TECNOLÓGICO DE MONTERREY®

Campus Ciudad de México

Escuela de Diseño, Ingeniería y Arquitectura

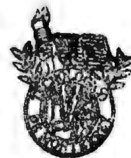
Maestría en Ciencias de la Computación

"Cifrado de datos en sistemas operativos utilizando el GPU como unidad de aceleración"

Autor:
Leonardo Antonio Naranjo Vega

Director de la tesis:
Dr. Moisés Alencastre-Miranda

Noviembre 2013



TECNOLÓGICO
DE MONTERREY

Biblioteca
Campus Ciudad de México

Contenido

Lista de Tablas IV

Lista de Figuras V

1. Introducción 1

1.1. Antecedentes 1

1.2. Definición del problema 2

1.3. Objetivo 3

1.4. Justificación 3

1.5. Hipótesis 3

1.6. Metodología 4

2. Marco Téorico 5

2.1. Elementos básicos de una computadora 5

2.2. Sistemas operativos 5

2.2.1. Como una máquina extendida (interfaz entre el usuario y el *hardware*) 6

2.2.2. Como un administrador de recursos 7

2.3. Unidades de procesamiento de gráficos (GPUs) 7

2.4. Introducción a CUDA 9

2.4.1. *Kernel* 10

2.4.2. Hilos, bloques y mallas 11
2.4.3. *Streaming Multiprocessors y Warps* 13
2.4.4. Jerarquía de memoria en el GPU 13
2.4.5. Flujo de ejecución (programación heterogénea) 15
2.5. Estado del arte 16

3. AES y su Implementación en CUDA **25**
3.1. Introducción a la criptografía 25
3.1.1. Algoritmos de llave asimétrica 26
3.1.2. Algoritmos de llave simétrica 27
3.1.2.1. Algoritmos de bloque 28
3.2. Descripción del algoritmo AES 28
3.2.1. Proceso de cifrado en AES 31
3.2.2. Operaciones utilizadas durante el proceso de cifrado 31
3.2.2.1. SubBytes 32
3.2.2.2. AddRoundKey 33
3.2.2.3. *Rijndael's Key Schedule* 33
3.2.2.4. ShiftRows 35
3.2.2.5. MixColumns 36
3.2.3. Proceso de descifrado en AES 37
3.2.4. Implementación básica del algoritmo AES en CUDA 38

4. Implementación **44**
4.1. El espacio del usuario y del núcleo en GNU/Linux 44
4.2. Modelo y mecanismos utilizados en la implementación 46
4.2.1. Ejecución indirecta de los *kernels* de cifrado y descifrado 47
4.2.2. Transferencia de datos entre núcleo del sistema operativo y el GPU . 48

4.2.3. Secuencia de ejecución en el cifrado y descifrado de datos 52

5. Resultados **56**

5.1. Gdev 56

5.2. *Crypto* API (GNU/Linux) 57

5.3. Descripción de las pruebas 57

6. Conclusiones y Trabajo a Futuro **63**

Bibliografía **65**

A. Vector Rcon **68**

B. Suma y Multiplicación en el Campo de *Galois* **70**

C. Derivación de la Subllave en AES **72**

D. Implementación de AES(ECB) en CUDA **74**

E. Instalación de Gdev **88**

Lista de Tablas

2.1. Jerarquía de memoria en un GPU. 14

3.1. Relación entre el tamaño de la llave y el número de rondas. 29

3.2. Relación entre llaves y subllaves. 34

3.3. Diferencias en las operaciones inversas. 38

5.1. Tiempos parciales considerados. 58

Lista de Figuras

2.1. Elementos básicos. 6

2.2. *Pipeline* de una Unidad de Procesamiento de Gráficos 8

2.3. Operaciones de punto flotante por segundo en CPU y GPU. 10

2.4. Definición de un *kernel* en CUDA. 11

2.5. Malla de un bloque de hilos. 12

2.6. Mapeo entre bloques de hilos y SMs. 14

2.7. Ejecución de un programa en CUDA. 16

2.8. Comparativa entre la implementación del algoritmo AES en CPU y GPU. . . 19

3.1. Criptografía de llave asimétrica. 26

3.2. Criptografía de llave simétrica. 27

3.3. Representación del *state*. 29

3.4. Representación gráfica de una llave de 128 *bits*. 30

3.5. Ejemplo de un *state*. 30

3.6. Ejemplo de una llave de 128 *bits*. 31

3.7. Proceso de cifrado en AES. 32

3.8. Matriz utilizada en la operación inversa de AddRoundKey. 38

3.9. Modo de operación ECB. 39

3.10. Implementación de la operación AddRoundKey en el GPU. 42

4.1. Arquitectura fundamental del sistema operativo GNU/Linux. 45

4.2. Modelo a implementar. 47

4.3. Región de memoria mapeada. 52

4.4. Secuencia de ejecución. 55

5.1. Cifrado (128 *bits*). 59

5.2. Descifrado (128 *bits*). 60

5.3. Cifrado (192 *bits*). 60

5.4. Descifrado (192 *bits*). 61

5.5. Cifrado (256 *bits*). 61

5.6. Descifrado (256 *bits*). 62

Capítulo 1

Introducción

1.1. Antecedentes

En la actualidad cada computadora que encontramos en el mercado posee una tarjeta gráfica (sin excepción alguna), que aunque no lo notemos estamos usando todo el tiempo debido a que su trabajo es procesar datos y enviarlos a nuestro monitor (sin importar de que tipo sea este). Debido a que el procesamiento de datos (imágenes) ha ido aumentando conforme a las necesidades de los usuarios (especialmente en el cómputo científico y los videojuegos), las tarjetas gráficas han pasado de ser simples *chips* integrados a las tarjetas madres de nuestra computadora a ser tarjetas totalmente dedicadas al procesamiento de imágenes. Dentro de dicha evolución, la arquitectura de la tarjeta gráfica ha cambiado y a la par han surgido tecnologías que permiten no solamente emplearla para procesar imágenes sino también datos numéricos en paralelo.

Dentro de la industria, una de las áreas en donde se ha aprovechado el buen rendimiento y la paralelización que las tarjetas de video o GPUs (*Graphics Processing Unit*) nos ofrecen, ha sido en la investigación científica pues el hecho de ejecutar aplicaciones en paralelo nos permite reducir en gran medida el tiempo de ejecución; sin embargo, los algoritmos de cómputo

intensivo también están presentes en otros campos, como es el caso de los sistemas operativos. Un sistema operativo de forma regular lleva a cabo un conjunto de tareas para administrar los recursos de una computadora, tales como: el control de la memoria física y el almacenamiento, el cambio de contexto de procesos, la selección de procesos a ejecutarse mediante algoritmos de planificación, la protección de la integridad tanto de los datos que residen localmente (ej. sistemas de archivos cifrados) como los datos que viajan a través de la red (ej. protocolos de cifrado de red), entre otros. Los procesos mencionados anteriormente, y otros más, pueden demandar un uso muy alto de CPU y si imaginamos que varios de ellos se ejecutan de forma concurrente entonces su tiempo de ejecución será prolongado.

El presente trabajo surge de la curiosidad de conocer más a fondo el núcleo de los sistemas operativos derivados de UNIX (en especial de GNU/Linux) y saber cómo es que estos pueden hacer uso del cómputo heterogéneo (combinación de CPU y GPU), que ha tenido un gran auge en años recientes, para reducir el tiempo de ejecución de algunas de sus tareas.

1.2. Definición del problema

Como se mencionó en la introducción, la investigación científica es una de las áreas que ha aprovechado en gran medida el surgimiento del cómputo heterogéneo, sin embargo no es ajeno pensar que el núcleo de algún sistema operativo moderno pueda hacer uso de este para acelerar sus tareas y por lo tanto así disminuir su tiempo de ejecución. El principal problema que surge al querer hacer uso del cómputo heterogéneo, en el caso anteriormente mencionado, es que las bibliotecas que se emplean para tal fin han sido diseñadas para utilizarse en el espacio donde se ejecutan las aplicaciones de los usuarios y en los sistemas operativos modernos, sobre todo con núcleo monolítico, hay un límite perfectamente definido entre este espacio y el núcleo del sistema operativo.

La presente tesis pretende proponer una solución al problema descrito en el párrafo ante-

rior y con esto permitir que el núcleo del sistema operativo pueda hacer uso (tal vez de forma indirecta) de alguna biblioteca de cómputo heterogéneo para acelerar el proceso de cifrado con el algoritmo AES.

1.3. Objetivo

El objetivo principal de este trabajo de investigación es implementar un modelo que le permita al núcleo del sistema operativo GNU/Linux tanto cifrar como descifrar un conjunto de datos a través de la utilización de la biblioteca de cómputo heterogéneo de CUDA.

1.4. Justificación

Estudios recientes, llevados a cabo en diferentes áreas en investigación científica, han demostrado que al emplear el GPU como una unidad de aceleración se puede reducir el tiempo de ejecución en aplicaciones que tienen un alto consumo de CPU y que además debido a su naturaleza son paralelizables; los sistemas operativos pueden verse beneficiados de tal aceleración debido a que distintos procesos dentro su núcleo presentan estas dos características, desafortunadamente las bibliotecas que han sido creadas para emplear el GPU como unidad de aceleración, como CUDA u OpenCL, no pueden ser utilizadas directamente por el núcleo y por lo tanto es necesario emplear otros mecanismos que permitan hacerlo.

1.5. Hipótesis

De acuerdo a lo explicado con anterioridad, este trabajo de investigación pretende saber qué mecanismos permitirían al núcleo del sistema operativo GNU/Linux utilizar (de forma directa o indirecta) la biblioteca de CUDA para acelerar el proceso del cifrado y descifrado utilizando el algoritmo AES.

1.6. Metodología

Para poder alcanzar la hipótesis planteada en la subsección anterior es necesario inicialmente establecer un modelo a ser implementado, una vez que han sido reconocidos los componentes que estarán presentes en dicho modelo será necesario estudiar a fondo mecanismos propios del sistema operativo GNU/Linux que permitan llevar a cabo una implementación. A continuación, para determinar si la implementación hecha es útil para el propósito de este estudio, será necesario implementar el algoritmo de cifrado AES en la arquitectura CUDA e integrarlo con la implementación del modelo establecido inicialmente. Finalmente, será necesario hacer una comparación de rendimiento con otros mecanismos que también permitan al núcleo de GNU/Linux cifrar datos con AES, todo esto para saber si nuestro modelo proporciona alguna mejora en cuanto a tiempo de ejecución o no.

Este trabajo está organizado de la siguiente forma, el siguiente capítulo proporciona una introducción a conceptos relacionados con sistemas operativos, GPUs y la arquitectura CUDA de NVIDIA; enseguida se presenta el estado del arte de los trabajos relacionados con aceleración de tareas propias de sistemas operativos a través del paralelismo en CPUs y GPUs. El capítulo 3 presenta una descripción más o menos detallada del proceso de cifrado y descifrado en el algoritmo AES y también explica cómo hacer una implementación básica de la versión ECB (*Electronic Codebook*) en la arquitectura de CUDA. En el capítulo 4 se describe de forma general el modelo a ser implementado y también los problemas y mecanismos encontrados para llevar a cabo una implementación correcta. Finalmente, en el capítulo 5 se presentan los tiempos obtenidos en el proceso de cifrado y descifrado con AES (ECB) utilizando el *Crypto* API de GNU/Linux, Gdev y la implementación descrita en el capítulo 4.

Capítulo 2

Marco Téorico

2.1. Elementos básicos de una computadora

Desde un punto de vista general, una computadora consta de un procesador, una memoria donde se almacenan los datos a ser operados y componentes de entrada/salida, todos ellos interconectados a través de un *bus* que les permite comunicarse para llevar a cabo la ejecución de un programa. La figura 2.1 muestra de manera general la interconexión de estos componentes siendo *Northbridge* y *Southbridge* parte del *bus* mencionado anteriormente y AGP (*Accelerated Graphics Port*), PCI, USB, ISA, IDE, etc., parte de los dispositivos de entrada/salida.

2.2. Sistemas operativos

El sistema operativo es el encargado de controlar los recursos de *hardware* en una computadora y su comportamiento puede ser pensado de las siguientes dos formas [16]:

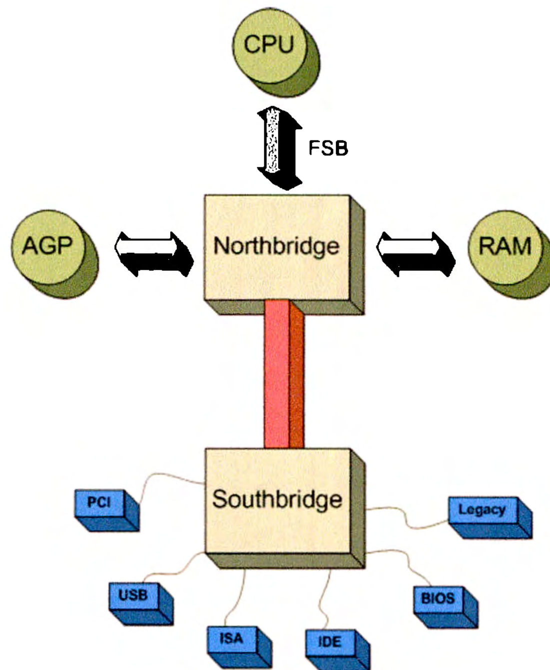


Figura 2.1: Elementos básicos.

Fuente: http://en.wikipedia.org/wiki/Southbridge_(computing)

2.2.1. Como una máquina extendida (interfaz entre el usuario y el *hardware*)

Uno de los objetivos del sistema operativo es presentar un conjunto de abstracciones (como las llamadas a sistemas o bibliotecas) que le permitan al usuario de una computadora utilizar los componentes de *hardware* sin conocer su funcionamiento preciso. Como ejemplo, supongamos que un usuario necesita utilizar la tarjeta gráfica para hacer despliegue en 3D, este no necesita conocer internamente y precisamente cómo funciona su tarjeta de video, sin embargo será capaz de realizar su tarea pues solamente es necesario que utilice alguna biblioteca de gráficos como OpenGL.

2.2.2. Como un administrador de recursos

Los sistemas operativos modernos son capaces de realizar multiprocesamiento, es decir, son capaces de ejecutar varias tareas al mismo tiempo y esto es debido a que los procesadores han evolucionado. Cuando hablamos de multiprocesamiento en sistemas operativos surge un inconveniente y este es la manera en como los recursos serán usados cuando existe concurrencia. Supongamos que dos procesos necesitan almacenar datos en el disco duro, para realizar dicha operación el sistema operativo debe de proveer un mecanismo que permita mantener la integridad de estos, es decir, dos procesos no podrían utilizar el disco duro al mismo tiempo debido a que podría darse el caso de que ambos estuvieran leyendo y escribiendo en el mismo espacio de almacenamiento. La tarea entonces del sistema operativo es administrar los recursos para evitar que sean usados de forma indebida.

2.3. Unidades de procesamiento de gráficos (GPUs)

Dentro de los recursos que el sistema operativo se encarga de administrar se encuentran las Unidades de Procesamiento de Gráficos (también conocidas como tarjetas de video o GPUs), todas las computadoras actuales poseen una unidad de este tipo. Esta sección pretende dar una pequeña introducción acerca del funcionamiento de este componente de *hardware*.

La tarea de cualquier tarjeta gráfica es la generación o síntesis de una imagen de acuerdo a la descripción de una escena [9], dicha escena contiene un conjunto de primitivas (esferas, conos, cubos, etc.) y una descripción de la iluminación (como cada objeto refleja la luz) de acuerdo a la posición y orientación del que ve la imagen. El proceso de síntesis de la imagen puede ser visto como un *pipeline* con varias etapas, el propósito de esta tesis no es explicar el funcionamiento de dicho *pipeline* pero si recalcar que a través de dicho conjunto de primitivas y la descripción de la escena es posible generar una imagen de forma paralela, es decir, el proceso actúa sobre cada píxel en la imagen al mismo tiempo. Como podemos ver en la

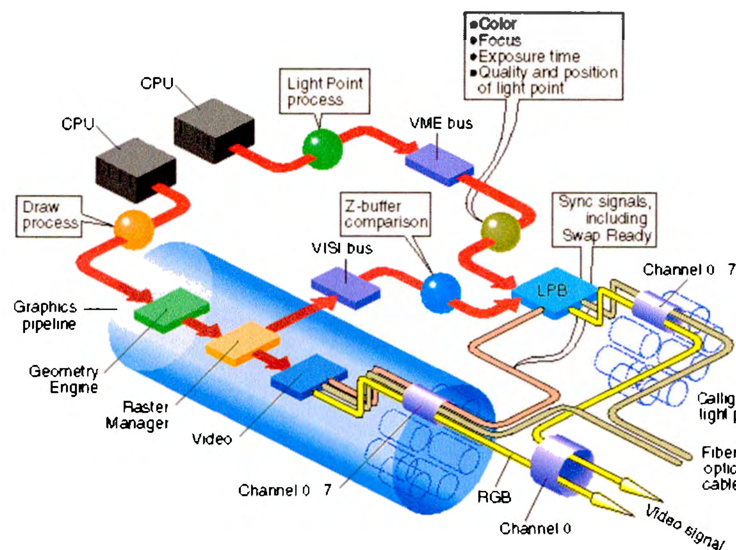


Figura 2.2: *Pipeline* de una Unidad de Procesamiento de Gráficos

Fuente: OpenGL *Performer Programmer's Guide.*

figura 2.2, todo comienza cuando el CPU ordena que se dibuje una imagen y a través de varias etapas en el *pipeline* la imagen se transforma para finalmente producir una señal de video que se puede observar en un monitor.

Cabe mencionar que el GPU es la segunda unidad de procesamiento que más lleva a cabo trabajo en una computadora; también es importante mencionar que mientras el CPU tiene como entrada datos numéricos y produce una salida numérica, el GPU tiene como entrada un conjunto de píxeles (primitivas y descripción de la escena) y produce un conjunto de píxeles ya transformados. Sin embargo, con la evolución del GPU ha sido posible utilizar el *pipeline* de la tarjeta gráfica para procesar datos numéricos, esto inicialmente fue conocido como *General Purpose Graphics Processing Unit Programming* (GPGPU por su siglas en inglés) o Programación de Tarjetas de Video para Aplicaciones de Propósito General en español.

