UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
Posgrado en Ciencia e Ingeniería de la Computación
Instituto de Investigaciones en Matemáticas
Aplicadas y en Sistemas
Teoría de la Computación

Criptosistemas en GPUs

T E S I S
QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

P R E S E N T A:
Mauricio Daniel Garza Rauda

Director de Tesis
Dr. José David Flores Peñaloza
Facultad de Ciencias, UNAM

CIUDAD UNIVERSITARIA, CDMX, 2017

Datos de quien presenta: Garza Rauda Mauricio Daniel mdgr.vf **@**gmail.com Universidad Nacional Autónoma de México Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas Posgrado en Ciencia e Ingeniería de la Computación 303139447
Datos del tutor: Dr. José David Flores Peñaloza
Datos del sinodal 1: Dr. Carlos Bruno Velarde Velázquez
Datos del sinodal 2: Dr. Sergio Rajsbaum Gorodezky
Datos del sinodal 3: Dr. Luis Miguel de la Cruz Salas
Datos del sinodal 4: Dr. Armando Castañeda Rojano
Datos de la tesis: Criptosistemas en GPUs 66 p. 2017

A mi padre y madre.

Por el amor y cariño que me han dado,
así como por todo su apoyo en mi vida.

Agradecimientos

Agradezco todos los consejos que me dieron mis padres, sin ellos y sin su apoyo no podría estar aquí.

Doy gracias a la Universidad Nacional Autónoma de México, por permitirme ser parte de ella, también agradezco al Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas por ser permitirme vivir en sus aulas y pasillos durante mis estudios.

A mi tutor, por su guía y consejos. A mis sinodales por revisar este trabajo, y ayudar a que fuera mejor con sus comentarios.

Gracias CONACyT por brindarme un apoyo para completar mis estudios de maestría.

A todas las personas que conocí en este tiempo, gracias, por recordarme que todos los dias se aprende algo nuevo.

A los nuevos y viejos amigos, gracias por estar ahí y soportarme.

A Raúl y Aura por apoyar las diversas *misiones* y *operaciones* en los últimos años.

A Tania, por aceptarme en su mesa a pesar de las diferencias de opinioń.

Las revelaciones del camino de Juan son difíciles de seguir, gracias por mostrarmelas.

Al Snark, por las largas pláticas depúes de comer, recuerda que Viktor está allá afuera, esperando.

Erik, que aún me toleras como ayudante, y Vero, que siempre está riendo, gracias.

A toda la (Van)da, calen.

Gracias a todos.

"Decimos 'silla' pero no queremos decir 'silla', y nos entienden.
O por lo menos nos entienden aquellos a quienes
está secretamente destinado el mensaje, ***críptico****,*
pasando indemne a través de las multitudes indiferentes y hostiles."

Sobre héroes y tumbas, Ernesto Sabato.

Índice

Índice de tablas

Índice de figuras

Índice de algoritmos

1
Introducción

En esta tesis se estudian implementaciones en CUDA de algunos criptosistemas por bloques (RC2, IDEA y AES, siendo estos propuestas para reemplazar a DES como estándar [4]), y se comparan con sus respectivas implementaciones en CPU.

La motivación de implementar criptosistemas usando CUDA es tratar de ganar una mejora en el desempeño de las aplicaciones que requieren operaciones criptográficas, ya que como menciona Neves [16], cerca del 70 % del tiempo en una transacción mediante DNSSEC/HTTPS se consume en operaciones criptográficas, ya que cuando fueron diseñados estos criptosistemas, los paquetes de datos eran pequeños en comparación con los paquetes de datos comunes en la actualidad (en la década de 1970, IBM anunció su sistema de almacenamiento IBM 3330 que ofrecía 100Mb de almacenamiento por disco [1], mientras que en la actualidad el proyecto LSST genera 15 Terabytes por noche [8]).

1.1. Estructura de la tesis

Actualmente es común que las computadoras tengan algún tipo de GPU, y con la posibilidad de usar esta arquitectura para cómputo de propósito general se tiene como objetivo de esta tesis el estudiar y analizar algunos de los criptosistemas por bloque propuestos como sucesores de DES, contemplando al criptosistema ganador Rjindael [14], proponer una implementación en GPU y comparar los rendimientos de estos con sus respectivas implementaciones en CPU.

Las secciones restantes de este capítulo introducen algunos conceptos básicos, mientras que el resto de esta sección revisa rápidamente el contenido del resto de los capítulos.

En el capítulo 2 se ofrece un poco de contexto histórico sobre los cifradores por bloque (o criptosistemas por bloque/simétricos) y sobre DES, así como sus modos de operación definidos en [3].

Los capítulos 3 a 5 presentan los criptosistemas RC2, IDEA y AES como son definidos en [17], [19] y [6] respectivamente.

En el capítulo 6 se analizan que etapas del cifrado o descifrado de los criptosistemas de pueden paralelizar.

El capítulo 7 revisa algunos detalles de implementación, así como los datos resultantes de las ejecuciones y sus respectivas comparaciones.

El capítulo 8 presenta conclusiones del trabajo realizado.

1.2. Criptografía

La palabra **criptografía** se deriva del griego para "escritura secreta", lo que en la actualidad generalmente implica que la información secreta o sensible es convertida de una forma legible a una forma no legible. La forma legible de la información es llamada **texto plano** o **texto en claro**, mientras que la forma no legible es llamada **texto cifrado**.

El proceso que toma un texto en claro y lo convierte en texto cifrado se conoce como **cifrado**, al proceso de convertir texto cifrado en su correspondiente texto en claro (proceso inverso del cifrado) se le conoce como **descifrado**. A la pareja de procesos, o algoritmos, de cifrado y descifrado correspondientes se les conoce como **criptosistema**.

Los algoritmos de un criptosistema suelen usar un valor secreto llamado **llave**, el conocimiento de este valor hace que el cifrado y el descifrado sean sencillos, pero el proceso de descifrado debe ser casi imposible sin el uso de la llave adecuada, al proceso de intentar encontrar una forma de descifrar un texto cifrado (no contemplada en el diseo del criptosistema) cuando no se conoce la llave se le conoce como **criptoanálisis** [18].

Se debe notar que un criptosistema no debería basar su seguridad en la no publicación de los algoritmos.

1.3. CUDA

Las unidades de procesamiento gráfico (GPU) originalmente fueron diseñadas para la generación y procesamiento de color para cada pixel en las pantallas de las computadoras mediante shaders. En general, un shader toma como entrada la posición (x, y) del pixel en la pantalla, junto con entradas adicionales y computa un nuevo color para el pixel. Eventualmente, algunos investigadores

se dieron cuenta de que estas entradas podían ser cualquier dato, explorando la posibilidad de usar GPU en cómputo de propósito general [13].

En el 2006, nVidia lanza una arquitectura llamada CUDA que permite ejecutar instrucciones entre CPU ("host") y GPU ("device"), dando una flexibilidad a los programadores para usar GPUs para cómputo general. Las instrucciones para el CPU son en ANSI C, mientras que para el GPU se extienden las instrucciones de C.

Desde su debut, las aplicaciones CUDA has gozado de beneficios que generalmente incluyen mejoras en el rendimiento en órdenes de magnitud sobre las mejores implementaciones en CPU en cuanto a tiempo de ejecución [13].

1.4. Trabajos previos

Investigaciones previas sobre el paralelismo en AES usando GPU [16] [11] presentan lo siguiente:

- Bobrov [11] encontró que la implementación de AES directa es mejor que la versión que usa tablas de sustitución ("look up tables") en un 10 %, ya que estas tablas requieren un uso excesivo de la operación xor para el cifrado,
- sólo se han comparado versiones que usan paralelismo trivial (capítulo 6),
- sólo presentan resultados sobre el algoritmo de cifrado, pero ya que AES no se trata de una red de Feistel también es importante el descifrado.

IDEA estuvo mucho tiempo bajo patente (hasta 2012), lo cual restringió su uso, además de que junto con RC2 no fueron aceptados como el nuevo estándar, lo que hace que su estudio sea meramente académico.

2

Sobre criptosistemas por bloque y DES

Hay dos tipos principales de algoritmos de cifrado y descifrado, o criptosistemas, los de llave secreta (o simétricos) y los de llave pública (o asimétricos).

Los criptosistemas de llave secreta son aquellos que usan la misma llave tanto para el cifrado como para el descifrado, o hay una pequeña transformación de la llave de cifrado en la llave de descifrado.

Los criptosistemas de llave pública son aquellos que requieren de dos llaves para su funcionamiento, la llave pública y la llave privada. La llave pública se disemina entre varios usuarios mientras que la privada sólo la conoce el "propietario", esto es con el fin de que cualquier usuario sea capaz de cifrar información usando la llave pública, pero sólo quien tiene la llave privada es capaz de descifrar los datos cifrados. La llave privada es difícil de obtener a partir de la pública.

2.1. Cifradores por bloque

"Los cifradores por bloque pueden ser de llave pública o privada [...] Un cifrador por bloque es una función que mapea bloques de texto en claro de n-bits a bloques de texto cifrado de n-bits; a n se le llama tamaño del bloque"[10], ésta función depende de un valor, llave, que se toma de un conjunto $\mathcal{K}$, llamado espacio de llaves. Para que dicha función sea viable debe ser invertible.

Definición 2.1.1. *Cifrador por bloque.* Un cifrador de bloques de n-bits es una función $C : V_n \times \mathcal{K} \to V_n$, donde V_n son vectores de tamaño n-bits, es decir los bloques. Para cada $k \in \mathcal{K}$, $C(B, k)$ o $C_k(B)$ es un mapeo invertible de V_n a V_n. El mapeo inverso D también está parametrizado por $\mathcal{K}$ y va de V_n a V_n, cumpliendo que $D_k(C_k(B)) = B$.

En un criptosistema por bloques se aplica el cifrador por bloque un total de r rondas, cada ronda usa una llave distinta.

2.2. DES

En la decada de 1970 IBM, inició un programa de investigación en criptografía dada la creciente necesidad de proteger información durante transacciones, sobre todo en las que había dinero involucrado [18].

Esto resultó en el diseño del criptosistema simétrico conocido como DES (Data Encryption Standard) [2] y su publicación como un estándar.

Este criptosistema fue diseñado para cifrar/descifrar bloques de 64 bits en 16 rondas, usando una llave especificada por 64 bits de los cuales sólo 56 bits son efectivos y los 8 restantes son bits de paridad. A continuación se explica brevemente el funcionamiento de DES.

A partir de los 56 bits efectivos de la llave, el espacio de llaves $\mathcal{K}$ generado consiste de 16 llaves (una para cada ronda) de 48 bits cada una.

Hay dos permutaciones de bits a destacar, llamadas IP y su inversa IP^{-1}. IP se aplica a la entrada antes de la primera ronda mientras que IP^{-1} se aplica después de la última ronda.

Para cifrar, en la ronda i se separa su entrada de 64 bits en dos paquetes de 32 bits cada uno, L y R siendo la entrada la secuencia de bits LR, el resultado de la ronda L' y R' se puede expresar como sigue

$$\begin{aligned} L' &= R \\ R' &= L \oplus f(R, k_i) \end{aligned}$$

donde $k_i \in \mathcal{K}$ es la subllave usada en la ronda. La función f realiza una expansión de los 32 bits de R a 48 bits, realiza permutaciones y sustituciones, y finalmente agrega la subllave de ronda usando la operación XOR. R' es calculado entonces mediante un XOR entre L y el resultado de la función f.

En el descifrado se realiza el mismo procedimiento que para el cifrado, pero la permutación inicial es IP^{-1} y la final es IP y las subllaves usadas se usan en orden inverso, es decir, en la primera ronda se usa la subllave k_{16}, en la segunda k_{15}, en la última k_1.

2.3. Modos de operación

Un cifrador por bloque provee seguridad en el cifrado para un sólo bloque de información, pero ¿qué pasa cuando el paquete de información a cifrar es mayor a n bits?. Si se aplica el mismo cifrador por bloque a todos los bloques existe la posibilidad de que en algún momento se cifre un bloque que ya se había cifrado

previamente, generando dos o más bloques de texto cifrado iguales, y de esta manera se puede dar información extra a un posible ataque del criptosistema.

Los modos de operación definen el proceso de cifrado para cuando hay más de un bloque a cifrar del mismo paquete de información. Los modos de operación están definidos en [3], a pesar de que fueron definidos para DES se pueden aplicar a cualquier criptosistema por bloques. Con la adopción de AES como nuevo estándar se incorporó un nuevo modo de operación en [15].

2.3.1. ECB

ECB (Electronic Code Book) es un modo que dada una llave, a cada bloque del texto en claro le asigna un único bloque del texto cifrado, como si se tratara de un diccionario.

