rijndael* [Fip197, 01], abreviación de los creadores belgas y está planteado para operar con longitudes

de claves y bloques variables de tamaño comprendido entre 128 y 256 bits mientras que AES tiene un bloque de tamaño fijo de 128 bits. La clave para *Rijndael* fue pensaba para trabajar con tamaños múltiplos de 32 bits mientras que en AES se permite tamaños de claves de 128, 192 ó 256 bits. Se realiza un número variable de rondas, dependiendo del largo de la clave, procesando bloques de 4x4 denominado etapas. Cada ronda es una secuencia de *AddRoundKey*, *SubBytes*, *ShiftRows* y *MixColumns*. Varias operaciones las realiza ocupando los bytes, siendo estos partes de cuerpos $\mathbb{F}_{2^8}$, también conocido como cuerpo de *Galois* y, otras las realiza ocupando registros de 32 bits. Es considerado un algoritmo seguro y el único tipo de ataque conocido es el de fuerza bruta que es casi imposible de realizar en la realidad.

TEA o Tiny Encryption Algorithm fue desarrollado por David Wheeler y Roger Needham en la universidad de Cambridge y se publicó en 1994 [Wheeler y Needham, 94]. Es un algoritmo con operaciones de 64 rondas de cifrado *Feistel* sobre mensajes de 64 bits con un clave de 128 bits. Fue diseñado de forma simple y eficiente. Todas sus operaciones son realizadas en palabras de 32 bits y en la función de cifrado ocupa operaciones aritmético/lógicas en vez de permutaciones y sustituciones (DES).

DES, FEAL y TEA comparten entre sí el hecho de que ocupan principios similares bajo los cuales fueron desarrollados que se conoce como red *Feistel*. Es por ello que ataques especiales han sido diseñados para explotar las debilidades que su estructura común permite. En el próximo capítulo se comenzará con el detalle de los algoritmos propuesto para el desarrollo sobre la GPU y de los ataques propiamente tal. También se detallarán mejor ciertas particularidades de los cifrados escogidos.

4.3.2 Cifrado por Flujo

Este tipo de cifrado encripta una cadena de bits o bytes uno a uno. La idea básica es combinar el mensaje con una clave formada por un flujo de bits pseudoaleatorio. Es muy útil cuando no se tiene conocimiento del largo de la data. En la Figura 4.3 se muestra el ejemplo básico de cómo se realiza el cifrado.

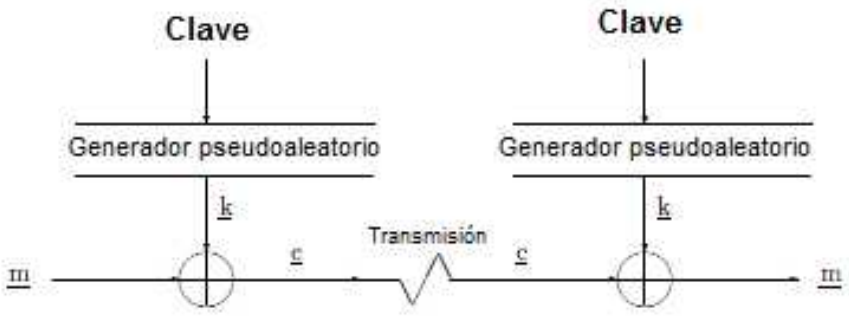


Figura 4.3 Esquema de un cifrado por flujo

Actualmente las implementaciones son realizadas en hardware especializado y en la mayoría de los casos a nivel binario. El campo de estudio de este tipo de cifrado se centra en la generación de secuencias pseudoaleatorias.

Uno de los algoritmos más conocidos en esta área es *RC4*, usado comúnmente en la encriptación de comunicación [Boeing, 08], como en los protocolos WEP, WPA o SSL. La gran ventaja que tiene es su simplicidad de implementación y rapidez. El algoritmo trabaja en tres pasos; Primero, un algoritmo de *key-scheduling* inicia la estructura por medio de una permutación de un array de 256 bits (de 0 a 255), el segundo paso es generar una secuencia pseudoaleatoria, que es combinada con la primera parte del algoritmo. Este algoritmo nunca ha sido liberado, pero en 1994 el código fuente fue posteado un *mailing list*, lo que permitió su rápida difusión gracias al internet. Este algoritmo es considerado seguro, pero su mal uso en WEP lo transforma en un encriptación insegura [Fluhrer *et al.*, 01] que ha sido explotada desde el 2001. Este es otro de los algoritmos elegidos para el desarrollo sobre la GPU y es por eso que en el capítulo próximo se ahondará más en los detalles que lo conforman, se explicarán algunas de sus debilidades y la forma en que se atacará el algoritmo.

4.4 Criptosistemas Asimétricos

También conocido como criptografía de llave pública es un concepto que nace del de la utilización de dos claves, una pública conocida en algunos casos como e y, una secreta, conocida como d. Además resulta prácticamente imposible obtener a partir de la clave pública la clave secreta. Una de las claves sirve para cifrar el mensaje y la otra para descifrarlo. Dentro de esta familia de algoritmos se encuentra RSA y ElGamal [Geiselmann *et al.*, 06].

El primero es uno de los algoritmos más conocidos en este ámbito, llamado RSA por sus creadores de apellido Rivest, Shamir y Adleman. Fue el primero en ser ocupado tanto para encriptar como para firmar digitalmente archivos. Su seguridad nace de la dificultad que existe en la factorización de grandes números. Es por ello que se eligen primero dos números primos p y q, de por lo menos 512 bits cada uno. Luego se calcula $n = p * q$. Se elije la llave pública e, de tal forma que el máximo común divisor entre e y $(p-1)(q-1)$ sea igual a 1. El cálculo de d se hace mediante $e * d \equiv 1 \, mod \, (p-1)(q-1)$. La dificultad nace del hecho que no conocemos los factores p y q. Finalmente para la verificación del cifrado se ocupa $c = m^e \, mod \, n$ y para comprobar el descifrado se ocupa $m = c^d \, mod \, n$.

ElGamal es otro cifrado, cuya dificultad radica en determinar los logaritmos discretos dentro de grupos cíclicos, concepto que se enmarca dentro de la teoría de grupo en la matemática discreta y su detalle será omitido, puesto que escapa del ámbito del proyecto. Los grupos (G) ocupados típicamente son de la forma $\mathbb{F}_{2^m}^* o \mathbb{F}_p^*$ (p es primo), con $E(\mathbb{F})$ una curva elíptica. Además g es generado por G y ambos son abiertamente conocidos. x e y son la clave secreta y clave pública respectivamente. El receptor elije x envia al receptor $y = g^x$. El emisor calcula k, de tal manera que el máximo común divisor entre $\#G$ y $k$sea 1. Luego calcula$a = g^k$y $b = y^k * m$ y envía el par (a, b) al receptor. Este puede con su clave secreta calcular m. Se puede ocupar también para la firma de mensajes.

Estos tipos de algoritmos no forman parte de los elegidos para implementar sobre la GPU, por lo cual no se profundizará más en su estructura, pero sin duda es un buen trabajo a futuro tanto para el desarrollo de los algoritmos como de los ataques conocidos.

4.5 Criptografía con GPU

La criptografía como tal lleva años desarrollándose, en muchos casos de manera teórica, pues la tecnología existente no es siempre suficiente para lo que se tiene, y últimamente ha avanzado mucho. En esta sección se busca comprender cómo se han unido los nuevos conceptos que aporta la GPU con los algoritmos criptográficos, de tal manera de ocupar en el capítulo siguiente algunos conceptos en el desarrollo de nuevas implementaciones.

Dentro de las primeras implementaciones en las que se ocuparon tarjetas de video para el procesamiento de algoritmos se encuentra la de AES usando OpenGL [Cook *et al.*, 05], a pesar de haber logrado el cometido, se dieron cuenta de que con las APIs disponibles en aquel entonces, no era del todo compatible. Esto principalmente a la dificultad que existía al momento de trabajar con operaciones a nivel de bit y de bit a bit, como también lo que sucedía con el soporte en la representación de enteros. Problemas que se debían a que el lenguaje no era compatible con el tipo de aplicación que se necesitaba.

Más adelante, una implementación presentada en [Manavski, 07] muestra un desarrollo eficiente al momento de ocupar la GPU para AES. Básicamente es la combinación de rondas y etapas, que permite ejecuciones rápidas en la tarjeta con palabras de largo de 32 bits. El alto rendimiento que se alcanza se debe a que los datos son guardados en la memoria global o en la memoria compartida. La reducción en la ejecución del algoritmo es la muestra de que una implementación eficiente sobre la GPU entre resultados que difícilmente se pueden obtener sobre una computadora normal.

El algoritmo RC4 descrito anteriormente también ha sido implementado exitosamente en la GPU ocupando CUDA [Boeing, 08]. Sin embargo, no todos los resultados obtenidos tenían una eficiencia mayor sobre la GPU, lo que dependía directamente en cómo se implementaba. Esto indica que no siempre un algoritmo sobre la GPU va a ser más rápido que su equivalente sobre una CPU. La eficiencia en este caso se alcanzó cuando se pudo ocupar de manera óptima la memoria con la que se dispone y, en este caso significa haber guardado los arrays en la memoria compartida de la GPU.

Claramente, el algoritmo que se quiera implementar sobre la GPU será más eficiente mientras mejor localizado se encuentren los datos y las operaciones dentro de la memoria. Es necesario saber cuáles son los datos que más procesamiento requieren y ocupar las ventajas del procesamiento paralelo que se puede obtener con los bloques e hilos.

4.6 Criptoanálisis

En criptoanálisis el fin principal es la obtención de la información cifrada transmitida por medio de un canal de comunicación por parte de un tercero no autorizado. Cuando esto sucede se habla de que se produjo un quiebre en el sistema. En la Figura 4.1 se observa que el atacante puede eventualmente tener acceso a lo que se transmite por medio del canal de comunicación que en primera instancia es inseguro. También puede tener acceso al mensaje al momento de ser cifrado o antes de ser descifrado. En cualquiera de los casos, un atacante tiene

conocimiento de al menos el algoritmo que se ocupa para cifrar la información. Más adelante se detallarán los principios que se ocupan en los ataques criptoanalíticos.

De una forma más general, el criptoanálisis se refiere a la comprensión de cómo funciona un sistema criptográfico. Debido a la existencia de una gran variedad de criptosistemas existen también diferentes técnicas que son adecuadas para comprender cada uno de los algoritmos. Cada una de estas técnicas son útiles al momento de implementar algoritmos criptográficos, puesto que ayudan a entender en qué momento se puede estar expuesto a alguna debilidad que provoque un quiebre.

4.6.1 Tipos de Ataques

El objetivo de un ataque es siempre el quiebre del criptosistema y para ello existen una serie de principios que se encuentran detrás análisis. La clasificación de las distintas técnicas que ocupará eventualmente un atacante se puede dividir en [Beth y Geiselmann, 07]:

- Conocimiento del texto cifrado (*ciphertext-only attack*): a disposición del atacante solo se encuentra el texto cifrado. Por lo que el atacante solo tuvo acceso al canal inseguro para obtener dicha información. Es uno de los ataques más débiles.

- Conocimiento del texto claro (*known-plaintext attack*): a disposición del atacante se encuentra el par *texto en claro, texto cifrado*, por lo cual es posible determinar la clave. Este tipo de ataque es siempre posible cuando el texto es cifrado contiene una frase de inicio y una frase final. Por ejemplo, "*Estimado señor…*", *"… saludos cordiales, Daniel*".

- Elección del texto claro (*chosen-plaintext attack*): en este caso existe a disposición del atacante diferentes pares de *textos en claro, textos cifrado*. Él puede elegir el texto claro que será encriptado. Esto es equivalente a que el atacante tenga acceso a la unidad de cifrado (en la Figura 4.1es la caja de encriptación).

- Elección del texto cifrado (*chosen-ciphertext attack*): este ataque es ocupado en criptosistema de llave pública (algoritmos asimétricos). Existen a disposición del atacante diferentes textos cifrados y el ataque se basa en encontrar la clave que descifre el texto.

Además de las diferentes técnicas, existen métodos que se han desarrollado para cada criptosistema en especial. Estos se pueden clasificar en:

- Búsqueda completa: también conocido como ataques de fuera bruta. Lo que intenta es probar todas las posibles combinaciones de claves, es decir, todo el conjunto $\mathcal{K}$. Es una técnica que es muchos casos requiere una gran cantidad de tiempo para el procesamiento, puesto que se tiene que elegir una clave, probarla y verificar el resultado. La obtención de una mejor eficiencia tiene relación directa con la capacidad computacional que tenga la máquina que la ejecuta.

- Prueba y error: probar distintas posibilidades conocidas y comprobar si no producen errores. Se puede unir a un ataque de fuerza bruta y reduce los campos en los cuales se buscan claves.

- Estadísticos: son ocupados análisis de frecuencia como por ejemplo la ocurrencia de ciertas letras o palabras de los textos en claro y cifrados. Ocupado ampliamente en los principios de la criptografía y del criptoanálisis.

- Análisis estructural del sistema: por ejemplo la resolución de sistemas de ecuaciones que sustentan un algoritmo ayuda a encontrar posibles debilidades.

4.6.2 Ataques con GPU

Muchos de los ataques que se han desarrollado de forma teórica han tenido la limitante que la capacidad computacional con que se dispone no es suficiente. Esto sucede principalmente porque se necesita del procesamiento de una gran cantidad de datos y el costo computacional asociado hasta hace poco hacía que las implementaciones no fueran realizables. Sin embargo, con las posibilidades que presentan las GPU nuevos desarrollos han comenzado a surgir.