2.4. Introducción a CUDA

Los productores de procesadores, como AMD e Intel, han tenido que buscar alternativas para incrementar el poder de cómputo en los procesadores debido a que elevar la frecuencia de reloj no es más una solución por la disipación que se tiene que hacer del calor. La solución que encontraron los productores fue aumentar el número de núcleos que un procesador contiene, aunque realmente una solución similar se había venido utilizando en las supercomputadoras desde años atrás. El hecho de que un procesador contenga múltiples núcleos permite que varias aplicaciones se puedan ejecutar paralelamente o que la misma aplicación pueda explotar el paralelismo para disminuir su tiempo de ejecución. Los GPUs no son ajenos al tema del paralelismo pues desde su concepción se tuvo en mente que pudieran procesar grandes cantidades de datos de forma masiva y paralela, un GPU normalmente procesa un conjunto de polígonos y les aplica ciertas operaciones como texturizado, sombreado e iluminación y por último se muestran en los monitores de las computadoras. Los grandes beneficiados del cómputo paralelo son en su mayoría las aplicaciones desarrolladas en cómputo científico debido a que realizan muchos cálculos y son en su mayoría paralelizables. La figura 2.3 muestra de manera comparativa cual ha sido la evolución de los CPUs (Intel) y los GPUs (NVIDIA) desde el año 2001 hasta el año 2013, en cuanto a la cantidad de operaciones de punto flotante que pueden realizar por segundo.

Como se puede ver en la figura 2.3, incluso para las operaciones de doble precisión, los GPUs presentan un mejor desempeño con respecto a los CPUs, sobre todo a partir del año 2003 que fue cuando surgió el término de GPGPU. Sin embargo, a pesar de que el modelo GPGPU demostró en un inicio que podía mejorar el desempeño para ciertas aplicaciones tenía ciertos inconvenientes debido a que era necesario que el programador expresara el problema a resolver por su aplicación en términos de programación para gráficos. Fue así que en el año 2006 la empresa NVIDIA introdujo la arquitectura CUDA (*Compute Unified Device Architecture*), que es definida como una plataforma de cómputo paralelo (*hardware*) y también

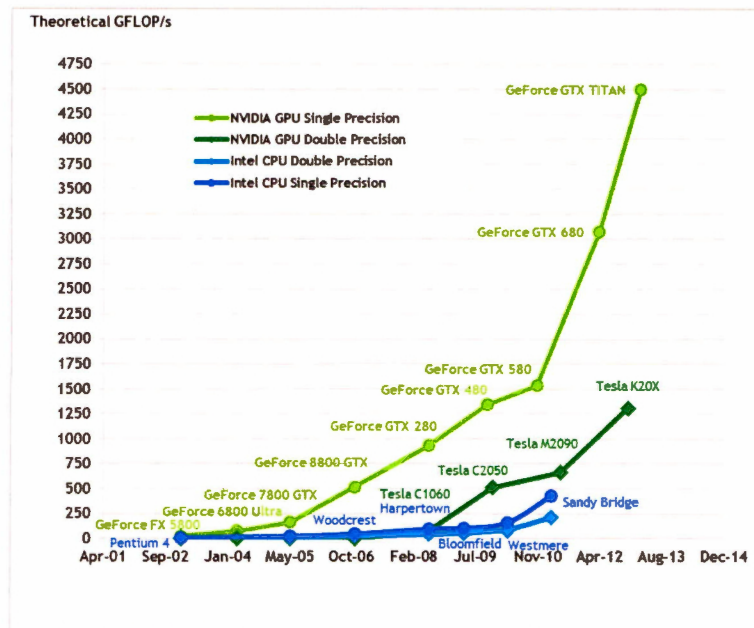


Figura 2.3: Operaciones de punto flotante por segundo en CPU y GPU.

Fuente: [10]

un modelo de programación (*software*) que permite resolver de una manera más eficiente (en comparación con un CPU) problemas computacionales complejos [10].

Como se mencionó anteriormente, CUDA es una arquitectura compuesta por *software* y *hardware*, la última se refiere obviamente al GPU, la primera se refiere al código desarrollado (o extensiones) para que las aplicaciones escritas en distintos lenguajes de programación (como C, C++, Fortran, Python, entre otros) puedan hacer uso del GPU como una unidad de procesamiento. Existen conceptos básicos y no tan básicos que deben ser explicados para entender mejor cómo es que funciona el modelo de programación propuesto por CUDA.

2.4.1. *Kernel*

CUDA propone extensiones en diferentes lenguajes de programación para permitir al programador definir "funciones", llamadas *kernels*, que cuando son llamados se ejecutan N veces en forma concurrente por N número de hilos (o CUDA *threads*) y de forma paralela, varios

Standard C Code

```c
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;

// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

C with CUDA extensions

```c
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(x, d_x, N, cudaMemcpyHostToDevice);
cudaMemcpy(y, d_y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, x, y);