Sea M_i el i-ésimo bloque a cifrar y N_i su respectivo cifrado, el cifrado usando ECB se define como

$$N_i = C_k(M_i)$$

mientras que el descifrado está dado por

$$M_i = D_k(N_i).$$

Es decir, el modo ECB es aplicar el cifrado por bloque sin modificaciones a cada bloque, por lo que se pueden aplicar varios cifrados a bloques en **paralelo**. La figura 2.1 muestra este modo de operación.

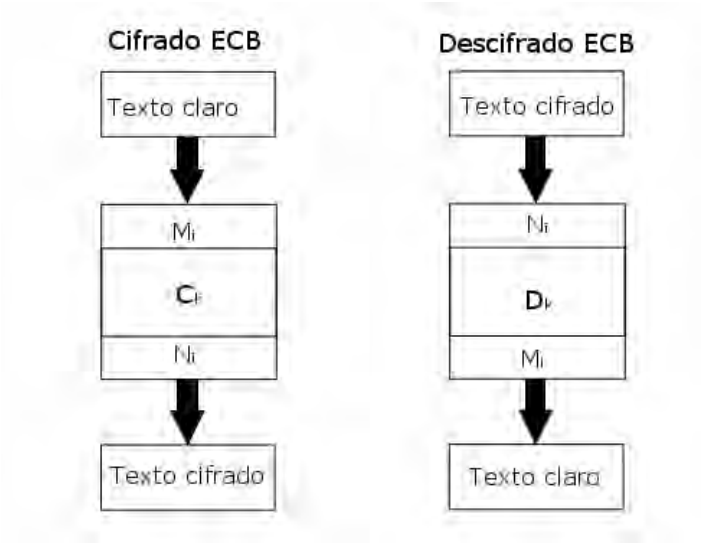

Figura 2.1: MODO: ECB

2.3.2. CBC

CBC (Cipher Block Chaining) es un modo que combina los bloques de texto claro con bloques de texto cifrado previos.

Para el primer bloque de texto claro se requiere de un vector de inicialización IV, este primer bloque se mezcla con IV usando la operación XOR y se realiza el cifrado a este nuevo bloque. El proceso anterior se repite pero ahora el resultado del cifrado anterior es usado como el nuevo IV para el siguiente bloque de texto claro hasta cifrar todo el paquete de información.

$$N_1 = C_k(M_1 \oplus IV)$$
$$N_i = C_k(M_i \oplus N_{i-1})$$

Para el descifrado, se le aplica la función de descifrado al primer bloque de texto cifrado y al resultado se le aplica la operación XOR con el vector IV, obteniendo así el primer bloque de texto claro. Los siguientes bloques se descifran usando la función de descifrado y al resultado se le aplica la operación XOR junto con el bloque de texto cifrado anterior, obteniendo el bloque de texto claro correspondiente.

$$M_1 = D_k(N_1) \oplus IV$$
$$M_i = D_k(N_i) \oplus C_{i-1}$$

En el modo CBC se necesita el resultado del cifrado del bloque i para cifrar el $i+1$ por lo que **no es paralelizable**, por otro lado en el descifrado se tienen todos los vectores necesarios desde el inicio, por lo que el descifrado se puede realizar en **paralelo**. La figura 2.2 muestra este modo de operación.

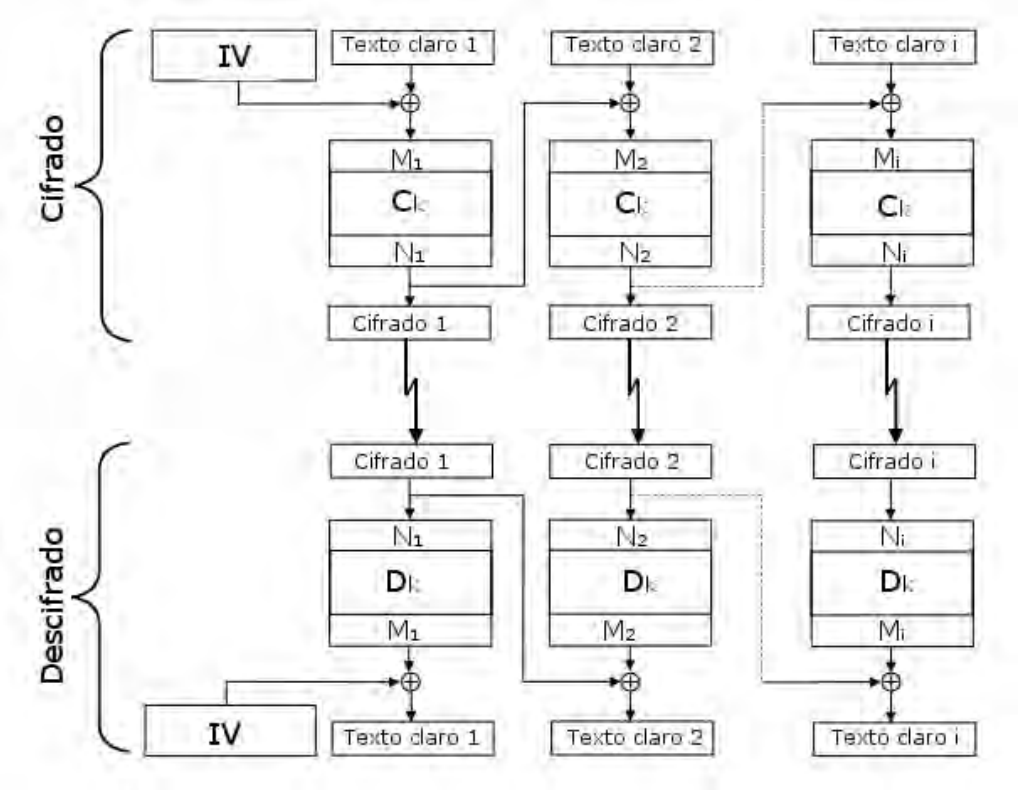

Figura 2.2: MODO: CBC

2.3.3. CFB

CFB (Cipher Feedback) es parecido a CBC en el sentido de usar los bloques anteriores para los nuevos cifrados. Se define un parámetro s, con $1 \leq s \leq t$, que indica cuantos bits se están pasando efectivamente como entrada para el nuevo bloque de texto a cifrar, suponiendo que el tamaño de los bloques a cifrar es de t bits.

La primer entrada a cifrar es el vector de inicialización IV, después de cifrado se seleccionan s bits y se mezclan con el primer bloque de texto en claro mediante

XOR obteniendo así el primer bloque de texto cifrado. El vector de entrada anterior se actualiza mediante un corrimiento a la izquierda de s posiciones y los s bits seleccionados de la etapa anterior se ponen en la parte derecha del vector de entrada, este nuevo vector se cifra, luego de cifrado se seleccionan sus s bits menos significativos y se mezclan con el siguiente bloque de texto en claro mediante XOR obteniendo un nuevo bloque de texto cifrado, este proceso se repite hasta cifrar todo el paquete de información.

$$\begin{aligned} I_1 &= IV \\ I_i &= I_{i-1}[s:t]^1 | N_{i-1} \\ N_i &= C_k(I_i)[1:s] \oplus M_i \end{aligned}$$

Para el descifrado, IV es la primera entrada y cada bloque de entrada se forma de la misma forma que en el cifrado usando la función de cifrado, obteniendo bloques cifrados de los cuales se toman los s bits que se usan para operar con los bloques de texto cifrado mediante la operación XOR.

$$\begin{aligned} I_1 &= IV \\ I_i &= I_{i-1}[s:t] | M_{i-1} \\ M_i &= C_k(I_i)[1:s] \oplus N_i \end{aligned}$$

Al igual que en CBC, el modo CFB necesita el resultado del cifrado del bloque i para cifrar el $i+1$ por lo que **no es paralelizable**, mientras que para el descifrado se tienen todos los vectores necesarios desde el inicio, por lo que el descifrado se puede realizar en **paralelo**. La figura 2.3 muestra este modo de operación.

[1] $A[x:y]$ representa tomar los bits de A en las posiciones x a y mientras que $|$ representa la concatenación de bits.

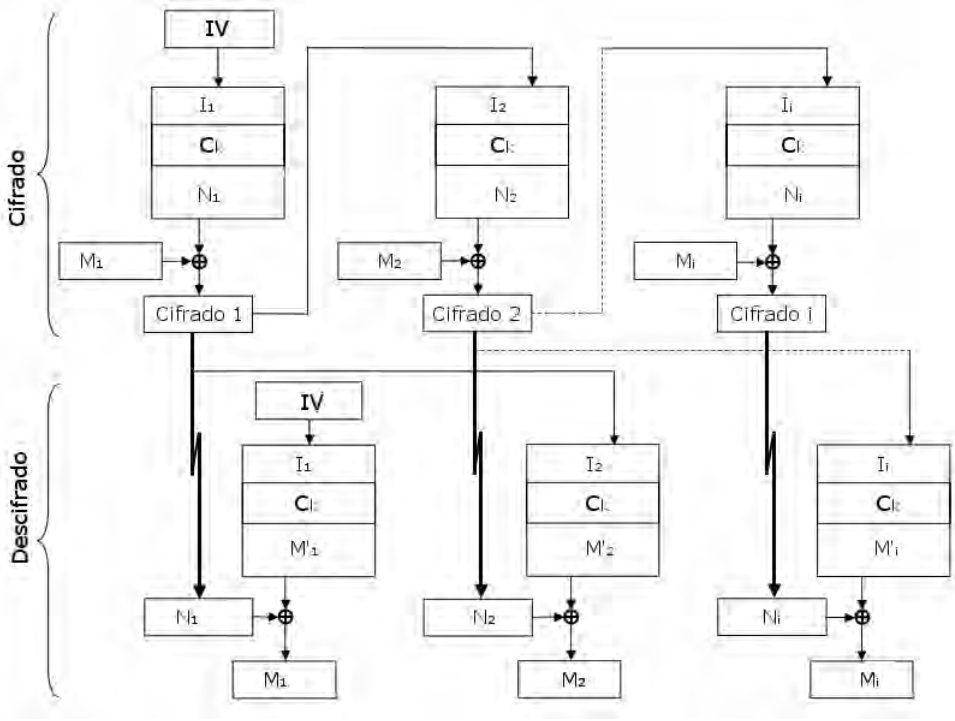

Figura 2.3: MODO: CFB

2.3.4. OFB

En el cifrado de OFB (Output Feedback) se toma el vector de inicialización IV como primera entrada y se cifra, el resultado cifrado será la entrada del cifrado de la siguiente etapa al mismo tiempo de ser operado con el primer bloque de texto claro mediante XOR para formar el primer bloque de texto cifrado. En la siguiente etapa se cifra la entrada, y el resultado se usa como entrada de la siguiente etapa así como operando para un XOR con el bloque de texto claro respectivo.

$$I_1 = IV$$
$$I_i = C_k(I_{i-1})$$
$$N_i = M_i \oplus C_k(I_i)$$

Para el descifrado se tiene el mismo proceso que para el cifrado, pero en vez de hacer XOR con el bloque de texto claro se hace con el bloque de texto cifrado.

$$I_1 = IV$$

$$I_i = C_k(I_{i-1})$$
$$M_i = N_i \oplus C_k(I_i)$$

En OFB, tanto el cifrado como el descifrado dependen de resultados de etapas anteriores por lo que **no son paralelizables** a menos que se precalcule el cifrado de IV y sus sucesivos cifrados. La figura 2.4 muestra este modo de operación.

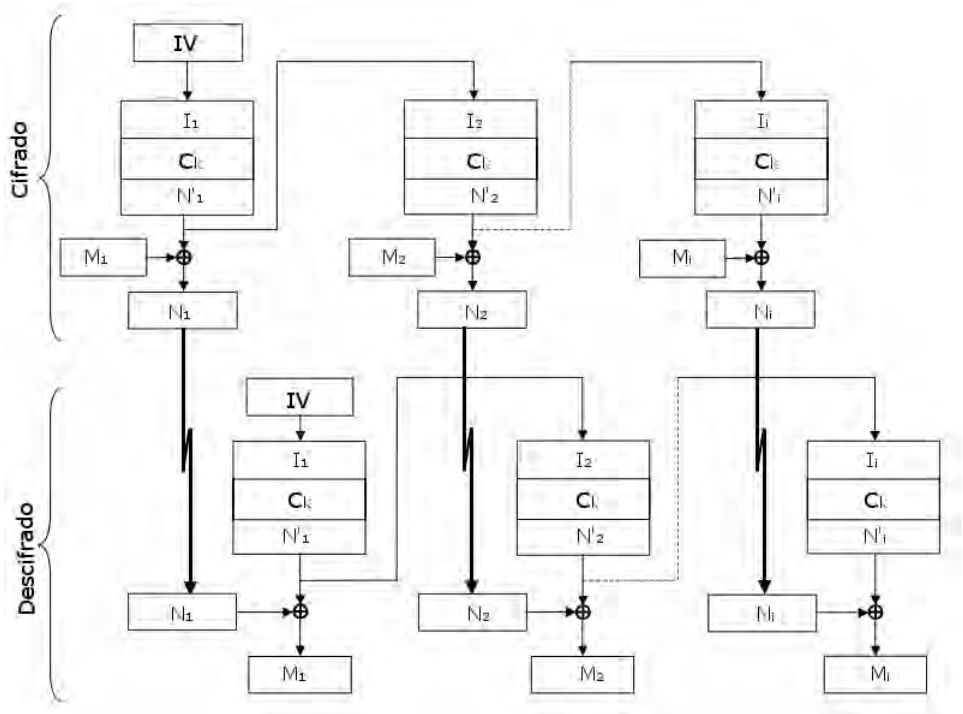

Figura 2.4: MODO: OFB

2.3.5. CTR

CTR (Counter) es un modo que requiere de un vector de inicialización distinto con relación a la llave usada en cada etapa llamado contador.