En RSA el mayor problema es tratar de factorizar n. La dificultad es que los factores son números primos formados por al menos 512 bits (actualmente se recomiendo que sean de mínimo 1024 bits) y su solución se puede lograr por métodos de curvas elípticas (*elliptic-curve method* o ECM). El método es capaz de encontrar factores que se encuentren entre 10^{10} y 10^{60}, pero el consumo de recursos computacionales ha sido siempre la limitante. En [Bernstein *et al.*, 09] se sugiere uno de los métodos más actuales para implementar ECM sobre GPU y además se compara el desarrollo bajo distintas tarjetas de video.

MD5, una función hash, que en sus principios fue utilizada para encriptar claves de Linux, pero desde que se encontraron colisiones [Wang y Yu, 05] su uso ha ido disminuyendo y actualmente es utilizado para chequear versiones en programas. Se pueden encontrar implementaciones sobre la GPU en internet y también para explotar las colisiones que se producen.

5 Uso de la tecnología GPU en la Criptografía

En este capítulo se intenta agrupar mucho de los conceptos que se explicaron en capítulos anteriores. El fin es que se puedan ocupar para el desarrollo de algoritmos criptográficos que funcionen sobre la GPU, pero específicamente para la implementación de ataques sobre los algoritmos elegidos. Es por eso que primero se explicarán con mayor detalle los algoritmos elegidos, TEA y RC4, las modificaciones que se realizaron, el ataque que se implementará para cada uno y el desarrollo que se hizo sobre la GPU.

5.1 TEA o Tiny Encryption Algorithm

En el capítulo anterior se explicó a grandes rasgos en qué consistía el algoritmo. Sin embargo, para un completo entendimiento de este es necesario clarificar ciertos conceptos importantes que en muchos casos son transversales a otros algoritmos similares. Como se verá más adelante se trata de un algoritmo simple con debilidades en el manejo de las claves, pero que ha sido actualizado con una versión más segura. Sin embargo, se utilizará una derivación del algoritmo que ha sido ocupado para desarrollar ataques que es una versión mucho más simple que no es la que se utiliza generalmente.

5.1.1 Cifrado Feistel

Uno de los primeros conceptos que es necesario aclarar tiene relación con el cifrado Feistel [Feistel, 73]. En el cifrado Feistel, el texto en claro es representado por un bloque llamado P, que en TEA es de 64 bits, que es dividido en una mitad izquierda L_0 y en otra derecha R_0. Se pasa por una serie de rondas para formar el cifrado final y, en este caso, cada ronda se encuentra representada por $i = 1, 2, \ldots n$, donde $n = 64$ bits. Finalmente, el nuevo valor de L_i y de R_i quedan representados por:

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i) \qquad (5.1)$$

$$R_i = L_{i-1} \oplus F(R_{i-1}, K_i) \qquad (5.1)$$

La subclave se obtiene de la clave K y el algoritmo llamado *key schedule* es el encargado de dividir la clave en las subclaves correspondientes. Donde K_i representa la subclave para cada ronda y F es la función de encriptación correspondiente. Un detalle importante que se puede visualizar es que en cada ronda solo se cifra la mitad del bloque de mensaje, esto es de suma importancia porque significa que al menos se tendrán que ejecutar tres rondas para que el mensaje en claro no pueda ser visualizado.

La Figura 5.1 muestra esquemáticamente cómo se encuentra formada una ronda de Feistel. En ella se ve que al lado derecho se le aplica junto con la subclave la función que

luego con un XOR con el lado izquierdo forma el lado derecho de la nueva ronda. El lado izquierdo de la nueva ronda lo forma el derecho de la ronda anterior.

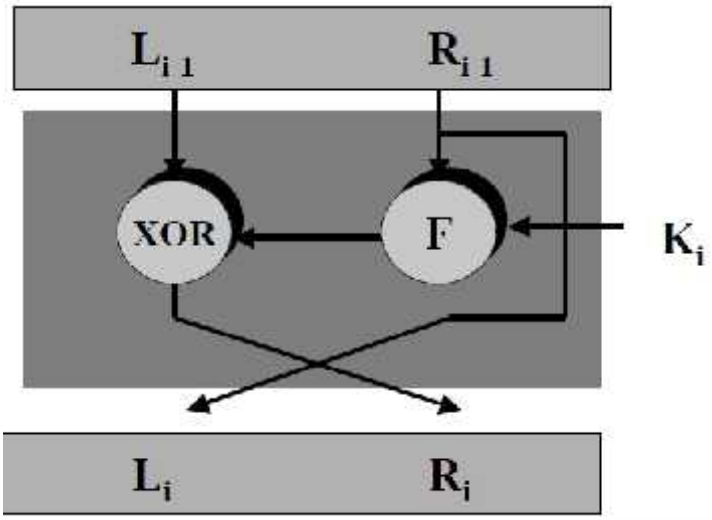


Figura 5.1 Ronda Feistel

El cifrado en la ronda final queda representado por $C = (L_n, R_n)$. Para desencriptar es necesario tomar $C = (L_n, R_n)$y aplicar las rondas en sentido inverso, es decir, i = n, n − 1, … , 1, para obtener $P = (L_0, R_0)$.

En la Figura 5.2 se visualiza completamente la red Feistel. Este tipo de cifrado es ocupado ampliamente en los algoritmos de cifrado por bloque como DES, FEAL, Blowfish, XTEA (versión mejorada de TEA), etc. La gran diferencia que hay entre cada uno de los algoritmos es en la función de cifrado F, siendo además ahí donde radica la seguridad.

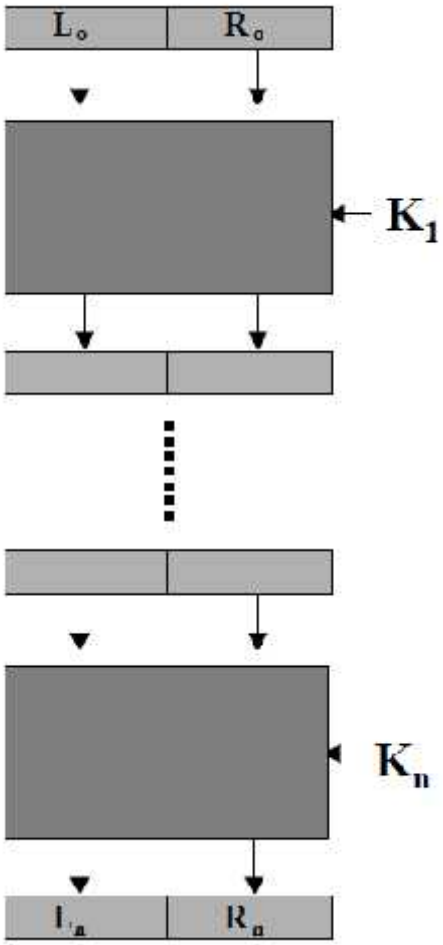


Figura 5.2 Red Feistel

5.1.2 Confusión y Difusión

Otros dos conceptos importantes, que en realidad se ocupan para todo los tipo de cifrado simétricos, son los llamados Confusión y Difusión [Shannon, 49]. Esto conceptos son los que se correspondían en los antiguos algoritmos a la sustitución y a la permutación, respectivamente.

Por un lado, la confusión tiene que ver con la relación que existe entre *texto en claro / texto cifrado* y cómo hacer que esta relación sea menos obvia. En los actuales cifrados, un bloque de texto en claro es sustituido por varios bloques de texto cifrado y, normalmente, la sustitución se basa en una parte del texto claro y en la llave del momento (la subclave).

Por otra parte, la difusión se refiere a la relación estadística entre los bits de salida y los bits de entrada y, cómo está relación tiene que ser lo más compleja posible para no poder deducir una clave. En otras palabras, los bits de salida (texto cifrado) tienen que estar influenciado por los bits de entrada previos y no tiene que haber una relación uniforme entre la salida y la entrada.

Ambos conceptos son ampliamente ocupados en este algoritmo y más adelante cómo uno de ellos afecta el resultado de un cifrado en especial.

5.1.3 Cifrado por Bloque TEA

TEA fue diseñado para ser implementado en software y es por eso que tiene una estructura simple. Por ejemplo, no requiere guardar en memoria grandes tablas (P-boxes o S-boxes) como AES o DES. El código fuente escrito en C para la encriptación en TEA se ve de la siguiente forma:

```
Void encryp_tea(ulong* V, ulong* K) {
      ulong V0=V[0], V1=V[1];
      uint i;
      ulong sum=0, delta=0x9e3779b9; // Constante para el manejo de
      claves
      ulong K0=K[0], K1=K[1], K2=K[2], K3=K[3];  //key separada en 4
      bloques de "32 bits"
      for (i=0; i < 32; i++) { // 64 Rondas
            sum += delta;
            V0 += ((V1<<4) + K0) ^ ((V1>>5) + K1)^ (V1 + sum); //XORs
            y Shifts
            V1 += ((V0<<4) + K2) ^ ((V0>>5) + K3)^ (V0 + sum);
            }
      v[0]=v0; v[1]=v1;
}
```

El texto en claro almacenado en V, es pasado a V_o, V_1 y se define una constante llamada delta (su valor no es relevante). La clave K es separada en cuatro bloques (subclaves) de 32 bits K_o, K_1, K_2, K_3, esto es llamado *Key Schedule Algorithm* y, siendo cada subclave diferente y de la clave inicial. Luego la rutina de encriptación sigue el mismo camino que se muestra en la Figura 5.2.

La función tiene una estructura que se ocupa tiene una estructura interna en donde las operaciones para la adición de enteros se realiza mediante XOR y mediante operaciones de desplazamiento a la izquierda y derecha. Esta estructura es similar para cada ronda con la diferencia que las subclaves K_o, K_1 son usadas en la ronda superior y K_2, K_3 en la ronda inferior, como se ve en el algoritmo.Esto se resume en la siguiente ecuación:

$$F(M,(T,U,V)) = \left((M \lhd 4) \boxplus T\right) \oplus \left((M \rhd 5) \boxplus U\right) \oplus (M \boxplus V) \qquad (5.2)$$

Donde la subclave está representada por la tupla (T, U, V).Haciendo una equivalencia con el código fuente presentado anteriormente se puede decir que: T es K_o o K_2, U es K_1, K_3 y V es sum.

La rutina de descifrado funciona de manera similar, pero lógicamente la entrada en este caso es el texto cifrado y las subclaves son utilizadas en orden inverso.

```
Void decryp_tea(ulong* V, ulong* K) {

      ulongV0=V[0], V1=V[1];
      uint i;
      ulong sum=0xC6EF3720, delta=0x9e3779b9;  // constante
      ulong K0=K[0], K1=K[1], K2=K[2], K3=K[3];// manejo de subclaves
      for (i=0; i<32; i++) {
      V1 -= ((V0<<4) + K2) ^ ((V0>>5) + K3) ^ (V0 + sum);
            V0 -= ((V1<<4) + K0) ^ ((V1>>5) + K1) ^ (V1 + sum);
            sum -= delta;
            }
            v[0]=v0; v[1]=v1;
}
```

Un problema que nace en esté algoritmo es que lo que se espera es tener 2^{128} clases de cifrado distintos, o lo que equivale a poder manejar 2^{128} diferentes claves, lo que sin embargo no es capaz de soportar [Mirza, 98]. Esta debilidad se debe a la función de la ronda de TEA y es muy fácil de probar cuyos resultados se pueden observar en la Tabla 5.1. De manera práctica, esto significa que es lo mismo ocupar TEA usando claves de 2^{128} que usando claves de 2^{126}. Lo que permite, para un ataque de búsqueda exhaustiva, probar sólo un cuarto de todas las claves posibles, pues las demás generan un mismo cifrado.

Tabla 5.1 Claves equivalentes en TEA

Clave	Texto en Claro	Cifrado
00000000 80000000 00000000 00000000	00000000 00000000	9327C497 31B08BBE
80000000 00000000 00000000 00000000	00000000 00000000	9327C497 31B08BBE
80000000 00000000 80000000 80000000	00000000 00000000	9327C497 31B08BBE
00000000 80000000 80000000 80000000	00000000 00000000	9327C497 31B08BBE

Una mejora para la debilidad anterior fue realizada por los diseñadores y nombrada Extended TEA o XTEA. La implementación de XTEA en tan simple como la de TEA, sin embargo, los cambios introducidos hacen que no se produzcan los problemas mostrados anteriormente. En la Tabla 5.2 se muestran los diferentes resultados obtenidos al utilizar TEA y XTEA.

Tabla 5.2 Comparación TEA y XTEA para claves equivalentes

Clave	Texto en Claro	Cifrado con TEA	Cifrado con XTEA
00000000 80000000 00000000 00000000	00000000 00000000	9327C497 31B08BBE	4F190CCF C8DEABFC
80000000 00000000 00000000 00000000	00000000 00000000	9327C497 31B08BBE	57E8C05 50151937
80000000 00000000 80000000 80000000	00000000 00000000	9327C497 31B08BBE	31C4E2C6 347B2DE

00000000 80000000 80000000 80000000	00000000 00000000	9327C497 31B08BBE	ED69B785 66781EF3

5.1.4 Cifrado STEA

El cifrado STEA es una derivación de TEA que tiene como finalidad ser un algoritmo de pruebas para las posibles debilidades que puedan existir en TEA, XTEA y otros de la familia. En [Mirza, 98] se detalla el algoritmo y, además, se dan a conocer una serie de ataques que se pueden realizar sobre él y su posibilidad de realizarlos sobre las versiones superiores. Para esta investigación se ocupó uno de los ataques descritos.

