Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Estudio sobre la implementación paralela de OCB de acceso aleatorio

Tesis que presenta

Luis Alejandro Pérez Sarmiento

para obtener el Grado de

Maestría en Ciencias

en Computación

Director de Tesis

Dr. Cuauhtemoc Mancillas López

Ciudad de México **Octubre de 2022**

Índice general

Índice de figuras

Índice de tablas

Resumen

En este trabajo se estudia el nivel de paralelismo de dos diferentes versiones del algoritmo de cifrado autenticado OCB (*Offset Code Book*). Las versiones incluidas en el presente trabajos son OCB3 y OCBRA (*random access OCB*). OCB es instanciado utilizando el cifrador por bloques AES (*Advanced Encryption Starndard*). El grado de paralelismo de OCB depende del cálculo de un valor Δ_i que sirve para enmascarar la entrada y la salida del cifrador por bloques de la siguiente manera: $E_K(P \oplus \Delta) \oplus \Delta$ donde $E_K(\cdot)$ es el cifrador por bloques y P es un bloque de mensaje. En OCB3 el Δ es calculado de una manera secuencial, es decir, Δ_i es dependiente de Δ_{i-1}. Como una solución a lo anterior se propuso OCBRA, en el cual todos los Δ_i pueden ser calculados de manera independiente.

Ya que todas las versiones de OCB pueden ser ejecutadas eficientemente en pipeline, se realizaron experimentos en el procesador Intel i7-11800H los cuales contienen registros de 512 bits, y los conjuntos de instrucciones AVX512 y VAES (AES-NI vectorial que permite cifrar 4 bloques de AES-128 en paralelo). La programación se realizó en lenguaje C utilizando las instrucciones intrínsecas de Intel.

Para demostrar que OCBRA tiene un grado mayor de paralelización se utilizaron GPUs (procesadores gráficos). OCB3 también puede ser paralelizado en GPUs pero requiere que todos los valores Δ_i sean precalculados. En este caso la programación se realizó en el lenguaje CUDA.

Los resultados obtenidos muestran que OCBRA presenta una mejora significativa en comparación con OCB3, OCBRA tiene un *Throughput* de 75.58 Gb/s contra los 54,57 Gb/s del OCB3 en sus respectivas versiones de 512 bits en procesadores Intel. Los experimentos muestran que OCBRA permite un mayor nivel de paralelización debido a la forma como se calculan los valores Δ_i, lo que permite un mejor uso del *pipeline* de instrucciones del procesador.

Abstract

In this work, we study the parallelism of two versions of the OCB (*Offset Code Book*) authenticated encryption algorithm. The versions included in this work are OCB3 and OCBRA (*random acces OCB*). OCB is instantiated using the block cipher AES(*Advanced Encryption Starndard*). The grade of parallelism depends on delta calculation which masks the input and output of the block cipher as follows $E_K(P \oplus \Delta) \oplus \Delta$ where $E_K(\cdot)$ is the block cipher, and P is a message block.

In OCB3, the Δ calculation is sequential. The Δ_i depends onΔ_{i-1}. As a solution to the above, OCBRA was proposed. In OCBRA, all delta can be calculated independently.

All versions of OCB could be executed efficiently in the pipeline. We perform different experiments on Intel processor I7-11800H, which contain 512-bit registers and AVX512 and VAES set of instructions (AES-NI vectorial, which allows encrypting four blocks in parallel). The codes are in C using an Intel intrinsics set of instructions.

To prove OCBRA has a more significant parallel grade. We use GPUs (Graphics Processors Units). OCB3 can also be parallelized on GPUs but requires computing all Δ_i values. In this case, all the codes are in CUDA.

The results obtained prove that OCBRA has the best performance than OCB3. OCBRA has a *Throughput* of 75.58 Gb/s against 54,57 Gb/s of OCB3 in theirs 512bits versions in Intel processor. The results prove that OCBRA allows a more significant parallel grade due to the way to calculate Δ_i values allowing better use of the processor's instruction *pipeline*.

Agradecimientos

Al Consejo Nacional de Ciencia y Tecnologia, CONACYT, por el apoyo económico otorgado al CVU: 1076902 para estudiar la maestría y al CINVESTAV por los conocimientos y espacios brindados.

Al Technology Innovation Institute, TII, por los conocimientos, el apoyo y espacios brindados.

Al Dr. Cuauhtemoc Mancillas López por ser mi asesor y mentor en esta nueva área de estudio.

Al Dr. Luis Rivera Zamarripa y al Dr. Gora Adj por el apoyo brindado durante la realización de los experimentos.

Al Dr. Francisco Rodríguez Henríquez por el apoyo durante este proceso de formación y darme la oportunidad de conocer nuevos horizontes.

A mis padres por el apoyo brindado durante estos dos años.

A Jazmín, Brian, Maza y Chuma por siempre darme el apoyo moral y emocional que necesitaba durante estos años.

A mi hermana por el apoyo incondicional que siempre estuvo presente.

A mis compañeros de la Maestría por estar siempre para pasar el rato.