En el cifrado, se cifra el contador y el resultado se opera con el bloque de texto claro mediante XOR.

$$N_i = M_i \oplus C_k(Contador_i)$$

En el descifrado, se cifra el contador y el resultado se opera con el bloque de texto cifrado mediante XOR.

$$M_i = N_i \oplus C_k(Contador_i)$$

En el modo CTR no hay dependencia de cifrado/descifrado entre bloques, por lo que es **paralelizable**. La figura 2.5 muestra este modo de operación.

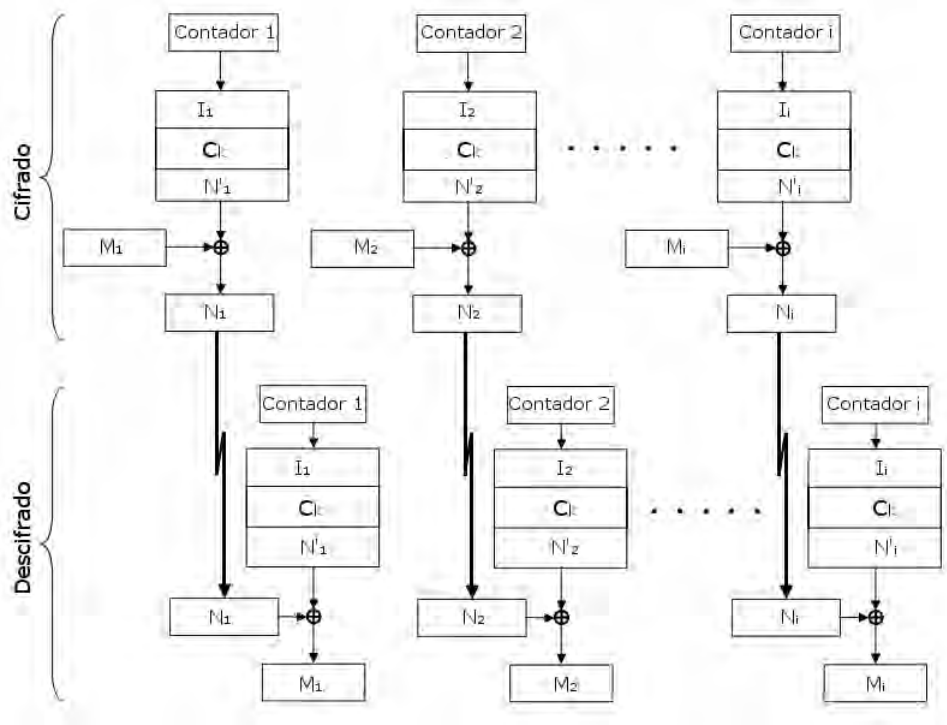

Figura 2.5: MODO: CTR

2.4. Sucesor de DES

Había muchas preocupaciones sobre el funcionamiento de DES, entre la velocidad de cifrado como en seguridad, a pesar de tener modificaciones (siendo la última propuesta en 1999 [5]) para mejorar el nivel de seguridad, en 1997 se inicia un proceso para seleccionar a los posibles candidatos para sustituir a DES como estándar [4].

Algunos de los criptosistemas propuestos fueron

- BlowFish
- NewDES
- RC2
- RC5
- IDEA
- Rijndael

La mayoría de las propuestas mantenían el tamaño del bloque en 64 bits para poder ser utilizados como reemplazos directos de DES.

En esta tesis se estudiaron y analizaron los criptosistemas RC2, IDEA y AES (Rjindael).

3
RC2

RC2 ("Ron's Code", "Rivest's Cipher") es un criptosistema de cifrado en bloque que usa llaves de tamaño variable propuesto por Rivest [17].

En RC2 a los grupos de 16 bits se les llama palabras.

En este algoritmo, como en otros cifradores en bloque, se tienen tres algoritmos involucrados: el algoritmo para la expansión de la llave, el algoritmo de cifrado y el algoritmo de descifrado. Para estos algoritmos se define un conjunto de funciones primitivas.

- Mix: función que afecta a una palabra, pág. 19.
- Mixing round: función que afecta mediante Mix a 4 palabras, pág. 19.
- Mash: función que afecta una palabra y agrega la subllave adecuada, pág. 20.
- Mashing round: función que afecta mediante Mash a 4 palabras, pág. 20.
- R-Mix: función inversa de Mix, pág. 20.
- R-Mixing round: función inversa a Mixing round, pág. 21.

- R-Mash: función inversa a Mash, pág. 21.
- R-Mashing round: función inversa a Mashing round, pág. 22.

3.1. Expansión de la llave

En este algoritmo se usan operaciones en grupos de 8 bits y en palabras, por ello se usa una notación distinta para cuando se estén usando operaciones en una u otra representación.

Para operaciones sobre la llave que sean en palabras se usará la notación

$$K[0], K[1], ..., K[63], \text{ donde } K[i] \text{ es una palabra.}$$

Para operaciones sobre la llave que sean en bytes se usará la notación

$$L[0], L[1], ..., L[127], \text{ donde } L[i] \text{ es un byte.}$$

Y en todo momento se cumple que

$$K[i] = L[2 * i] + 256 * L[2 * i + 1].$$

Suponemos que la llave a expandir en RC2 es de exactamente T bytes, recordando que T es variable en un rango de 8 a 1024 bits ($1 \leq T \leq 128$), también se tiene un parámetro $T1$ que indica cuántos bits efectivos hay en la llave, es decir el algoritmo usará un espacio de búsqueda dentro de la llave de $min(2^{8T}, 2^{T1})$.

El algoritmo 3.1 muestra el proceso para expandir la llave.

Algoritmo 3.1 RC2: *expansionLlave*

Entrada: la llave K de T bytes, T1 bits efectivos de la llave
Salida: la llave expandida L de tamaño 1024 bits

begin
$copiarEnSuLugar(K, L)$
$T8 \leftarrow (T1 + 7)/8$
$TM \leftarrow 255\ MOD\ 2^{8+T1-8*T8}$
$for\ i = T, T + 1, .., 127$
. $\quad L[i] = PITABLE[L[i - 1] + L[i - T]]$
$L[128 - T8] = PITABLE[L[128 - T8]\ \&\ TM]$

$for\ i = 127 - T8, .., 0$
. $\quad L[i] = PITABLE[L[i+1]\ XOR\ L[i+T8]]$
end

Donde el arreglo $PITABLE$ contiene una permutación de los bytes $0, ..., 255$, presuntamente obtenidos usando el número π, en el cuadro 3.1 se muestra la permutación.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
00	d9	78	f9	c4	19	dd	b5	ed	28	e9	fd	79	4a	a0	d8	9d
10	c6	7e	37	83	2b	76	53	8e	62	4c	64	88	44	8b	fb	a2
20	17	9a	59	f5	87	b3	4f	13	61	45	6d	8d	09	81	7d	32
30	bd	8f	40	eb	86	b7	7b	0b	f0	95	21	22	5c	6b	4e	82
40	54	d6	65	93	ce	60	b2	1c	73	56	c0	14	a7	8c	f1	dc
50	12	75	ca	1f	3b	be	e4	d1	42	3d	d4	30	a3	3c	b6	26
60	6f	bf	0e	da	46	69	07	57	27	f2	1d	9b	bc	94	43	03
70	f8	11	c7	f6	90	ef	3e	e7	06	c3	d5	2f	c8	66	1e	d7
80	08	e8	ea	de	80	52	ee	f7	84	aa	72	ac	35	4d	6a	2a
90	96	1a	d2	71	5a	15	49	74	4b	9f	d0	5e	04	18	a4	ec
a0	c2	e0	41	6e	0f	51	cb	cc	24	91	af	50	a1	f4	70	39
b0	99	7c	3a	85	23	b8	b4	7a	fc	02	36	5b	25	55	97	31
c0	2d	5d	fa	98	e3	8a	92	ae	05	df	29	10	67	6c	ba	c9
d0	d3	00	e6	cf	e1	9e	a8	2c	63	16	01	3f	58	e2	89	a9
e0	0d	38	34	1b	ab	33	ff	b0	bb	48	0c	5f	b9	b1	cd	2e
f0	c5	f3	db	47	e5	a5	9c	77	0a	a6	20	68	fe	7f	c1	ad

Cuadro 3.1: RC2: PITABLE

3.2. Mix

Para esta primitiva se supone una variable j que puede ser usada por todos los Mix y que se inicia en 0.

$R[0], R[1], R[2], R[3]$ son las palabras del bloque a cifrar, por lo que los índices en R se toman módulo 4.

$\overline{R}$ denota la operación de complemento a nivel de bit, & es la operación "and" en bits.

Algoritmo 3.2 RC2: *mix*

Entrada: $R[0], R[1], R[2], R[3]$ las palabras del bloque a cifrar, $R[i]$ la palabra a procesar.
Salida: $R[i]$ la palabra procesada.

begin
$s[0] \leftarrow 1$
$s[1] \leftarrow 2$
$s[2] \leftarrow 3$
$s[3] \leftarrow 5$
$R[i] = R[i] + K[j] + (R[i-1] \ \& \ R[i-2]) + ((\overline{R[i-1]}) \ \& \ R[i-3])$
$j \leftarrow j + 1$
$corrimientoCiclicoIzquierda(R[i], s[i])$
end

3.3. Mixing round

Una ronda de funciones Mix se define como la secuencia de Mix aplicada a los distintos $R[i]$

Algoritmo 3.3 RC2: *mixRound*

Entrada: $R[0], R[1], R[2], R[3]$ las palabras del bloque a cifrar.
Salida: $R[0], R[1], R[2], R[3]$ procesadas.

begin
$mix(R[0])$
$mix(R[1])$
$mix(R[2])$
$mix(R[3])$
end

3.4. Mash

Se define la operación Mash como el agregar la subllave a la palabra que se está procesando.

Algoritmo 3.4 RC2: *mash*

Entrada: $R[0], R[1], R[2], R[3]$ las palabras del bloque a cifrar, $R[i]$ la palabra a procesar.
Salida: $R[i]$ la palabra procesada.

begin
$R[i] = R[i] + K[R[i-1] \ \& \ 63]$
end

3.5. Mashing round

Una ronda de funciones Mash se define como la secuencia de Mash aplicada a los distintos $R[i]$

Algoritmo 3.5 RC2: *mashRound*

Entrada: $R[0], R[1], R[2], R[3]$ las palabras del bloque a cifrar.
Salida: $R[0], R[1], R[2], R[3]$ procesadas.

begin
$mash(R[0])$
$mash(R[1])$
$mash(R[2])$
$mash(R[3])$
end

3.6. R-Mix

Esta es la función inversa de la función Mix.

Algoritmo 3.6 RC2: *rmix*

Entrada: $R[0], R[1], R[2], R[3]$ las palabras del bloque a cifrar, $R[i]$ la palabra a procesar.
Salida: $R[i]$ la palabra procesada.

begin
$s[0] \leftarrow 1$

$s[1] \leftarrow 2$
$s[2] \leftarrow 3$
$s[3] \leftarrow 5$
$corrimientoCiclicoDerecha(R[i], s[i])$
$R[i] = R[i] - K[j] - (R[i-1] \ \& \ R[i-2]) - ((\overline{R[i-1]}) \ \& \ R[i-3])$
$j \leftarrow j - 1$
end

3.7. R-Mixing round

Una ronda de funciones R-Mix se define como la secuencia de R-Mix aplicada a los distintos $R[i]$

Algoritmo 3.7 RC2: *rMixingRound*

Entrada: $R[0], R[1], R[2], R[3]$ las palabras del bloque a cifrar.
Salida: $R[0], R[1], R[2], R[3]$ procesadas.

begin
$rmix(R[3])$
$rmix(R[2])$
$rmix(R[1])$
$rmix(R[0])$
end

3.8. R-Mash

Se define la operación R-Mash como la inversa de Mash al extraer la subllave a la palabra que se está procesando.