STEA está formado por un cifrado Feistel de 64 Rondas que opera sobre mensajes de 64 bits y, a diferencia de TEA, ocupa una clave de solo 64 bits. En definitiva la función para cada ronda queda de la siguiente forma:

$$F(M, K_i) = M \square K_i \qquad (5.3)$$

Donde K_i es una subclave de 32 bits, es decir que la clave se divide en dos bloques de 32 bits $K = (K_0, K_1)$. Por lo que, las subclave para la ronda superior sería donde $K_i = K_0$ y en el caso de la ronda inferior sería $K_i = K_1$ (para $1 \leq i \leq 64$).

Matemáticamente STEA es mucho más simple que TEA y esto también queda reflejado en el código fuente que se presenta a continuación:

```
void stea_enc(ulong* m, ulong* k) {
      ulong m0=m[0], m1=m[1];
      uint i;
      ulong  k0=k[0],  k1=k[1];   //key separada en 2 bloques de 32
      bits
      for (i = 0; i < RONDAS; i++) { // N° Rondas
            m0 += m1 ^ k0;
            m1 += + m0 ^ k1;
      }
      m[0]=m0; m[1]=m1;
}
```

Sin embargo de esta implementación se puede identificar un problema que surge casi de manera inmediata y es que para diferentes textos planos con una misma clave se puede alcanzar el mismo cifrado (similar en casi en su totalidad), tal cual se muestra en la Tabla 5.3. Esto se debe a un problema la difusión que entrega este algoritmo y es el problema que más adelante explotará el ataque. En TEA no se producirían los mismos resultados, puesto que las rotaciones que se producen en su función hacen que la difusión sea mayor.

Tabla 5.3 Cifrado con STEA

Clave	Texto en Claro	Cifrado
00000000 00000000	00000000 00000000	00000000 00000000
00000000 00000000	00000000 00000001	61CA20BB 297A859D
00000000 00000000	00000001 00000001	297A859D 8B44A658
00000000 00000000	00000001 00000000	C7B064E2 61CA20BB
00000001 00000003	00000000 00000000	57112EA4 255E6231

00000001 00000003	00000000 00000001	F12AEA7D ED0EC712
00000001 00000003	00000001 00000001	**C7B064E3 61CA20BB**
00000001 00000003	00000001 00000000	**C7B064E2 61CA20BA**
55555555 55555555	22222222 22222222	0F3102B4 83B32497

5.1.5 Ataque Dividir y Conquistar sobre STEA

Este es un método efectivo de texto en claro conocido (*Known-Plaintext attack*), mediante es posible recuperar la clave con solo 32 encriptaciones [Mirza, 98]. El ataque se basa en el principio de dividir y conquistar (*divide-and-conquer*) donde la estructura de cifrado es separada en partes pequeñas y estas son atacadas independientemente.

Para realizar el ataque es necesario definir el concepto del bit menos significativo, *LSB* (por su sigla en inglés). Donde i es el i-bit de A también denotado por $A[i]$, el LSB de A es $A[0]$. Con esto, la función para una ronda de STEA para el bit menos significativo quedaría como:

$$L_i[0] = R_{i-1}[0]$$

$$R_i[0] = L_{i-1}[0] \oplus R_{i-1}[0] \square K_i[0] \qquad (5.4)$$

De esa manera será posible recuperar los 2 bits de la clave (los menos significativos de K_0 y K_1) usando solamente un texto plano conocido y encontrando las expresiones lineales para los bits menos significativos del texto plano, texto cifrado y la clave. Si denotamos el texto en claro como $P = (L_0, R_0)$ y el texto cifrado como $C = (L_{64}, R_{64})$, se obtiene en la ronda 64 la siguiente ecuación:

$$K_0[0] = L_{64}[0] \oplus R_{64}[0] \square L_0[0],$$

$$K_1[0] = L_0[0] \oplus R_0[0] \square R_{64}[0] \qquad (5.5)$$

En estas ecuaciones los términos de la parte derecha son conocidos y, por lo tanto, se hace posible la recuperación de $K_0[0]$ y $K_1[0]$.

Extendiendo el ataque y comprendiendo que cada bit del mensaje es independiente de los otros excepto por el *carry* (acarreo)que produce una interferencia, es posible generalizar las ecuaciones anteriores para los demás bits de la clave. Estas quedarían de la siguiente forma:

$$K_1[i] = L_{64}[i] \oplus R_{64}[i] \oplus L_0[i] \oplus C_1[i],$$

$$K_2[i] = L_0[i] \oplus R_0[i] \oplus R_{64}[i] \oplus C_2[i] \qquad (5.6)$$

Donde i representa la posición a recuperar y C_1 y C_2 son el *carry* para cada subclave respectivamente.

Suponiendo que se recuperó el primer bit de forma correcta y queremos recuperar el siguiente bit, el bit 1, ese se encuentra afectado por el *carry* del bit 0. Si es que el *carry* llegase a ser eliminado para la recuperación del bit 1, se volvería a la ecuación 5.5, lo cual permitiría recuperar dos bits más de la clave. Así se podría seguir hasta obtener la clave completa. La

forma más fácil de eliminar el *carry* es calcular su valor para el bit apropiado y luego aplicar XOR con el bit del texto cifrado. Para calcular el *carry* que tendrá el bit i-esimo es necesario encriptar el $(i-1)$ bit menos significativo del texto en claro, usando el bit conocido $(i-1)$ de la clave.

Ahora es posible recuperar los 64 bits de la clave usando solo un texto en claro conocido y 32 encriptaciones. En general, para un mensaje de largo m, el ataque recuperará el bit *m* de la llave con solo $m / 2$ encriptaciones, independiente de la cantidad de rondas.

Debido al problema de difusión que se mostró en la Tabla 5.3 hace que este ataque sea posible solo sobre STEA, ya que en TEA las rotaciones en la función de cifrado proveen una mejor difusión. En cada ejecución de sus rondas, cada bit del bloque del mensaje es cambiado por otro en el cifrado dependiendo del bloque anterior y de los bits de la clave. En otras palabras, cada bit del cifrado es una compleja función no-lineal de un número de textos en claro y bits de la clave, lo que hace que la ecuación no sea simple de resolver.

En la literatura se encuentran otros tipos de ataques que es posible realizar sobre STEA, sin embargo, para esta investigación se eligió el descrito anteriormente básicamente por la simplicidad. Los resultados del capítulo siguiente muestran, a partir de las implementaciones realizadas, que existen casos en que la CPU trabaja más rápido que la GPU.

5.2 RC4

Fue diseñado en 1987 por Ron Rivest en los laboratorios de RSA Security. Es posiblemente uno de los más populares cifrados por flujo que existen y su uso, gracias a su fácil implementación en software, se extiende a una gran gama de aplicaciones y protocolos como: Microsoft, Apple, SSL (*Secure Sockets Layer*), WEP, entre otros. Hasta 1994, el algoritmo era propietario, pero se filtró en internet una versión que producía salidas idénticas y se hizo público. A pesar de eso siguió siendo utilizado y se considera un algoritmo seguro. El problema, y que se presentará más adelante, nace cuando se efectúan malas implementaciones del algoritmo.

5.2.1 Descripción de RC4

RC4 se forma de dos partes. La primera llamada algoritmo de manejo de llave o *Key-Scheduling Algorithm* (KSA) que genera una permutación inicial desde una llave *random* de largo n, donde normalmente el largo se encuentra entre 40 y 256 bytes. Se inicializa una tabla de estados, denotada por Scon valores de 0 a 255. Este arreglo también es conocido como tabla de estados internos.

La implementación de KSA para C se puede ver a continuación:

```
void rc4_ksa() {
for (i = 0; i < 256; i++)
      S[i] = i;
j = 0;
for (i = 0; i < 256; i++) {
      j = (j + clave[i % clave_largo] + S[i]) & 255;
```

```
        swap(S, i, j); // Intercambia el valor para S
        }
i = j = 0;
}
```

La segunda parte, que es la principal del algoritmo, está formada por el llamado algoritmo de generación pseudo-aleatorio o *Pseudo-Random Generation Algorithm* (PRGA) que produce cómo salida un byte en cada paso. Este byte es combinado mediante la función XOR con el texto en claro, formando el texto cifrado.

Un poco más en detalle, en PRGA se inicializan dos índices, i y j, en 0, luego se incrementa i como un contador, j de forma pseudo-aleatorio, se intercambian los valores (swap) de S entre i y j. La salida es el valor de S de $S[i] + S[j]$.

En el algoritmo siguiente, se puede ser una implementación de PRGA:

```
uchar rc4_prga() {
      i = (i + 1) & 255;
      j = (j + S[i]) & 255;// Incremento pseudo-aleatorio
      swap(S, i, j);
      return S[(S[i] + S[j]) & 255];
}
```

5.2.2 Debilidad del IV en el KSA en WEP

En el 2001 se publicó un ataque sobre el algoritmo RC4 ocupado en el protocolo WEP (*Wired Equivalent Privacy*), o también conocido como ataque FMS [Fluhrer *et al.*, 01]. WEP nace de un estándar conocido como 802.11, que se ha extendido por años para el uso en redes inalámbricas. La mayoría de las tarjetas que adoptaron este estándar incluían el protocolo de seguridad WEP, simple de administrar, usando una clave que se distribuía por todos los dispositivos.

El problema radica en que WEP crea un vector de inicialización (IV), de 24 bits, que no es secreto y que se antepone a la clave secreta, formando una clave de sección. Un esquema general de cómo se combina la clave secreta con el IV para formar una clave se sección y encriptar un mensaje se muestra en la Figura 5.3. Una manera similar se ocupa para desencriptar el mensaje.

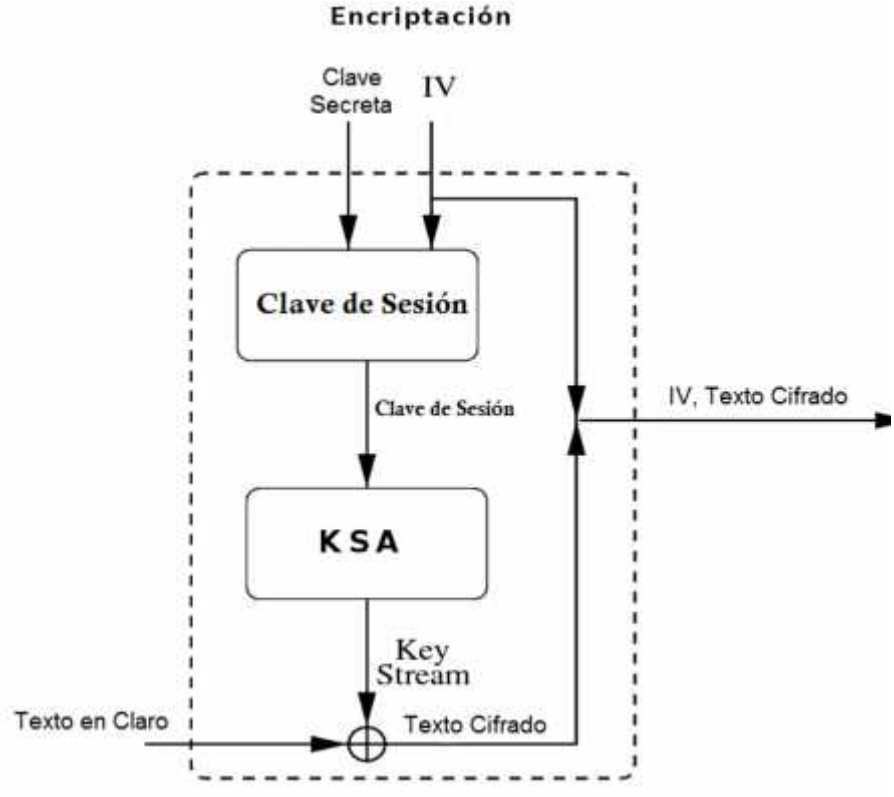


Figura 5.3 Manejo de clave en un cifrado por flujo

En WEP además de crear una clave de sesión mediante la concatenación del IV y la clave secreta, se pueden utilizar otras dos formas que también permiten explotar la debilidad anterior. Una forma sería concatenar la clave secreta con los IV y la otra sería hacer un XOR entre los IVs y la clave secreta. Sin importar el método que se ocupe para la clave de sección, el ataque que se implementa se preocupa de observar la primera palabra de *keystream* para un número reducido de claves se sección.

En las pruebas realizadas se implementaron los algoritmos siguiendo como base lo que sucede en WEP, es decir, se conocen los textos cifrados, se conocen los IV y se obtendrá en base a eso la clave secreta. En las secciones siguientes se buscan entender los ataques que se pueden llevar a cabo para este algoritmo.

5.2.3 Ataque FMS

FMS fue uno de los primeros ataques publicados y recibía su nombre pos los autores Fluhrer, Mantin y Shamir [Fluhrer *et al.*, 01]. En este método el atacante junta los paquetes encriptados y junto con ellos los IV correspondientes. Debido a que los primero bytes del texto plano son fáciles de predecir, el atacante puede recuperar los primeros bytes de la salida de KSA (*keystream*) usada para encriptar. Los vectores de inicialización son transmitidos sin protección con los paquetes, por lo que los primero 3 bytes de la clave son conocidos. Sin embargo, los demás bytes son desconocidos para el atacante.