cudaMemcpy(d_y, y, N, cudaMemcpyDeviceToHost);
```

Figura 2.4: Definición de un *kernel* en CUDA.

Fuente: http://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/

de ellos al mismo tiempo, en los diferentes *cores* del GPU. La figura 2.4 muestra cómo son utilizadas las extensiones de CUDA en el lenguaje de programación C, en ella podemos distinguir la diferencia entre un código C estándar (del lado izquierdo) y la forma de utilizar las extensiones (del lado derecho) para definir un *kernel*; la palabra reservada __global__ es utilizada para tal fin.

2.4.2. Hilos, bloques y mallas

El concepto de hilo dentro de la arquitectura de CUDA está íntimamente relacionado con el mismo concepto en la programación en CPU, debido a que a un hilo de CUDA también le son asignados recursos para que se pueda ejecutar y porque existen varias instancias de estos ejecutándose concurrentemente. Para su organización y ejecución en el GPU, un conjunto de hilos de CUDA es agrupado en bloques (o *Thread blocks*) y los bloques a su vez en mallas (*Grids*) bidimensionales y tridimensionales. Cada hilo posee un identificador único dentro del bloque y la malla a la que pertenece (la figura 2.5 muestra un ejemplo de dicha organización), dichos identificadores pueden ser obtenidos a través de las variables predefinidas threadIdx.x, threadIdx.y (identificadores dentro de un bloque), blockIdx.x y blockIdx.y (identificadores

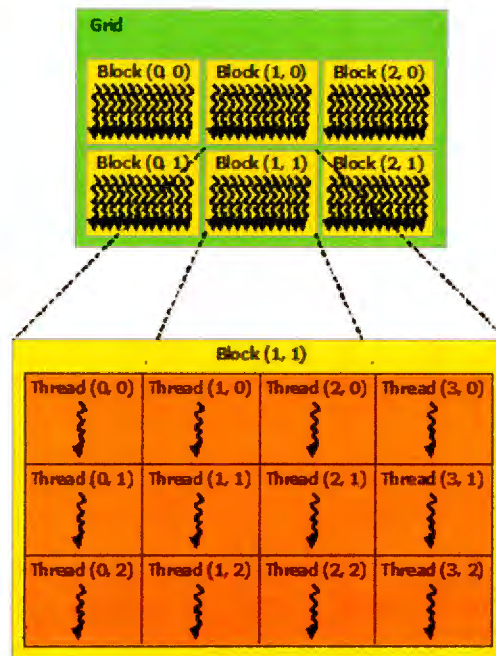


Figura 2.5: Malla de un bloque de hilos.

Fuente: [10]

dentro de una malla) en CUDA. Dentro del bloque los hilos pueden cooperar a través de mecanismos de comunicación interproceso como memoria compartida y sincronización por barreras.

Al momento de enviar una malla a ejecución cada hilo dentro de un bloque ejecutará una instancia de un *kernel*, lo que permitirá que cada uno de estos hilos trabaje en una pequeña parte del problema a resolver. Una de las ventajas de agrupar los hilos en bloques y estos a su vez en mallas es que el problema atacado por la aplicación paralela puede ser dividido en subproblemas y cada subproblema dividido en partes más pequeñas que pueden resolverse de manera cooperativa o independiente por el total de hilos que se especificaron antes de lanzar el *kernel* de CUDA. El modelo anteriormente descrito, permite hacer un escalamiento de la aplicación de manera casi automática pues si la cantidad de datos de entrada crece, el número de hilos puede ser incrementado hasta una cantidad considerablemente grande; por ejemplo, para el caso de una tarjeta NVIDIA GTX 555 el número máximo de hilos que esta

puede ejecutar concurrentemente es de 9216.

2.4.3. *Streaming Multiprocessors y Warps*

Para su ejecución, el conjunto de bloques definidos en una malla son enumerados y enviados a cada una de las unidades de ejecución dentro del GPU, llamadas *Streaming Multiprocessors* (SMs). A nivel de *hardware* cada SM contiene un cierto número de *cores* (múltiplo de 8) y se encarga de crear, manejar, calendarizar y ejecutar grupos de treinta y dos hilos llamados *warps*; un *warp* es la unidad básica de ejecución dentro un SM. Conceptualmente podríamos decir que este grupo de treinta y dos hilos es ejecutado paralelamente por cada una de los SMs en el GPU, cada uno de los hilos que compone al *warp* comienza en la misma instrucción y sigue en conjunto con los demás una misma trayectoria de ejecución dentro del conjunto de instrucciones definidas en el *kernel*, a esto se le denomina *Single Instruction Multiple Thread* en la arquitectura de CUDA; sin embargo, cada hilo es independiente debido a que cada uno posee su contador de programa y sus propios registros.

La figura 2.6 muestra de manera conceptual la forma en cómo cada uno de los bloques y mallas definidos en el contexto de CUDA son mapeados a cada uno de los *Streaming Multiprocessors* dentro del GPU.

2.4.4. Jerarquía de memoria en el GPU

Cada hilo definido en el contexto de CUDA puede acceder cinco espacios de memoria diferentes durante su ejecución como se muestra en la tabla 2.1, es muy importante recalcar la diferencia entre cada uno de estos espacios porque el rendimiento de la aplicación desarrollada en CUDA puede verse afectado dependiendo del espacio que el hilo acceda para obtener datos. El espacio más rápido en cuanto a velocidad de acceso son los registros que cada uno de los hilos posee, enseguida se encuentra la memoria compartida por los hilos dentro de un bloque (utilizada para trabajar de manera conjunta) y finalmente la memoria de textura, la

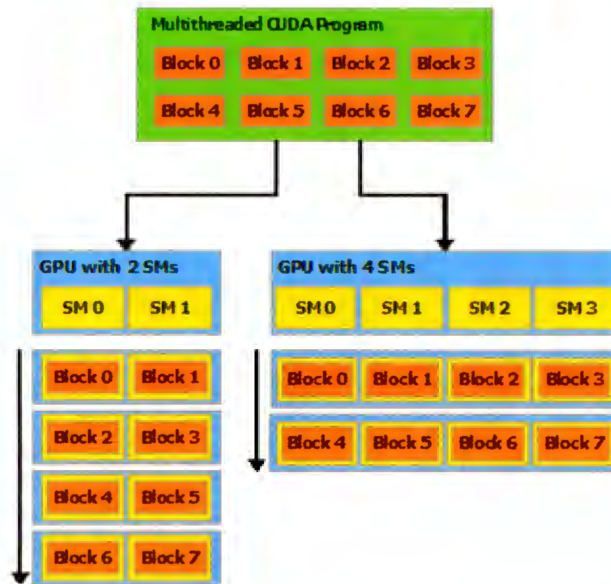


Figura 2.6: Mapeo entre bloques de hilos y SMs.

Fuente: [10]

Tipo de almacenamiento	Ancho de banda	Latencia (ciclos)
Registros	~8 TB/s	1 ciclo
Memoria Compartida	~1.5 TB/s	1 a 32
Memoria de Textura	~200 MB/s	~400 a 600
Memoria Constante	~200 MB/s	~400 a 600
Memoria Global	~200 MB/s	~400 a 600

Tabla 2.1: Jerarquía de memoria en un GPU.

Fuente: [3]

memoria constante y la memoria global con la que cuenta el GPU.

En la tabla 2.1, la latencia esta expresada en el número de ciclos de reloj que le toma traer a cada SM los datos desde los distintos tipos de memoria de almacenamiento.

2.4.5. Flujo de ejecución (programación heterogénea)

El modelo de programación de CUDA asume que existen dos unidades de procesamiento en los que el código de una aplicación puede ser ejecutado, por un lado se define el *host* (CPU) y por otro el *device* (GPU). La distinción entre cada uno de estos códigos se hace utilizando palabras reservadas dentro del lenguaje de programación utilizado y se define en el momento de compilación de la aplicación, por lo tanto el código destinado a ejecutarse en el *host* no puede ejecutarse en el *device* y viceversa. La distinción entre los dos tipos de códigos se le conoce como programación heterogénea y permite que el GPU pueda y deba ser considerado como una unidad de coprocesamiento (obviamente adicional al CPU) con su propia jerarquía de memoria y unidades de procesamiento.

Si alguna o varias partes de la aplicación necesitan hacer uso del GPU para ejecutar código, normalmente deberán seguir los siguientes pasos:

1. Inicializar el *device* (este punto solamente es necesario realizarlo una vez).

2. Transferir los datos a ser procesados por cada hilo de la memoria del *host* (memoria RAM) a la memoria del *device* (memoria de la tarjeta de video), utilizando DMA (*Direct Memory Access*).

3. Ejecutar en el GPU el *kernel* de CUDA definido.

4. Copiar los resultados de la memoria del *device* hacia memoria del *host*, nuevamente utilizando DMA.

5. Liberar recursos solicitados.

La figura 2.7 muestra gráficamente un ejemplo de cómo sería la secuencia de ejecución en una aplicación que utiliza el modelo de programación heterogénea de CUDA. El ejemplo (en este particular caso) está divido en cuatro partes, las partes marcadas como *"Serial code"* es

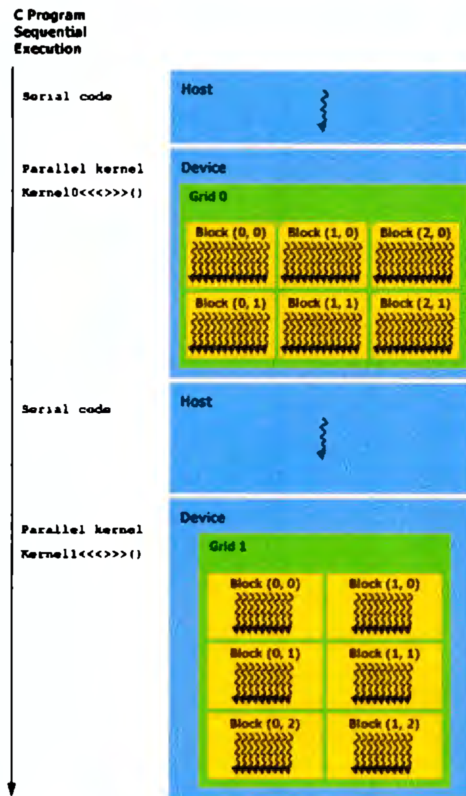


Figura 2.7: Ejecución de un programa en CUDA.

Fuente: [10]

el código a ejecutarse en el *host* o CPU, las partes marcadas como *"Parallel Kernel"* es el código a ejecutarse en el *device* o GPU.

2.5. Estado del arte

En años recientes el número de transistores y la velocidad de reloj en los circuitos integrados ha crecido en gran medida, sin embargo la velocidad de reloj ha llegado a un límite debido a la disipación que se tiene que hacer de calor. Las predicciones en el 2005 consideraron que para el 2008 la velocidad de reloj de los procesadores excedería los 10 GHz y llegaría a un tope de 15 GHz para el 2010. Con tales frecuencias es casi imposible disipar el calor y es

por eso que los arquitectos de *hardware* necesitaban resolver este problema urgentemente [1]. Una de las soluciones al problema de la disipación del calor es la tecnología multinúcleo, en la cual múltiples núcleos lógicos son adicionados a un sólo circuito integrado y cada uno de ellos es considerado como un procesador independiente. A lo anterior, en cómputo se le denomina Multiprocesamiento Simétrico (*Symmetric MultiProcessing*, en inglés, SMP) y no es más que el hecho de que dos o más procesadores idénticos comparten una memoria principal y pueden ejecutar instrucciones en paralelo.

Con el surgimiento de la tecnología multinúcleo los sistemas operativos han tenido que hacer adaptaciones en su arquitectura para poder aprovechar la capacidad de cómputo que los procesadores actuales ofrecen, el principal objetivo de utilizar una arquitectura de multiprocesamiento simétrico es el hecho de que las aplicaciones puedan escalar conforme el número de núcleos aumenta, sin embargo esto no siempre es así. El estudio que llevó a cabo Yuan Qingbo [12], el cual comprende los resultados obtenidos de un conjunto de pruebas, reconoce que hay ciertos problemas que afectan el rendimiento de aplicaciones cuando se aumenta el número de núcleos en las plataformas SMP. Dos de los problemas mencionados son que el ancho de banda de la memoria se convierte en un cuello de botella debido a que los diferentes procesadores necesitan traer datos de la memoria principal, y también que los sistemas operativos actuales, que hacen uso de SMP, deben de implementar una serie de estructuras para la contención de recursos compartidos.

Otro estudio donde se puede corroborar que el aumento de núcleos lógicos no es una solución directa para el tema de escalamiento de aplicaciones es el que llevó a cabo Raffaele Bolla [2], su estudio consiste en implementar un modelo de *router* en GNU/Linux que permite llevar a cabo el procesamiento en paralelo de los paquetes que son tanto transmitidos como recibidos por el sistema operativo, para que realmente el procesamiento fuera en paralelo el autor tuvo que modificar el *software* que hace esta tarea para aumentar la capacidad computacional y también permitir hacer un escalamiento cuando la carga de trabajo es

demasiado alta. La carga de trabajo es balanceada en todos los procesadores utilizando un término denominado afinidad, el cual establece que es necesario asociar las tareas con un procesador en particular para tener una eficiencia en lo que se refiere a *cache*, es decir, las tareas no son migradas de procesador a procesador pues esto introduce latencia que finalmente afecta el tiempo total de ejecución de la tarea. De acuerdo al estudio, para hacer un uso óptimo de la arquitectura SMP es necesario explotar el concepto de afinidad, pues con esto se puede asegurar que un solo procesador se encargará de realizar las operaciones pertinentes que involucran enviar o recibir un paquete; también es importante que haya una buena distribución de trabajo entre los procesadores que están marcados como activos y por último eliminar las estructuras de contención que proporciona el sistema operativo. Para corroborar la hipótesis planteada, el autor realizó dos tipos de pruebas: una con la versión estándar de SMP en GNU/Linux y otra con una versión modificada teniendo en cuenta los conceptos de afinidad, distribución de cargas y las estructuras de contención. Los resultados demuestran que existe una brecha bastante amplia entre el primer tipo de pruebas y el segundo, pues como se ha comentado el sistema operativo introduce latencia en el tiempo total de ejecución.

Otra posible solución al problema del escalamiento en los sistemas operativos es el llamado cómputo heterogéneo, es decir, la combinación entre unidades de procesamiento de diferentes tipos. En este caso, para los fines de esta tesis nos concentraremos en la combinación entre CPU y GPU que ha tenido un gran auge debido a dos cosas, la primera es el surgimiento de tecnologías que permiten realizar cómputo de propósito general en GPUs y también al crecimiento exponencial en el número de núcleos de ejecución presentes en los mismos. A pesar de que las plataformas heterogéneas han recibido poca atención en la investigación de sistemas operativos, Weibin y Robert Ricci [14] han demostrado que el *performance* de los algoritmos de cifrado puede ser incrementado hasta 6 veces (los resultados fueron obtenidos con una tarjeta de video NVIDIA GTX 480 y un procesador Intel *Core* i7) cuando una

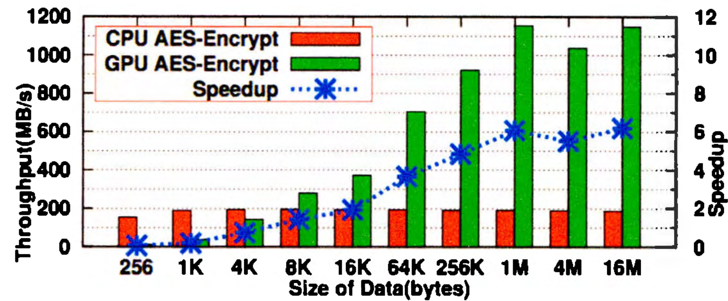


Figura 2.8: Comparativa entre la implementación del algoritmo AES en CPU y GPU.

Fuente: [14]

comparativa es hecha entre la implementación del algoritmo AES en CPU y GPU. Los autores anteriormente mencionados han propuesto un marco de trabajo llamado kgpu en GNU/Linux, el cual ofrece servicios (como algoritmos de cifrado, uno de ellos basados en CUDA AES[1]) para acelerar ciertas tareas que llevan a cabo los sistemas operativos y que tienen un alto consumo de CPU. En la figura 2.8 es posible ver los resultados que ellos obtuvieron, en dichos resultados las diferencias de rendimiento entre la implementación en CPU y GPU se pueden ver hasta que la cantidad de datos a cifrar es superior a los 8 *Kilobytes.*

Otro trabajo referente a cómputo heterogéneo es GPUStore [15], el cual es una implementación que propone la aceleración de servicios en GPU relacionados con dispositivos de almacenamiento, en especial para las capas de cifrado (dm-crypt y ecryptfs) y también para el *software* que se encarga de gestionar las operaciones de almacenamiento en arreglos de discos (RAID). Ambas operaciones, cifrado y operaciones de paridad, son tareas que demandan un alto grado de computación y también son perfectas para la paralelización debido a que trabajan en bloques independientes de datos. Algo que es importante mencionar de este trabajo es que los autores decidieron no modificar ningún subsistema del núcleo de GNU/-Linux y solamente abstrajeron un conjunto de servicios básicos llamados también GPUStore (como manejo de memoria, creación de *buffers*, servicios de cifrado, etc.) a un nivel arriba del

[1]OpenSSL CUDA AES Engine. http//code.google.com/p/engine-cuda.

marco de trabajo de CUDA, estos servicios son utilizados por los subsistemas de GNU/Linux a través de una aplicación *"helper"* que se encuentra en el espacio del usuario del sistema operativo. Los resultados de las pruebas realizadas son bastante buenos para la parte del algoritmo de cifrado en AES, debido a que de acuerdo a los autores, con algunas optimizaciones (de asignación de memoria, manejo de *buffers* y en los dispositivos de almacenamiento) el sistema es capaz de tener un *throughput* de casi 4GB/s, es decir, treinta y seis veces más que la implementación hecha en CPU. Para el caso de dm-crypt, la versión para CPU es 60 % más lenta y en el caso de recuperación de RAID el rendimiento se ve aumentado hasta 6 veces conforme el tamaño de bloque se incrementa. Finalmente, es importante notar que todas estas pruebas fueron realizadas en dos equipos con procesador Intel *Core* i7 de cuatro núcleos de procesamiento, 6GB de memoria RAM, discos duros de estado sólido de 32GB y una tarjeta NVIDIA GTX 480.

El cómputo heterogéneo también ha sido aplicado a otras partes del sistema operativo GNU/Linux, en el estudio de Sangjin Han [6] se utilizó para el procesamiento de paquetes de red. Las pruebas que se realizaron, tanto para CPU como para CPU+GPU, consistieron en el reenvío de paquetes en IPv4, IPv6 y también para el protocolo IPsec en una estación de trabajo con dos procesadores Intel Nehalem *Quad-Core Xeon* X5550 de 2.6 GHz con 12 GB de memoria RAM y dos tarjetas NVIDIA GTX480 con 1.5 GB de memoria DDR5. De acuerdo a las resultados, el rendimiento en la implementación de CPU+GPU, para el caso del reenvío de paquetes en IPv4 y en IPv6, fue un poco menor a la implementación hecha solamente para el CPU; sin embargo el autor refiere que es debido a un problema de *hardware*. En cambio, para la parte de IPsec se menciona que la implementación de CPU+GPU es 3.5 veces mayor a la implementación hecha solamente en el CPU y presenta un mayor rendimiento cuando el tamaño de los paquetes se incrementa. Es importante mencionar que para alcanzar un alto desempeño los autores desarrollaron un modelo altamente optimizado para enviar y recibir paquetes que permite disminuir en gran medida la latencia causada por el manejo de estos en

memoria, pues de acuerdo al autor este manejo representa un gran porcentaje en el tiempo de procesamiento de paquetes en *routers* actuales.

Un trabajo que pretende abstraer los GPUs como dispositivos de cómputo en sistemas operativos es PTask [13], el cual puede ser definido como una serie de abstracciones en los que GPUs y otros dispositivos de aceleramiento son utilizados como recursos de computación. La motivación principal de este trabajo es que los GPUs son completamente tratados como dispositivos de entrada/salida por el controlador y no como unidades de procesamiento (como lo son los CPUs), lo que da como resultado que el sistema operativo no sea capaz de garantizar la igualdad en las tareas ejecutadas en el GPU. PTask consiste en una serie de interfaces y un entorno de ejecución construido teniendo como base CUDA, dichas interfaces y entornos de ejecución incluyen elementos como funciones para la creación de *buffers*, copiado de datos entre memoria RAM y memoria del GPU y una infraestructura para tratar cada uno de los *kernels* de CUDA como vértices en un grafo acíclico dirigido, es decir, se crean cadenas de ejecución en donde las salidas de un *kernel* de CUDA son conectadas a las entradas del siguiente *kernel*. El modelo propuesto en PTask fue probado principalmente en un sistema de reconocimiento de gestos, el cual se utiliza para reconocer el movimiento de manos, es decir, hay un sistema de captura de imágenes (catusb), después se hace el análisis de dichas imágenes de forma paralela (como transformaciones geométricas -xform- y filtros de ruido –filter-) y por último el sistema operativo recibe toda esa información para mover un dispositivo de señalamiento, las tareas paralelizables se encuentras conectadas para que la salida de una sea la entrada de la otra. La implementación del sistema de reconocimiento de gestos fue hecho en cinco modos diferentes, sin embargo, las pruebas que interesan al presente estudio son *host-based* (completamente en CPU) y ptask (en el que xform y filter fueron implementadas en el GPU). De acuerdo al autor, la implementación ptask muestra mucho mejor rendimiento que la implementación hecha en el CPU, pues su tasa de rendimiento es de 53.8 MB/s a diferencia de la implementación en GPU que es de 35 MB/s. Las pruebas realizadas en este caso fueron

llevadas a cabo en una procesador Intel *Core* i5 de 3.2 GHz y una tarjeta NVIDIA GTX580.

Son varios los estudios que se han realizado con respecto al cómputo heterogéneo en sistemas operativos usando GPUs como coprocesadores, sin embargo es importante mencionar el trabajo realizado en *"A GPU accelerated storage system"* [5], pues en él existe una sección dedicada a los retos que deben enfrentarse cuando se diseña una aplicación heterogénea y aunque dichos retos son mencionados por otros autores, en este trabajo son especificados puntualmente. Los retos se mencionan a continuación:

- Minimizar el volumen de código generado para evitar que el rendimiento se vea afectado.
- Separación de la lógica en la aplicación, es decir, deberá ser casi inmediato la adición de nuevas funcionalidades.
- Disminuir latencia en la transferencia de datos entre el *host* y el *device*.
- Utilizar, si es posible, el método de asignación de memoria no paginable pues esto ayuda a disminuir la latencia creada al transferir los datos utilizando DMA.
- Disminuir la latencia producida al asignar memoria en el *device*.
- Cuando se tienen más de dos GPUs, balancear las cargas entre ellos.
- Usar de manera eficiente la memoria compartida.

El trabajo *"A GPU accelerated storage system"* se refiere en si a un sistema de almacenamiento distribuido que implementa funciones *hash*, en el que el trabajo de la función de digestión es cedido al GPU para comprobar la integridad de los archivos almacenados. Una de las cosas a resaltar en este trabajo es que es el único que se ha encontrado hasta el momento que considera el uso de varios GPUs; diferentes GPUs son utilizados para calcular el valor *hash* de los bloques en los que son divididos los archivos y en base a eso se almacenan en algún medio (a esto se le conoce como *content-based storage*).

Por último, es de suma importancia mencionar el trabajo realizado en *"Gdev: First-Class GPU Resource Management in the Operating System"* [7], pues es el trabajo más aproximado a lo que se refiere a la utilización de las tarjetas gráficas como una unidad más de procesamiento para el sistema operativo. Dicho trabajo argumenta que la mayoría de las implementaciones que se han hecho para acelerar tareas en el GPU dependen en gran medida de lo que ofrecen ciertos marcos de trabajo como CUDA en el espacio de usuario (y no para las tareas propias del sistema operativo). Gdev no necesariamente aporta una mejora en el rendimiento en relación con CUDA, pero si lo hace en lo que se refiere al control que el sistema operativo tiene sobre el GPU cuando se necesita hacer multiprocesamiento. En el modelo que proponen los autores son cuatro cosas sobre las que Gdev le proporciona control al sistema operativo con respecto a la tarjeta gráfica, estas son:

- Manejo de memoria: Para la optimización en la transferencia de datos entre el *host* y el *device*.
- Compartición de memoria: Para proveer un mecanismo que permita la comunicación interproceso.
- Área de intercambio de datos: Para mejorar la administración de la memoria global de la tarjeta gráfica.
- Calendarización de tareas: Para poder asignar correctamente los tiempos de computación y los tiempos de transmisión de datos.
- Virtualización de la tarjeta gráfica: Para poder fomentar el multiprocesamiento.

Gdev, de forma más específica, es un módulo dentro del núcleo del sistema operativo que proporciona una interface para controlar los puntos mencionados anteriormente, otros marcos de trabajo pueden hacer uso de esta interface y como ejemplo se encuentra la implementación parcial de CUDA que los autores llevaron a cabo dentro del núcleo de GNU/Linux. Son varias

las pruebas de rendimiento que el autor realizó con una tarjeta NVIDIA GeForce GTX 480 y un procesador Intel *Core* 2 *Extreme* QX9650, sin embargo para los fines de este estudio solo se mencionarán dos que fueron implementadas en tres contextos diferentes: el primer contexto fue utilizando el controlador propietario de NVIDIA para GPU, el segundo fue utilizando Gdev en el espacio del usuario y el último utilizando Gdev en el espacio del núcleo del sistema operativo. La primer prueba se refiere a la transferencia de datos entre la memoria del GPU y la memoria principal (RAM), los resultados muestran que la implementación hecha dentro del núcleo es la más lenta de las tres (hasta 50 % más lenta) pues existe una latencia producida por la función memcpy dentro del núcleo de GNU/Linux; la segunda prueba se refiere a un conjunto de *benchmarks* para medir la carga de trabajo en aplicaciones realizadas en CUDA, para este caso se pudo concluir que en ocasiones el rendimiento se ve afectado hasta en un 20 % en la implementación hecha dentro del núcleo. La mejor implementación en cuanto a rendimiento, en cualquiera de las dos pruebas realizadas, es la hecha con el controlador propietario de NVIDIA para GPUs y es debido a que el autor refiere que el fabricante ha hecho optimizaciones a su *hardware* cuando dicho controlador es usado.