Introducción

Planteamiento del problema

El mantener la confidencialidad, integridad y autenticación de información es una de las problemáticas actuales de todo tipo de comunicación. Dicha problemática se busca resolver usando múltiples herramientas entre ellas la implementación de modos de operación. Si bien algunos de estos pueden ofrecer dicha seguridad, muchos de ellos solo se encargan de mantener la confidencialidad del mensaje e incluso algunos otros tienen la desventaja de que toman un tiempo de ejecución bastante largo ya que son exclusivamente secuenciales, los modos de operación que presentan estas desventajas son: Libro de Códigos Electrónicos, Modo de Encadenamiento de Bloques, Retroalimentación del Cifrado y Salida Retro-alimentada.

El Libro de Códigos Electrónicos, Modo de Encadenamiento de Bloques, Retroalimentación del Cifrado y Salida Retro-alimentada, están pensados para procesar conjuntos de datos que no necesiten una autenticación, debido a esto solo son usados para el cifrado del mensaje, en caso de requerir una autenticación es necesario hacer uso de otro tipo de construcciones estas pueden estar basadas en los mismos tales como: PMAC y CBC-MAC, la principal desventaja de estas construcciones es la doble lectura requerida del mensaje para obtener una autenticación. Un detalle importante de estos modos de operación es la construcción secuencial en la que estan basados, obviando al Libro de Códigos Electrónicos, muchas construcciones criptográficas no fueron diseñadas con propiedades de paralelización, debido a que no se contaba con arquitecturas paralelas, multi-núcleos o vectoriales.

Cuando se tiene el problema de cifrar grandes cantidades de datos en poco tiempo, es necesario considerar si las construcciones secuenciales son una opción viable, además estas construcciones tomaran el doble de tiempo si es necesario autenticar el mensaje debido a la doble lectura. Una propuesta para la solución de este problema fue el Offset Codebook (OCB), un modo de operación que permite obtener la autenticación haciendo una única lectura al mensaje, no obstante este modo de operación sigue presentando problemas para la paralelización a gran escala, es por ello que se presento una nueva versión del OCB, el *OCB de Acceso Aleatorio*, que sus autores afirman que puede tomar provecho de un paralelismo a gran escala.

En esta tesis se plantea responder la siguiente pregunta: ¿Es posible obtener una mejora en el rendimiento usando *OCB de Acceso Aleatorio* comparado con las versiones previas del OCB cuando se utiliza un paralelismo a gran escala? Para ello se realizaran implementaciones de *OCB de Acceso Aleatorio*, tanto en GPUs como en AES-NI vectoriales presentes en ciertos procesadores de uso general. Se harán comparaciones con otras versiones de OCB tomando en cuenta las características de cada uno.

Objetivos

General

Comprobar el rendimiento del modo de operación *OCB de Acceso Aleatorio* contra las implementaciones existentes de OCB, utilizando GPUs y AES-NI vectorial.

Particulares

1. Comprender el modo de operación OCB, su funcionamiento y características junto a sus distintas implementaciones.

2. Analizar las distintas arquitecturas de computadoras recientes, tanto en CPU como GPU.

3. Lograr una implementación eficiente del *OCB de Acceso Aleatorio* en CPU y GPU.

4. Comparar la eficiencia del *OCB de Acceso Aleatorio* frente a las versiones anteriores de OCB.

Estado del arte

Existen muchos trabajos en la literatura sobre diseño de modos de operación de cifrado autenticado y sus implementaciones. En la Tabla 1 se presentan los trabajos con mayor relevancia al tema de tesis.

En el artículo realizado por Jha et al. [3], se efectuó una implementación del *OCB de Acceso Aleatorio*, en el mismo explican la eficiencia del modo de operación, el funcionamiento y cómo se lleva a cabo una paralelización del mismo, realizan una comparación con otras versiones de OCB, no obstante, su implementación es a nivel de procesador y usando arquitecturas diferentes a la pensada en este trabajo de tesis, además, de añadir el aditamento de las tarjetas aceleradoras gráficas.

Los artículos de Nishikawa et al.[5], Wang et al. [9] y Zhong et al. [6] presentan la paralelización de AES en tarjetas aceleradoras gráficas, a pesar de que todos hablan del mismo tema, el enfoque de cada uno es distinto. Nishikawa et al.[5] explica cómo utilizar

Título	Autor(es)	Año	Características
OCB Mode	Phillip Rogaway [4]	2001	Implementación del OCB Esquema del OCB
Acceleration of AES encryption on CUDA GPU	Nishikawa et al. [5]	2012	Implementación de AES en GPU Esquema de AES menores a 128 bits Comparación del rendimiento entre esquemas
Implementation and Analysis of AES Encryption on GPU	Zhong et al. [6]	2012	Comparativas de rendimiento de AES
Modes of operation of the AES algorithm	Blazhevski et al. [7]	2013	Esquemas de modos de operación Paralelización de modos de operación
Bitsliced High-Performance AES-ECB on GPUs	Kwei Lim et al. [8]	2016	Implementación de AES con Bitsliced
On Random Read Access in OCB	Jha et al. [3]	2019	Implementación del OCB Random Access Esquema del OCB Random Acces
GPU Accelerated AES Algorithm	Wang et al. [9]	2019	Implementación ECB en GPU Implementación de las tablas xtime
Making AES great again: the forthcoming vectorized AES instruction	Drucker et al. [10]	2019	Esquema de AES vectorizado