Para continuar con este ataque es necesario asumir ciertas condiciones. Primero, que el atacante tiene conocimientos de los primeros $l$bytes de la clave secreta que se ocupará para generar el keystream X. Por lo cual puede simular los primeros l pasos de la función KSA y conocer K_l y j_l. Para el siguiente paso, $j_{l+1} = j_l + K[l] + S_l[l]$y $S_l[l]$ es intercambiado por $S_l[j_{l+1}]$. Si el atacante puede revelar el valor de $S_{l+1}[l]$, el cálculo de $K[l]$ significaría calcular la diferencia entre $S_l^{-1}[S_{l+1}[l]] - j_l - S_l[l]$. Para realizar esto se definen las siguientes reglas que se realizan antes de l pasos en la función KSA:

1. $S_l[1] < l$
2. $S_l[1] + S_l[S_l[1]] = l$
3. $S_l^{-1}[X[0]] \neq 1$
4. $S_l^{-1}[X[0]] \neq S_l[1]$

A continuación el valor $k = S_l[j_{l+1}]$ será intercambiado por $S_{l+1}[l]$. Como $j$cambia aleatoriamente durante KSA, los valores $S[1]$, $S[S[1]]$y $S[l]$ no se verán alterados con una probabilidad de $\left(\frac{1}{e}\right)^3 \approx 5\%$.

La primera salida producida en PRGA, j toma el valor de $S_n[1]$ y $S_n[1]$ y $S_n[j]$ son intercambiados. El resultado después del intercambio sería que $S[1] + S[S[1]] = l$ se mantiene y que el primer byte de salida de PRGA, $X[0]$, es $S[l]$. Se desprende entonces que manteniendo las 4 condiciones anteriores, la aplicación de los pasos anteriores entregará el valor de $K[l]$ con una probabilidad de $\left(\frac{1}{e}\right)^3 \approx 5\%$.

Sin embargo, un ataque publicado en un foro bajo el nombre de Korek [Korek, 04] es una aproximación mucho más poderosa a FMS y fue este método el que se utilizó como ataque para la implementación en la GPU.

5.2.4 Ataque Korek

Korek es una técnica estadística que ocupa una serie de correlaciones para obtener un byte de la clave secreta de WEP. En el ataque publicado inicialmente había 5 correlaciones a las que con el tiempo se le agregaron otras y llegando a ser un total de 16. Cada correlación se realiza entre el primer l byte de la clave RC4 (clave de sesión), los dos primeros bytes de la clave generada (*keystream*) y el siguiente byte $K[l]$. Korek asignó nombre a cada una de las correlaciones como por ejemplo A_s13, A_u15 o A_s5_1, este último hace referencia a FMS.

La mayoría de las correlaciones encontradas por Korek asumen que el primero y segundo byte del *keystream* revela el valor de j_{l+1} bajo ciertas condiciones, si 2-4 valores en S tienen una configuración especial y no cambian durante las siguientes fases de KSA después de $l + 1$ pasos. Un caso especial se presenta para la correlación A_neg, que ayuda a reducir el espacio de búsqueda de la clave indicando que valores son los menos probables.

En [Chaabouni, 06] se explican detalladamente las 16 correlaciones existentes. La presente investigación no es un detalle de cada una de las ataques, sino más bien se trata del

uso de la GPU junto a los ataques, solo se han elegido alguno de los ataques disponibles dejando abierta la puerta a seguir con el desarrollo en el futuro.

Los métodos que se ocupan tienen la particularidad que no están basados en identificar el IV, sino en el comportamiento que tiene el KSA y PRGA. Es por eso que se asumen ciertas precondiciones para los ataques. Asumiendo que los tres primeros bytes corresponden al IV y que el primer byte de la clave comienza en la posición 0. Si se quiere atacar el byte n de la clave, el primer byte de la clave es desconocido para el atacante, por lo que el KSA tiene que llegar hasta ese byte y así se podrá tener una aproximación del estado final que tiene S para los elementos que se encuentran al inicio. Mientras se realiza este proceso es necesario ir guardando los valores que toma j y con ello construir una tabla con la información que contenga la posición de cada elemento con su valor aproximando en la tabla de estados. El último valor será S_i.

La aproximación realizada es el foco principal que atacan las correlaciones, puesto que exploran la probabilidad de ocurrencia de que un elemento de la tabla de estados permanezca inalterable hasta el final del KSA.

Los métodos que se detallan a continuación exploran cómo es posible que suceda cierto estado. Los ataques elegidos son:

1. A_s5_1: tiene una probabilidad de éxito de aproximadamente 5.07% luego de l pasos en KSA. Este es el ataque FMS descrito anteriormente, por lo que cumple las mismas condiciones:
   a. $S[1] < l$: se ocupa para aumentar las posibilidades de que la primera parte de los estados internos permanezca invariable.
   b. $(S[1] + S[S[1]])\ mod\ 256 = l$: encuentra el caso en que el primer byte de salida del keystream (llamado $o1$) dependerá de los valores conocidos y del valor de nuestro byte objetivo.
   c. $Si[o1] \neq 1$ y $Si[o1] \neq S[S[1]]$: así el valor de $S[1]$ o $S[S[1]]$ no es sobre escrito.

   Finalmente, la clave atacada puede ser obtenida calculando
   $$K[l] = Si[o1] - S[l] - j_{l-1}.$$

2. A_s13: tiene una probabilidad de éxito de aproximadamente 13,75%. En este ataque se ocupa un truco en la función PRGA, puesto que si se termina antes el KSA con un estado $S[1] = l$ y $S[l] = 0$ el PRGA intercambiara esos valores a una salida donde $o1 = l$. Para ellos es necesario que se cumplan:
   a. $S[1] = l$: hace posible que el PRGA encuentre el elemento l en el estado interno.
   b. $o1 = l$: suponiendo que el primer byte de salida será igual a l, se asume que después del intercambio en el PRGA el valor de l estará en la posición l, forzando la suposición de que $j_l$es igual al valor del índice 0.

   Finalmente, la clave atacada puede ser obtenida calculando
   $$K[l] = Si[0] - S[l] - j_{l-1}.$$

3. A_u15: tiene una probabilidad de éxito de aproximadamente 13,75%.Se ocupa el conocimiento del segundo byte de salida, $o2$, y del comportamiento que tenga el PRGA. Una consideración importante es que antes del cálculo para cada salida hay un intercambio donde $j = j + S[i]$, el primer j es igual a $S[1]$ y el segundo será igual al anterior más $S[2]$. En definitiva, si en el PRGA $S[2] = 0$, $o2$ será igual a 0. Para explotar esta situación es necesario mantener un valor invariable durante los pasos desconocidos del KSA. Para eso, este ataque supone que $j_3 = 2$ y usa las siguientes condiciones:
    a. $o2 = 0$: es el caso se quiere.
    b. $S[l] = 0$: de esta manera se puede cambiar a 0 $S[2]$, asumiendo que $j_l = 2$ en el KSA.
    c. $S[2] = 0$: no es una condición muy útil, puesto que el 0 se coloca en $S[l]$ y l es mayor o igual que 3. Sin embargo, es utilizada como optimización de la ejecución.

    Finalmente, la clave atacada puede ser obtenida calculando
    $$K[l] = 2 - S[l] - j_{l-1}.$$

4. A_s5_2: la probabilidad de éxito es aproximadamente de 5,07%. Es otra forma de explotar el conocimiento del segundo byte de salida, $o2$. Para realizarlo es necesario determinar el valor que toma j en el segundo paso del PRGA. Las condiciones que se necesitan son:
    a. $S[1] > l$: seleccionar los IV que se colocan dentro de $S[1]$con un valor mayor al, para que el PRGA no varíe los valores con índices menores o iguales a l durante las primeras fases.
    b. $(S[2] + S[1])\ mod\ 256 = l$: el valor j en el segundo paso del PRGA toma el valor de l, de manera de incluir lo que fue puesto en $S[l]$ en la ejecución de KSA.
    c. $o2 = S[1]$: se comprueba el caso donde la salida $S[1]$ es $o2$. Esta condición permite identificar la suposición hecha para el valor de j_l en KSA: $j_l = Si[(S[1] - S[2])\ mod\ 256]$.
    d. $Si[(S[1] - S[2])\ mod\ 256] \neq 1\ y\ Si[(S[1] - S[2])\ mod\ 256] \neq 2$: para j_l no tome ni el valor 1 ni 2, de tal manera de mantener el contenido de $S[1]$.
    La idea general es la colocación de un valor específico en un lugar determinado ($o2 = S[1]$, donde $S[1]$ se coloca al inicio del KSA). Para que esto suceda $S[1], S[2]\ y\ S[l]$ deben mantenerse invariables antes del paso l del KSA para dejar que el PRGA se haga cargo.
    Finalmente, la clave atacada puede ser obtenida calculando
    $$K[l] = Si[(S[1] - S[2])\ mod\ 256] - S[l] - j_{l-1}.$$

5. A_s5_3: la probabilidad de éxito es aproximadamente de 5,07%. Las condiciones en este ataque son para que el valor de $S[j_l]$ sea la salida en $o2$. Para realizar esto se ocupan las siguientes condiciones:

a. $S[1] > l$: igual a la condición de A_s5_2, no se quiere que el PRGA cambie los valores inferiores a l, por lo que se filtran los casos en donde el KSA coloca un valor mayor a l en $S[1]$.
b. $(S[2] + S[1])\ mod\ 256 = l$: igual a la condición del ataque anterior, se quiere que j en el segundo paso del PRGA sea igual a l.
c. $o2 = 2 - S[2]$: Es necesario suponer el valor de j_l en los casos en que es igual a $2 - S[2]\ mod\ 256$, que será un valor que se colocará en $S[l]$. De acuerdo a la condición anterior, en el segundo paso del PRGA el valor de $S[l]$ será intercambiado por $S[2]$ $(j = l)$. Luego la suma de $S[2]$ y $S[l]$ entregará el índice de la salida de $o2$, es decir, $S[2]$. El valor de $S[2]$ cambió y contiene el valor de $S[l]$.
d. $Si[o2] \neq 1$ y $Si[o2] \neq 2$: para alcanzar todos los eventos anteriores, $o2$ no debe ser salida ni de $S[1]$ ni de $S[2]$, lo que significa que $S[1]$ no debe contener $(2 - S[2])\ mod\ 256$ y $S[2]$ no debe ser igual ni a 1 ni a 129 $(S[2] = (2 - S[2])\ mod\ 256)$.
Finalmente, la clave atacada puede ser obtenida calculando

$$K[l] = Si[(2 - S[2])\ mod\ 256] - S[l] - j_{l-1}.$$

6. A_neg: el último grupo de ataques de Korek que tiene como finalidad disminuir el tamaño del espacio de búsqueda. Se encuentra dividió en 4 subgrupos.
   a. $j = S[1] = 2$ y $o1 = S[2] = 2$: permite deducir que el valor de j_l no ha a alterado ni $S[1]$ ni $S[2]$. De este modo, se puede descartar algunas bytes de la clave que cumplan lo siguiente $K[l] \neq 1 - S[l] - j_{l-1}$ y $K[l] \neq 2 - S[l] - j_{l-1}$.
   b. $S[2] = 0$, $o2 = 0$ y $S[1] \neq 2$ o $o1 \neq 2$: se seleccionan el segundo byte de salida, es decir, donde $o2 = 0$. De este modo, se puede descartar algunas bytes de la clave que cumplan lo siguiente $K[l] \neq 2 - S[l] - j_{l-1}$. Utilizada en Aircrack.
   c. $S[1] = 1$ y $o1 = S[2]$: la primera condición verifica que si $S[1]$ no es modificado, forzaría a que el valor de $S[2]$ sea la primera salida, lo que llevaría a que se cumpliera la segunda condición. Esto significa que el byte de la llave que los modifica tiene que ser descartado, lo que se cumple cuando $K[l] \neq 1 - S[l] - j_{l-1}$ y $K[l] \neq 2 - S[l] - j_{l-1}$. Esta condición es ocupada en Aircrack.
   d. $S[1] = 0$, $S[0] = 1$ y $o1 = 1$: son los casos cuando los valores 0 y 1 se mantiene al inicio de los estados internos. Si estos valores no varían después de la ejecución de KSA, en el PRGA se obtendrá una salida $o1 = 1$. Lo anterior se cumple cuando se calcula $K[l] \neq 1 - S[l] - j_{l-1}$.

La existencia de las 11 correlaciones restantes hace que un ataque sea mucho más eficiente y rápido, pero en las pruebas realizadas con las condiciones anteriores se obtuvieron resultados positivos.

La forma de contabilizar estas correlaciones se hace por medio de votos, en donde el cumplimiento de cada condición entrega un voto. Los votos se van acumulando para cada correlación y, finalmente entrega posibles bytes que puede ser parte de la clave. Entre más votos acumule una clave potencial, más probable es que sea correcta. Por otra parte, los bytes que acumulen una gran cantidad de votos negativos son descartados inmediatamente.

En el caso de que existan bytes con una cantidad de votos alta y/o semejante, se pueden aplicar ataques de fuerza bruta ocupando estos bytes. Esta técnica se ocupó en las primeras versiones de *Aircrack* que implementaron el ataque Korek y para elegir entre las bytes candidatos se elegía un factor llamado *fugde*. Este factor indicaba que tan amplio iba a hacer el ataque de fuerza bruta. Entre mayor sea el factor más claves prueba. Esta técnica no se ocupó en las implementaciones realizadas, pero puede ser de mucha ayuda en futuros desarrollos.