Como el lector podrá darse cuenta, el tema de la aceleración de tareas propias del sistema operativo (como cifrado, compresión, ruteo de paquetes, verificación de integridad de sistemas de archivos, etc.) a través del paralelismo se ha venido desarrollando desde hace algún tiempo, sin embargo no existen muchos trabajos en el que el GPU se utilice como unidad de aceleración. De hecho, de todos los trabajos que se han mencionado hasta este momento ninguno explica de manera detallada que mecanismos (como llamadas al sistema, mecanismos de memoria compartida o de comunicación interproceso, etc.) pueden ser utilizados cuando se pretende que el sistema operativo utilice el GPU como una unidad adicional de procesamiento. Es tarea, por lo tanto, de este trabajo de investigación describir de manera puntual que mecanismos son útiles al momento de llevar a cabo una implementación propia.

Capítulo 3

AES y su Implementación en CUDA

3.1. Introducción a la criptografía

Criptografía es la ciencia de la escritura secreta con el objetivo de esconder el significado de un mensaje [11]. Dentro de la ciencia de la criptografía se encuentran dos ramas que interesan a nuestro estudio, estas son la Criptografía Simétrica y la Criptografía Asimétrica, las cuales a su vez se encargan del estudio de los algoritmos de cifrado con llave simétrica y algoritmos de cifrado con llave asimétrica, respectivamente.

Antes de adentrarnos en una explicación un poco más detallada de los algoritmos de llave simétrica y asimétrica, es preciso definir en que consiste un algoritmo de cifrado. Si buscamos en el diccionario de la Real Academia de la Lengua la palabra cifrado y algoritmo, encontraremos que el significado del primero es "Transcribir en guarismos, letras o símbolos, de acuerdo con una clave, un mensaje cuyo contenido se quiere ocultar" y el significado del segundo es "Conjunto ordenado y finito de operaciones que permiten hallar la solución a un problema". Uniendo de manera simple estas dos definiciones, podemos definir entonces un algoritmo de cifrado como un conjunto de pasos finitos u operaciones que al aplicarlos a un mensaje permite ocultar el contenido de este. Es importante mencionar que así como

el algoritmo de cifrado nos permite ocultar (cifrar) el contenido de un mensaje, también este nos debe permitir revelar (descifrar) el contenido del mismo mensaje utilizando también una serie de pasos. Dentro de la criptografía, al conjunto de datos sin cifrar se le denomina normalmente texto plano y al conjunto de datos cifrados se le denomina texto cifrado.

3.1.1. Algoritmos de llave asimétrica

Estos algoritmos se caracterizan principalmente por utilizar dos llaves (entiéndase por llave una serie de *bits*), una de estas llaves se utiliza en conjunto con un algoritmo para cifrar texto plano y la otra llave se utiliza de la misma manera para descifrar texto cifrado; la figura 3.1 es una representación gráfica del funcionamiento de los algoritmos de llave asimétrica. Ejemplos de estos algoritmos son: RSA, ElGamal y DSA (*Digital Signature Algorithm*).

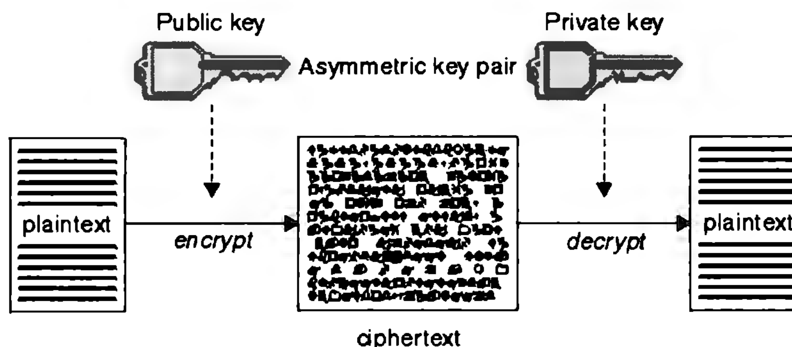


Figura 3.1: Criptografía de llave asimétrica.

Fuente: http://publib.boulder.ibm.com/infocenter/wmqv6/v6r0/index.jsp?topic=/com.ibm.mq.csqzas.doc/sy10500_.htm

En la figura 3.1 podemos notar que dos llaves son necesarias tanto para cifrar como para descifrar, en la primera parte de la figura la combinación de texto plano (*plaintext*) con la llave (mediante un algoritmo) nos da como resultado texto cifrado (*ciphertext*); en un proceso inverso la combinación de una llave con el texto cifrado nos da como resultado texto plano nuevamente. A los algoritmos de llave asimétrica también se les denomina algoritmos de llave pública, pues una de las llaves es distribuida públicamente y es utilizada para cifrar datos

que después solamente podrán ser descifrados con su correspondiente llave privada.

Los algoritmos de llave asimétrica normalmente son utilizados cuando dos entidades (un emisor y un receptor) desean intercambiar datos a través de un canal inseguro, como es el caso de redes TCP/IP, redes de telefonía celular, redes inalámbricas, etc.

3.1.2. Algoritmos de llave simétrica

En los algoritmos de llave simétrica solamente una llave es utilizada para cifrar tanto texto plano como para descifrar texto cifrado. Ejemplos de este tipo de algoritmos son: DES *(Data Encryption Standard)*, Triple-DES, BLOWFISH, XTEA *(Extended Tiny Encryption Algorithm)* y AES *(Advanced Encryption Standard)*. La figura 3.2 muestra el funcionamiento de los algoritmos de llave simétrica, en ella podemos ver que la misma llave es aplicada en conjunto con un algoritmo para llevar a cabo tanto el proceso de cifrado como el de descifrado.

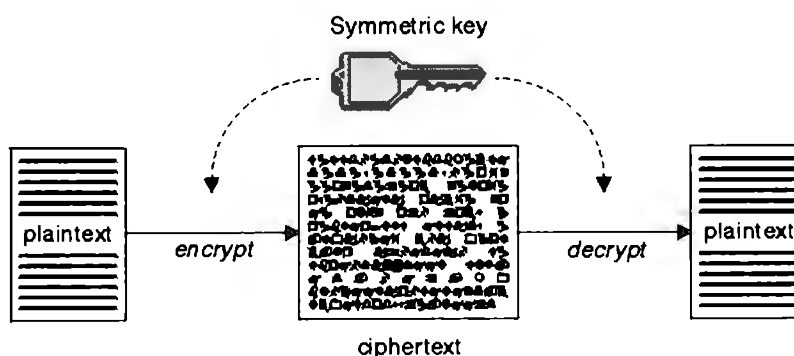


Figura 3.2: Criptografía de llave simétrica.

Fuente: http://publib.boulder.ibm.com/infocenter/wmqv6/v6r0/index.jsp?topic=/com.ibm.mq.csqzas.doc/sy10500_.htm

Una aplicación de los algoritmos de llave simétrica es el cifrado de los datos almacenados en sistemas de archivos, como ejemplo tenemos a ecryptfs.

3.1.2.1. Algoritmos de bloque

Los algoritmos de llave simétrica a su vez se clasifican en algoritmos de flujo *(stream ciphers)* y en algoritmos de bloque *(block ciphers)* [11]. Es de interés del presente trabajo hablar sobre el funcionamiento de los algoritmos de bloque y no de los algoritmos de flujo, debido a la naturaleza de su funcionamiento. En términos generales, un algoritmo de bloque opera sobre un conjunto de *bits* (llamado bloque), es decir, los datos a cifrar son divididos en bloques y cada bloque puede ser cifrado o descifrado utilizando la misma llave; en algunos modos de operación el cifrado de cada bloque es completamente independiente de otros. Normalmente el tamaño del bloque en los algoritmos de bloque es de 8 *bytes* (64 *bits*), como es el caso de DES o Triple-DES, o de 16 *bytes* (128 *bits*) como es el caso de AES.

3.2. Descripción del algoritmo AES

AES es el algoritmo de llave simétrica más usado hoy en día [11] y está basado en el algoritmo de bloque Rijndael, desarrollado por Joan Daemen y Vincent Rijmen de Bélgica. La única diferencia entre AES y Rijndael es el tamaño del bloque sobre el que el algoritmo opera, mientras que en Rijndael el tamaño puede ser algún múltiplo de 32 *bits* (con un mínimo de 128 y un máximo de 256), en AES el tamaño de bloque es siempre de 128 *bits*; sin embargo, el tamaño de las llaves puede variar siendo estas de 128, 192 y 256 *bits*. En AES es posible cifrar mensajes mayores al tamaño de un bloque, para llevar a cabo este proceso es necesario utilizar alguno de los siguientes modos de operación: ECB *(Electronic Codebook)*, CBC *(Cipher-block chaining)*, PCBC *(Propagating cipher-block chaining)*, CFB *(Cipher Feedback)*, OFB *(Output Feedback)* y CTR *(Counter)*.

Como se mencionó en la introducción de este capítulo, un algoritmo de cifrado aplica un conjunto de operaciones a un mensaje para cifrarlo, normalmente dichas operaciones se refieren a operaciones lógicas, operaciones aritméticas y operaciones de substitución. AES

siendo un algoritmo iterativo, aplica un conjunto de operaciones (descritas más abajo) a resultados intermedios (llamados *state*) durante un número determinado de rondas. El número de rondas está definido por el tamaño de la llave que se está utilizando; la tabla 3.1 muestra esta relación.

Tamaño llave *(bits)*	Rondas (n)
128	10
192	12
256	14

Tabla 3.1: Relación entre el tamaño de la llave y el número de rondas.

En el algoritmo AES, el *state* es definido como una matriz rectangular (de valores hexadecimales) de tamaño 4x4 (32 *bits* por renglón); como lo muestra la figura 3.3.

$$\begin{bmatrix} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{10} & s_{11} & s_{12} & s_{13} \\ s_{20} & s_{21} & s_{22} & s_{23} \\ s_{30} & s_{31} & s_{32} & s_{33} \end{bmatrix}$$

Figura 3.3: Representación del *state.*

La llave se puede representar también como una matriz de valores hexadecimales, sin embargo, esta contiene 4 renglones y el número de columnas (N_k) varía dependiendo del tamaño de la llave. Para llaves de 128 *bits*, N_k es 4, para llaves de 192 *bits*, N_k es 6 y para 256 *bits*, N_k es 8. La figura 3.4 muestra la forma de representar una llave de 128 *bits* de acuerdo a lo explicado en este párrafo.

$$\begin{bmatrix} k_{00} & k_{01} & k_{02} & k_{03} \\ k_{10} & k_{11} & k_{12} & k_{13} \\ k_{20} & k_{21} & k_{22} & k_{23} \\ k_{30} & k_{31} & k_{32} & k_{33} \end{bmatrix}$$

Figura 3.4: Representación gráfica de una llave de 128 *bits.*

Si se tiene en cuenta que computacionalmente tanto la llave como el *state* pueden ser representados como un bloque de números hexadecimales consecutivos, entonces la forma de convertir dicha llave y el *state* (de acuerdo a la figura 3.3 y 3.4) a su representación matricial en el algoritmo AES sería de la siguiente forma: s_{00}, s_{10}, s_{20}, s_{30}, s_{01}... y k_{00}, k_{10}, k_{20}, k_{30}, k_{01}..., en donde cada elemento representa un valor hexadecimal. Es decir, si tenemos un *state* como 3243f6a8885a308d313198a2e0370734 y una llave de entrada de 128 *bits* como 2b7e151628aed2a6abf7158809cf4f3c entonces las respectivas matrices serían como las figuras 3.5 y 3.6.

$$\begin{bmatrix} 32 & 88 & 31 & e0 \\ 43 & 5a & 31 & 37 \\ f6 & 30 & 98 & 07 \\ a8 & 8d & a2 & 34 \end{bmatrix}$$

Figura 3.5: Ejemplo de un *state.*

$$\begin{bmatrix} 2b & 28 & ab & 09 \\ 7e & ae & f7 & cf \\ 15 & d2 & 15 & 4f \\ 16 & a6 & 88 & 3c \end{bmatrix}$$

Figura 3.6: Ejemplo de una llave de 128 *bits*.

3.2.1. Proceso de cifrado en AES

Para tener una idea general del proceso de cifrado, la figura 3.7 muestra un diagrama en donde se pueden ver los pasos a seguir y las operaciones a aplicar durante este. La figura muestra cada una de las secciones o rondas en que está dividido el algoritmo, en la primera sección o ronda inicial solamente se aplica la operación de AddRoundKey a la entrada que es el *state*, en la siguiente sección o rondas intermedias las operaciones SubBytes, ShiftRows, MixColumns y AddRoundKey se repiten n-1 número de veces (de acuerdo a la tabla 3.1) y por último en la sección final o ronda final se aplican las operaciones SubBytes, ShiftRows y AddRoundKey.

Durante las rondas intermedias y la ronda final no se utilizará la llave principal en la operación AddRoundKey (descrita más adelante), sino los diferentes elementos (referidos en la figura 3.7 como *Round Key*) de una llave derivada a partir de la principal.

3.2.2. Operaciones utilizadas durante el proceso de cifrado

A continuación se hace una descripción detallada de cada una de las operaciones utilizadas en el proceso de cifrado en AES y las cuales se pueden ver de manera gráfica en la figura 3.7.

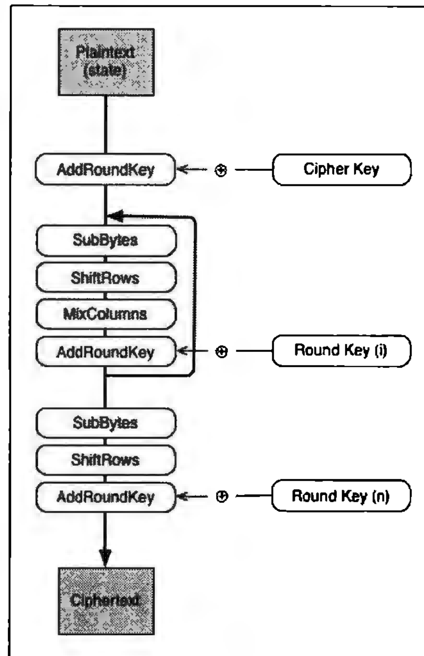


Figura 3.7: Proceso de cifrado en AES.

3.2.2.1. SubBytes

La operación SubBytes consiste en sustituir cada *byte* en el *state* por otro valor, dicho valor se obtiene utilizando un vector llamado S-Box *(Substitution Box)*. Debido a que el cálculo del vector S-Box es independiente de cualquier valor de entrada, este puede ser precalculado fuera del mismo algoritmo y guardado en memoria (su tamaño es solamente de 256 *bytes*), para posteriormente ser utilizado en el proceso de cifrado. El cálculo del vector S-Box es resultado de un proceso matemático, el cual no será descrito aquí. Suponiendo que se tiene un *state* (s_{ij}) como el siguiente:

$$\begin{bmatrix} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{10} & s_{11} & s_{12} & s_{13} \\ s_{20} & s_{21} & s_{22} & s_{23} \\ s_{30} & s_{31} & s_{32} & s_{33} \end{bmatrix}$$

La operación SubBytes estaría definida como:

$$s'_{ij} = SBox(s_{ij})$$

3.2.2.2. AddRoundKey

La operación AddRoundKey simplemente consiste en combinar el *state* con la llave derivada utilizando la operación lógica XOR, es decir, suponiendo que se tiene el siguiente *state* (s_{ij}) y la llave (k_{ij}) entonces la operación quedaría definida como:

$$\begin{bmatrix} s'_{00} & s'_{01} & s'_{02} & s'_{03} \\ s'_{10} & s'_{11} & s'_{12} & s'_{13} \\ s'_{20} & s'_{21} & s'_{22} & s'_{23} \\ s'_{30} & s'_{31} & s'_{32} & s'_{33} \end{bmatrix} = \begin{bmatrix} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{10} & s_{11} & s_{12} & s_{13} \\ s_{20} & s_{21} & s_{22} & s_{23} \\ s_{30} & s_{31} & s_{32} & s_{33} \end{bmatrix} \oplus \begin{bmatrix} k_{00} & k_{01} & k_{02} & k_{03} \\ k_{10} & k_{11} & k_{12} & k_{13} \\ k_{20} & k_{21} & k_{22} & k_{23} \\ k_{30} & k_{31} & k_{32} & k_{33} \end{bmatrix}$$

Donde:

$$s'_{ij} = s_{ij} \oplus k_{ij}$$

Como se mencionó anteriormente, la llave utilizada durante las rondas del proceso de cifrado es derivada de la llave principal a través de un método llamado *Rijndael's Key Schedule*, en cada iteración la operación AddRoundKey utilizará 16 *bytes* de la llave derivada (cada uno de estos 16 *bytes* corresponde a un elemento de la subllave). El método *Rijndael's Key Schedule* es explicado en la siguiente sección.

3.2.2.3. *Rijndael's Key Schedule*

El tamaño de la llave derivada a utilizarse en la operación AddRoundKey estará determinado por el tamaño de la llave principal, la relación que existe entre estos tamaños es fácil de obtener y básicamente depende del número de rondas en el proceso de cifrado. Las relaciones entre llaves y subllaves se muestran en la tabla 3.2.

	128 *bits*		192 *bits*		192 *bits*	
Ronda(s)	Iteraciones	*Bytes*	Iteraciones	*Bytes*	Iteraciones	*Bytes*
Inicial	1	16	1	16	1	16
Intermedias	9	144	11	176	13	208
Final	1	16	1	16	1	16
Tamaño subllave	Suma	176	Suma	208	Suma	240

Tabla 3.2: Relación entre llaves y subllaves.

Como ejemplo, para el caso de llaves de 192 *bits* la operación AddRoundKey utilizará 16 *bytes* (128 *bits*) en la ronda inicial, para las rondas intermedias utilizará 11*16 *bytes* (176 *bytes*) y para la ronda final utilizará 16 *bytes* más; lo anterior da como total 208 *bytes*.

Para fines ilustrativos de cómo obtener la llave derivada, supongamos que esta es un vector x con n elementos de 32 *bits* (n se obtiene al dividir el tamaño en *bits* de la llave derivada entre 32) como se muestra a continuación:

$$\begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 & \cdots & x_n \end{bmatrix}$$

Y que tenemos también una llave principal k con m elementos de 32 *bits* (m se obtiene al dividir el tamaño en *bits* de la llave principal entre 32 o al tomar el valor de $\mathrm{N_k}$):

$$\begin{bmatrix} k_1 & k_2 & \cdots & k_m \end{bmatrix}$$

El primer paso para obtener la llave derivada es copiar los m elementos de la llave principal a la llave derivada. Posteriormente, para obtener los elementos i que sean múltiplos de m+1 se utiliza la siguiente expresión (también llamada *Key Schedule Core*):

$$x_i = x_{i-m} \oplus SubBytes(Rotate(x_{i-1})) \oplus Rcon(l)$$

Donde: $m+1 \leq i \leq n, l \geq 1$ y aumenta en uno por cada múltiplo de m+1.

La operación de SubBytes es similar a la utilizada en el proceso de cifrado, la diferencia radica en que se sustituye cada uno de los cuatro *bytes* del elemento al que se le aplica la operación. La operación Rotate rota el elemento hacia la izquierda una posición y Rcon es una operación que regresa un valor de un vector que fue previamente calculado y el cual se muestra en el apéndice A de esta tesis.

Para obtener todos los elementos restantes, es decir, los que no sean múltiplos de m+1, se utiliza la siguiente expresión:

$$x_i = x_{i-m} \oplus x_{i-1}$$

El método anteriormente descrito sirve para obtener subllaves cuyas llaves principales sean de tamaños de 128 y 192 *bits*, el proceso para obtener subllaves a partir de llaves de 256 *bits* es prácticamente igual y la única diferencia que existe es que para obtener los elementos que sean múltiplos de $(m+1)/2$ (pero no múltiplos de m+1) se utiliza la siguiente expresión:

$$x_i = SBox(x_{i-1}) \oplus x_{i-m}$$

3.2.2.4. ShiftRows

La operación de ShiftRows consiste en rotar hacia la izquierda un número determinado de posiciones cada uno de los renglones del *state*; el primer renglón se rota cero posiciones, el segundo se rota una posición, el tercero se rota dos y el cuarto se rota tres posiciones. Suponiendo que se tiene el siguiente *state*:

$$\begin{bmatrix} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{10} & s_{11} & s_{12} & s_{13} \\ s_{20} & s_{21} & s_{22} & s_{23} \\ s_{30} & s_{31} & s_{32} & s_{33} \end{bmatrix}$$

El resultado de aplicar la operación de ShiftRows al *state* anterior sería:

$$\begin{bmatrix} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{11} & s_{12} & s_{13} & s_{10} \\ s_{22} & s_{23} & s_{20} & s_{21} \\ s_{33} & s_{30} & s_{31} & s_{32} \end{bmatrix}$$

3.2.2.5. MixColumns

La operación de MixColumns consiste tomar cada una de las columnas del *state* y multiplicarla por la siguiente matriz:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

Es decir, suponiendo que se tiene el siguiente *state*:

$$\begin{bmatrix} s_{00} & s_{01} & s_{02} & s_{03} \\ s_{11} & s_{12} & s_{13} & s_{10} \\ s_{22} & s_{23} & s_{20} & s_{21} \\ s_{33} & s_{30} & s_{31} & s_{32} \end{bmatrix}$$

La primera columna de la matriz resultante sería el resultado de realizar la siguiente

multiplicación de matrices:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} * \begin{bmatrix} s_{00} \\ s_{11} \\ s_{22} \\ s_{33} \end{bmatrix} = \begin{bmatrix} 2 \times s_{00} + 3 \times s_{11} + 1 \times s_{22} + 1 \times s_{33} \\ 1 \times s_{00} + 2 \times s_{11} + 3 \times s_{22} + 1 \times s_{33} \\ 1 \times s_{00} + 1 \times s_{11} + 2 \times s_{22} + 3 \times s_{33} \\ 3 \times s_{00} + 1 \times s_{11} + 1 \times s_{22} + 2 \times s_{33} \end{bmatrix}$$

La última columna sería el producto de la siguiente multiplicación:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} * \begin{bmatrix} s_{03} \\ s_{10} \\ s_{21} \\ s_{32} \end{bmatrix} = \begin{bmatrix} 2 \times s_{03} + 3 \times s_{10} + 1 \times s_{21} + 1 \times s_{32} \\ 1 \times s_{03} + 2 \times s_{10} + 3 \times s_{21} + 1 \times s_{32} \\ 1 \times s_{03} + 1 \times s_{10} + 2 \times s_{21} + 3 \times s_{32} \\ 3 \times s_{03} + 1 \times s_{10} + 1 \times s_{21} + 2 \times s_{32} \end{bmatrix}$$

Es importante mencionar que tanto la operación de suma como la de multiplicación utilizadas en la operación de MixColumns están definidas en el campo de *Galois*, es decir, ambas operaciones han sido redefinidas en este campo. Para una explicación detallada de cómo obtener la suma y la multiplicación de dos números vea el apéndice B.

3.2.3. Proceso de descifrado en AES

Todas las operaciones descritas hasta este momento son utilizadas durante el proceso de cifrado en AES, el descifrado en términos generales consiste en seguir el flujo del diagrama presentado en la figura 3.7 en orden inverso y aplicar las operaciones inversas de AddRoundKey, ShiftRows, SubBytes y MixColumns al *state*; es decir, en la primera etapa del algoritmo se aplican las operaciones inversas de AddRoundKey, ShiftRows, SubBytes, en las rondas intermedias las operaciones inversas de AddRoundKey, MixColumns, ShiftRows y SubBytes y en la última ronda se aplica la operación inversa de AddRoundKey.

Las operaciones utilizadas durante el proceso de cifrado y sus inversos son muy similares,

las diferencias se muestran en la tabla 3.3

Operación	**Diferencia con la operación inversa.**
ShiftRows	Los renglones del *state* son rotados hacia la derecha.
SubBytes	Se utiliza el vector rsbox en lugar de sbox; el vector rsbox es calculado también previamente.
MixColumns	Se utiliza una matriz diferente, esta se muestra en la figura 3.8.
AddRoundKey	Los elementos de la llave derivada son utilizados en orden inverso, es decir, en la ronda inicial se utiliza el último elemento de la llave derivada y en la ronda final el primer elemento de esta.

Tabla 3.3: Diferencias en las operaciones inversas.

$$\begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix}$$

Figura 3.8: Matriz utilizada en la operación inversa de AddRoundKey.

3.2.4. Implementación básica del algoritmo AES en CUDA

El paralelismo se refiere a una propiedad de un programa donde muchas operaciones aritméticas pueden ser llevadas a cabo en estructuras de datos de una forma simultánea [8], algunos

modos de operación de los algoritmos de cifrado de bloque (como AES -ECB-) poseen esa característica y por lo tanto son candidatos ideales para implementarse en la arquitectura de CUDA. La figura 3.9 muestra el funcionamiento del modo de operación ECB, en esta figura se puede apreciar que al aplicar el algoritmo de cifrado a bloques de datos de tamaño fijo *(plaintext)*, los cuales son independientes entre sí, en conjunto con la llave *(Key)* se obtiene texto cifrado *(ciphertext)*.

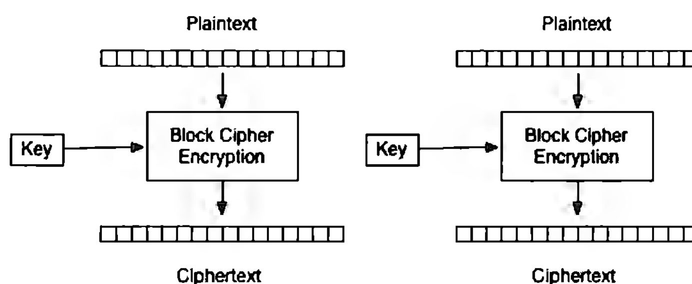


Figura 3.9: Modo de operación ECB.

De acuerdo a lo explicado en la sección 2.4.2, el modelo de programación en la arquitectura CUDA permite descomponer el problema a resolver en pequeños subproblemas para poder paralelizarlo; en la implementación hecha en este trabajo de investigación cada hilo de CUDA se encargará de cifrar un bloque de datos de 128 *bits* (16 *bytes*), un *state* en términos de AES. A su vez, cada uno de estos hilos de CUDA será agrupado en bloques de hilos de tamaño de 16x16, lo anterior permitirá que cada uno de los bloques cifre 4096 *bytes* (16 hilos x 16 hilos x 16 *bytes*). El hecho de que el problema de cifrado se haya descompuesto como se explicó con anterioridad es porque los 4096 *bytes* que cada bloque de CUDA cifrará representan una página, que en sistemas operativos es la unidad básica de manejo de memoria y es la forma en como la memoria física de una computadora está dividida para su administración; en caso de que el sistema operativo requiera cifrar más de una página entonces el tamaño de la malla en CUDA puede adecuarse.

Una vez que se ha establecido el tamaño del bloque de hilos y la malla a ejecutarse, el siguiente paso en la implementación de AES es escribir el *kernel* de CUDA que cifrará los

datos, por cuestiones de rendimiento es importante tener muy presente que el código escrito deberá evitar en la medida de lo posible que los hilos en el GPU tengan distintas trayectorias de ejecución, es decir, debe existir una divergencia muy pequeña entre las instrucciones que cada hilo ejecutará. El siguiente listado de código muestra la función general del *kernel* que se utilizará en las pruebas descritas más adelante:

```
1  __global__ void cypher(uint32_t *expandedKey, uint8_t *global_state, int rounds) {
2
3          __shared__ uint32_t roundKey[60];
4
5          int idx = 16*(blockIdx.x*blockDim.x*blockDim.y+(blockDim.x*threadIdx.y+threadIdx.x));
6          int i = 0;
7          uint32_t state [4];
8
9          get_state(state, global_state+idx);
10
11         if(idx%PAGE_SIZE==0)
12                 for(i=0; i<((rounds+1)*16*8)/32; i++)
13                         roundKey[i] = expandedKey[i];
14
15         __syncthreads();
16
17         AddRoundKey(state, roundKey);
18
19         for(i=1; i<rounds; i++) {
20                 SubBytes(state);
21                 ShiftRows(state);
22                 MixColumns(state);
23                 AddRoundKey(state, roundKey+4*i);
24         }
```

```
25
26         SubBytes(state);
27         ShiftRows(state);
28         AddRoundKey(state, roundKey+4*rounds);
29
30         put_state(state, global_state+idx);
31 }
```

En la declaración del *kernel* (línea 1) del código anterior se establece que este recibirá tres argumentos, los primeros dos son apuntadores a la memoria global del GPU y el último es un entero que indica el número de rondas que se llevarán a cabo en el proceso de cifrado (de acuerdo a la tabla 3.1). La llave derivada que se utilizará en cada ronda, contenida en el primer apuntador, se obtendrá mediante un proceso que se ejecutará totalmente en el CPU, la razón de que se ejecute completamente ahí es porque no es posible paralelizarlo y además porque es suficiente con que se realice una vez y no por cada uno de los hilos en el GPU (véase el apéndice C para una referencia completa de las funciones utilizadas en esta tesis para derivar subllaves). El segundo apuntador contendrá los datos a cifrar y cada uno de sus elementos será del tipo *byte* sin signo, esto permitirá tener un acceso individual y evitará tener problemas de alineación de memoria.

Es necesario que cada uno de los hilos ejecutados durante el proceso de cifrado tenga una copia de la llave derivada debido a que esta será empleada en la operación AddRoundKey, por este motivo y para evitar problemas de latencia si se accede a memoria global, el primer hilo de cada bloque copiará la llave (líneas 11, 12 y 13 del código anterior) a una región de memoria compartida (declarada en la línea 3) por los hilos de un bloque.

Las funciones get_state y put_state (utilizadas en las líneas 9 y 30) serán las encargadas de copiar los datos a cifrar de y hacia la memoria global, cada hilo copiará 16 elementos (128 *bits*) a partir del índice almacenado en la variable idx (calculado en base a las variables blockIdx.x, blockDim.x, blockDim.y, blockDim.x, threadIdx.y y threadIdx.x, predefinidas en

CUDA); los elementos copiados se transformarán en valores de 32 *bits* y serán almacenados en el arreglo *state*, lo anterior es para mejorar el rendimiento en algunas de las operaciones del proceso de cifrado como es el caso de AddRoundKey (mostrada en la figura 3.10).

En el resto del código (líneas 17 en adelante) se pueden apreciar la ronda inicial, las rondas intermedias (líneas 19-24) y la ronda final (líneas 26-28) del proceso de cifrado.

```
__device__ void AddRoundKey(uint32_t *state, uint32_t *roundKey) {
        state [0] ^= *roundKey;
        state [1] ^= *(roundKey+1);
        state [2] ^= *(roundKey+2);
        state [3] ^= *(roundKey+3);
}
```

Figura 3.10: Implementación de la operación AddRoundKey en el GPU.

El *kernel* utilizado para descifrar los datos (mostrado en el siguiente listado de código) es muy similar al utilizado para cifrar, las diferencias entre uno y otro han sido mencionadas en la sección 3.2.3.

```
__global__ void decypher(uint32_t *expandedKey, uint8_t *global_state, int rounds) {

        __shared__ uint32_t roundKey[60];

        int idx = 16*(blockIdx.x*blockDim.x*blockDim.y+(blockDim.x*threadIdx.y+threadIdx.x));
        int i=0;
        uint32_t state [4];

        get_state(state, global_state+idx);

        if(idx%PAGE_SIZE==0)
                for(i=0; i<((rounds+1)*16*8)/32; i++)
                        roundKey[i] = expandedKey[i];
```

```
        __syncthreads();

        AddRoundKey(state, roundKey+rounds*4);
        invShiftRows(state);
        invSubBytes(state);

        for(i=rounds-1; i>0; i--) {
                AddRoundKey(state, roundKey+i*4);
                invMixColumns(state);
                invShiftRows(state);
                invSubBytes(state);
        }

        AddRoundKey(state, roundKey);

        put_state(state, global_state+idx);

}
```

Véase el apéndice D de esta tesis para la versión completa de la implementación de AES (ECB) en CUDA utilizada.

Capítulo 4

Implementación

4.1. El espacio del usuario y del núcleo en GNU/Linux

Como se mencionó en el capítulo 2, una de las principales funciones del sistema operativo es la de administrar los recursos de una computadora, dentro de esas tareas de administración se encuentra la de evitar accesos no autorizados al *hardware*; para hacer cumplir lo anterior el sistema operativo se vale del CPU. En la actualidad, la mayoría de los CPUs implementan varios modos de operación y en cada uno de estos modos solo algunas instrucciones son permitidas. Por lo regular existen dos modos, el modo supervisor (o modo núcleo) en donde todas las instrucciones están permitidas y el modo usuario en donde instrucciones de entrada/salida y otras instrucciones no están permitidas [16]. En teoría de sistemas operativos a los modos de operación implementados por el CPU normalmente se les conoce como espacio del núcleo y espacio del usuario; en GNU/Linux (como lo muestra figura 4.1) en el espacio del núcleo encontramos el código ejecutado por el mismo núcleo (como el manejador de memoria, el manejador de procesos, los controladores de dispositivos, etc.), las llamadas al sistema (como open, write, fork, etc.) y el código propio de cada arquitectura que está escrito en ensamblador. En el espacio del usuario encontramos las aplicaciones programadas por los

usuarios y las bibliotecas que dichas aplicaciones utilizan para ejecutarse, como es el caso de glibc (la biblioteca estándar de C en GNU/Linux), CUDA, etc.

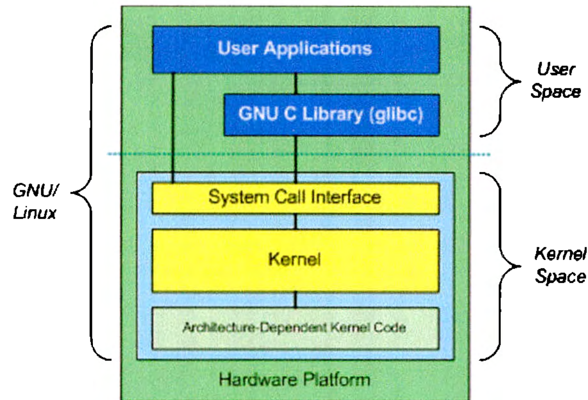


Figura 4.1: Arquitectura fundamental del sistema operativo GNU/Linux.

Fuente: https://www.ibm.com/developerworks/linux/library/l-linux-kernel/

Con el uso de bibliotecas, una aplicación de usuario puede hacer llamadas a funciones que no defina, en la etapa de enlace (durante el proceso de compilación) todas las referencias hacia funciones externas son resueltas. En GNU/Linux, el código escrito para el núcleo también puede hacer uso de funciones externas, sin embargo las únicas disponibles son las exportadas por el mismo núcleo [4]. En el espacio del núcleo no hay bibliotecas y no existe una forma directa de utilizar las que se encuentran en el espacio del usuario, por lo tanto tampoco es posible utilizar las bibliotecas de cómputo heterogéneo de CUDA.

Adicionalmente a como se resuelven los símbolos en los dos espacios, existen más diferencias que son importantes mencionar, en especial nos enfocaremos en tres para hacer notar la diferencia entre el espacio del núcleo y el espacio del usuario. La primer diferencia es que mientras en el espacio del núcleo tenemos direcciones virtuales, direcciones lógicas y direcciones físicas, en el espacio del usuario solamente tenemos direcciones virtuales, por lo tanto una aplicación programada en cualquiera de los dos espacios no puede acceder fácilmente a una región de memoria que se encuentre en el otro espacio. Las otras dos diferencias son que los APIs *(Application Programming Interface)* que se proporcionan y los ciclos de vida de las

aplicaciones programadas en cada uno de los espacios son muy diferentes, lo que hace que lo que ha sido programado en un espacio no necesariamente funcione en el otro.

En base a lo explicado anteriormente, la siguiente sección explica el modelo a implementar y también los mecanismos utilizados para que el núcleo de GNU/Linux emplee el GPU como una unidad de aceleración y de esta forma sea capaz de cifrar y descifrar datos a través de las bibliotecas de CUDA.

4.2. Modelo y mecanismos utilizados en la implementación

La interacción entre el núcleo del sistema operativo y el GPU es el principal problema a resolver en la implementación, en específico es necesario encontrar mecanismos que permitan al núcleo de GNU/Linux tanto ejecutar los kernels de cifrado y descifrado, presentados en la sección 3.2.4, como poder copiar datos desde una región de memoria en el núcleo hacia la memoria del GPU y de regreso.

La figura 4.2 muestra el modelo a usarse en la implementación, este modelo ha sido propuesto por algunos autores mencionados en el estado del arte de este trabajo de investigación y se pretende usar como punto de partida. En dicha figura podemos observar una clara diferencia entre el espacio del núcleo *(OS Kernel Space)* y el espacio del usuario *(User Space)*, en el espacio del usuario encontramos una aplicación *"helper"* que interactúa con el GPU, en representación del núcleo, a través de las bibliotecas de CUDA. Por otro lado, en el espacio del núcleo se encuentra un módulo (aes_cuda) que mantiene un canal de comunicación con el proceso *"helper"* y que permite ejecutar aplicaciones programadas en CUDA de manera indirecta. Es importante mencionar que un módulo en GNU/Linux no es más que una pieza de código que se inserta utilizando el comando insmod y que le agrega alguna funcionalidad de forma dinámica al núcleo.

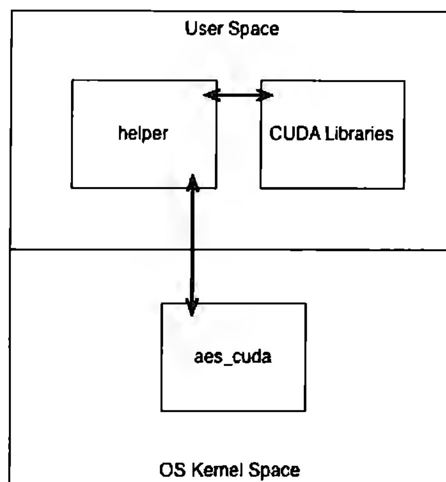


Figura 4.2: Modelo a implementar.

4.2.1. Ejecución indirecta de los *kernels* de cifrado y descifrado

Con el modelo presentado en la figura 4.2 surge la primera interrogante a resolver, ¿qué mecanismo utilizar para que el proceso *"helper"* y el módulo en el núcleo puedan comunicarse? Son varios los mecanismos que permiten la comunicación interproceso entre el espacio del núcleo y el espacio del usuario en GNU/Linux, para el propósito de este estudio se ha escogido utilizar el API *Generic Netlink*. La principal razón de que se haya escogido *Generic Netlink* es porque guarda una gran similitud con la programación de *sockets* en redes TCP/IP y también porque proporciona una comunicación *full-duplex* entre los dos espacios.

En *Generic Netlink*, la comunicación entre dos instancias que se encuentran en el mismo espacio o en diferentes espacios se hace a través del envío de paquetes (similares a los utilizados en TCP/IP), dichos paquetes están compuestos por un encabezado que contiene información de control y por un *payload*. Para saber qué tipo de paquete o mensaje una instancia le ha enviado a la otra, un identificador (número) puede ser asociado con un campo que se encuentra en el encabezado del paquete; los siguientes cuatro identificadores fueron definidos en la implementación del modelo utilizado:

Tecnológico de Monterrey, Campus Ciudad de México
Biblioteca

```
enum {
        AESHLPCUDA_CMD_REG,
        AESHLPCUDA_CMD_DONE,
        AESHLPCUDA_CMD_ENC,
        AESHLPCUDA_CMD_DEC,
};
```

Los identificadores AESHLPCUDA_CMD_REG y AESHLPCUDA_CMD_DONE son utilizados en mensajes que son enviados desde el proceso *"helper"* hacia el módulo aes_cuda en el núcleo. El primer identificador denota un tipo de mensaje inicial entre las instancias que se encuentran en los dos espacios, este mensaje es de suma importancia debido a que el módulo aes_cuda recupera el PID *(Process Identifier)* del proceso *"helper"* y lo utiliza como método de direccionamiento para futuros mensajes; el segundo identificador es utilizado cuando el proceso *"helper"* le notifica al núcleo que ha terminado de ejecutar algo en el GPU. Los identificadores AESHLPCUDA_CMD_ENC y AESHLPCUDA_CMD_DEC son utilizados por el núcleo cuando este le solicita al *"helper"* que lleve a cabo el cifrado o descifrado de datos.

Con el intercambio de mensajes en *Generic Netlink*, el núcleo es capaz de notificarle al proceso *"helper"* que ciertos datos necesitan ser cifrados o descifrados utilizando CUDA, sin embargo, la parte de transferencia de datos no queda resuelta con este mecanismo y por lo tanto es preciso emplear otro.

4.2.2. Transferencia de datos entre núcleo del sistema operativo y el GPU

De acuerdo a lo explicado en la sección 2.4.5, una parte importante en el flujo de ejecución de una aplicación en CUDA es la transferencia de datos que se tiene que hacer entre *host* y *device*. En base a lo anterior, es necesario implementar algún mecanismo que permita llevar datos desde alguna región de memoria en el núcleo de GNU/Linux hacia el GPU y

de regreso con la ayuda del proceso *"helper"*. Una solución inicial podría sugerir utilizar las funciones copy_from_user y copy_to_user exportadas por el núcleo de GNU/Linux, las cuales permiten copiar bloques de datos desde el núcleo hacia el espacio del usuario y viceversa. En esta solución, el proceso *"helper"* tendría que declarar una región de memoria que debería ser conocida por el núcleo y entonces así, este último, podría hacer transferencias entre el espacio del usuario y el espacio del núcleo y el proceso *"helper"* haría lo propio hacia el GPU. Si consideramos la transferencia de datos como una parte del proceso de cifrado, esta solución inicial presentaría un problema de latencia por los datos que se tienen que mover del espacio del núcleo hacia el espacio del usuario.

Otro mecanismo que podría ayudar a resolver el problema de la transferencia de datos sería el proporcionado por la operación mmap *(memory mapping)* en los controladores de dispositivos de caracteres, este mecanismo permite mapear una región de memoria del espacio de direcciones del núcleo al espacio de direcciones virtual de un proceso en el espacio del usuario.

Cuando se crea un controlador para un dispositivo de caracteres es necesario registrarlo en el núcleo de GNU/Linux utilizando la función register_chrdev, esta función recibe tres argumentos, el primer argumento es un número único asociado al dispositivo, el segundo es el nombre del archivo que estará asociado a este y el tercero es una estructura (mostrada en el siguiente listado de código) que tiene como miembros apuntadores a funciones que representan las operaciones soportadas (read, write, lseek, open, close, etc.) por el dispositivo. Cuando un proceso en el espacio del usuario trata, por ejemplo, de leer o escribir en el dispositivo por medio de un descriptor (con las llamadas a sistema read y write), las funciones especificadas en los miembros read y write serán ejecutadas; lo mismo sucede cuando la función mmap es llamada en el espacio del usuario.