Algoritmo 3.8 RC2: *rmash*

Entrada: $R[0], R[1], R[2], R[3]$ las palabras del bloque a cifrar, $R[i]$ la palabra a procesar.
Salida: $R[i]$ la palabra procesada.

begin
$R[i] = R[i] - K[R[i-1] \ \& \ 63]$
end

3.9. R-Mashing round

Una ronda de funciones R-Mash se define como la secuencia de R-Mash aplicada a los distintos $R[i]$

Algoritmo 3.9 RC2: *rMashRound*

Entrada: $R[0], R[1], R[2], R[3]$ las palabras del bloque a cifrar.
Salida: $R[0], R[1], R[2], R[3]$ procesadas.

begin
$rmash(R[3])$
$rmash(R[2])$
$rmash(R[1])$
$rmash(R[0])$
end

3.10. Cifrado de un bloque

El proceso de cifrado de un bloque de 64 bits se define en el algoritmo 3.10.

Algoritmo 3.10 RC2: *cifrado*

Entrada: Bloque B con palabras $R[0], R[1], R[2], R[3]$.
Salida: $R[0], R[1], R[2], R[3]$ palabras del bloque B cifrado.

begin
$j \leftarrow 0$
$for\ i = 0, .., 4$
. $mixRound(B)$
$mashRound(B)$
$for\ i = 0, .., 5$
. $mixRound(B)$
$mashRound(B)$
$for\ i = 0, .., 4$
. $mixRound(B)$
end

3.11. Descifrado de un bloque

El proceso de descifrado de un bloque de 64 bits se define en el algoritmo 3.11.

Algoritmo 3.11 RC2: *descifrado*

Entrada: Bloque B con palabras $R[0], R[1], R[2], R[3]$.
Salida: $R[0], R[1], R[2], R[3]$ palabras del bloque B cifrado.

begin
$j \leftarrow 63$
$for\ i = 0, .., 4$
. $rMixRound(B)$
$rMashRound(B)$
$for\ i = 0, .., 5$
. $rMixRound(B)$
$mashRound(B)$
$for\ i = 0, .., 4$
. $rMixRound(B)$
end

4
IDEA

IDEA (International Data Encryption Algorithm) es un algoritmo, en 8 rondas y una transformación final (llamada media ronda) de cifrado en bloques de 64 bits usando una llave secreta de 128 bits propuesto por Xuejia Lai y James L. Massey [19]. Este algoritmo es simétrico ya que se necesita de la misma llave que se usó en el cifrado para realizar el descifrado.

En este algoritmo no hay presentes cajas de sustitución como en otros algoritmos de cifrado en bloque, en su lugar se realizan operaciones en tres estructuras algebraicas distintas: adición módulo 2^{16}, multiplicación módulo $2^{16}+1$ y XOR.

IDEA presenta la cualidad de que el proceso de cifrado y el de descifrado son idénticos, dependiendo sólo de la llave y la "llave invertida"para el cifrado y descifrado respectivamente.

4.1. Expansión de la llave

Se tiene como entrada un llave de 128 bits, y el objetivo es obtener 52 subllaves de 16 bits cada una a partir de los 128 originales. Para esto, se toman los 128 bits originales y se dividen en 8 subllaves, o subbloques, y se copian a la llave expandida, se hace un corrimiento cíclico a la izquierda de 25 posiciones de la llave y se vuelve a dividir en 8 subllaves que se agregan a la llave expandida, repitiéndose el corrimiento hasta generarse las 52 necesarias. Este proceso se presenta en el algoritmo 4.1.

Algoritmo 4.1 IDEA: *expansionLlave*

Entrada: la llave K de 128 bits
Salida: la llave expandida llaveExp de tamaño 52*16 bits

begin
$tmp \leftarrow K$
$copiarEnSuLugar(llaveExp, tmp)$
$subBloquesGenerados \leftarrow 8$
while $subBloquesGenerados < 52$
. $corrimientoCiclicoIzquierda(tmp, 25)$
. $subBloquesGenerados \leftarrow subBloquesGenerados + 8$
. $copiarEnSuLugar(llaveExp, tmp)$
endwhile
end

La llave expandida contiene 52 subllaves, cada una de ellas a usarse en un momento de la operación de cifrado, el cuadro 4.1 indica en que orden y en que momento se usan estas subllaves. $Z_i^{(r)}$ es la i-ésima subllave de la ronda $r = 1, ..,8$, mientras que $Z_1^{(9)}, Z_2^{(9)}, Z_3^{(9)}, Z_4^{(9)}$ son las llaves usadas en la transofrmación final del cifrado.

Ronda 1	$Z_1^{(1)}, Z_2^{(1)}, Z_3^{(1)}, Z_4^{(1)}, Z_5^{(1)}, Z_6^{(1)}$
Ronda 2	$Z_1^{(2)}, Z_2^{(2)}, Z_3^{(2)}, Z_4^{(2)}, Z_5^{(2)}, Z_6^{(2)}$
Ronda 3	$Z_1^{(3)}, Z_2^{(3)}, Z_3^{(3)}, Z_4^{(3)}, Z_5^{(3)}, Z_6^{(3)}$
Ronda 4	$Z_1^{(4)}, Z_2^{(4)}, Z_3^{(4)}, Z_4^{(4)}, Z_5^{(4)}, Z_6^{(4)}$
Ronda 5	$Z_1^{(5)}, Z_2^{(5)}, Z_3^{(5)}, Z_4^{(5)}, Z_5^{(5)}, Z_6^{(5)}$
Ronda 6	$Z_1^{(6)}, Z_2^{(6)}, Z_3^{(6)}, Z_4^{(6)}, Z_5^{(6)}, Z_6^{(6)}$
Ronda 7	$Z_1^{(7)}, Z_2^{(7)}, Z_3^{(7)}, Z_4^{(7)}, Z_5^{(7)}, Z_6^{(7)}$
Ronda 8	$Z_1^{(8)}, Z_2^{(8)}, Z_3^{(8)}, Z_4^{(8)}, Z_5^{(8)}, Z_6^{(8)}$
Ronda 9	$Z_1^{(9)}, Z_2^{(9)}, Z_3^{(9)}, Z_4^{(9)}$

Cuadro 4.1: IDEA: orden de subllaves

4.2. Llave invertida

Para el proceso de descifrado se usa la misma llave que se usó para cifrar ya que se trata de un criptosistema simétrico, pero en vez de usarse la llave expandida se usan los inversos de las subllaves usadas en orden inverso, inversos correspondientes a las operaciones usadas en el cifrado. $-Z$ representa el inverso aditivo de Z módulo 2^{16}, Z^{-1} es el inverso multiplicativo de Z módulo $2^{16} + 1$, es decir, $-Z + Z = 0$ y $Z^{-1} * Z = 1$, y siguiendo la notación de la sección anterior el cuadro 4.2 denota el orden de uso de las subllaves en el proceso de descifrado.

Hay que notar que usando la operación XOR, Z es su propio inverso, es decir, $X\ XOR\ Z\ XOR\ Z = X$.

Ronda 1	$Z_1^{(9)^{-1}}, Z_2^{(9)^{-1}}, -Z_3^{(9)}, -Z_4^{(9)}, Z_5^{(8)}, Z_6^{(8)}$
Ronda 2	$Z_1^{(8)^{-1}}, Z_2^{(8)^{-1}}, -Z_3^{(8)}, -Z_4^{(8)}, Z_5^{(7)}, Z_6^{(7)}$
Ronda 3	$Z_1^{(7)^{-1}}, Z_2^{(7)^{-1}}, -Z_3^{(7)}, -Z_4^{(7)}, Z_5^{(6)}, Z_6^{(6)}$
Ronda 4	$Z_1^{(6)^{-1}}, Z_2^{(6)^{-1}}, -Z_3^{(6)}, -Z_4^{(6)}, Z_5^{(5)}, Z_6^{(5)}$
Ronda 5	$Z_1^{(5)^{-1}}, Z_2^{(5)^{-1}}, -Z_3^{(5)}, -Z_4^{(5)}, Z_5^{(4)}, Z_6^{(4)}$
Ronda 6	$Z_1^{(4)^{-1}}, Z_2^{(4)^{-1}}, -Z_3^{(4)}, -Z_4^{(4)}, Z_5^{(3)}, Z_6^{(3)}$
Ronda 7	$Z_1^{(3)^{-1}}, Z_2^{(3)^{-1}}, -Z_3^{(3)}, -Z_4^{(3)}, Z_5^{(2)}, Z_6^{(2)}$
Ronda 8	$Z_1^{(2)^{-1}}, Z_2^{(2)^{-1}}, -Z_3^{(2)}, -Z_4^{(2)}, Z_5^{(1)}, Z_6^{(1)}$
Ronda 9	$Z_1^{(1)^{-1}}, Z_2^{(1)^{-1}}, -Z_3^{(1)}, -Z_4^{(1)}$

Cuadro 4.2: IDEA: orden de subllaves inversas

Es decir, dada la llave de cifrado, ésta se expande y sobre la llave expandida se calculan los inversos en orden correspondiente.

4.3. Cifrado y descifrado de un bloque

El proceso de cifrado y descifrado se lleva a cabo en 8 rondas y una transformación al final. Este proceso se puede ver en la figura 4.1, donde $\bigodot$ es la multiplicación módulo $2^{16}+1$, $\boxplus$ es la adición módulo 2^{16} y $\bigoplus$ es la operación XOR.

X_1, X_2, X_3, X_4 son 4 subbloques resultantes de dividir la entrada de 64 bits en 4 grupos de 16 bits, Y_1, Y_2, Y_3, Y_4 es el resultado del proceso.

De acuerdo a la figura 4.1, se nota la necesidad de tener un registro de las operaciones realizadas en la ronda, el resultado de las primeras sumas y multiplicaciones se usa para realizar operaciones al final de la ronda, para la siguiente ronda se permutan los subbloques y se repite el proceso con las subllaves adecuadas. Este proceso se especifica en el algoritmo 4.3.

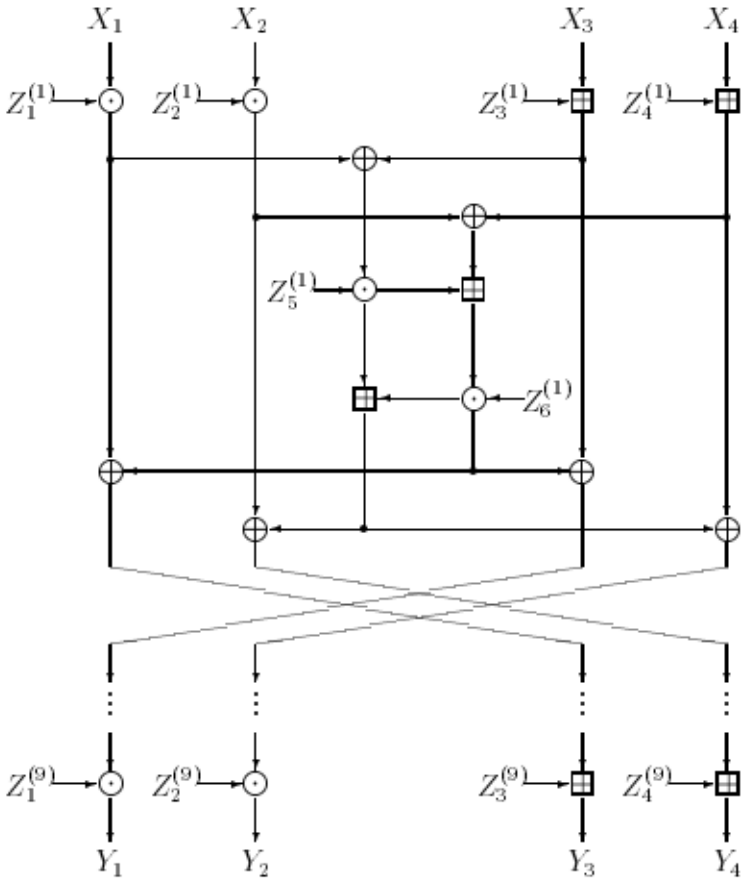

Figura 4.1: IDEA: Cifrado y descifrado en idea

Se nota que en IDEA no hay uso de tablas de sustitución de ningún tipo en ningún momento, como la tabla PITABLE de RC2 (cuadro 3.1) o las tablas S-Box e InvS-Box de AES (figuras 5.2 y 5.3 del siguiente capítulo), si no que todas las operaciones realizadas son sobre el bloque a cifrar y las llaves de ronda.

Algoritmo 4.2 IDEA: *cifrarDescifrar*

Entrada: Bloque B con subbloques, o palabras, de 16 bits w_0, w_1, w_2, w_3, la llave de cifrado *key*.