6 Resultados Obtenidos

En esta sección se presentan los resultados obtenidos durante la ejecución de diferentes implementaciones de STEA, RC4-WEP y de sus respectivos ataques tanto para la CPU como para la GPU. Para ello se utilizó un computador que cuenta con las siguientes características:

- Sistema Operativo: Linux Ubuntu 9.04
- Procesador: AMD Athlon II X2 240
- Memoria RAM: 2,0 GB
- Tarjeta Gráfica: NVidia GTX 275

Los algoritmos que se ejecutaron en la CPU fueron programados en C con el compilador GCC 4.3.3 y aquellos que fueron ejecutados en la GPU fueron escritos de la misma forma en C, con las modificaciones correspondientes para las funciones, usando el compilador NVCC 0.2.1221.

6.1 Desarrollo de los algoritmos

Al momento de implementar los algoritmos se ocuparon diferentes métodos y combinaciones entre la CPU y la GPU, siendo con todas posible obtener conclusiones de lo que será importante hacer en un futuro y qué puede ser omitido. Sin embargo, lo principal es ver el rendimiento que tenga la tarjeta gráfica.

6.1.1 Implementación de STEA

Como se explicó anteriormente, el algoritmo de STEA es sencillo de implementar y el ataque que se le aplica también lo es desde el punto de vista de la codificación. Es por eso que ambos algoritmos pueden ser implementados directamente en la GPU. A grandes rasgos se utilizan 2 métodos al momento de la implementación:

- La primera forma es implementar ambas rutinas, de cifrado y ataque, sobre la tarjeta de video, de forma que se pueda tener una referencia de la eficiencia completa que se tiene sobre el dispositivo.
- Las otras alternativas probadas intercambian entre el desarrollo sobre la CPU y el desarrollo sobre la GPU. Cabe mencionar que hasta la fecha de esta investigación no existían implementaciones de los algoritmos ocupados en la GPU. Además para el desarrollo de STEA se ocupó una rutina que genera textos en claro de forma automática con la cual igual se hicieron pruebas para ver la velocidad de genere.

La Figura 6.1 ejemplifica cómo se realizaron los diferentes desarrollos para este algoritmo. Para el otro RC4 se ocupó una implementación distinta que nace de un diseño diferente de las pruebas que se realizan.

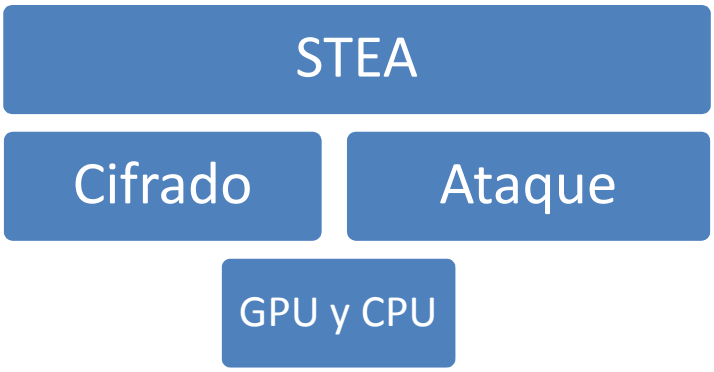


Figura 6.1 Implementación de STEA

Se ocuparon diferentes metodologías en las implementaciones realizadas del algoritmo. Cada una tenía en común que la definición de las funciones principales, cifrado, descifrado y quiebre junto con algunas definiciones de tipos de datos, quedaban guardadas en una librería llamada *stea.h*. Cada versión del algoritmo desarrollado tiene al inicio un comentario con todos los detalles relevantes de cómo se había hecho la implementación. Por lo general, se ocupaba un archivo para guardar el o las claves que se ocupaban, otro para guardar texto cifrado y otro para guardar los tiempos que había requerido cada ejecución. En un inicio la clave se ingresa por consola, pero para hacer la ejecución más automática se optó por un archivo.

6.1.1.1 Headers

La definición de las funciones utilizadas para la implementación del algoritmo sobre la CPU queda reflejada en el siguiente algoritmo:

```
/* stea.h */
#ifndef STEA_H_
#define STEA_H_
#include <stdint.h>
typedef unsigned char uchar;
typedef unsigned int uint;
typedef unsigned long ulong;

/*
* m:    Bloque a Cifrar/Descifrar (64 bit)
* k:    Key (64 bit)
 */

/*Funciones para STEA */
void stea_enc(ulong* m, ulong* k);
void stea_dec(ulong* m, ulong* k);

/*
  *   Implementación de la encriptacionencriptación de STEA y del
ataque 'divide-   and-conquer known-plaintext' por
  *   Fauzan Mirza <fauzan.mirza@seecs.nust.edu.pk>
*/
void crack_stea (ulong [2], ulong [2], ulong [2]);
void stea_enc2 (ulong [2], ulong [2], ulong [2]);
```

```
void zero(ulong X[2]){
      X[0] = 0;
      X[1] = 0;
}

#endif
```

En el caso de los headers para la GPU, se utilizaron nombres similares en varias funciones, pero el cambio se reflejó en los tipos de datos especiales usados. Estos se pueden ver a continuación:

```
/* gpu_stea.h */
#ifndef STEA_H_
#define STEA_H_
#include <stdint.h>

typedef unsigned char uchar;
typedef unsigned int uint;
typedef unsigned long ulong;
/* Estructuras */
typedef struct {
ulong k0, k1;
} STEA_KEY;
typedef struct{
      ulong m0, m1;
} STEA_BLOCK;
#define THREADS_PER_THREADBLOCK 128 #define BLOCKBUFFER_SIZE ( 2048 *
THREADS_PER_THREADBLOCK)
#define DATASIZE 512 // Mayor a THREADS_PER_THREADBLOCK

/* FUNCIONES OMITIDAS PARA LA CPU POR SER IDÉNTICAS A LAS DEFINIDAS EN
STEA.H */

/*GPU */
#define STEA_ROUND(block,key)
{ \
      (block).m0 += (block).m1 ^ (key).k0; \
      (block).m1 += (block).m0 ^ (key).k1; \
}

#define STEA_ROUND2(block,key) \
{ \
      (block).m1 -= (block).m0 ^ (key).k1; \
      (block).m0 -= (block).m1 ^ (key).k0; \
}
__global__ void gpu_stea_enc (STEA_BLOCK *m, STEA_KEY key){
      STEA_BLOCK tmp_m;
      int idx = (blockIdx.x * blockDim.x + threadIdx.x);
      int y;
      tmp_m = m[idx];
      for(y = 0; y < 32; y++)
      STEA_ROUND(tmp_m, key);
      m[idx] = tmp_m;
}
/*Sirve para ser llamada por un kernel __global__*/
__device__ float gpu_stea_enc2 (STEA_BLOCK *m, STEA_KEY key){
```

```
STEA_BLOCK tmp_m;
int idx = (blockIdx.x * blockDim.x + threadIdx.x);
int y;
tmp_m = m[idx];
__syncthreads();
for(y = 0; y < 32; y++)
    STEA_ROUND(tmp_m, key);
m[idx] = tmp_m;
return 0;
}

#endif
```

Mediante las rutinas anteriores, se realizar las diferentes pruebas alternando el uso de la CPU y GPU.

6.1.1.2 Textos planos

En un principio los textos en claro fueron leídos desde un archivo plano, pero finalmente se optó por crear una rutina dentro del código para su creación, la cual sirvió más adelante para ser utilizada dentro de la GPU. A su vez, el texto cifrado que se guardaba en un archivo, era luego descifrado y con ello se realizaba la comparación con el texto original. Para la generación mediante de la GPU se ocupó la siguiente rutina:

```
__global__ void cuda_plaintext( ulong cu_pt[NUMPT][2], ulong
Cantidad){
      const ulong i = blockDim.x * blockIdx.x + threadIdx.x;
      const ulong k = i;
      const ulong N = Cantidad;
      shared__ ulong cero;
      cero = 0;
      cu_pt[k][0] = cero;
      if(i < N)
            cu_pt[k][1] = i;
}
```

Por otro lado, la generación por medio la CPU y que finalmente se ocupó para las pruebas finales es la siguiente:

```
void plaintexts(){
      ulong i, num_pt;
      ulong plaintext[2];
      num_pt = NUMPT;
      plaintext[0] = 0x10000000;
      for(i = 0x0; i < num_pt; i++){
            plaintext[1] = i;
            pt[i][0] = plaintext[0];
            pt[i][1] = plaintext[1];
      }
      printf("Creados %d textos en claro\n\n", (int)num_pt);
}
```

6.1.1.3 Quiebre STEA

Para la implementación de la rutina de quiebre del algoritmo se utilizó la siguiente implementación en la GPU:

```
/* RUTINA DE QUIEBRE IMPLEMENTADA EN GPU */
      __global__ void crack_stea_gpu (ulong *P, ulong *C ,ulong K[2]){
            ulong mask ;
            __shared__ ulong TP[2], TK[2], TC[2];
            __shared__ ulong k[2];
            STEA_BLOCK m;
            __shared__ STEA_KEY keya;
            mask = 1;
            for (int i=0; i<32; i++){
                  TP[0] = P[0] & (mask - 1);
                  TP[1] = P[1] & (mask - 1);
                  TK[0] = K[0] & (mask - 1);
                  TK[1] = K[1] & (mask - 1);
                  TC[0] = 0; TC[1] = 0;
                  m.m0 = TP[0]; m.m1 = TP[1];
                  keya.k0 = TK[0]; keya.k1 = TK[1];
                  gpu_stea_enc2 (&m, keya);
                  TC[0] = m.m0; TC[1] = m.m1;
                  TC[0] = TC[0] ^ C[0];
                  TC[1] = TC[1] ^ C[1];
                  k[0] = P[0] ^ TC[0] ^ TC[1];
                  k[1] = P[0] ^ P[1] ^ TC[1];
                  K[0] |= (k[0] & mask);
                  K[1] |= (k[1] & mask);
                  mask<<= 1;
            }
      }
```

La rutina utilizada en la CPU es similar a la que se mostró anteriormente, básicamente los cambios se reflejan en la forma en que se ocupa la memoria dentro la GPU. En este caso la implementación es la siguiente:

```
      void crack_stea (ulong P[2], ulong C[2], ulong K[2]){
            ulong mask = 1;
            ulong TP[2], TK[2], TC[2];
            ulong k[2];
            int i;
            /* Recuperación de un bit de K[0] y de  K[1] por iteración */

            for (i=0; i<32; i++){
                  /* Encriptación del bit menos significativo */
                  TP[0] = P[0] & (mask - 1);
                  TP[1] = P[1] & (mask - 1);
                  TK[0] = K[0] & (mask - 1);
                  TK[1] = K[1] & (mask - 1);
                  zero(TC);
                  stea_enc2(TP, TK, TC);
                  TC[0] = TC[0] ^ C[0];
                  TC[1] = TC[1] ^ C[1];
                  /* Resolución de la ecuación lineal */
                  k[0] = P[0] ^ TC[0] ^ TC[1];
```

```
                k[1] = P[0] ^ P[1] ^ TC[1];
                K[0] |= (k[0] & mask);
                K[1] |= (k[1] & mask);
                mask<<= 1;
        }
        return;
}
```

Los tiempos de ejecución fueron medidos en diferentes puntos: en la encriptación, desencriptación, creación de textos en claro, ataque, tiempo total. Las tablas y gráficos de la sección de los resultados muestran los valores junto con la respectiva comparación entre CPU y GPU.

En la sección 6.2se muestran los resultados obtenidos donde se ve que el rendimiento de la tarjeta gráfica no supera, en muchos casos, a lo que se obtuvo con la CPU.

6.1.2 Implementación de RC4

A diferencia de la implementación de STEA, en el desarrollo utilizado para RC4 el algoritmo de cifrado se ejecutó únicamente sobre la CPU y el ataque se hizo sobre ambos dispositivos, tal cual se ve en laFigura 6.2. Este cambio radica en que RC4 fue pensado para ser eficiente sobre software.

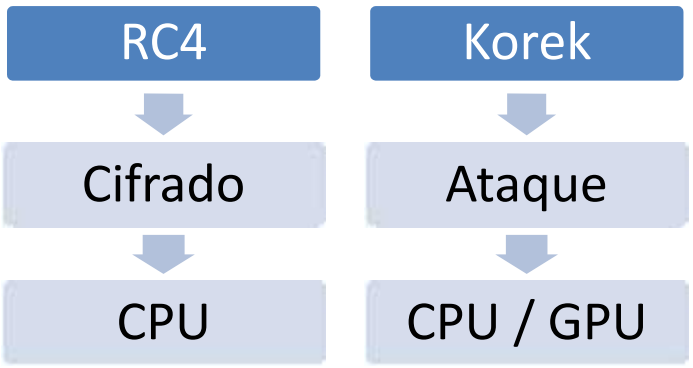


Figura 6.2 Implementación de RC4

La adaptación hecha a RC4 para que funcionara de la manera más similar a WEP llevó a que se tuvieran ciertas consideraciones especiales en la implementación. En la sección 5.2.2 se explicó cómo RC4 es manipulado para que funcione en WEP y con la misma modificación se trabajó para hacer posible un ataque.

6.1.2.1 Headers

Cada desarrollo tiene una librería de definición llamada *rc4.h* en la cual se detallaban funciones y tipos de datos que se ocupan. La clave y el texto en claro son leídos desde un archivo de texto. Con cada texto en claro leído se crea un IV (vector de inicialización), el cual

junto a la clave forman la clave de sección que se ocupa para encriptar cada texto en claro que se tiene.