```
struct file_operations {
```

```
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t , int);
        ssize_t (*read) (struct file *, char __user *, size_t , loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t , loff_t *);

        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);

};
```

Si un proceso en el espacio del usuario hace una llamada a la función mmap, este tiene que especificar una serie de argumentos que se presentan en el siguiente prototipo de función:

```
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

El primer argumento representa la dirección virtual inicial del segmento de memoria mapeado, el segundo la cantidad de *bytes* que el proceso está pidiendo sean mapeados, el tercero describe los permisos del área de memoria, el cuarto es un conjunto de banderas asociadas al comportamiento de la misma, el quinto es un descriptor de archivo asociado al dispositivo y el último es el desplazamiento en la región de memoria dentro del núcleo a partir del cual el mapeo deberá ser hecho.

Es importante poner especial atención al segundo argumento que recibe la función que tiene que ser especificada en el miembro mmap de la estructura file_operations, este argumento es una estructura que describe un área de memoria *(virtual memory area)* que será creada por el sistema operativo, a partir de las especificaciones proporcionadas por el proceso en el espacio del usuario en la función mmap, previo a ejecutar la función especificada en el miembro de la estructura. Una vez dentro del núcleo, la función que realmente se encarga de hacer el mapeo de las regiones de memoria es remap_pfn_range; el código empleado en la implementación de esta tesis de la función que se especificó en el miembro mmap de la

estructura file_operations se muestra a continuación:

```
int cudahlp_mmap(struct file *fd, struct vm_area_struct *vma) {
        if(remap_pfn_range(vma, vma->vm_start, __pa(helper_buffer)>>PAGE_SHIFT,
                vma->vm_end - vma->vm_start,
                vma->vm_page_prot))
                        return -EAGAIN;
        return 0;
}
```

Los argumentos que recibe la función remap_pfn_range se obtienen en su mayoría de la estructura vm_area_struct que la función ya recibe como argumento, el que es de mayor importancia mencionar es el tercero, pues este se refiere al número de página que le corresponde a la dirección física inicial de la región de memoria a mapear dentro del núcleo. Para obtener el número de página es necesario hacer un desplazamiento de *bits* en la dirección física, obtenida a través de la macro __pa *-physical address-*, utilizando la constante PAGE_SHIFT definida dentro del núcleo. Al término del mapeo, tanto el proceso *"helper"* como el módulo en núcleo serán capaces de acceder a los mismos datos como se muestra gráficamente en la figura 4.3.

Los mecanismos descritos hasta este momento resuelven el problema planteado al inicio de la sección 4.2, es decir, con ellos el núcleo es capaz de ejecutar los *kernels* de cifrado y descifrado de manera indirecta y también transferir datos hacia el GPU. La siguiente subsección presenta de manera general la secuencia de ejecución que se sigue cuando el núcleo requiere cifrar o descifrar datos utilizando el GPU.

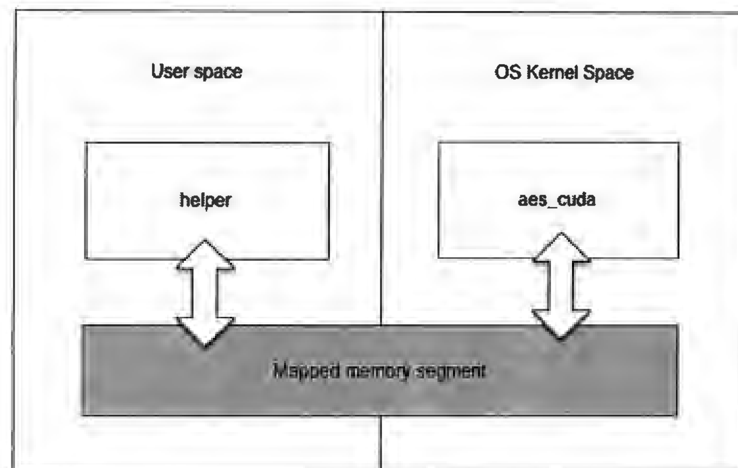


Figura 4.3: Región de memoria mapeada.

4.2.3. Secuencia de ejecución en el cifrado y descifrado de datos

La secuencia descrita en esta subsección es muy general, para una referencia completa de todo el proceso es necesario consultar el código que se incluye en el CD que acompaña a esta tesis.

Todo el proceso comienza con la inserción del módulo aes_cuda en el núcleo y la ejecución del proceso *"helper"* en el espacio del usuario con privilegios de superusuario; una vez que este último ha terminado de inicializar varios elementos (tales como *Generic Netlink*, el GPU y el área de memoria mapeada), envía un mensaje del tipo AESHLPCUDA_CMD_REG al módulo en el núcleo, entonces la función cudahelper_register es ejecutada a través de funciones *callback* en *Generic Netlink* y el PID del *"helper"* es guardado en una variable global para enviarle mensajes posteriores. A continuación, el proceso *"helper"* entra en un ciclo infinito esperando algún mensaje por parte del núcleo, como se muestra en el siguiente listado de código:

```
struct {
        struct nlmsghdr n;
        struct genlmsghdr g;
```

```
        char buf[256];
} ans;

...
// Registration of the CUDA AES Helper
send_aes_module_msg(nl_sd, id, AESHLPCUDA_CMD_REG, AES_HLPREG_A_MSG, ..., sizeof(uint64_t));

while(1) {
        int rep_len = recv(nl_sd, &ans, sizeof(ans), 0);

        key_payload = (uint32_t *)NLA_DATA(...);

        switch(ans.g.cmd) {
                case AESHLPCUDA_CMD_ENC:
                                expand256BitKey(key_payload, expanded_key);
                                ret_code = AESCUDAkernel(cypher);
                                send_aes_module_msg(nl_sd, id, AESHLPCUDA_CMD_DONE,
                                    AES_HLPDONE_A_MSG, &ret_code, sizeof(uint32_t));
                                printf("Done encrypting data...\n");
                                break;
                case AESHLPCUDA_CMD_DEC:
                                expand256BitKey(key_payload, expanded_key);
                                ret_code = AESCUDAkernel(decypher);
                                send_aes_module_msg(nl_sd, id, AESHLPCUDA_CMD_DONE,
                                    AES_HLPDONE_A_MSG, &ret_code, sizeof(uint32_t));
                                printf("Done decrypting data...\n");
                                break;
        }
...
}
```

El núcleo debe enviar un mensaje AESHLPCUDA_CMD_ENC o AESHLPCUDA_CMD_DEC

cuando necesita cifrar o descifrar datos, en el mensaje la llave que se utilizará durante el proceso de cifrado es enviada como *payload.* Como se puede ver en el código anterior, una vez que el proceso *"helper"* ha recibido un mensaje por parte del núcleo, este extrae la llave que contiene el paquete e inspecciona un campo en el encabezado para saber si el núcleo le está pidiendo cifrar o descifrar datos. Cuando el proceso *"helper"* conoce que operación debe realizar entonces la llave principal es expandida (de acuerdo a lo explicado en la sección 3.2.2.3) y posteriormente la función AESCUDAkernel es llamada con una referencia hacia el *kernel* de cifrado o descifrado como argumento. Dentro de la función AESCUDAkernel (cuyo código se muestra en el siguiente listado), la llave expandida y los datos a ser cifrados o descifrados (los cuales se encuentran en el área de memoria mapeada) son enviados hacia el GPU; finalmente el *kernel* es ejecutado con una serie de argumentos empleando la función cuLaunchKernel del CUDA *Driver* API. Una vez que la ejecución del *kernel* ha terminado, los datos en el GPU son copiados de regreso a la región de memoria mapeada.

Cuando la ejecución de la función AESCUDAkernel ha terminado, el proceso *"helper"* le envía finalmente un mensaje del tipo AESHLPCUDA_CMD_DONE al núcleo con un código que indica si el proceso de cifrado o descifrado fue llevado a cabo con éxito o no.

```
int AESCUDAkernel(CUfunction kernel) {

    // CUDA kernel arguments
    void *kernel_args[] = { &key_dev, &state_dev, &rounds };
    CUresult res;

    res = cuMemcpyHtoD(key_dev, expanded_key, EXPANDED_KEY_SIZE_BYTES );

    res = cuMemcpyHtoD(state_dev, state, NEEDED_THREADS * BYTES_PER_THREAD);

    res = cuLaunchKernel(kernel, grid_x, grid_y, 1, block_x, block_y, 1, 0, NULL, kernel_args,
```

```
        NULL);

    cuCtxSynchronize();

    res = cuMemcpyDtoH(state, state_dev, NEEDED_THREADS * BYTES_PER_THREAD);


    return 0;
}
```

La secuencia de ejecución explicada en esta sección se muestra de manera gráfica en la figura 4.4.

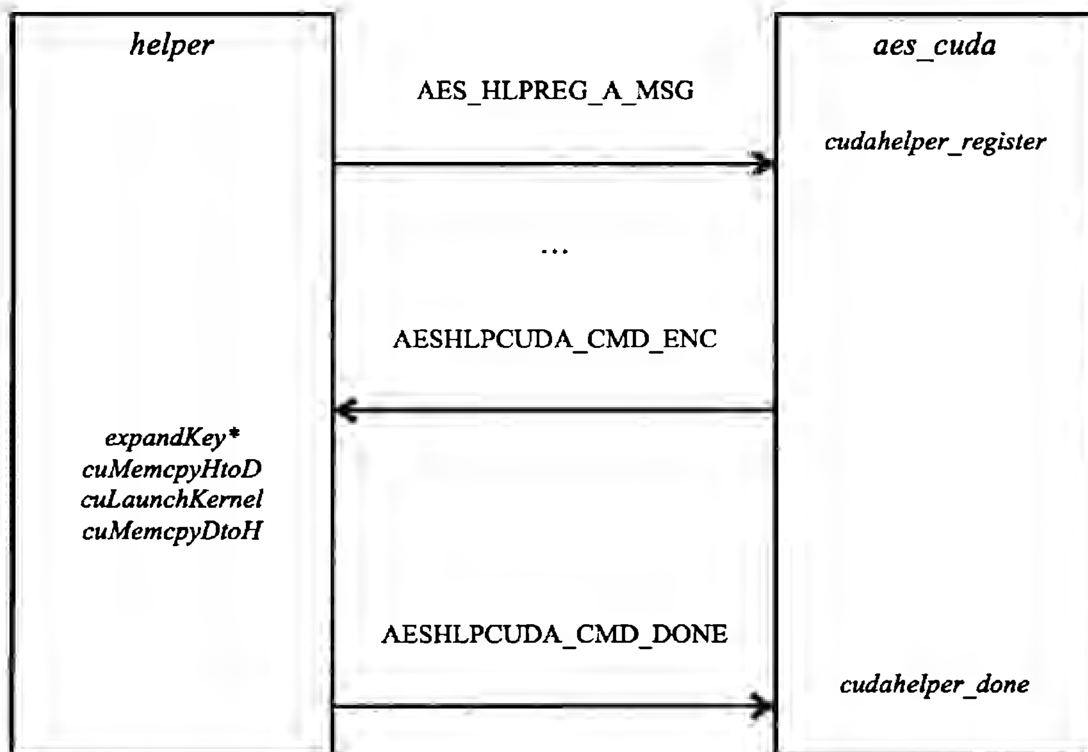


Figura 4.4: Secuencia de ejecución.

Capítulo 5

Resultados

Para saber si la implementación del modelo propuesto es útil, es necesario compararla con otros mecanismos que permitan el cifrado de datos dentro del núcleo de GNU/Linux, para hacer pruebas comparativas de rendimiento se ha decidido utilizar Gdev y *Crypto* API. Ambos mecanismos son explicados en las dos subsecciones siguientes.

5.1. Gdev

Gdev es un entorno de ejecución que propone la mejora del manejo de los recursos del GPU y la integración parcial de la arquitectura de CUDA dentro del núcleo de GNU/Linux, lo último permite al desarrollador de aplicaciones en CUDA ejecutarlas dentro del mismo núcleo. Utilizando la terminología de GNU/Linux, Gdev es un módulo del núcleo que está compuesto por un controlador de dispositivo y un entorno de ejecución, el controlador se encarga de administrar los recursos de *hardware* de bajo nivel (valiéndose del controlador de la tarjeta de video nouveau) y el entorno de ejecución se encarga de manejar las llamadas al API de Gdev (gopen, gclose, gmalloc, gfree, etc.). En un nivel más arriba del API de Gdev se encuentra kcuda que exporta un subconjunto de símbolos del API de CUDA (cuInit, cuMalloc, cuLaunchGrid, cuMemAlloc, cuMemFree, etc.) dentro del núcleo para que otros

módulos sean capaces de hacer llamadas a estos símbolos.

Para poder utilizar Gdev en el núcleo de GNU/Linux es necesario compilarlo e instalarlo, para una referencia de cómo hacer lo anterior consulte el apéndice E de esta tesis.

5.2. *Crypto* API (GNU/Linux)

Crypto API es un marco de trabajo presente en el núcleo de GNU/Linux que proporciona servicios de criptografía a diferentes componentes, originalmente fue diseñado para soportar una implementación de IPSec realizada por Dave Miller and Alexey Kuznetzov aunque actualmente es utilizado para otros propósitos como el cifrado de sistemas de archivos en ecryptfs.

Crypto API exporta un conjunto de funciones o símbolos que permiten utilizar algoritmos de cifrado (AES, DES, TEA, BLOWFISH, etc.), compresión (LZO, Deflate, etc.), digestión (SHA1, MD5, etc.), entre otros.

5.3. Descripción de las pruebas

Lo resultados presentados en esta sección se obtuvieron utilizando un equipo con un procesador Intel(R) Xeon(R) E5502 de 1.87GHz, con 4 GB de memoria RAM, una tarjeta NVIDIA Tesla C2050 y con la distribución de GNU/Linux Fedora 17 instalada. Las pruebas consistieron en el cifrado y descifrado de un número variable de páginas (unidad básica de manejo de memoria en sistemas operativos) de tamaño de 4096 *bytes*, empleando llaves de 128, 192 y 256 *bits* en los siguientes contextos: con la implementación del modelo descrito en el capítulo 4 y de AES (ECB) descrita en la sección 3.2.4 (denominado cuda_helper), utilizando gdev en conjunto también con la implementación de AES (ECB) en CUDA (denominado gdev) y utilizando el *Crypto* API de GNU/Linux (denominado CPU).

Las gráficas presentadas más adelante muestran en el eje de las x el número de páginas

cifradas o descifradas, en el eje de las y se presenta el tiempo total (t) en microsegundos que tomó el proceso en los diferentes contextos; es importante mencionar que en cada contexto el tiempo total (t) se obtuvo a través de la suma de diferentes tiempos parciales de acuerdo a lo que se presenta en la tabla 5.1.

Contexto.	**Tiempos parciales.**
CPU	Ejecución de la función crypto_blkcipher_encrypt o crypto_blkcipher_decrypt.
gdev	Copia de la llave derivada en AES hacia el GPU. Copia de datos hacia el GPU. Ejecución del *kernel* de CUDA de cifrado o descifrado. Copia de datos hacia la memoria RAM.
cuda_helper	Envío de mensaje AESHLP_CMD_ENC o AESHLP_CMD_DEC hacia el proceso *"helper"*. Derivación de la llave utilizada en AES. Copia de la llave derivada de AES y de los datos hacia el GPU. Ejecución del *kernel* de CUDA de cifrado o descifrado. Copia de datos hacia la memoria RAM. Envío de mensaje AESHLP_CMD_DONE.

Tabla 5.1: Tiempos parciales considerados.

Aun cuando son más los tiempos parciales considerados en el contexto de cuda_helper, se tiene plena confianza en que esta implementación tendrá un mejor rendimiento que los otros dos contextos debido a que el controlador proporcionado por NVIDIA es empleado y este presenta un mejor rendimiento.

Las siguientes gráficas muestran los resultados obtenidos, en cada una de ellas el comportamiento de cada contexto es muy similar, es decir, para un número pequeño de páginas el contexto del CPU tiene un mejor rendimiento, lo anterior es debido a que tanto gdev como cuda_helper tienen que hacer una copia de la llave y los datos hacia el GPU previo a la ejecución del *kernel* de cifrado o descifrado. Tanto gdev como cuda_helper tienen un mejor rendimiento conforme el número de páginas empieza a aumentar, sin embargo, este último contexto llega a ser hasta entre 4 y 7 veces menor cuando el número de páginas llega a 1024.

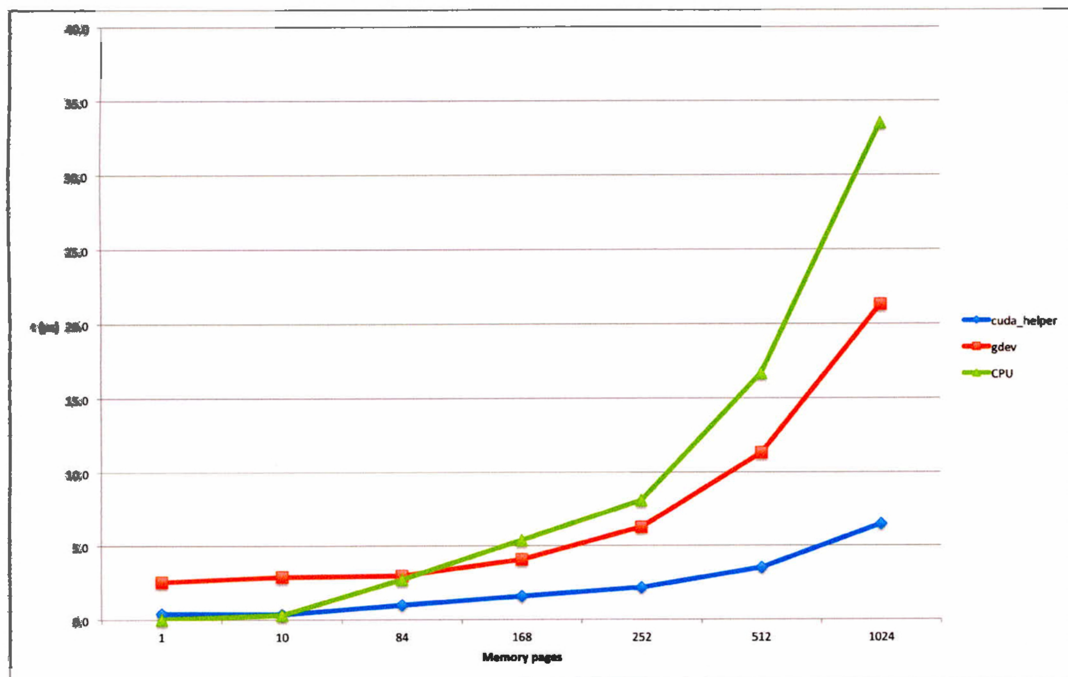


Figura 5.1: Cifrado (128 *bits*).