Salida: El cifrado del bloque B.

begin
$keyIndex \leftarrow 0$

$for\ ronda = 0, .., 7$
. $a_0 \leftarrow multiplicar(w_0, key[keyIndex++])$
. $a_1 \leftarrow sumar(w_1, key[keyIndex++])$
. $a_2 \leftarrow sumar(w_2, key[keyIndex++])$
. $a_3 \leftarrow multiplicar(w_3, key[keyIndex++])$
. $b_0 \leftarrow multiplicar(a_0\ XOR\ a_2, key[keyIndex++])$
. $b_1 \leftarrow sumar(a_1\ XOR\ a_3, b_0)$
. $b_2 \leftarrow multiplicar(b_1, key[keyIndex++])$
. $b_3 \leftarrow sumar(b_0, b_2)$
. $w_0 \leftarrow a_0\ XOR\ b_2$
. $w_1 \leftarrow a_2\ XOR\ b_2$
. $w_2 \leftarrow a_1\ XOR\ b_3$
. $w_3 \leftarrow a_3\ XOR\ b_3$
$endfor$
$tw_0 \leftarrow multiplicar(w_0, key[keyIndex++])$
$tw_1 \leftarrow sumar(w_2, key[keyIndex++])$
$tw_2 \leftarrow sumar(w_1, key[keyIndex++])$
$tw_3 \leftarrow multiplicar(w_3, key[keyIndex++])$
$w_0 \leftarrow tw_0$
$w_1 \leftarrow tw_1$
$w_2 \leftarrow tw_2$
$w_3 \leftarrow tw_3$
end

Donde *sumar* es la suma módulo 2^{16} y *multiplicar* es la multiplicación módulo $2^{16} + 1$, se asume que *key* contiene las subllaves en el orden adecuado.

5
AES

AES (Advance Encryption Standard) es un algoritmo de cifrado en bloques de 128 bits usando llaves de 128, 192 o 256 bits (en 10, 12 o 14 rondas dependiendo del tamaño de la llave) propuesto por Joan Daemen y Vincent Rijmen [14] y adoptado como estándar en 2002 [6].

Dentro de AES, se le llama **estado** al bloque que se esta cifrando o descifrando visto como un arreglo bidimensional de bytes, de tal forma que si se está cifrando la secuencia de bytes $b_0b_1b_2b_3b_4b_5b_6b_7b_8b_9b_{10}b_{11}b_{12}b_{13}b_{14}b_{15}$, el estado que representa es el indicado en el cuadro 5.1.

b_0	b_4	b_8	b_{12}
b_1	b_5	b_9	b_{13}
b_2	b_6	b_{10}	b_{14}
b_3	b_7	b_{11}	b_{15}

Cuadro 5.1: AES: representación de un estado

En AES se definen las siguientes funciones primitivas y valores que se usan en el proceso de cifrado/descifrado y expansión de llave.

- addRoundKey: agrega la subllave de la ronda al bloque que se está cifrando o descifrando, pág. 38,
- mixColumns transformación en el cifrado que toma las columnas del estado y las mezcla para obtener nuevas columnas, pág. 37,
- invMixColumns transformación en el descifrado que es inversa a mixColumns, pág. 38,
- shiftRows transformación en el cifrado que toma los renglones del estado y les aplica un corrimiento de acuerdo a un offset, pág. 36,
- invShiftRows transformación en el descifrado que es inversa a shiftRows, pág. 37,
- subBytes transformación en el cifrado que toma los bytes del estado y los substituye uno por uno usando una caja de sustitución (S-box), pág. 34,
- invSubBytes transformación en el descifrado inversa a subBytes, pág. 35,

- K la llave de cifrado/descifrado,
- Nb el número de columnas del estado (cada columna es una palabra de 32 bits), en este estándar esta fijado en Nb=4,
- Nk número de palabras de 32 bits que componen la llave K,
- Nr el número de rondas de cifrado/descifrado para cada estado, está dado en términos de Nb y Nk,
- Rcon un arreglo de potencias de elementos en GF(256),
- rotWord permutación en un conjunto de 4 bytes, se usa en la función de expansión de la llave, pág. 39,
- subWord función que substituye 4 bytes de acuerdo a la caja de substitución S-box para generar la llave expandida, pág. 39.

5.1. Campos finitos en AES

En el algoritmo de AES cada byte se trata como un elemento en el campo $GF(2^8 = 256)$, de tal forma que la representación binaria del byte corresponde a los coeficientes de su respectivo elemento en $GF(256)$.

El polinomio irreducible que se usa para construir $GF(256)$ es

$$m(x) = x^8 + x^4 + x^3 + x + 1,$$

las operaciones de adición y multiplicación por tanto se realizan módulo $m(x)$.

5.1.1. Adición

Para la suma de elemento en $GF(256)$ es importante notar que al sumar o restar se obtiene el mismo resultado, ya que se está construyendo a partir de Z_2, por ejemplo,

$$(x^7 + x^5 + x^3 + x + 1) + (x^7 + x^4 + x^2 + x + 1) = x^5 + x^4 + x^3 + x^2$$
$$(x^7 + x^5 + x^3 + x + 1) - (x^7 + x^4 + x^2 + x + 1) = x^5 + x^4 + x^3 + x^2$$

Y en su representación en bits se tiene que

$$x^7 + x^5 + x^3 + x + 1 = \{10101011\}$$
$$x^7 + x^4 + x^2 + x + 1 = \{10010111\}$$
$$x^5 + x^4 + x^3 + x^2 = \{00111100\}$$

y trabajando con bits la operación XOR presenta el mismo comportamiento

$$\{10101011\} XOR \{10010111\} = \{00111100\}$$

por lo que es suficiente hacer un XOR para operar los elementos.

5.1.2. Multiplicación

Para la multiplicación no es tan sencillo, ya que no hay una operación a nivel de bits que nos lo permita.

Para realizar una reducción modular se necesita utilizar el algoritmo extendido de Euclides, hacer multiplicaciones entre los elementos y sumas, lo que aumenta la complejidad de esta operación.

Ante este problema se observa que la multiplicación por el polinomio $x = \{00000010\}$ es un corrimiento en una posición a la izquierda, y esto si se puede implementar a nivel de bits.

Con lo anterior se puede implementar la multiplicación usando corrimientos a la izquierda conjuntamente con la operación XOR.

5.1.3. Polinomios con coeficientes en $GF(256)$

Se pueden definir polinomios de 4 términos con coeficientes en $GF(256)$ de la siguiente forma

$$a_3x^3 + a_2x^2 + a_1x + a_0, \text{ con } a_i \text{ en } GF(256)$$

Estos polinomios se comportan distinto que los mencionados anteriormente, ya que estos tienen como coeficientes a elementos en un campo finito, que a su vez pueden ser vistos como polinomios, estos elementos pueden ser vistos como palabras de 32 bits, y la aritmética se calcula módulo $x^4 + 1$.

Es importante notar que los polinomios con coeficientes en $GF(256)$ módulo $x^4 + 1$, que se denotan como

$$\frac{GF(256)[x]}{x^4+1}$$

no forman un campo ya que no todos los elementos tienen inverso.

Se definen los siguientes polinomios en $\frac{GF(256)[x]}{x^4+1}$, expresando los coeficientes en hexadecimal

$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \ldots \text{(pol1)}$$
$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\} \ldots \text{(pol2)}$$

y las primitivas mixColumns e invMixColumns quedan definidas como transformaciones en términos de $a(x)$ y $a^{-1}(x)$ respectivamente.

5.2. subBytes

La primitiva subBytes es una transformación afín que toma como entrada un vector de 16 bytes y regresa otro vector de 16 bytes, es decir, substituye cada byte dentro de un estado por otro byte como se indica en la figura 5.1.

Esta transformación se puede hacer mediante una tabla evitando hacer las multiplicaciones y sumas en el campo cada vez, a esta tabla se le llama S-box y se muestra en la figura 5.2.

Hay que notar que esta transformación afín es invertible pero no nos centraremos en demostrarlo.

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Figura 5.1: AES: Transformación subBytes

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figura 5.2: AES: Tabla de sustitución S-box para el byte xy

5.3. invSubBytes

Esta transformación afín es la inversa de subBytes, y también se puede dar con una tabla, a la que llamaremos invS-box, la figura 5.3 contiene esta transformación.

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figura 5.3: AES: Tabla de sustitución invS-box para el byte xy

5.4. shiftRows

Esta primitiva toma un estado de entrada y regresa este mismo estado después de hacer un corrimiento cíclico de los renglones, numerados de arriba hacia abajo con $r = 0, 1, 2, 3$, de manera que el renglón r se corre a la izquierda en r posiciones.

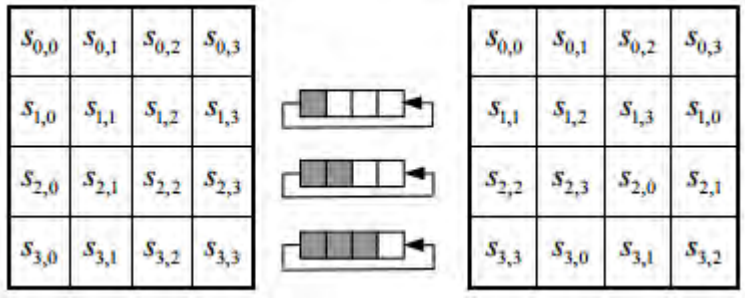

Figura 5.4: AES: shiftRows

5.5. invShiftRows

Primitiva inversa de shiftRows que al tomar un estado lo regresa después de hacer un corrimiento cícliclo sobre los renglones en $Nb - r$ posiciones a la derecha, donde r es el número de renglón dentro del estado.

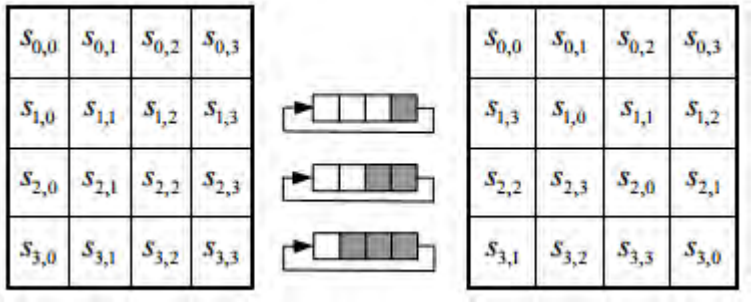

Figura 5.5: AES: invShiftRows

5.6. mixColumns

Esta primitiva opera el estado de manera que toma cada columna como un sólo elemento en $\frac{GF(256)[x]}{x^4+1}$, usando el polinomio $a(x)$ (pol1) esta transformación queda dada por

$$s'(x) = a(x) * s(x)$$

donde $s(x)$ es el estado a cifrar.

Dado que el polinomio $a(x)$ está fijo, esto se puede escribir como la multiplicación de matrices

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Figura 5.6: AES: mixColumns

5.7. invMixColumns

La transformación invMixColumns es la inversa de mixColumns, nuevamente aquí se trata a cada columna del estado como un elemento de $\frac{GF(256)}{x^4+1}$, pero aquí se usa el polinomio $a^{-1}(x)$ (pol2), dando la transformación

$$s'(x) = a^{-1}(x) * s(x)$$

con $s(x)$ el estado a descifrar.

Nuevamente $a^{-1}(x)$ está fijo y se puede reescribir como multiplicación de matrices.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Figura 5.7: AES: invMixColumns

5.8. addRoundKey

Esta primitiva mezcla el estado con una llave de ronda usando la operación XOR. Las llaves de ronda se generan con el algoritmo de expansión de llave y están acotadas por Nr, que toma valores de 10, 12 y 14 para llaves de 128, 192 y 256 bits respectivamente.

Las llaves de ronda se agregan de la siguiente forma

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] XOR [Kexp_{ronda*Nb+c}]$$

donde $Kexp$ es la llave expandida, $0 \leq ronda \leq Nr$ y c es la columna del estado que se está cifrado/descifrando.

Como esta operación es un XOR, es su propia operación inversa.

Las llaves de ronda se calculan a partir de la llave de entrada mediante un algoritmo de expansión de llave.

5.9. Algoritmo de expansión de llave

Algoritmo 5.1 AES: *expansionLlave*
Entrada: la llave K de Nk*4 bytes
Salida: la llave expandida llaveExp de tamaño 4*Nb*(Nr+1) bytes

begin
word tmp
$i \leftarrow 0$
while$(i < Nk)$
. $llaveExp[i] \leftarrow word(K[4*i], K[4*i+1], K[4*i+2], K[4*i+3])$
. $i \leftarrow i+1$
while$(i < Nb*(Nr+1))$
. $tmp \leftarrow llaveExp[i-1]$
. $if(i \bmod Nk = 0)$
. $\quad tmp \leftarrow subWord(rotWord(tmp))$ xor $Rcon[i/Nk]$
. $elseif(Nk > 6$ and $i \bmod Nk = 4)$
. $\quad tmp \leftarrow subWord(tmp)$
. $llaveExp[i] \leftarrow llaveExp[i-Nk]$ xor tmp
. $endif$
. $i \leftarrow i+1$
endwhile
end

La función $rotWord$ toma una palabra de 32 bits, vistos como bytes $[a_0, a_1, a_2, a_3]$, y regresa el corrimiento cíclico $[a_1, a_2, a_3, a_0]$.