Para el caso de RC4, la librería definida es la siguiente:

```
/* rc4.h */
#ifndef RC4_H_
#define RC4_H_
#include <stdint.h>
typedef unsigned char uchar;
typedef unsigned int uint;
typedef unsigned long ulong;

void swap(uchar *, uint , uint );
void rc4_ksa(uchar *, uint ); /* key-scheduling Algorithm */
uchar rc4_prga(); /* Pseudo-Random Generation Algorithm */
uchartest_hexa[] =
{'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};

#ifndef AMOD /* absolute value modulus, so we don't reference neg
values */
#define AMOD(x, y) ((x) % (y) <0 ? ((x) % (y)) + (y) : (x) % (y))
#endif

uchar opt_keys[] =
{'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','
R','S','T','U','V','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','x','y','z'};
/* Ataques Korek utilizados */
enum KoreK_attacks{
      A_u15, /* semi-stable  15% */
      A_s13, /* stable  13%             */
      A_u13_1, /* unstable 13% */
      A_u13_2, /*unstable?13%             */
      A_u13_3,/*unstable? 13%             */
      A_s5_1,/* standard   5% (~FMS)      */
      A_s5_2,/* other stable  5%             */
      A_s5_3,/* other stable  5%             */
      A_u5_1,/* unstable  5% no good ?   */
      A_u5_2,/* unstable  5%             */
      A_u5_3,/* unstable  5% no good     */
      A_u5_4, /* unstable  5%             */
      A_s3, /*stable3%      */
      A_4_s13,/*stable  13% on q = 4    */
      A_4_u5_1,/* unstable  5% on q = 4    */
      A_4_u5_2,/* unstable      5% on q = 4    */
      A_neg /* helps reject false positives */
};
const uchar R[256] =
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,2
6,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49
,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,
73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,9
6,97,98,99,100,101,102,103,104,105,106,107,108,109,110,111,112,113,114
,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130,131,1
32,133,134,135,136,137,138,139,140,141,142,143,144,145,146,147,148,149
```

```
,150,151,152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,1
67,168,169,170,171,172,173,174,175,176,177,178,179,180,181,182,183,184
,185,186,187,188,189,190,191,192,193,194,195,196,197,198,199,200,201,2
02,203,204,205,206,207,208,209,210,211,212,213,214,215,216,217,218,219
,220,221,222,223,224,225,226,227,228,229,230,231,232,233,234,235,236,2
37,238,239,240,241,242,243,244,245,246,247,248,249,250,251,252,253,254
,255};

/* Distintas funciones de Swaping (intercambio) utilizadas */
#define SWAP(x,y) { unsigned char tmp = x; x = y; y = tmp; }
void swap(uchar *s, uint i, uint j) {
      uchar temp = s[i];
      s[i] = s[j];
      s[j] = temp;
}
void swap2(uint *a, uint *b){
      uint c;
      c=*a;
      *a=*b;
      *b=c;
}
/* Definición de funciones adicionales utilizadas normalmente para
transformaciones de hexadecimal */
/* Equivalencia Hex-Char (*/
uchar dev_hex(ulong )
ulong dev_val(uchar c);
/* transforma mensaje hexadecimal a string*/
char * inthexTOstrhex(ulong);
/* pasa un string a hexadecimal*/
ulong * strhexTOinthex(uchar *,int);

#endif
```

Para el ataque se ocupa una librería llamada *gpu_wep.h* con algunas definiciones de funciones y tipos de datos necesarios. Las pruebas realizadas tanto sobre la CPU como sobre la GPU utilizan la función *korek_attack*, la cual tiene como parámetros la clave y el bit que se va a atacar, de la misma forma que se realiza en Aircrack [Aircrack, 2006]. También se ocupan las 6 correlaciones descritas en la sección 5.2.4, cuyo código también se obtuvo de la versión de Aircrack utilizada.

En el caso de la librería utilizada para el desarrollo de ataque Korek sobre WEP, la librería que se ocupa es la siguiente:

```
#ifndef WEP_H
#define WEP_H
typedef unsigned char uchar;
typedef unsigned int uint;
typedef unsigned long ulong;
#define LEN_KEY 8

/*Función en la CPU que llama a las rutinas de la GPU*/
void pre_korek(ulong [LEN_KEY], uint );

/* Rutinas en la GPU */
__global__ void gpu_korek(ulong Key[LEN_KEY], uint p, ulong *guess, int
```

```
     *, int [256], int [256], int [256], int [256], int [256]);
     __global__ void gpu_korek2(ulong Key[LEN_KEY], uint p, ulong *guess);

     __device__ float swap(uint *a, uint *b){
            uint c;
            c=*a;
            *a=*b;
            *b=c;
            return 0;
      }
#endif
```

La diferencia entre las implementaciones radica en que para la GPU se ocupa las ventajas de los diferentes espacios de memoria existente. Por ejemplo se ocupa una gran cantidad de variables compartidas (*shared*) de manera de optimizar los accesos a memoria que se realizan. Este punto es donde se pueden hacer las primeras mejoras al momento de optimizar el algoritmo y es un tema que a futuro se puede tratar.

6.1.2.2 Ataque Korek

En el caso del algoritmo utilizado sobre la CPU, la rutina de quiebre es similar a la que utiliza en Aircrack. Salvo algunas personalizaciones del algoritmo, a grandes rasgos no presenta mayor diferencia. En la implementación hecha sobre la GPU se hicieron las transformaciones necesarias para un correcto funcionamiento. Se probaron diferentes configuraciones con los parámetros permitidos en el kernel y la rutina utilizada fue la siguiente:

```
      __global__ void gpu_korek(ulong (*Key)[LEN_KEY], uint p, ulong
*guess){
            uint cont = 0;
            uint S[256];
            __shared__  ulong clave[LEN_KEY];
            __shared__ uint i , j, o1, o2, tmp, jp;
            __shared__ int stat1[256], stat2[256], stat3[256],
            stat4[256],stat5[256], stat6[256], pstat2[256];
            __shared__ int tstat1, tstat2, tstat3, tstat4, tstat5;
            __shared__ ulong bestchance;
            ulong value;
            __shared__ int chance[5];
            ulong tx = blockIdx.x * blockDim.x + threadIdx.x;
            bestchance = 0x0;
            clave[0] = Key[tx][0]; clave[1] = Key[tx][1]; clave[2] =
            Key[tx][2]; clave[3] = Key[tx][3];
            clave[4] = Key[tx][4]; clave[5] = Key[tx][5]; clave[6] =
            Key[tx][6]; clave[7] = Key[tx][7];
            for(i = 0; i < 256; i++){
                  stat1[i]=0; stat2[i]=0; pstat2[i]=0; stat3[i]=0;
                  stat4[i]=0; stat5[i]=0; stat6[i]=0;
            }
            clave[0] = tx; // -> for (K[0] = 0; K[0] < 256; K[0]++)
            for (clave[1] = 0;clave[1] < 256; clave[1]++){
                  for (clave[2] = 0; clave[2] < 256; clave[2]++) {
                        for (i = 0; i < 256; i++) S[i] = i;
                        for (i = 0, j = 0; i < 256; i++) {
```

```
      j = (j + clave[i & 0x0f] + S[i]) & 0xff;
      swap(S+i, S+j);
}
i = 1;
j = S[1];
tmp = (S[i] + S[j]) & 0xff;
swap(S+i, S+j);
o1 = S[tmp]; //primer byte
i = 2;
j = (j+S[2]) & 0xff;
tmp = (S[i] + S[j]) & 0xff;
swap(S+i,S+j);
o2 = S[tmp];
for (i = 0; i < 256; i++) S[i] = i;
for (i = 0,j = 0; i < p; i++) {
    j = (j + clave[i & 0x0f] + S[i]) & 0xff;
swap(S+i, S+j);
}
/* FMS 5% (~A_s5_1) standard */
if ((S[1] < p) && ((S[1]+S[S[1]]) == p)) {
jp=o1;
stat1[(jp-j-S[p]) & 0xff]++;
}
/* A_s13 estable */
if (S[1] == p) {
for (jp = 0; S[jp] != 0; jp++);
if (o1 == p) {
            stat2[(jp-j-S[p]) & 0xff]++;
      }
      pstat2[(jp-j-S[p]) & 0xff]++;
}
/* A_u15 semi-stable */
if ((o2 == 0) && (S[p] == 0) && (S[2] != 0))
stat3[(2-j-S[p]) & 0xff]++;
if ((S[1] > p) && (((S[2]+S[1]-p) & 0xff) == 0)) {
if (o2 == S[1]) {  /* A_s5_2 other stable */
            for (jp = 0; S[jp] != ((S[1]-S[2]) &
                  0xff); jp++);
                  if ((jp!=1) && (jp!=2)) stat4[(jp-
                  j-S[p]) & 0xff]++;
}else
      if (o2 == ((2-S[2]) & 0xff)) {/* A_s5_3
other stable */
            for (jp=0; S[jp] != o2; jp++);
                  if ((jp!=1) && (jp!=2)) stat5[(jp-
                  j-S[p]) & 0xff]++;
      }
}
__syncthreads();
/* A_neg */
if (S[2] == 0) {
      if ((S[1] == 2) && (o1 == 2)) {
            stat6[(1-j-S[p]) & 0xff]++;
            stat6[(2-j-S[p]) & 0xff]++;
      }else
            if (o2==0) {/* A_neg */
                  stat6[(2-j-S[p]) & 0xff]++;
```

```
                        }
                  }
                  if ((S[1] == 1) && (o1 == S[2])) {
                        stat6[(1-j-S[p]) & 0xff]++;
                        stat6[(2-j-S[p]) & 0xff]++;
                  }
                  if ((S[1] == 0) && (S[0] == 1) && (o1 == 1)){
                  stat6[(-j-S[p]) & 0xff]++;
                        stat6[(1-j-S[p]) & 0xff]++;
                  }
            }
      }           cont++;
      tstat1=0;
      tstat2=0;
      tstat3=0;
      tstat4=0;
      tstat5=0;
      for (i = 0 ; i < 256 ;i++) { // Para todas las posibilidades
            tstat1+=stat1[i];
            tstat2+=stat2[i];
            tstat3+=stat3[i];
            tstat4+=stat4[i];
            tstat5+=stat5[i];
      }
      chance[0] = stat1[0];
      chance[1] = stat2[0];
      chance[2] = stat3[0];
      chance[3] = stat4[0];
      chance[4] = stat5[0];
      for (i = 0; i < 256; i++){
            if(stat6[i] <= 32){ /* caso A_neg */
                  if(stat1[i] >= chance[0]) chance[0] = stat1[i];
                  if(stat2[i] >= chance[1]) chance[1] = stat2[i];
                  if(stat3[i] >= chance[2]) chance[2] = stat3[i];
                  if(stat4[i] >= chance[3]) chance[3] = stat4[i];
                  if(stat5[i] >= chance[4]) chance[4] = stat5[i];
                  if(((ulong)chance[0]+(ulong)chance[1]+(ulong)
                  chance[2]+(ulong)chance[3]+(ulong)chance[4]) >=
                  bestchance){
                        bestchance = (ulong) chance[0] + (ulong)
                        chance[1] + (ulong) chance[2] + (ulong)
                        chance[3] + (ulong) chance[4];
                        value = (ulong)i;
                  }
            chance[0] = 0;
            chance[1] = 0;
            chance[2] = 0;
            chance[3] = 0;
            chance[4] = 0;
            }
      }
      guess[tx] = value;
}
```

La función que realiza el llamado al kernel tiene una la configuración que siempre fue variando para hacer diferentes tipos de pruebas. Básicamente la estructura era la siguiente:

```
      void pre_korek(ulong K[LEN_KEY], uint p){
            ulong (*g_key)[LEN_KEY], *g_guess, g_aux[LEN_KEY]; /*Clave y
            acierto */
            ulong *guess, clave[CANT][LEN_KEY];
            cudaError_t error;
            volatile double t1 = 0, t2 = 0;
            FILE *times;
            t1 = clock();
            cudaMalloc((void **) &g_key, sizeof(ulong) * CANT * LEN_KEY );
            cudaMalloc((void **) &g_guess, sizeof(ulong) * CANT); /*Testing
*/
            cudaMallocHost((void**) &guess, sizeof(ulong) * CANT); /*
Testing */

            for(int i = 0; i < CANT; i++){
                  clave[i][0] = K[0]; clave[i][1] = K[1]; clave[i][2] =
                  K[2]; clave[i][3] = K[3];
                  clave[i][4] = K[4]; clave[i][5] = K[5]; clave[i][6] =
                  K[6]; clave[i][7] = K[7];
            }
            cudaMemcpy(g_key,K,sizeof(ulong)*LEN_KEY,
      cudaMemcpyHostToDevice);
            printf("\n  A resolver: %lX\n", K[p]);
            t2 = clock();
            gpu_korek<<<2,CANT>>>(g_key, p, g_guess); /* ATAQUE */
            t2 = (clock()-t2)/CLOCKS_PER_SEC;
            cudaMemcpy(guess, g_guess, sizeof(ulong) * CANT,
            cudaMemcpyDeviceToHost);
            cudaFree(g_guess);
            cudaFree(g_key);
            t1 = (clock()-t1)/CLOCKS_PER_SEC;
            printf(" \nt1: %.15f \t t2: %.15f \n", t1, t2);
            for (int i = 0; i < CANT; i++)
                  printf(" %lX ", guess[i]);
            printf("\n");
            error=cudaGetLastError();
            printf("\n\nCUDA error code is %i: %s.\n\n",
            error,cudaGetErrorString(error));
            times = fopen(rc4_times,"a");
            if (! times) {
                  perror ("Error creating file " rc4_times);
                  exit (2);
            }
            fprintf(times,"%.5f %.5f\n", t1, t2);
            fclose(times);
      }
```

En los resultados de la sección 6.3.1 se mostrará cómo fue el funcionamiento de las diferentes implementaciones propuestas y se verá que los resultados fueron mejores que los obtenidos con STEA.