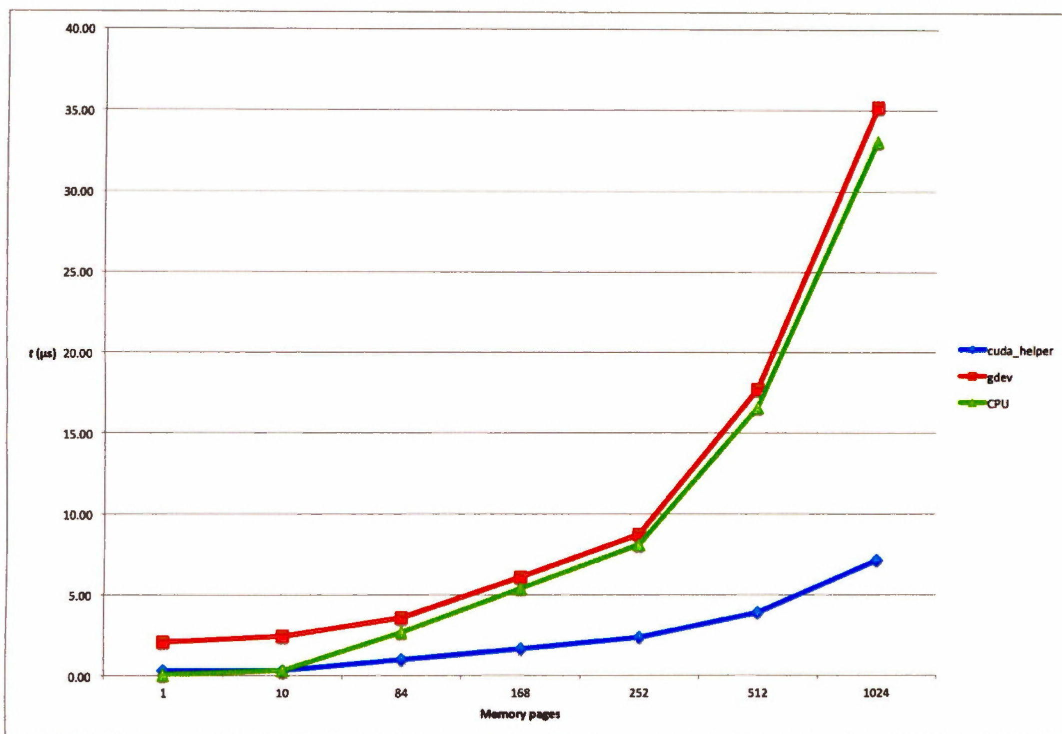


Figura 5.2: Descifrado (128 *bits*).

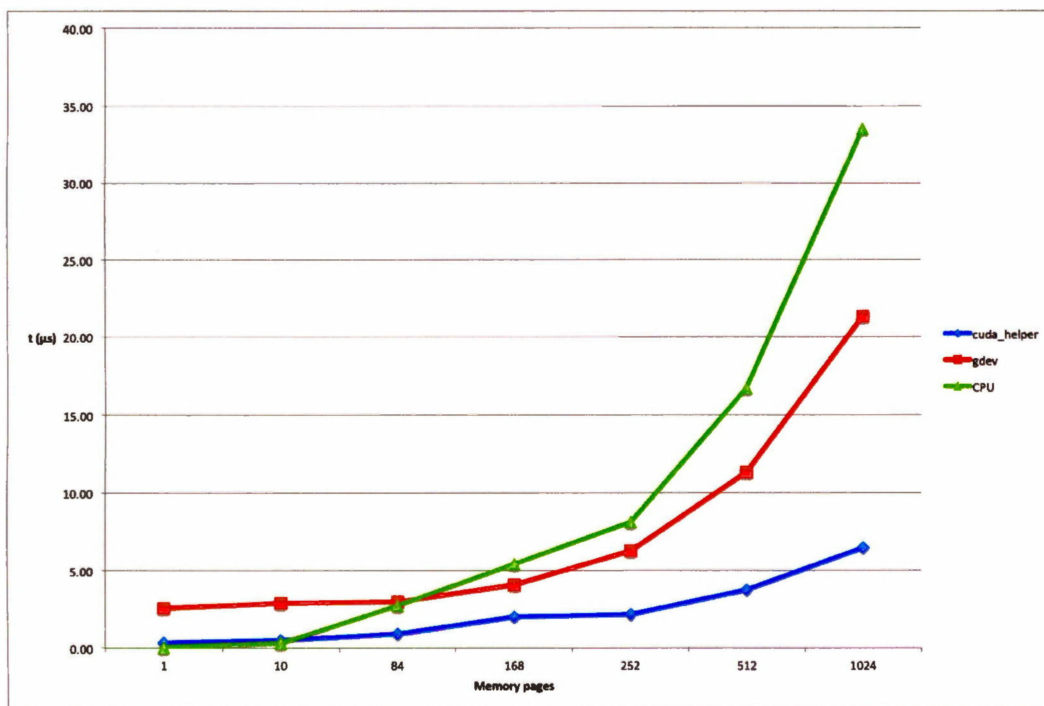


Figura 5.3: Cifrado (192 *bits*).

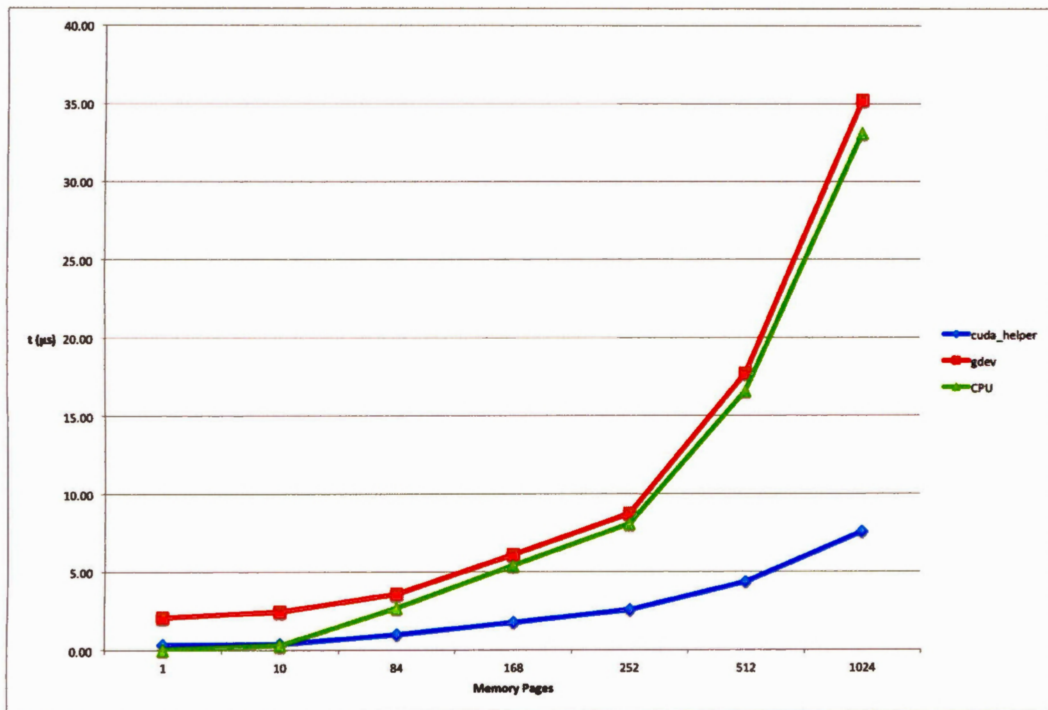


Figura 5.4: Descifrado (192 *bits*).

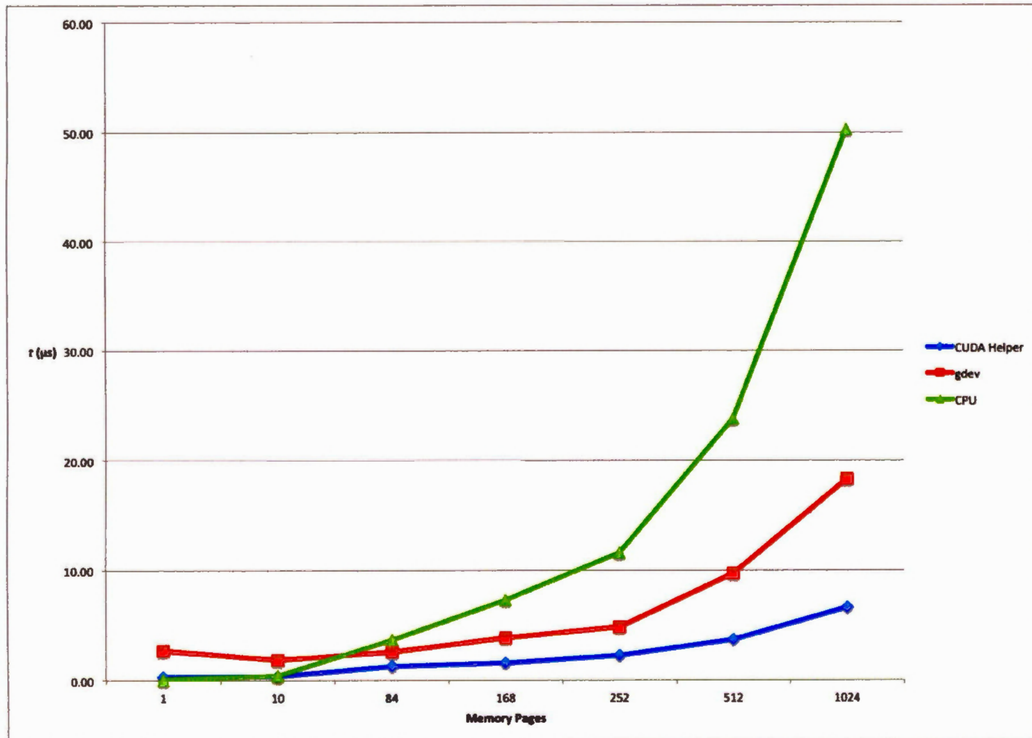


Figura 5.5: Cifrado (256 *bits*).

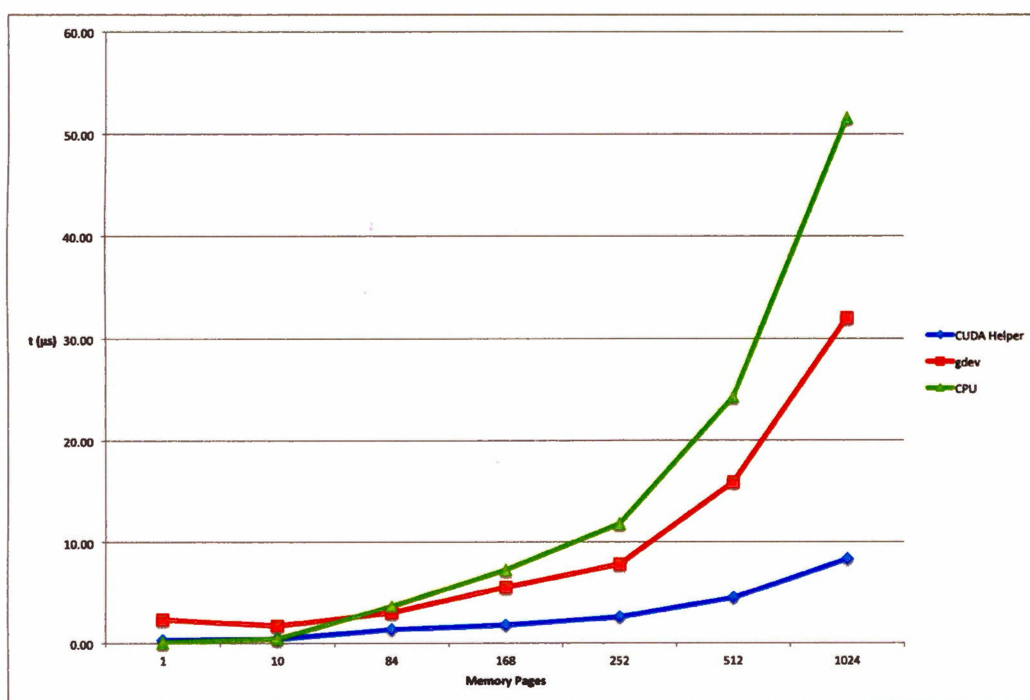


Figura 5.6: Descifrado (256 *bits*).

Capítulo 6

Conclusiones y Trabajo a Futuro

Con el modelo implementado y los resultados presentados en el capítulo anterior, no solamente se ha demostrado que es factible acelerar el proceso de cifrado y descifrado con el algoritmo AES dentro del núcleo de GNU/Linux (empleando de manera indirecta la biblioteca de CUDA con mecanismos como mmap y *Generic Netlink*), sino que de hecho es posible alcanzar un rendimiento entre 4 y 7 veces mejor si se hace una comparación con las pruebas realizadas con el *Crypto* API (CPU) y Gdev en conjunto con el controlador nouveau.

De hecho se puede concluir que el rendimiento que se obtuvo, se debe principalmente a tres cosas: la primera es el buen rendimiento que el controlador propietario de NVIDIA proporciona en comparación con otros (e.j. nouveau), la segunda es la optimización parcial que se llevó a cabo en la implementación del algoritmo AES (ECB) en CUDA y por último el mapeo de memoria que se hizo con mmap; esto último debido a que permitió reducir la latencia que se hubiera producido al querer copiar los datos desde el núcleo del sistema operativo hacia el GPU.

Como se mencionó al inicio de esta tesis, no existe mucha investigación relacionada con el uso del GPU como una unidad de aceleración por parte del núcleo de algún sistema operativo y de hecho los artículos consultados no proporcionan detalles técnicos de cómo llevar

a cabo lo anterior, es por eso que podemos decir que los mecanismos empleados en el modelo implementado representan una muy buena solución al problema planteado originalmente en este trabajo de investigación.

Finalmente, es importante enfatizar que los resultados obtenidos en este estudio provienen de un prototipo y que un mejor rendimiento puede ser alcanzado si una optimización más exhaustiva es aplicada a cada uno de los componentes del modelo implementado, de hecho la optimización mencionada anteriormente constituye solamente una de las áreas de oportunidad para un trabajo futuro; entre otras podemos encontrar: las modificaciones que se tendrían que llevar a cabo para que el modelo permitiera cifrar un número mayor y variable de páginas, la adición de nuevas tareas del núcleo del sistema operativo que puedan ser paralelizables y que de hecho podrían ser fácilmente agregadas debido a la flexibilidad que el modelo implementado proporciona (solamente sería necesario sustituir el *kernel* de cifrado por algún otro); finalmente, debido a que el controlador del GPU se encuentra dentro del mismo núcleo de GNU/Linux sería factible pensar en eliminar el proceso *"helper"* que se encuentra en el espacio del usuario y permitir que un módulo dentro del núcleo empleara funciones de bajo nivel en el GPU para utilizarlo como unidad de aceleración.

Bibliografía

[1] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[2] R. Bolla and R. Bruschi. An effective forwarding architecture for smp linux routers. In *Telecommunication Networking Workshop on QoS in Multiservice IP Networks, 2008. IT-NEWS 2008. 4th International*, pages 210–216, 2008.

[3] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[4] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[5] Abdullah Gharaibeh, Samer Al-Kiswany, Sathish Gopalakrishnan, and Matei Ripeanu. A gpu accelerated storage system. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 167–178, New York, NY, USA, 2010. ACM.

[6] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: A gpu-accelerated software router. *SIGCOMM Comput. Commun. Rev.*, 40(4):195–206, August 2010.

[7] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class gpu resource management in the operating system. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 37–37, Berkeley, CA, USA, 2012. USENIX Association.

[8] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[9] David Luebke and Greg Humphreys. How gpus work. *Computer*, 40(2):96–100, 2007.

[10] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, July 2013.

[11] Christof Paar and Jan Pelzl. *Understanding cryptography: A Textbook for Students and Practitioners.* Springer, 2010.

[12] Yuan Qingbo, Bao Yungang, Chen Mingyu, and Sun Ninghui. A scalability analysis of the symmetric multiprocessing architecture in multi-core system. In *Proceedings of the 2009 IEEE International Conference on Networking, Architecture, and Storage*, NAS '09, pages 231–234, Washington, DC, USA, 2009. IEEE Computer Society.

[13] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. Ptask: Operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[14] Weibin Sun and Robert Ricci. Augmenting operating systems with the gpu. *CoRR*, abs/1305.3345, 2011.

[15] Weibin Sun, Robert Ricci, and Matthew L. Curry. Gpustore: harnessing gpu computing for storage systems in the os kernel. In *Proceedings of the 5th Annual International*

Systems and Storage Conference, SYSTOR '12, pages 9:1–9:12, New York, NY, USA, 2012. ACM.

[16] Andrew S Tanenbaum and Albert S Woodhull. *Operating Systems Design and Implementation (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

Apéndice A

Vector Rcon

```
uint8_t Rcon[255] = {
0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,
0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,
0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d,
0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab,
0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25,
0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01,
0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d,
0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa,
0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a,
0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02,
0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,
0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,
0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
```

```
0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f,
0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5,
0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33,
0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb};
```

Apéndice B

Suma y Multiplicación en el Campo de *Galois*

El resultado de sumar dos números en el campo de *Galois* es muy simple, por ejemplo si queremos obtener la suma s de dos números a y b, entonces solamente es necesario aplicarle la operación XOR a los dos operandos, es decir:

$$s = a \oplus b$$

La multiplicación de dos números es un proceso más complejo, el algoritmo para obtener dicho resultado se muestra a continuación:

- El procedimiento toma como argumentos dos números de ocho *bits*, a y b. El resultado de la multiplicación se almacena en una variable también de ocho *bits* llamada p.
- La variable p se iguala a cero.
- El siguiente bloque de instrucciones se ejecuta ocho veces:
  - Si el *bit* menos significativo de b es 1 entonces aplicar la operación XOR a la

variable a y al producto p, el resultado de la operación es almacenado nuevamente en p.

- Almacenar el valor del *bit* más significativo de a en una variable temporal *tmp*.
- Rotar a un *bit* hacia la izquierda, descartando el *bit* más significativo y haciendo el *bit* menos significativo cero.
- Si el valor de la variable *tmp* es 1, entonces aplicar la operación XOR a la variable a con el valor 0x1b y almacenar el resultado en a.
- Rotar b un *bit* hacia la derecha, descartando el *bit* menos significativo y haciendo el *bit* más significativo cero.

- Regresar el valor de p.