La función $subWord$ toma una palabra de 32 bits, vistos como bytes, y regresa la palabra después de sustituir los bytes usando la caja de sustitución S-box.

El arreglo $Rcon$ está definido como un arreglo de potencias del elemento primitivo α del campo $GF(256)$, explícitamente este arreglo es
$Rcon = [0, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36]$, el 0 de la primera entrada es para evitar redireccionamientos en la implementación de este proyecto.

5.10. Cifrado de un estado

Algoritmo 5.2 AES: *cifrarEstado*

Entrada: el estado state a cifrar y la llave expandida llaveExp
Salida: el estado state cifrado

begin
$addRoundKey(state, llaveExp[0, Nb-1])$
$for\ round = 1\ to\ Nr-1\ by\ 1$
. $subBytes(state)$
. $shiftRows(state)$
. $mixColumns(state)$
. $addRoundKey(state, llaveExp[round*Nb,\ (round+1)*Nb-1])$
endfor
$subBytes(state)$
$shiftRows(state)$
$addRoundKey(state, llaveExp[Nr*Nb, (Nr+1)*Nb-1])$
end

5.11. Descifrado de un estado

Algoritmo 5.3 AES: *descifrarEstado*

Entrada: el estado state a descifrar y la llave expandida llaveExp
Salida: el estado state descifrado

begin
$addRoundKey(state, llaveExp[Nr*Nb, (Nr+1)*Nb-1])$
$for\ round = Nr-1\ to\ 1\ by\ -1$
. $invShiftRows(state)$
. $invSubBytes(state)$
. $addRoundKey(state, llaveExp[round*Nb,\ (round+1)*Nb-1])$
. $invMixColumns(state)$
endfor
$invShiftRows(state)$
$invSubBytes(state)$
$addRoundKey(state, llaveExp[0, Nb-1])$
end

6
Paralelización

Al paralelizar se deben tomar en cuenta algunos aspectos de los criptosistemas, algunos de los aspectos que tienen en común los criptosistemas revisados son los modos de operación (estos criptosistemas por bloque funcionan usando cualquiera de los modos de operación definidos en [3] y revisados en el capítulo 2) y el que todos tengan un algoritmo de expansión de llave.

6.1. Expansión de llave

Los algoritmos de expansión de llave de los criptosistemas revisados presentan varios problemas si se quisiera paralelizarlos. El primer problema que se presenta es el costo administrativo de hacer copias de memoria entre CPU y GPU para realizar la expansión en la GPU, las operaciones en la expansión de llave son para obtener un conjunto de entre 100 y 240 bytes (104 para IDEA, 128 para RC2 y 240 para AES de 256 bits), otro problema que presenta es que la generación de la llave de ronda i requiere fuertemente de las llaves de rondas anteriores, lo que lo convierte en un algoritmo secuencial.

Por estos motivos, la expansión de llave se realiza en CPU y el resultado se copia a la memoria de la GPU para el uso dentro del algoritmo de cifrado o descifrado.

6.2. Modos de operación: paralelización trivial

Los modos de operación que se contemplaron en la paralelización fueron ECB y CTR, ya que revisando las definiciones de los modos de operación, ECB y CTR se pueden paralelizar tanto en el cifrado como en el descifrado, ya que el procesamiento de un bloque es independiente de los demás bloques; por otro lado, los modos CBC y CFB no son paralelizables en el cifrado, pero el descifrado se puede paralelizar sólo si se precalculan los vectores de inicialización y aún así, este precálculo es secuencial; por último, el modo OFB no es paralelizable en cifrado ni descifrado, ya que siempre se necesitan resultados anteriores para cifrar/descifrar bloques subsecuentes.

Al usar los modos de operación ECB y CTR se puede definir una "paralelización trivial", que consiste en paralelizar el cifrado y descifrado de cada bloque, es decir, cada hilo de la GPU se encargará de procesar un bloque a la vez.

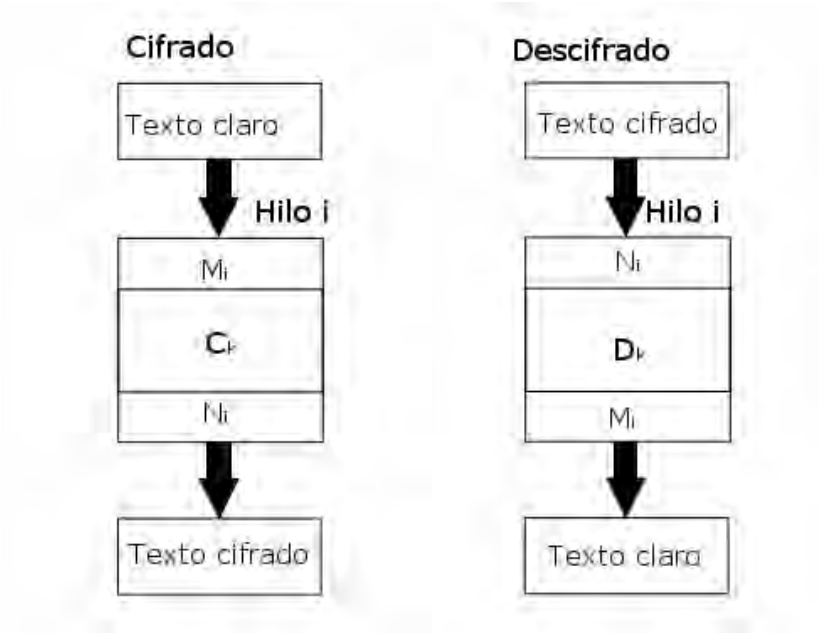

Figura 6.1: Paralelización trivial

Los criptosistemas RC2, IDEA y AES son paralelizables usando paralelización trivial. En las próximas secciones se revisa la posibilidad de paralelizar a nivel de instrucción.

6.3. RC2

RC2 no es paralelizable a nivel de instrucción por los siguientes motivos:

- En el algoritmo 3.2 se ve que el procesamiento de cada subbloque, o palabra, es dependiente de los demás subbloques. Para procesar el subbloque $i + 1$ se necesita que el i ya esté procesado. Este algoritmo se podría paralelizar usando un hilo por subbloque y el uso de sincronización para garantizar la correctez de los datos, pero el hilo $i + 1$ no podría iniciar hasta que el hilo i termine, lo que convierte al algoritmo en secuencial.
- Las rondas 5 y 11 del cifrado necesitan del algoritmo 3.4, que requiere a todos los subbloques ya procesados, es decir, se necesita de sincronización adicional por cada subbloque.

- Algo similar ocurre con los algoritmos 3.6 y 3.8 durante el descifrado, paralelizarlo necesitaría sincronización en el procesamiento de cada subbloque, convirtiendo el descifrado en secuencial.
- Para solucionar los problemas anteriores se podría calcular un conjunto de tablas de sustitución, potencialmente evitando la sincronización, pero ese espacio de búsqueda es de orden 2^{64}, por la forma en que se expande la llave, debido a los distintos valores que puede tener el algoritmo de expansión para los bits efectivos.

Algoritmo 6.1 CUDA RC2: *cifrado*
Entrada: paquete de datos *in*, llave *key*, *tam* tamaño en bytes de *in*.
Salida: paquete de datos *in* cifrado.

begin
$for\ tid \in HILOSVALIDOS$
. $B \leftarrow identificarBloque(tid, in)$
. $cifradoBloqueRC2(B)$
end

Algoritmo 6.2 CUDA RC2: *descifrado*
Entrada: paquete de datos *in*, llave *key*, *tam* tamaño en bytes de *in*.
Salida: paquete de datos *in* cifrado.

begin
$for\ tid \in HILOSVALIDOS$
. $B \leftarrow identificarBloque(tid, in)$
. $descifradoBloqueRC2(B)$
end

En los algoritmos 6.1 y 6.2 *tid* es el identificador del hilo de ejecución que esta realizando los cálculos, $cifradoBloqueRC2$ y $descifradoBloqueRC2$ son copias de los algoritmos 3.10 y 3.11 respectivamente en la GPU, ya que se trata de implementaciones con paralelismo trivial.

La implementación que se siguió realiza todos los *mix* y *mash* dentro de la misma rutina, de la misma forma para el descifrado, ya que realizar varias llamadas a subrutinas en GPU para realizar pocas operaciones sobre bits significa una gran carga administrativa.

6.4. IDEA

IDEA es paralelizable a nivel instrucción, con gran carga a un sólo hilo por bloque, por las siguientes razones:

- De la figura 4.1, y suponiendo la ronda i, las operaciones relacionadas con las subllaves $Z_1^{(i)}$, $Z_2^{(i)}$, $Z_3^{(i)}$ y $Z_4^{(i)}$ se pueden realizar de manera independiente. Se pueden realizar estas operaciones una por hilo en la GPU.
- Las operaciones relacionadas con $Z_5^{(i)}$ y $Z_6^{(i)}$ necesitarían de previa y posterior sincronización si se tuvieran varios hilos procesando el mismo bloque, además se necesita de un registro de resultados intermedios de las subllaves anteriores, para realizar estas operaciones se usa sólo un hilo para evitar tener varias sincronizaciones para la misma operación.
- Las operaciones de la media ronda final se pueden realizar de manera independiente, es decir, una operación se puede realizar por hilo previa sincronización.

La versión trivial para cifrar y descifrar IDEA se muestra en el algoritmo 6.3, la versión con paralelización a nivel instrucción se presenta en el algoritmo 6.4.

Algoritmo 6.3 CUDA IDEA: *cifrarDescifrarTrivial*
Entrada: paquete de datos in, llave key.
Salida: paquete de datos in procesado.

$begin$
$for\ tid \in HILOSVALIDOS$
. $B \leftarrow identificarBloque(tid, in)$
. $cifrarDescifrarIDEA(B)$
end

$cifrarDescifrarIDEA$ es una copia del algoritmo 4.3 en la GPU.

Algoritmo 6.4 CUDA IDEA: *cifrarDescifrarInstrucción*
Entrada: paquete de datos in, llave key.
Salida: paquete de datos in procesado.

$begin$
$for\ tid \in HILOSVALIDOS$

. $B \leftarrow identificarBloque(tid, in)$
. $cifrarDescifrar(B)$
end

Algoritmo 6.5 CUDA IDEA: *cifrarDescifrar*

Entrada: Bloque B con subbloques, o palabras, de 16 bits w_0, w_1, w_2, w_3, la llave de cifrado key.
Salida: El cifrado del bloque B.

begin
$threadIn \leftarrow tid$ mód 4
$tidloc \leftarrow tid * 2$
$word \leftarrow in[tidloc]|in[tidloc + 1]$
$i \leftarrow 0$
$for\ i = 0;\ i < rondas;\ i++$
. $switch(threadIn)$
. *case* 0:
. $a[tid] \leftarrow multiplicar(word, key[i * 6])$
. *case* 1:
. $a[tid] \leftarrow sumar(word, key[i * 6 + 1])$
. *case* 2:
. $a[tid] \leftarrow sumar(word, key[i * 6 + 2])$
. *case* 3:
. $a[tid] \leftarrow multiplicar(word, key[i * 6 + 3])$
. $endswitch$
. $synchthreads()$
. $if\ threadIn = 0$
. $b[tid] \leftarrow multiplicar(a[tid]^a[tid + 2], key[i * 6 + 4])$
. $b[tid + 1] \leftarrow sumar(a[tid + 1]^a[tid + 3], b[tid])$
. $b[tid + 2] \leftarrow multiplicar(b[tid + 1], key[i * 6 + 5])$
. $b[tid + 3] \leftarrow sumar(b[tid], b[tid + 2])$
. $endif$
. $synchthreads()$
. $tmp \leftarrow tid/4$
. $tmp \leftarrow tmp * 4$
. $switch(threadIn)$
. *case* 0:
. $word \leftarrow [tmp]$ xor $b[tmp + 2]$
. *case* 1:

```
.         word ← a[tmp + 2] xor b[tmp + 2]
.       case 2:
.         word ← a[tmp + 1] xor b[tmp + 3]
.       case 3:
.         word ← a[tmp + 3] xor b[tmp + 3]
.     endswitch
.     synchthreads()
endfor
switch(threadIn)
.   case 0:
.     word ← multiplicar(word, key[i ∗ 6])
.     in[tidloc] ← (word >> 8) & 0xffff
.     in[tidloc + 1] ← word & 0xffff
.   case 1:
.     word ← sumar(word, key[i ∗ 6 + 2])
.     in[tidloc + 2] ← (word >> 8) & 0xffff
.     in[tidloc + 3] ← word & 0xffff
.   case 2:
.     word ← sumar(word, key[i ∗ 6 + 1])
.     in[tidloc − 2] ← (word >> 8) & 0xffff
.     in[tidloc − 1] ← word & 0xffff
.   case 3:
.     word ← multiplicar(word, key[i ∗ 6 + 3])
.     in[tidloc] ← (word >> 8) & 0xffff
.     in[tidloc + 1] ← word & 0xffff
endswitch
end
```

La sincronización se lleva a cabo con *synchthreads*() y los arreglos a y b almacenan los datos intermedios del procesamiento del bloque.