6.2 Resultados STEA

6.2.1 STEA

La primera prueba que sea realizó fue la implementación de STEA para C. Para ello, se ocuparon las 32 rondas del algoritmo, diferentes cantidades de textos en claro y una misma clave. De forma similar, se realizó una implementación del algoritmo sobre la GPU.

La Tabla 6.1 presenta los resultados ocupando 4 cantidades diferentes de textos en claro y ocupando solamente la función de encriptación para las comparaciones.

Tabla 6.1 Encriptado STEA CPU v/s GPU en seg

Encriptado CPU	Encriptado GPU	# Textos en claro
0,82	137,70	983040
7,17	354,61	8388480
14,26	500,30	15728640
14,26	**>500,30**	16776960

Los tiempos después de cierta cantidad de textos en claro se dispararon ampliamente en comparación a lo que sucedía con la CPU, donde los tiempos eran similares y en el último caso superaba ni siquiera fue contabilizado por su alto valor. Después de varias pruebas realizadas con distintas alternativas en el manejo de la memoria, se llegó a la conclusión de que los malos resultados sobre la GPU se deben a los altos requerimientos de memoria que tiene el algoritmo.

Como los resultados anteriores no muestran que la implementación sobre la GPU sea más eficiente que la hecha sobre la CPU, se optó por utilizar una función del algoritmo y optimizarla. La generación de rápida de textos planos es importante para el desarrollo de las pruebas y es por eso que se implementó tanto para la CPU como para la GPU. El gráfico de la Figura 6.3 muestra los resultados obtenidos en base a la cantidad de textos en claro que se creaban, la misma cantidad ocupada para la encriptación.

A pesar de que los resultados no son significativamente mejores, se puede ver que después de cierta cantidad de textos planos producidos los tiempos mejoran y tienden a mantenerse estables.

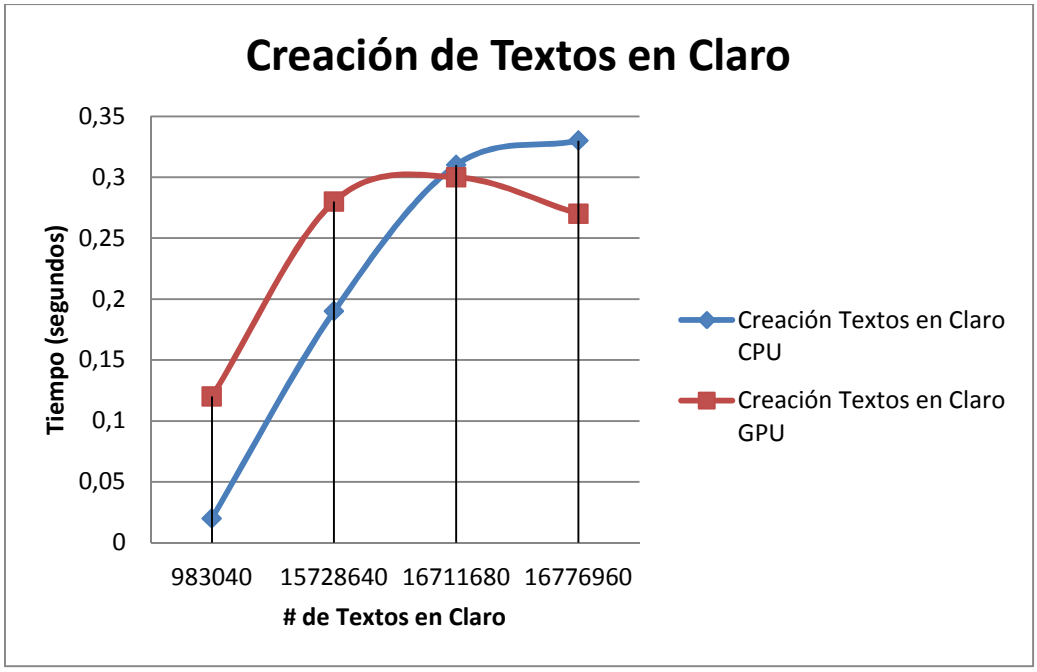


Figura 6.3 Creación de Textos en Claro para STEA

Los valores del gráfico representan los mejores tiempos con las diferentes implementaciones utilizadas. Alguna variantes de las implementaciones hechas ocupan la memoria global para almacenar valores que se ocuparan constantemente, otras ocupan la memoria compartida, siendo este el caso que se gráfica.

De manera global se desprende que la implementación de STEA incluyendo la creación de textos en claro es mucho más eficiente sobre la CPU, puesto que las diferencias en esta última función no son significativas.

6.2.2 Ataque Divide-and-Conquer sobre STEA

El ataque Divide-and-Conquer para STEA sobre la CPU tiene un rendimiento tan eficiente en la recuperación de la clave como lo tiene el algoritmo de encriptación. Es importante mencionar que para el ataque solo es necesario tomar un texto en claro para la recuperación de la clave. Sin embargo en los resultados graficados en la Figura 6.4 se toman distintas cantidades de textos en claro y se calcula el tiempo global que toma la recuperación de la clave.

Por lo tanto si consideramos el caso de tomar un solo texto en claro y atacarlo para recuperar la clave, ambas implementaciones realizadas entregan un tiempo computacional igual a 0 y significa que no hay necesidad de utilizar la GPU para implementación. La razón por la que la eficiencia baja considerablemente, cuando se hacen mediciones para un gran número de textos en claro, es la carga de datos en la memoria del dispositivo. Es necesario ir cargando datos en la GPU para cada texto en claro diferente y este proceso es más lento.

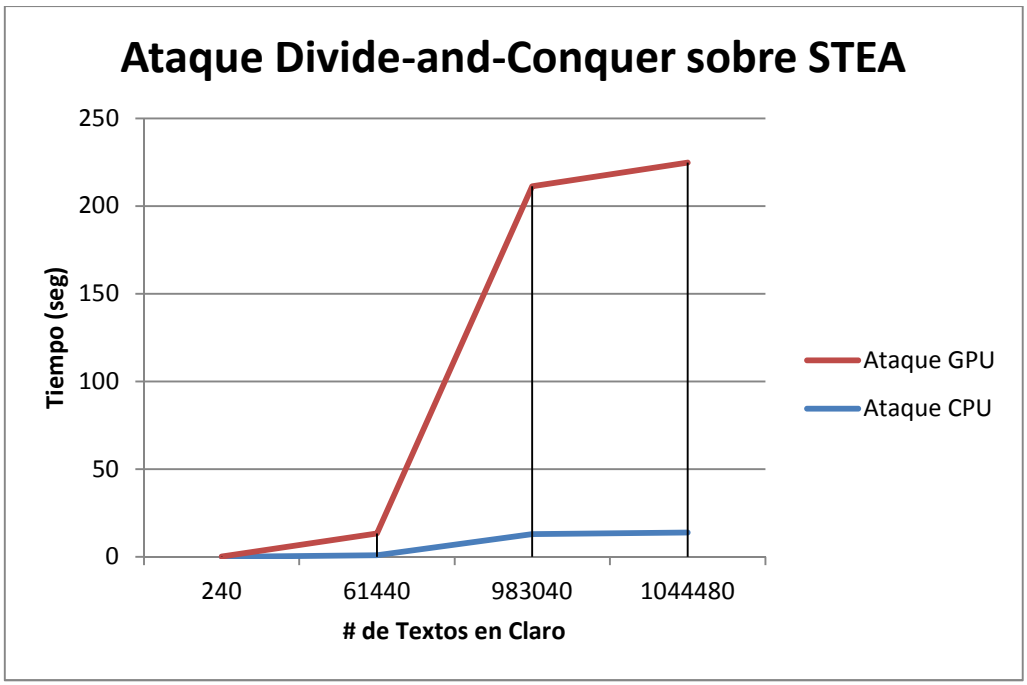


Figura 6.4 Ataque Divide-and-Conquer

Mejoras futuras podrían apuntar a la modificación en la forma de cargar los datos, pero al igual que para el algoritmo de encriptado, este ataque ya fue pensado para ser eficiente sobre la CPU, por lo cual un futura mejora a lo realizado sería sin duda implementar el algoritmo original de TEA o alguna variante más compleja como XTEA. En el caso de los ataques sería conveniente utilizar un número reducido de rondas y comparar los resultados.

6.3 Resultados RC4-WEP

6.3.1 Ataque Korek

La visualización de los principales resultados obtenidos se hace de dos formas. Primero se muestra un gráfico comparativo entre los tiempos de recuperación para la CPU y la GPU para distintas cantidades de bytes. Para este último el tiempo corresponde a un promedio de lo obtenido por el mejor y el peor caso. Y segundo, un gráfico con los tiempos de recuperación de uno, tres y cinco bytes del mejor, del peor y del promedio en la implementación con CUDA.

En la Figura 6.5 se aprecia los tiempos obtenidos en ambos caso donde se aprecia claramente cómo la implementación hecha sobre la GPU para el ataque Korek resulta ser más eficiente de manera significativa. Es el caso contrario a lo obtenido con el ataque a STEA, puesto que el uso de la GPU acelera de manera significativa el tiempo de recuperación de uno o varios bytes de la clave. Es lógico de asumir que existe un incremente lineal en los tiempos de recuperación, donde en el caso de la CPU se puede ver una diferencia aproximada de 90 segundos entre byte y en el caso de la GPU corresponde a cerca de 8 segundos.