Apéndice C

Derivación de la Subllave en AES

A continuación se muestran las funciones (utilizadas en la implementación de esta tesis) que llevan a cabo el proceso de la derivación de la subllave utilizada en AES (ECB). Es importante mencionar que estas funciones fueron implementadas de acuerdo a lo explicado en la sección 3.2.2.3.

```
void expand128_192Key(uint32_t *key, uint32_t *expandedKey, int keysize, int expandedkeysize) {
        int rcon=1, i=0;
        uint32_t t=0;

        for(i=0; i<keysize; i++) expandedKey[i] = key[i];

        for(i=keysize; i<expandedkeysize; i++) {
                t = expandedKey[i-1];
                if( i % keysize == 0 ) coreKeySchedule(&t, rcon++);
                expandedKey[i] = t ^ expandedKey[i-keysize];
        }
}

void expand192BitKey(uint32_t *key, uint32_t *expandedKey) {
```

```
        expand128_192Key(key, expandedKey, 6, 52);
}

void expand128BitKey(uint32_t *key, uint32_t *expandedKey) {
        expand128_192Key(key, expandedKey, 4, 44);
}

void expand256BitKey(uint32_t *key, uint32_t *expandedKey) {
        int rcon=1, i=0;
        uint32_t t=0;

        for(i=0; i<8; i++) expandedKey[i] = key[i];

        for(i=8; i<60; i++) {
                t = expandedKey[i-1];
                if(i % 8 == 0) coreKeySchedule(&t, rcon++);
                else if(i % 4 == 0) sboxWord(&t, 4);
                expandedKey[i] = t ^ expandedKey[i-8];
        }
}
```

Apéndice D

Implementación de AES(ECB) en CUDA

```
#include <inttypes.h>

#define FIRST_ROW(x) ((x&r0)>>24)
#define SECOND_ROW(x) ((x&r1)>>16)
#define THIRD_ROW(x) ((x&r2)>>8)
#define FORTH_ROW(x) (x&r3)
#define WORD_TOGETHER(x,y,z,w) (((x)<<24)|((y)<<16)|((z)<<8)|w)

#ifndef PAGE_SIZE
#define PAGE_SIZE 4096
#endif

__device__ __constant__ uint8_t sbox[256] = {
//0   1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
```

```
0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 }; //F


__device__ __constant__ uint8_t rsbox[256] = {
//0 1 2 3 4 5 6 7 8 9 A B C D E F
0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb, //0
0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb, //1
0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e, //2
0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25, //3
0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92, //4
0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84, //5
0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06, //6
0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b, //7
0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73, //8
0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e, //9
0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b, //A
0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4, //B
0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f, //C
0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef, //D
0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61, //E
```

```
0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d }; //F

/*
 * Masks to obtain rows 0, 1, 2 and 3 in the ShiftRows Operation
 */
#define r0 0xff000000
#define r1 0x00ff0000
#define r2 0x0000ff00
#define r3 0x000000ff

/*
* The algorithm for multiplication in the Galois' field
* was partially borrowed from http://www.samiam.org/galois.html
*/

 __forceinline__ __device__ uint32_t gmul(uint32_t a, uint32_t b) {
        uint32_t p = 0;
        uint32_t i;
        uint32_t hi_bit_set;

        for(i = 0; i < 8; i++) {
                if((b & 1) == 1)
                        p ^= a;
                hi_bit_set = (a & 0x80);
                a = (a << 1) & 0x000000ff;
                if( hi_bit_set == 0x80)
                        a ^= 0x1b;
                b = (b >> 1) & 0x000000ff;
        }
        return p;
}
```

```
/*
 * The MixColumsCore operation operates on each one of the columns of the state.
 * This code was partially borrowed from http://www.samiam.org/mix-column.html,
 * it does not completely implement a regular multiplication matrix, it
 * has been optimized to perform the MixColumns operation.
 */

__forceinline__ __device__ void MixColumnsCore(uint32_t *state) {
        uint32_t a[4], b[4];

        //First column of the state
        a[0] = FIRST_ROW(*state); a[1] = SECOND_ROW(*state);
        a[2] = THIRD_ROW(*state); a[3] = FORTH_ROW(*state);

        b[0] = gmul(a[0], 2); b[1] = gmul(a[1], 2);
        b[2] = gmul(a[2], 2); b[3] = gmul(a[3], 2);

        *state = WORD_TOGETHER( b[0]^a[3]^a[2]^b[1]^a[1],
                                b[1]^a[0]^a[3]^b[2]^a[2],
                                b[2]^a[1]^a[0]^b[3]^a[3],
                                b[3]^a[2]^a[1]^b[0]^a[0]);
}

/*
 * MixColumns operation (argument: the whole state)
 */

__forceinline__ __device__ void MixColumns(uint32_t *state) {
        // The function operates on each one of the columns of the state
        MixColumnsCore(state);
```

```
        MixColumnsCore(state+1);
        MixColumnsCore(state+2);
        MixColumnsCore(state+3);
}

/*
 * Function useful to rotate a 32 bit word to the left
 */
__forceinline__ __device__ void rotate(uint32_t *word, int positions) {
        int i=0;
        uint32_t mask=0xFF000000, tmp=0x0;

        for(i=0; i<positions; i++) {
                tmp = *word & mask;
                *word = *word << 8;
                tmp = tmp >> 24;
                *word |= tmp;
        }
}

/*
 * Subbytes operation
 * Input: the state
 * Output: the state with the SubBytes operation carried out on it
 */
__forceinline__ __device__ void SubBytes(uint32_t *state) {
        int i=0;

        for(i=0; i<4; i++)
        state[i] = WORD_TOGETHER(sbox[FIRST_ROW(state[i])],sbox[SECOND_ROW(state[i])],sbox[
            THIRD_ROW(state[i])],sbox[FORTH_ROW(state[i])]);
```

```
}

__forceinline__ __device__ void ShiftRows(uint32_t *state) {
	uint32_t row = 0x0;

	//Second Row
	row = WORD_TOGETHER(SECOND_ROW(state[0]),SECOND_ROW(state[1]),SECOND_ROW(
		state[2]),SECOND_ROW(state[3]));
	rotate(&row, 1);
	state[0] = ((row & r0) >> 8) | (state[0] & ~r1);
	state[1] = (row & r1) | (state[1] & ~r1);
	state[2] = ((row & r2) << 8) | (state[2] & ~r1);
	state[3] = ((row & r3) << 16) | (state[3] & ~r1);

	//Third Row
	row = WORD_TOGETHER(THIRD_ROW(state[0]),THIRD_ROW(state[1]),THIRD_ROW(state
		[2]),THIRD_ROW(state[3]));
	rotate(&row, 2);
	state[0] = ((row & r0) >> 16) | (state[0] & ~r2);
	state[1] = ((row & r1) >> 8) | (state[1] & ~r2);
	state[2] = (row & r2) | (state[2] & ~r2);
	state[3] = ((row & r3) << 8) | (state[3] & ~r2);

	//Fourth row
	row = WORD_TOGETHER(FORTH_ROW(state[0]),FORTH_ROW(state[1]),FORTH_ROW(state
		[2]),FORTH_ROW(state[3]));
	rotate(&row, 3);
	state[0] = ((row & r0) >> 24) | (state[0] & ~r3);
	state[1] = ((row & r1) >> 16) | (state[1] & ~r3);
	state[2] = ((row & r2) >> 8) | (state[2] & ~r3);
	state[3] = ( row & r3 ) | (state[3] & ~r3);
```

```
}

/*
 * The AddRoundKey operation combines, with the XOR operation, the state with the
 * current round key. It operates on the entire state.
 */

__forceinline__ __device__ void AddRoundKey(uint32_t *state, uint32_t *roundKey) {
        state [0] ^= *roundKey;
        state [1] ^= *(roundKey+1);
        state [2] ^= *(roundKey+2);
        state [3] ^= *(roundKey+3);
}

/*
 * This function just copies the local state to the global state
 */
__forceinline__ __device__ void put_state(uint32_t *state, uint8_t *global_state) {
        int i=0;
        for(i=0; i<4; i++) {
                global_state [0] = FIRST_ROW(state[i]);
                global_state [1] = SECOND_ROW(state[i]);
                global_state [2] = THIRD_ROW(state[i]);
                global_state [3] = FORTH_ROW(state[i]);
                global_state +=4;
        }
}

/*
 * This function just copies the global state to the local state
 */
```

```
__forceinline__ __device__ void get_state(uint32_t *state, uint8_t *global_state) {
        int i=0;
        for(i=0; i<4; i++) {
                state[i] = WORD_TOGETHER(global_state[0],global_state[1],
                                        global_state [2], global_state [3]) ;
                global_state +=4;
        }
}

/*
 * Kernel used to cipher data, it receives the following parameters:
 * The expanded key (performed in CPU)
 * The data to be encrypted
 * The number of rounds to be performed
 */
extern "C" __global__ void cypher(uint32_t *expandedKey, uint8_t *global_state, int rounds) {
        // The expanded key will be 60 columns (max)
        __shared__ uint32_t roundKey[60];

        /* Each thread will work with 4 elements */
        int idx = 16*(blockIdx.x*blockDim.x*blockDim.y+(blockDim.x*threadIdx.y+threadIdx.x));
        int i = 0;
        uint32_t state [4];

        // Copy the state from global memory
        get_state(state, global_state+idx);

        // The first thread of each block will copy the expanded key to shared mem
        if(idx %PAGE_SIZE==0) for(i=0; i<((rounds+1)*16*8)/32; i++) roundKey[i] = expandedKey[i];

        // We wait for all the other threads
```

```
        __syncthreads();

        //Initial round, AddRoundKey
        AddRoundKey(state, roundKey);

        //Then the number of rounds-1 are carried out (according to the key size),
        //It's rounds-1 since there's a final round in the process
        for(i=1; i<rounds; i++) {
                //The SubBytes operation has to be performed in every
                //Column of the state, so the MixColumns Operation
                SubBytes(state);
                ShiftRows(state);
                //The MixColumns operation is applied to each column in the
                //state
                MixColumns(state);
                //AddRoundKey
                AddRoundKey(state, roundKey+4*i);
        }

        //The final round, SubBytes, ShiftRows and AddRoundKey
        SubBytes(state);
        ShiftRows(state);
        //Final AddRoundKey
        AddRoundKey(state, roundKey+4*rounds);
        //Finally, the state has to be copied back to global memory
        put_state(state, global_state+idx);
}

/*
 * Decryption methods
 */
```

```
/*
 * Function to rotate a word to the right
 */

__forceinline__ __device__ void rotateRight(uint32_t *word, int positions) {
        int i=0;
        uint32_t mask=0x000000FF, tmp=0x0;

        for(i=0; i<positions; i++) {
                tmp = *word & mask;
                *word = *word >> 8;
                tmp = tmp << 24;
                *word |= tmp;
        }
}

/*
 * Inverse SubBytes method
 */

__forceinline__ __device__ void invSubBytes(uint32_t *state) {
        int i=0;

        for(i=0; i<4; i++)
        state[i] = WORD_TOGETHER(rsbox[FIRST_ROW(state[i])],rsbox[SECOND_ROW(state[i])],
             rsbox[THIRD_ROW(state[i])],rsbox[FORTH_ROW(state[i])]);
}

/*
 * Inverse ShiftRows method
```

```
*/

__forceinline__ __device__ void invShiftRows(uint32_t *state) {
        uint32_t row = 0x0;

        //Second Row
        row = WORD_TOGETHER(SECOND_ROW(state[0]),SECOND_ROW(state[1]),SECOND_ROW(
              state[2]),SECOND_ROW(state[3]));
        rotateRight(&row, 1);
        state [0] = ((row & r0) >> 8) | (state[0] & ~r1);
        state [1] = (row & r1) | (state [1] & ~r1);
        state [2] = ((row & r2) << 8) | (state[2] & ~r1);
        state [3] = ((row & r3) << 16) | (state[3] & ~r1);

        //Third Row
        row = WORD_TOGETHER(THIRD_ROW(state[0]),THIRD_ROW(state[1]),THIRD_ROW(state
              [2]),THIRD_ROW(state[3]));
        rotateRight(&row, 2);
        state [0] = ((row & r0) >> 16) | (state[0] & ~r2);
        state [1] = ((row & r1) >> 8) | (state[1] & ~r2);
        state [2] = (row & r2) | (state [2] & ~r2);
        state [3] = ((row & r3) << 8) | (state[3] & ~r2);

        //Fourth row
        row = WORD_TOGETHER(FORTH_ROW(state[0]),FORTH_ROW(state[1]),FORTH_ROW(state
              [2]),FORTH_ROW(state[3]));
        rotateRight(&row, 3);
        state [0] = ((row & r0) >> 24) | (state[0] & ~r3);
        state [1] = ((row & r1) >> 16) | (state[1] & ~r3);
        state [2] = ((row & r2) >> 8) | (state[2] & ~r3);
        state [3] = ( row & r3 ) | (state [3] & ~r3);
```

```
}

/*
 * Inverse MixColumn Core Operation
 * This is the core of the inverse MixColumn operation, it affects one column
 * at a time
 */
__forceinline__ __device__ void invMixColumnsCore(uint32_t *state) {
        uint32_t a[4];

        a[0] = FIRST_ROW(*state); a[1] = SECOND_ROW(*state);
        a[2] = THIRD_ROW(*state); a[3] = FORTH_ROW(*state);

        *state = WORD_TOGETHER( gmul(a[0],14) ^ gmul(a[3],9) ^ gmul(a[2],13) ^ gmul(a[1],11),
                                gmul(a[1],14) ^ gmul(a[0],9) ^ gmul(a[3],13) ^ gmul(a[2],11),
                                gmul(a[2],14) ^ gmul(a[1],9) ^ gmul(a[0],13) ^ gmul(a[3],11),
                                gmul(a[3],14) ^ gmul(a[2],9) ^ gmul(a[1],13) ^ gmul(a[0],11));
}

/*
 * Inverse Mix Column operation, it operates on the whole state
 */
__forceinline__ __device__ void invMixColumns(uint32_t *state) {
        invMixColumnsCore(state);
        invMixColumnsCore(state+1);
        invMixColumnsCore(state+2);
        invMixColumnsCore(state+3);
}

extern "C" __global__ void decypher(uint32_t *expandedKey, uint8_t *global_state, int rounds) {
        // To copy the expanded key to shared memory
```

```
__shared__ uint32_t roundKey[60];

/* Each thread will work with 4 elements */
int idx = 16*(blockIdx.x*blockDim.x*blockDim.y+(blockDim.x*threadIdx.y+threadIdx.x));
int i=0;
uint32_t state [4];

/* We copy the state to local memory */
get_state(state, global_state+idx);

//The first thread of each block copies the expanded key
if(idx %PAGE_SIZE==0) for(i=0; i<((rounds+1)*16*8)/32; i++) roundKey[i] = expandedKey[i];

//We wait for all the other threads
__syncthreads();

//The operations have to be performed in reverse order,
//according to the diagram shown here:
//http://www.cs.bc.edu/~straubin/cs381-05/blockciphers/rijndael_ingles2004.swf
//AddRound Key (the inverse operation for the XOR operation is the XOR operation)
AddRoundKey(state, roundKey+rounds*4);
invShiftRows(state);
invSubBytes(state);

//The use of the expanded key is backwards
for(i=rounds-1; i>0; i--) {
        AddRoundKey(state, roundKey+i*4);
        invMixColumns(state);
        invShiftRows(state);
        invSubBytes(state);
}
```

```
    AddRoundKey(state, roundKey);

    //Copying back the local state to the global state
    put_state(state, global_state+idx);
}
```

Apéndice E

Instalación de Gdev

El proceso de compilación e instalación para poder utilizar gdev dentro del núcleo de GNU/-Linux se explica a continuación, todos los comandos han sido ejecutados desde una terminal.

El primer paso en la instalación consiste en compilar e instalar la versión 3.3.8 del núcleo de GNU/Linux (el código puede ser descargado en https://www.kernel.org/pub/linux/kernel/v3.x), pero antes de hacer esto es necesario desactivar el componente DRM *(Direct Rendering Manager)* entrando al menú de configuración del núcleo con los siguientes comandos:

```
$ make oldconfig
$ sudo make modules_install && sudo make install
```

En la interfaz de ncurses hay que ir a la opción de *Load an Alternate Configuration File* y especificar el nombre del archivo .config (este paso es para asegurar que la configuración que tenemos actualmente se tome como punto de partida). Para desactivar la opción de DRM es necesario entrar al menú *Device Drivers*, posteriormente a *Graphics Support* y por último desactivar la casilla referente a este componente. La compilación e instalación del nuevo núcleo se hace introduciendo los siguientes comandos en la terminal:

```
$ make -j3
$ sudo make modules_install && sudo make install
```

El siguiente paso en la instalación es reiniciar la computadora, en algunas distribuciones de GNU/Linux será necesario ejecutar comandos adicionales y editar algunos archivos de configuración para completar la instalación y que el nuevo núcleo sea el que inicie por defecto.

A continuación se deberá instalar el controlador nouveau para tarjetas de video de NVIDIA, afortunadamente dentro del directorio del código fuente de gdev está incluido también el código fuente de este controlador. La instalación del controlador está compuesta por los siguientes comandos:

```
$ cd gdev/mod/nouveau-3.0.0/
$ make NOUVEAUROOTDIR=.
$ sudo make install NOUVEAUROOTDIR=.
```

Al término de la instalación es necesario reiniciar la computadora y comprobar con el comando lsmod que el controlador nouveau haya sido cargado durante el proceso de inicialización, en caso de que no haya sido así es necesario hacer lo propio con el comando modprobe.

El último paso en este proceso de instalación consiste en instalar gdev como tal, pero primero el código fuente de este y las herramientas envytools deben ser descargadas utilizando git como se muestra enseguida:

```
$ git clone https://github.com/pathscale/envytools.git
$ git clone https://github.com/shinpei0208/gdev.git
```

La instalación de envytools consiste en introducir los comandos que se muestran a continuación en la terminal, para completar la instalación será necesario modificar las variables

de ambiente PATH y LD_LIBRARY_PATH.

```
$ cd envytools; mkdir build; cd build
$ cmake .. -D CMAKE_INSTALL_PREFIX=/usr/local/envytools
$ make && sudo make install
```

El último paso consiste en compilar e insertar los módulos de gdev y kcuda en el núcleo de GNU/Linux, para tal propósito los siguientes comandos deberián ser suficientes:

```
$ cd gdev/mod
$ mkdir build; cd build; ../configure
$ make
$ sudo insmod gdev.ko
$ sudo sh gdev.sh
$ cd ..
$ cd cuda
$ mkdir kbuild; cd kbuild; ../configure --target=kcuda
$ make
$ sudo insmod kcuda.ko
```

El proceso instalación de gdev descrito en este apéndice puede haber cambiado al momento al momento de la publicación de esta tesis, para una mejor referencia del proceso se puede consultar el archivo README incluido en el código fuente del proyecto.