6.5. AES

AES es paralelizable a nivel instrucción ya que el bloque se procesa como una matriz bidimensional (estado).

- Los renglones del estado se operan de manera independiente.
- Las columnas se operan de manera independiente.

- Sustituir los bytes del estado es de manera independiente.
- La llave de ronda se puede operar con el estado de manera independiente entre bytes.

Se usan 4 hilos para procesar cada estado en AES, ya que el estado es de 4 renglones por 4 columnas, para pasar de procesar columnas a renglones se necesita de sincronización, lo mismo que con las sustituciones de bytes.

El cifrado y descifrado trivial con AES se presentan con los algoritmos 6.6 y 6.7, mientras que los algoritmos 6.8 y 6.10 corresponden a la paralelización a nivel de instrucción.

Algoritmo 6.6 CUDA AES: *cifrarTrivial*
Entrada: paquete de datos *in*, llave *key*.
Salida: paquete de datos *in* procesado.

begin
$for\ tid \in HILOSVALIDOS$
. $B \leftarrow identificarBloque(tid, in)$
. $cifrarAES(B)$
end

Algoritmo 6.7 CUDA AES: *descifrarTrivial*
Entrada: paquete de datos in, llave key.
Salida: paquete de datos in procesado.

begin
$for\ tid \in HILOSVALIDOS$
. $B \leftarrow identificarBloque(tid, in)$
. $descifrarAES(B)$
end

$cifrarAES$ y $descifrarAES$ son copias de los algoritmos 5.10 y 5.11 en la GPU.

Algoritmo 6.8 CUDA AES: *cifrarInstrucción*
Entrada: paquete de datos in, llave key.
Salida: paquete de datos in procesado.

begin
$for\ tid \in HILOSVALIDOS$
. $B \leftarrow identificarBloque(tid, in)$
. $cifrar(B)$
end

Algoritmo 6.9 CUDA AES: *cifrar*
Entrada: bloque B a cifrar.
Salida: bloque B cifrado.

begin
$addRoundKey()$
$j \leftarrow 1$
$for\ j = 1;\ j < Nr;\ j++$
. $subBytes()$
. $shiftRows()$
. $syncthreads()$
. $mixColumns()$
. $addRoundKey()$
endfor
$subBytes()$

$shiftRows()$
$syncthreads()$
$addRoundKey()$
end

Algoritmo 6.10 CUDA AES: *descifrarInstrucción*
Entrada: paquete de datos in, llave key.
Salida: paquete de datos in procesado.

begin
$for\ tid \in HILOSVALIDOS$
. $B \leftarrow identificarBloque(tid, in)$
. $descifrar(B)$
end

Algoritmo 6.11 CUDA AES: *descifrar*
Entrada: bloque B a cifrar.
Salida: bloque B cifrado.

begin
$addRoundKey()$
$syncthreads()$
$j \leftarrow Nr - 1$
$for\ j;\ j > 0;\ j--$
. $invShiftRows()$
. $syncthreads()$
. $invSubBytes()$
. $addRoundKey()$
. $invMixColumns()$
. $syncthreads()$
endfor
$invShiftRows()$
$syncthreads()$
$invSubBytes()$
$addRoundKey()$
end

6.6. Transferencia de memoria

Se puede estudiar la capacidad de paralelismo de los criptosistemas anteriores a nivel de algoritmo, pero si se desea hacer una comparación "justa" entre las versiones de CPU y GPU, entonces de debe contemplar las transferencias de memoria hacia y desde la GPU, ya que estas pueden representar un cuello de botella para la ejecución de los programas en la GPU.

La arquitectura CUDA ofrece la opción de realizar transferencias de datos de manera no sincronizada, permitiendo que el CPU pueda realizar operaciones mientras la GPU sigue con sus cálculos, esto se logra a través de un *stream*.

Un *stream* de CUDA es una secuencia de operaciones que se ejecutan en la GPU en el orden que el CPU registra, pero tiene la ventaja de que si hay más streams, estos pueden ejecutarse de manera concurrente [12].

El esquema que se sigue para poder usar estos streams es el siguiente:

1. Leer un fragmento del archivo a procesar en un buffer.
2. Inicializar los n streams que se usarán.
3. Dividir el buffer en n partes y copiarlas a la GPU, una en cada stream.
4. Procesar el buffer mediante los streams.
5. Copiar el buffer al CPU.
6. Escribir el buffer en el archivo.

7
Implementación y resultados

A continuación se examinan varios detalles de la implementación, variables y valores dentro de los programas.

- Lo primero que se puede observar en los criptosistemas en el capítulo anterior es que los hilos de ejecución no requieren de memoria compartida, por lo que se le puede dar preferencia a la memoria cache del dispositivo CUDA mediante la directiva

  cudaDeviceSetCacheConfig(cudaFuncCachePreferL1),

  esta característica es común a los 3 criptosistemas.

- Una parte importante de los programas CUDA, o *kernels*, es la configuración de lanzamiento, donde se especifica cuantos hilos ejecutarán el programa [7]. Una herramienta útil para definir estos parámetros es la "CUDA Occupancy Calculator"de NVIDIA [9], que en esta implementación reporta máxima ocupación con una configuración de 128, 256 y 128 hilos por bloque para RC2, IDEA y AES respectivamente. Las tablas 7.1, 7.2 y 7.3 presentan la ocupación teórica máxima con esta herramienta y la ocupación real alcanzada (reportada con la herramienta *nvprof* de CUDA) para diversas configuraciones de hilos por bloque. Lograr una ocupación alta es importante para propósitos de ocultar latencia de accesos a memoria.

- El paquete de datos a procesar se divide en N paquetes más pequeños, en esta implementación el tamaño de cada parte es de 100 Mb. Se realizaron pruebas para otros tamaños pero la diferencia en tiempo de ejecución entre GPU y CPU no fue significativa.

- Cada *stream* se encarga de procesar $104857600/nStreams$ (100 Mb / $nStreams$) bytes, donde $nStreams$ es el número de *streams* que se están usando.

- Las ejecuciones de los programas que se usaron para los resultados reportados en este capítulo fueron usando $nStreams = 4$. Las tablas 7.12, 7.13 y 7.14 muestran datos de las ejecuciones para $nStreams$ =2, 6 y 8.

- En el caso de AES, las cajas de sustitución SBOX e INVSBOX se copian al dispositivo CUDA para que todos los hilos de ejecución tengan acceso a ellas en memoria constante mediante la directiva

cudaMemcpyToSymbol

ahorrando uso de memoria compartida y mejorando el acceso a estas tablas en comparación a [16].

Hilos por Bloque	Teórica	Min	Max	Promedio
64	50 %	41 %	48 %	47 %
128	100 %	88 %	99 %	91 %
192	94 %	82 %	91 %	90 %
256	100 %	88 %	98 %	91 %

Cuadro 7.1: Resultados: Ocupación teórica vs ocupación alcanzada en RC2

Hilos por Bloque	Teórica	Min	Max	Promedio
64	50 %	39 %	46 %	45 %
128	75 %	70 %	73 %	72 %
192	75 %	68 %	73 %	71 %
256	75 %	71 %	74 %	72 %

Cuadro 7.2: Resultados: Ocupación teórica vs ocupación alcanzada en IDEA

Hilos por Bloque	Teórica	Min	Max	Promedio
64	50 %	42 %	47 %	43 %
128	100 %	88 %	98 %	89 %
192	94 %	81 %	89 %	87 %
256	100 %	86 %	97 %	89 %

Cuadro 7.3: Resultados: Ocupación teórica vs ocupación alcanzada en AES

7.1. Resultados

Para los criptosistemas analizados definimos como *CRi* y *CRt* a las versiones del criptosistema *CR* con paralelización a nivel de (i)nstrucción y paralelización (t)rivial respectivamente.

Observación: la paralelización trivial es equivalente a la versión del criptosistema en cuestión en CPU, salvo por el cálculo del identificador del hilo de ejecución y el indexado del paquete de datos que se esta cifrando o descifrando.

El dispositivo CUDA en que se realizaron las pruebas para las versiones GPU es GeForce GTX 780, arquitectura 3.5, permitiendo ejecución y copiado concurrente. Mientras que las pruebas para CPU se realizaron en un Intel(R) Xeon(R) CPU E5-2680 v2.

A continuación se presentan tablas con resultados referentes a las ejecuciones de los programas para distintos tamaños de paquetes de datos a cifrar o descifrar: el tamaño de estos paquetes esta en Mb; las columnas marcadas como "CPU", "*CRt*", "*CRi*"son mediciones en milisegundos; las columnas "factor *t*", "factor *i*"denotan la mejora entre las versiones *triviales*-CPU e *instrucción*-CPU respectivamente.

Tamaño	CPU	RC2t	factor t
100	2241.56	69.669	32.17x
200	4550.41	139.344	32.66x
500	11213.9	348.338	32.19x
1000	22288.1	696.731	31.99x

Cuadro 7.4: Resultados: Cifrado RC2

Tamaño	CPU	RC2t	factor t
100	1379.99	70.276	19.64x
200	2818.77	140.595	20.05x
500	6789.87	351.432	19.32x
1000	13338.3	702.905	18.98x

Cuadro 7.5: Resultados: Descifrado RC2

Los factores de aceleramiento de las tablas 7.4 y 7.5 eran de esperar, ya que RC2 usa operaciones a nivel bit en su mayoría.

Tamaño	CPU	IDEAt	IDEAi	factor t	factor i
100	1459.98	229.421	775.888	6.36x	1.88x
200	2996.14	456.478	1560.9	6.56x	1.92x
500	7298.49	1127.45	4105.58	6.47x	1.78x
1000	14470.7	2196.52	8319.3	6.59x	1.74x

Cuadro 7.6: Resultados: Cifrado IDEA

Tamaño	CPU	IDEAt	IDEAi	factor t	factor i
100	1470.38	228.099	774.332	6.45x	1.90x
200	3009.94	453.983	1588.53	6.63x	1.89x
500	7353.68	1115	4105.38	6.60x	1.79x
1000	14462	2221.44	8368.31	6.51x	1.73x

Cuadro 7.7: Resultados: Descifrado IDEA

En las tablas 7.6 y 7.7 se aprecian valores muy similares si se hace una comparación entrada por entrada, esto se debe a que la diferencia entre el cifrado y el descifrado es solamente la llave que se esta usando, el algoritmo es exactamente el mismo. También se observa una aceleración promedio de 6.5x para la versión trivial, sin embargo para versión a nivel de instrucción se tiene 1.8x, basta revisar un fragmento del código para ver el por que.

Sabemos que "los multiprocesadores crean, gestionan, agendan y ejecutan los hilos ejecución en grupos de 32 hilos llamados *warp*, los hilos que pertenecen a un mismo *warp* comienzan en la misma dirección del programa, pero tienen su propio contador de instrucciones y estado de registros [...] Un warp ejecuta una instrucción común a la vez, por lo que la eficiencia máxima se alcanza cuando todos los 32 hilos de un warp coinciden en su camino de ejecución. Si

los hilos de un warp divergen a través de una rama condicional, el warp ejecuta en forma secuencial cada camino de la rama tomada, deshabilitando los hilos que no están en ese camino" [7]. En el fragmento que se muestra en la figura 7.1 se observa una dependencia sobre el identificador del hilo de ejecución para la elección de la operación (*switch*), causando divergencia en los warp, además de tener poca carga aritmética por hilo y una instrucción *__syncthreads()* que representa gran carga administrativa.

```
switch(numThreadInBlock) {
    case 0:
        a[tidInArray] = multiplicar(word, key[i * 6]);
        break;
    case 1:
        a[tidInArray] = sumar(word, key[i * 6 + 1]);
        break;
    case 2:
        a[tidInArray] = sumar(word, key[i * 6 + 2]);
        break;
    case 3:
        a[tidInArray] = multiplicar(word, key[i * 6 + 3]);
        break;
}

__syncthreads();
```

Figura 7.1: IDEA: Fragmento de código

Las tablas 7.8 y 7.9 muestran los resultados de las ejecuciones para AES, cabe mencionar que la versión AES con paralelismo a nivel de instrucción presentada en el capítulo anterior sufre del mismo detalle que la versión de IDEA, hay una fuerte dependencia de las operaciones respecto de los identificadores de los hilos de ejecución. Sin embargo, presenta una cualidad que IDEA no, en cada renglón y cada columna se aplica la misma operación que en las demás con la única diferencia de ser renglón 1,2,3 ó 4, y ser columna 1,2,3 ó 4, lo que nos lleva a preguntarnos si ¿se pueden reescribir las instrucciones condicionales de forma que se minimice la divergencia y se obtenga un rendimiento mejor?.