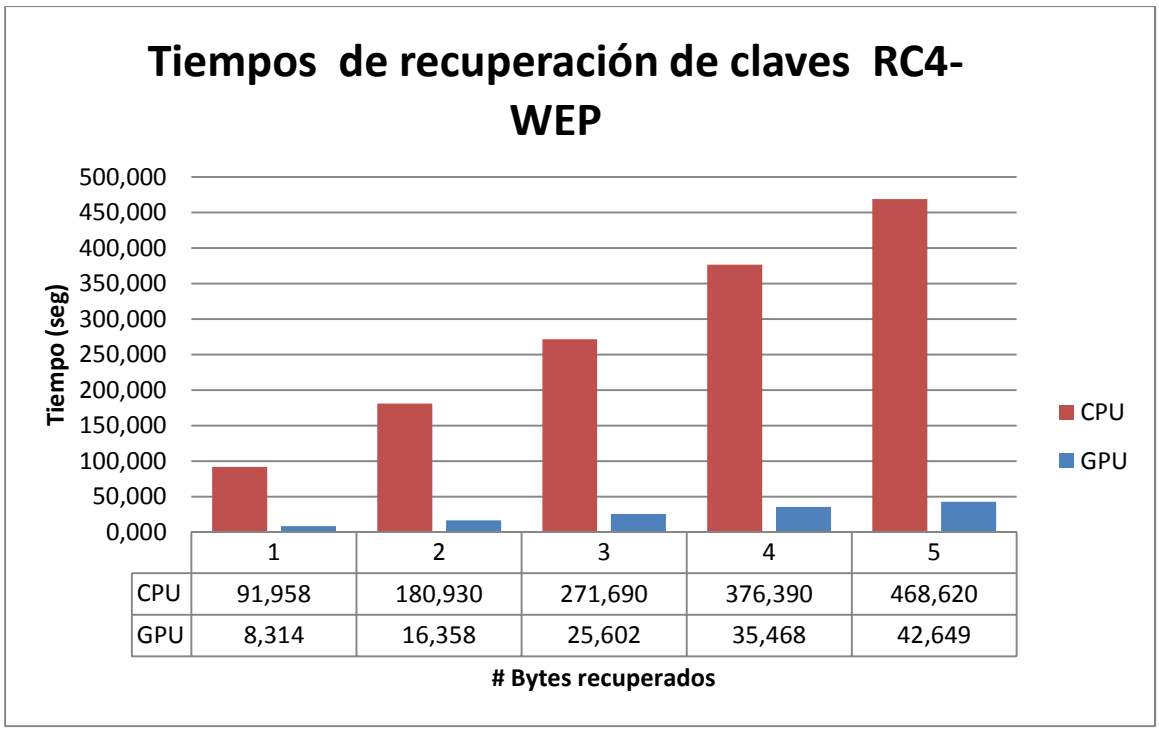


Figura 6.5 Tiempo ataque Korek CPU y GPU

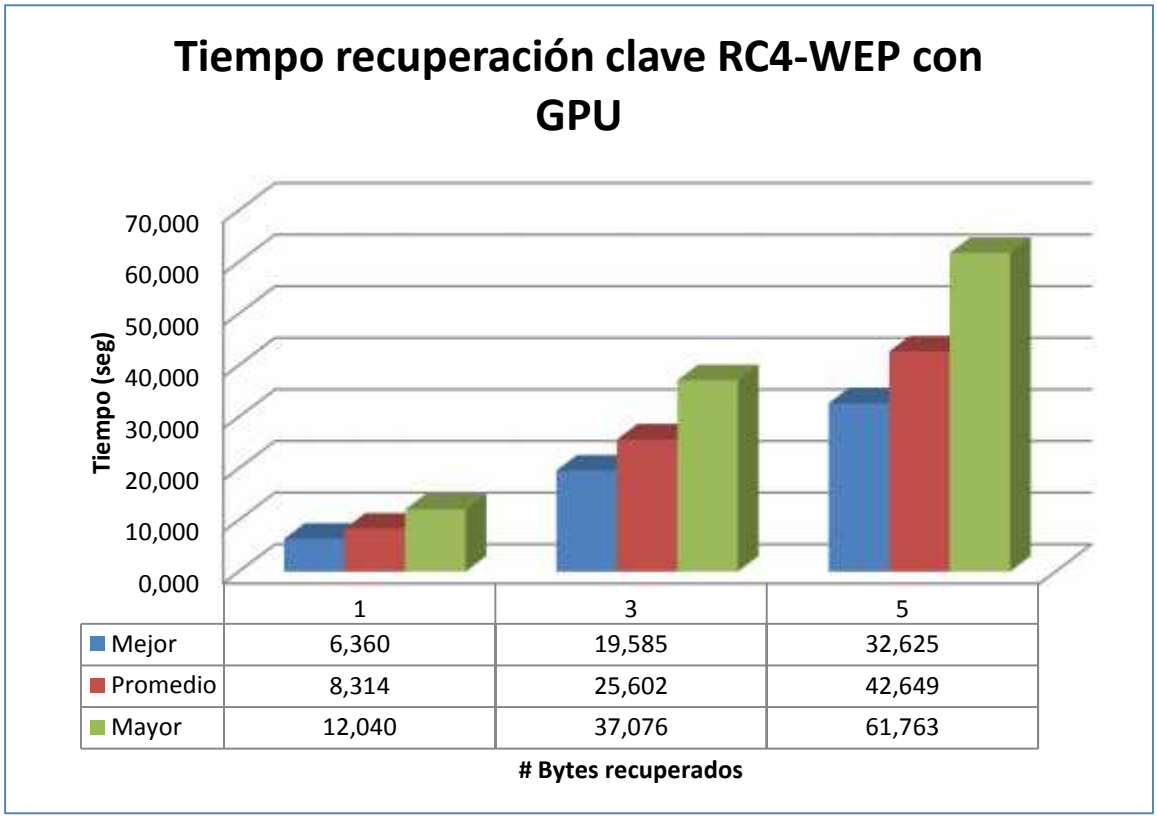


Figura 6.6 Tiempo ataque Korek GPU

Tal como sucedió en todas las implementaciones varias fueron las alternativas que se ocuparon hasta que se llegó a un resultado satisfactorio. En la Figura 6.6 se presenta un resumen de dos tiempos obtenidos, donde uno representa el más bajo correspondiente a una implementación poco depurada y un segundo que representa una ya mejor depurada. Además se da un promedio entre ambos de modo de ejemplificar un tercer caso representativo de los resultados obtenidos.

Mejoras a las implementaciones apuntan primero a la inclusión de todas las correlaciones identificadas en Korek, lo que sin duda disminuiría los tiempos de recuperación de un bytes y, por consiguiente, de la clave en conjunto. También la ampliación del tamaño de la clave secreta que en este caso es de 5 bytes. Pero sin duda, una mejora importante sería la implementación del ataque PTW, propuesto en [Tews *et al.*, 07], y que actualmente se encuentra en las implementaciones de Aircrack en desmedro de las correlaciones de Korek.

7 Conclusiones

La primera parte de este proyecto se centra en definir los componentes principales de GPU, CUDA y algoritmos criptográficos y las técnicas más comunes para el quiebre de algoritmos, de tal manera de abarcar todos los tópicos generales con los que se trabajará y que servirán como conocimientos básicos para futuros trabajos. Cumpliendo con el primero de los objetivos específicos definidos.

Mediante la definición de los algoritmos criptográficos que se utilizarían, se pudo completar el segundo objetivo específico definido. Se tomaron de la familia de los algoritmos simétricos dos cifrados diferentes, uno de ellos fue TEA y el otro RC4. Se explicó cada uno de ellos, además de ciertos conceptos que es necesario manejar para su comprensión. Luego de eso, para el fin de la investigación se explicaron los cambios que se realizarían a los algoritmos. En el caso del primero, se tomó una versión reducida llamada STEA que tiene una serie de debilidades conocidas, siendo una de esas debilidades llamada ataque dividir-y-conquistar la que se ocupó para hacer las pruebas. Para el caso de RC4, se ocupó una implementación famosa ocupada en WEP cuya débil implementación ha sido explotada hace un tiempo y se simuló el comportamiento para el caso de estudio.

La parte final del trabajo incluye tanto la especificación de las rutinas utilizadas para el desarrollo de los algoritmos criptográficos como para los ataques realizados y los resultados obtenidos con las pruebas realizadas. En la parte final, se muestran los resultados obtenidos después de las implementaciones realizadas. Para el caso de STEA, considerando que su implementación fue pensada para ser rápida sobre software, los resultados muestran que para diferentes cantidades de textos en claro el rendimiento de la implementación sobre la GPU es menos eficiente que su implementación en la CPU.

En el caso del ataque dividir-y-conquistar, los resultados muestran que la aplicación del ataque en la GPU para diferentes cantidades de textos en claro es igual de ineficiente que en el caso anterior, a excepción del caso en que se quiera aplicar a un solo texto en claro. Los resultados demostraron ser mejores, no ampliamente, en la creación de una gran cantidad de textos en claro, lo que queda sustentado con el hecho de que las GPU están pensadas para manejar una gran cantidad de datos.

En el caso de RC4 y su adaptación cómo se utiliza en WEP, los resultados fueron mejores que los anteriores. La recuperación de la clave resultó ser mucho más eficiente sobre la GPU y la CPU muestra tiempo cerca de 10 veces mayores. También dependiendo de la configuración que se utilizó, puede haber mejoras en la implementación del ataque realizado sobre la tarjeta gráfica.

Finalmente, el uso de la tecnología GPU sobre los algoritmos seleccionados demostró ser un poco dispar. Por una parte, la utilización sobre STEA demostró ser innecesaria. Esto debido a la simpleza del algoritmo y de la rutina de ataque. Por otro lado, la implementación hecha de WEP generó resultados mejores al utilizar la GPU. Se ven las dos caras sobre la posible existencia de ventajas en la utilización de tarjetas gráficas para desarrollar algoritmos criptográficos. Sin duda, la elección tanto del algoritmo criptográfico como de la rutina de

quiebre tienen mucho que ver con los resultados que se puedan obtener, lo que deja un campo muy amplio de trabajo futuro.

7.1 Recomendaciones

En base a lo propuesto anteriormente se pueden definir algunos puntos importantes en un futuro trabajo. Algunos de ellos relacionados directamente con el desarrollo de algoritmos criptográficos y otros que no estuvieron dentro del ámbito de la investigación, pero que sí darían un valor agregado.

Lo primero, sería aplicar mejoras en la implementación de STEA para mejorar el rendimiento, de forma de asemejarse a lo logrado con la CPU o simplemente la utilización de otras variantes dentro de la familia de TEA (TEA, XTEA, Block TEA). Del punto de vista de los ataques, un trabajo futuro sería el desarrollo de otro tipo de ataques sobre estos algoritmos. Criptoanálisis diferencial (*Differential Cryptoanalysis*), o de llave correlacionada (*Related Key Cryptoanalysis*), o análisis estadístico (*Statistical Analysis*) son alguno de los ataques que podrían ser utilizados sobre la GPU y poder utilizar algunas de sus ventajas, por ejemplo en el manejo de grandes cantidades de datos.

Para lo que concierne a RC4 y su implementación en WEP, un trabajo futuro importante sería la aplicación de todas las correlaciones propuestas hasta la fecha por Korek, Lo que podría ser comparado con las seis utilizadas en este trabajo. Y una segunda propuesta, sería la implementación del ataque PTW, propuesto en [Tews *et al.*, 07], el cual ya tiene un buen rendimiento sobre la CPU.

Un tema no visto en este trabajo tiene relación con ocupar una mayor cantidad de tarjetas gráficas y hacerlas trabajar en forma paralela. Esto es conocido como Cluster de GPU (*GPU Cluster*) y es un concepto que se está utilizando ampliamente en todo tipo de áreas. Contar con un rack con varias GPU funcionando, es en realidad el gran paso que se puede dar después de conocer cómo funciona CUDA y que en criptografía podría significar muchos avances.

Todas estas propuestas fueron en su momento parte de la investigación, pero dejadas de lado para limitar el ámbito de trabajo. Sin embargo, los trabajos futuros sobre GPU y CUDA tendrán que ver mucho con las propuestas antes mencionadas.

8 Bibliografía

[Amorim *et al.*, 08], R. Amorim, G. Haase, M. Liebmann, R. dos Santos. "ComparingCUDA and OpenGL implementations for a Jacobi iteration", SFB-Report No. 2008-25.

[Bernstein*et al.*, 09], D. Bernstein, T. Chen, C. Cheng, T, Lange, B. Yang, "ECM on Graphics Cards", EUROCRYPT '09, 2009.

[Beth y Geiselmann,07], Prof. Dr. Th. Beth, Dr. W. Geiselmann, "Datensicherheitstechnik (Signale, Codes und Chiffren II)", Institut für Algorithmen und Kognitive Systeme – Universität Karlsruhe, 2007.

[Boeing, 08], A. Boeing, "Survey and future trends of efficient cryptographic function implementations on GPGPUs, 2008.

[Chaabouni, 06], R. Chaabouni."Break wep faster with statistical analysis", Technical report, EPFL, LASEC, Junio 2006.

[Cook *et al.*,05], D. L. Cook, J. Ioannidis, A. D. Keromytis, J. Luck, "CryptoGraphics: Secret Ket Cryptography Using Graphics Cards", RSA Conference, Cryptographer's Trac (CT-RSA), 2005.

[DirectX], Microsoft DirectX, http://www.microsoft.com/windows/directx.

[Feistel, 73], H. Feistel, "Cryptography and Computer Privacy", Scientific American, Vol. 228(5), 1973.

[Fip64, 93], "Data Encryption Standard", http://www.itl.nist.gov/fipspubs/fip46-2.htm, 1993.

[Fip197, 01], National Institute of Standars and Technology (NIST), "FIPS 197: Advanced Encryption Standard (AES)", 2001.

[Fluhrer *et al.*, 01],S. Fluhrer, I. Mantin, A. Shamir, "Weaknesses in the Key Scheduling Algorithm of RC4", Eighth Annual Workshop on Selected Areas in Cryptography, Toronto, Canada, Agosto, 2001.

[Geiselmann *et al.*, 06], W. Geiselmann, J-M.Bohli, S. Röhrih,"Public Key Kryptographie")", Institut für Algorithmen und Kognitive Systeme – Universität Karlsruhe, 2006.

[Gratton, 07], Steven Gratton, "The Cholesky factorization of a matrix", www.ast.cam.ac.uk/~stg20, 2007.

[Korek, 04], Korek,"Experimental WEP Attacks",http://www.netstumbler.org/showthread.php?t=12489, 2004.

[Lucena, 03], M. Lucena, "Criptografía y seguridad en computadores", Tercera Edición, 2003.

[Manavski y Valle, 08], S. Manavski, G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment", BMC Bioinformatics, 2008.

[Manavski, 07], S. Manavski, "CUDA compatible GPU as an efficient hardware for AES cryptography", 2007.

[Mantin, 01], I. Mantin, "Analysis of the Stream Cipher RC4", Master's Thesis, Weizmann Institute of Science, 2001.

[Mirza, 98], F. Mirza, "Block Ciphers and Cryptanalysis", Royal Holloway University of London, 1998.

[NVidia, 09],"NVidia CUDA Programming Guide 2.3", 2009.

[NVidia, 09a], "NVidia CUDA C Programming Best Practices Guide", 2009.

[OpenGL], "OpenGL", http://www.opengl.org.

[Pankratius, 09], Dr. V. Pankratius, "9. GPGPUs: Grafikkarten als Parallelrechner", Vorlesung Software Engineering für modern, parallele Plattformen, Fakultät für Informatik – Universität Karlsruhe, 2009.

[Preis*et al.*, 09], T. Preis, P. Virnau, W. Paul, J. Schneider, "Accelerated fluctuation analysis by Graphics cards and complex pattern formation in financial markets, New Journal of Physics 11, 2009.

[Rosenberg, 07], U. Rosenberg, "Using Graphic Processing Unit in Block Cipher Calculations", Master's Tesis, 2007.

[Schive*et al.*, 07], H. Schive, C. Chien, S. Wong, Y. Tsai, T. Chiueh, "Graphic-Card Cluster for Astrophysics (GraCCA) --- Perfomance Test)", 2007.

[Shannon, 49], C.E. Shannon, "Communication Theory of Secrecy Systems", Bell System Technical Journal, V. 28, n. 4, 1949.

[Tews *et al.*, 07] E. Tews, R. Weinmann y A. Pyshkin. "Breaking 104 bit wepin less than 60 seconds", in Sehun Kim, Moti Yung, and Hyung-Woo Lee, editors, WISA, volume 4867 of Lecture Notes in Computer Science, páginas 188-202.Springer, 2007.

[Wang y Yu, 05], X. Wang, H. Yu, "How to Break MD5 and Other Hash Functions", In *Proc. Advances in Crytology-- EUROCRYPT 2005, LNCS3494*, Cramer R (ed.), Springer-Verlag, pp. 19-35.

[Wheeler y Needham, 94], D. Wheeler, R. Needham, "TEA, a tiny encryption algorithm", In Fast Software Encryption – Proceedings of the 2nd International Workshop, 1008, 1994.