Tamaño	CPU	AESt	AESi	factor t	factor i
100	5473.37	492.094	466.869	11.12x	11.72x
200	9949.69	982.552	942.052	10.13x	10.56x
500	24788.5	2463.01	2499.29	10.06x	9.92x
1000	50192.7	5080.39	4919.05	9.88x	10.20x

Cuadro 7.8: Resultados: Cifrado AES

Tamaño	CPU	AESt	AESi	factor t	factor i
100	7669.23	1765.79	1871.18	4.34x	4.10x
200	15298	3597.92	3819.38	4.25x	4.01x
500	38232	9027.06	9748.4	4.24x	3.92x
1000	76245.2	18126.9	20096.2	4.21x	3.79x

Cuadro 7.9: Resultados: Descifrado AES

Llamaremos AESi *v2* al intento de mejorar la implementación de AESi, algorítmicamente se puede observar que cada vez que en el cifrado se aplica la operación *subBytes* se aplica inmediatamente después la operación *shifRrows*, la primera modificación es combinar estas dos operaciones de forma que cuando se realizan las lecturas para hacer los corrimientos de los renglones, en la escritura se escriben los valores de la caja de sustitución SBOX. Esta modificación resultó en una mejora de 2 %.

La siguiente modificación, tratando de minimizar la divergencia, es quitar la instrucción de control de las operaciones, la operaciones *addRoundKey* y *mixColumns* están libres de instrucciones de control ya que operan bajo la premisa de que el identificador del hilo de ejecución indica que columna se va a operar, pero la operación *shiftRows* tiene una instrucción *switch* para identificar el renglón (el renglón 1 se corre 0 bits, el 2 se corre 8 bits, el 3 en 16 y el 4 en 24). Usando aritmética modular se leen los datos y se guardan en un entero de 32 bits, en el que se realiza el corrimiento y se escribe el resultado en el renglón correspondiente. La versión de *shiftRows* de AESi está en la figura 7.2 mientras que la de AESi *v2* esta en la figura 7.3.

```
B8 tmp;
switch(row) {
    case 1:
        tmp = in[tid - 3];
        in[tid - 3] = in[tid + 1];
        in[tid + 1] = in[tid + 5];
        in[tid + 5] = in[tid + 9];
        in[tid + 9] = tmp;
        break;
    case 2:
        tmp = in[tid - 6];
        in[tid - 6] = in[tid + 2];
        in[tid + 2] = tmp;
        tmp = in[tid - 2];
        in[tid - 2] = in[tid + 6];
        in[tid + 6] = tmp;
        break;
    case 3:
        tmp = in[tid - 9];
        in[tid - 9] = in[tid + 3];
        in[tid + 3] = in[tid - 1];
        in[tid - 1] = in[tid - 5];
        in[tid - 5] = tmp;
        break;
}
```

Figura 7.2: AES*i*: Fragmento de código

```
B32 aRotar = ((B32)(in[tid - 3*row]) << 24) | ((B32)(in[tid - 3*row + 4]) << 16)
           | ((B32)(in[tid - 3*row + 8]) << 8) | ((B32)(in[tid - 3*row + 12]));
aRotar = (aRotar << (8*row)) | (aRotar >> (8*(4-row)));
in[tid - 3*row] = SBOX[(aRotar >> 24) & 0xff];
in[tid - 3*row + 4] = SBOX[(aRotar >> 16) & 0xff];
in[tid - 3*row + 8] = SBOX[(aRotar >> 8) & 0xff];
in[tid - 3*row + 12] = SBOX[aRotar & 0xff];
```

Figura 7.3: AES*i* *v*2: Fragmento de código

De esta forma todos los hilos de los warp tienen exactamente el mismo código, por lo que incluso ya no es necesario el uso de *__syncthreads*, ya que todos los hilos del mismo warp avanzan al mismo tiempo siempre y cuando todas las instrucciones sean comunes [7].

La modificación anterior se aplica de manera análoga al descifrado, las tablas 7.10 y 7.11 presentan la comparación entre las versiones AES*i* y AES*i* *v*2, junto con los factores de ganancia contra la versión en CPU.

Tamaño	AES*i*	AES*i* *v*2	factor *i*	factor *i* *v*2
100	466.869	376.361	11.72x	14.54x
200	942.052	741.976	10.56x	13.41x
500	2499.29	1908.77	9.92x	12.99x
1000	4919.05	3982.35	10.20x	12.6x

Cuadro 7.10: Resultados: Cifrado AES*i* vs AES*i* *v*2

Tamaño	AES*i*	AES*i* *v*2	factor *i*	factor *i* *v*2
100	1871.18	1727.94	4.10x	4.44x
200	3819.38	3469.26	4.01x	4.41x
500	9748.4	8554.77	3.92x	4.47x
1000	20096.2	18008.8	3.79x	4.23x

Cuadro 7.11: Resultados: Descifrado AES*i* vs AES*i* *v*2

De lo anterior se puede ver que la versión AES*i* *v2* es mejor que AES*i*. Contra la versión AES*t* también es mejor en el cifrado por un factor $\approx$ 3x, pero para el descifrado apenas y es mejor por un factor de $\approx$ .4x.

Ya se mencionó que las transferencias de memoria pueden representar un problema ya que podría ocurrir un evento de cuello de botella en estas, a lo que se propuso usar *streams*, en los resultados hasta ahora mostrados se usaron 4 *streams*, pero ¿aumentar o disminuir el número de *streams* afecta el desempeño del cifrado o descifrado?. La respuesta es sí, aunque sea en un pequeño factor, usar distinto número de streams puede afectar el rendimiento. Las tablas 7.12, 7.13 y 7.14 muestran los resultados obtenidos para distintos valores de la variable *nStreams*[1].

[1]En estas tablas RC2c representa el cifrado usando RC2 mientras que RC2d el descifrado, en general la letra c es para el cifrado y la letra d para el descifrado. Además, los resultados obtenidos son usando las versiones IDEA*t* y AES*i* *v*2.

Tamaño	RC2c	RC2d	IDEAc	IDEAd	AESc	AESd
100	1.05x	1.05x	1.01x	1.02x	1.01x	1.02x
200	1.03x	1.05x	1.02x	1.02x	1.08x	1.1x
500	1.05x	1.05x	1.01x	1.01x	1.1x	1.12x
1000	1.05x	1.05x	1.01x	1.02x	1.07x	1.1x

Cuadro 7.12: Resultados: Ganancia 4 streams de CUDA sobre 2 streams

Tamaño	RC2c	RC2d	IDEAc	IDEAd	AESc	AESd
100	0.98x	0.98x	1x	0.99x	1.02x	1.01x
200	0.95x	0.98x	1.01x	1x	1.03x	1.04x
500	0.99x	1.01x	1x	0.99x	1.1x	1.1x
1000	1x	1.001x	0.99x	1x	1.07x	1.3x

Cuadro 7.13: Resultados: Ganancia 4 streams de CUDA sobre 6 streams

Tamaño	RC2c	RC2d	IDEAc	IDEAd	AESc	AESd
100	0.99x	1x	0.98x	0.99x	1x	1.01x
200	0.99x	0.98x	0.99x	1x	1.01x	1.05x
500	1x	1x	1.01x	1.02x	1.02x	1.2x
1000	1x	1x	1.01x	1.02x	1.03x	1.5x

Cuadro 7.14: Resultados: Ganancia 4 streams de CUDA sobre 8 streams

De las tablas anteriores se puede apreciar que usar 4 streams es mejor que usar 2 para cualquier criptosistema presentado. Usar 6 es mejor que 4 en pocos casos, sin embargo los casos donde el desempeño de usar 4 son muchos más. Para el uso de 8 streams se puede observar una relación entre el tamaño del paquete procesado y el rendimiento: mientras más grande es el paquete mayor es el desempeño de usar 4 streams.

8
Conclusiones

El los criptosistemas estudiados, y en general para cualquier criptosistema por bloque, se observó potencial de paralelización a nivel de bloque al usar los modos de operación ECB y CTR, tanto para cifrado como para descifrado.

El usar *streams* de CUDA mejoraron el rendimiento en al menos 10 % que si no se usaran. Después de realizar pruebas variando el número de *streams* se encontró que para estos criptosistemas los mejores resultados se obtenían con 4 *streams*.

Para el cifrado y descifrado de paquetes de datos de gran tamaño es importante lograr una alta ocupación, ya que el cifrado/descifrado de los datos requiere de un gran número de lecturas y escrituras de memoria. Se alcanzó una ocupación de 100 % para RC2, 75 % para IDEA y 100 % para AES.

En RC2 no es viable dar una paralelización a nivel de instrucción dada la gran dependencia de operaciones, tanto en el cifrado como en el descifrado. Sin embargo, al usar operaciones a nivel de bit en su mayoría, la paralelización a nivel de bloque (o paralelización trivial) presentó factores de ganancia de $\approx$ 32x y $\approx$ 19x, en cifrado y descifrado respectivamente comparando con la versión secuencial del criptosistema.

En IDEA se observó potencial de paralelización a nivel de instrucción, ya en la implementación se obtuvieron factores de $\approx$ 6,4x y de $\approx$ 1,8 para las versiones paralelización-trivial y paralelización-instrucción comparando ambas versiones con la versión secuencial, estos factores son los mismos para el cifrado y el descifrado (en IDEA el cifrado es exactamente el mismo procedimiento que el descifrado salvo por el uso de las subllaves), el pobre desempeño de la paralelización a nivel de instrucción se debe a la relación/dependencia que hay entre las distintas operaciones realizadas y las palabras del bloque donde se realizan, haciendo que los hilos se ejecuten de manera secuencial.

En AES también se observó potencial de paralelización a nivel de instrucción, obteniendo factores de $\approx$ 10x y $\approx$ 4,2x para el cifrado y descifrado en la versión trivial, mientras que la versión con paralelización a nivel de instrucción obtuvo factores de $\approx$ 10x y $\approx$ 4x (por los mismos motivos que en IDEA), estos factores resultaron de comparar con la versión secuencial.

Un análisis más profundo del algoritmo de AES mostró las siguientes mejo-

ras en la versión con paralelización a nivel de instrucción: combinación de las operaciones $shiftRows$ y $subBytes$, obteniendo una mejora de 2 % sobre la primera versión a nivel de instrucción; usando una variable auxiliar se eliminó la divergencia de los *warp* y la necesidad de sincronizar los hilos. Esta nueva versión de AES obtuvo factores de $\approx$ 13x y $\approx$ 4,4x en cifrado y descifrado contra la versión secuencial.

Todo este trabajo fue en GPUs, hay otros dispositivos que podrían ser usados para este propósito: los coprocesadores Xeon Phi. Se esperarían resultados similares usando estos coprocesadores dada su arquitectura SIMD (Single Instruction Multiple Data) y su subsistema de memoria, que es de tarjeta gráfica (GDDR, teniendo un gran ancho de banda pero una gran latencia), sin embargo, su estudio y la implementación de los criptosistemas en estos dispositivos escapó al alcance de la tesis.

Bibliografía

[1] IBM 3330 data storage. URL `https://www-03.ibm.com/ibm/history/exhibits/storage/storage_3330.html`.

[2] Federal Information Processing Standards Publication 46: Data Encryption Standard (DES), 1977.

[3] Federal Information Processing Standards Publication 81: DES modes of operation, 1980.

[4] Announcing Development Of A Federal Information Processing Standard For Advanced Encryption Standard. Federal Register, 1997.

[5] Federal Information Processing Standards Publication 46-3: Data Encryption Standard (DES), 1999.

[6] Federal Information Processing Standards Publication 197: the official AES standard, 2001.

[7] CUDA C Programming Guide, 2007. URL `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[8] LSST, 2014. URL `https://www.lsst.org/`.

[9] CUDA Occupancy Calculator, 2015. URL `developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.

[10] A. J. Menezes, P. C. Van Oorschot y S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press.

[11] Maksim Bobrov. Cryptographic algorithm acceleration using CUDA enabled GPUs in typical system configurations. Master's thesis, Rochester Institute of Technology, 2010.

[12] Mark Harris. Overlap Data Transfers in CUDA C/C++, 2012. URL `https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/`.

[13] Jason Sanders y Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2010.

[14] Joan Daemen y Vincent Rijmen. AES proposal: Rijndael, 1999.

[15] Morris Dworkin. Recommendation for Block Cipher Modes of Operation. NIST Special Publication 800-38A, 2001.

[16] Samuel Neves. Cryptography in GPUs. Master's thesis, University of Coimbra, 2009.

[17] Ronald L. Rivest. A description of the RC2(r) encryption algorithm. Internet Network Working Group Request for Comments: RFC 2268., 1998.

[18] Gustavus J. Simmons. *Contemporary Cryptology: The Science of Information Tntegrity*. IEEE Press, 1992.

[19] Xuejia Lai y James L. Massey. *Advances in Cryptology — EUROCRYPT '90: Workshop on the Theory and Application of Cryptographic Techniques Aarhus, Denmark, May 21–24, 1990 Proceedings*, chapter A Proposal for a New Block Encryption Standard, pages 389–404. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.