Tabla 1: Trabajos relacionados.

varios AES dentro de un solo bloque en las tarjetas aceleradoras gráficas y el desempeño que tiene ejecutar esto; Wang et al. [9] explica cómo paralelizar AES usando las tablas T en las tarjetas aceleradoras gráficas y el desempeño que tiene con estas; Zhong et al. [6] detalla el conjunto del desempeño de distintas tarjetas aceleradoras gráficas con múltiples arquitecturas sobre la paralelización en los modos de operación básicos. Es importante tener en cuenta las distintas formas de implementación de AES para entender de qué manera es más conveniente implementarlo en las tarjetas aceleradoras gráficas, debido a que una buena implementación de AES puede resultar en la diferencia entre un buen o mal rendimiento para el *OCB de Acceso Aleatorio*.

Otro modo de paralelizar AES es utilizar el conjunto de instrucciones del procesador, en el artículo de Drucker et al. [10] explica cómo se logra la paralelización de AES haciendo uso de estas instrucciones y cómo se espera a que mejore con instrucciones a añadidas a futuro.

El trabajo de Blazhevski et al. [7] explica cómo funcionan los modos de operación la diferencia entre ellos y cómo implementarlos, por otro lado Phillip Rogaway [4], explica exclusivamente cómo funciona el OCB y las ventajas que tiene, además, de dar una demostración de la seguridad del mismo. Entender los modos de operación permite tener una mejor visión de la implementación a realizar.

Por último, Kwei Lim et al. [8] explica cómo logró implementar el Bitsliced en el modo de operación ECB, con esto logro aprovechar cada instrucción de la arquitectura para obtener rendimientos excepcionales.

Esta demás decir que los modos de operación junto con AES es un tema altamente

estudiado en el paralelismo y cómo estos se comportan sobre distintos entornos paralelos, no obstante el modo de operación usado constantemente en todas estas pruebas es el conocido ECB un modo de operación altamente paralelizable de fácil implementación sin embargo, existen otros modos de operación altamente paralelizables los cuales no se han investigado a profundidad.

Organización de la tesis

En el capítulo 1 se expone todo el conocimiento requerido para el entendimiento de esta tesis. En el capítulo 2 se presenta una breve investigación de las arquitecturas y la evolución de estas hasta llegar a las arquitecturas actuales. En el capítulo 3 se explica a detalle las distintas versiones del OCB y sus características, así como su estructura. En el capítulo 4 se presentan las distintas versiones realizadas del OCB junto con las distintas técnicas usadas en sus implementaciones. En el capítulo 5 se expone los resultados obtenidos de las distintas pruebas, además de explicar cómo se realizaron estas junto con las características de los equipos usados. Por último en el capítulo 6 se describen las conclusiones obtenidas junto con el trabajo a futuro que se podría realizar.

Capítulo 1

Preliminares

En este capítulo se describe que es un modo de operación junto con los cifradores por bloques, se explica detalladamente el cifrador por bloques que se usara en todo el trabajo de tesis en este caso AES. Se definirá la construcción de estos a la par con los cifradores por bloque entonados, después se desarrollara la obtención del cifrado autenticado.

1.1. Notación

Sea $\{0,1\}^n$ el conjunto de todas las cadenas de n-bits, análogamente se considera como el campo $\mathbb{F}_2^n$, sus elementos pueden ser también considerados como polinomios de grado a lo más $n-1$. Sean $a, b \in \{0,1\}^n$, la adición es denotada por $a \oplus b$, que representa una or-exclusiva a nivel de bits. La concatenación es denotada como $a||b$, y la longitud en bits como $|a|$. La multiplicación $a \cdot b \bmod q(x)$ se realiza de forma polinomial módulo un polinomio irreducible $q(x)$. Sea $L \in \mathbb{F}_2^n$, $x^i L$ es una multiplicación de L por el elemento primitivo x[1] elevado a la potencia i y se denota como $xtimes(L, i)$. El monomio x visto como un número entero representa el 2, así que análogamente $xtimes(L, i)$ es $2^i L$. Dicha operación se calcula eficientemente con un corrimiento de bits a la izquierda y una xor condicional con el polinomio irreducible.

Una cadena de bits M de longitud arbitraria es dividida en bloques $M_1, ..., M_m$ donde $|M1| = |M_2| = ... = |M_{m-1}| = n$ y $|M_m| \leq n$, dicha cadena puede ser un mensaje o datos asociados. [2]

1.2. Cifradores por bloque

Un cifrador por bloques es un algoritmo determinista de cifrado de llave secreta, recibe como entrada un bloque de datos y una llave, entrega como salida un bloque cifrado.

[1] Si x se eleva a todas las potencias $i \in \{0, 2^n - 1\}$ generará todos los elementos de $\mathbb{F}_2^n$.

[2] Datos que se autentican pero no deben cifrarse.

A dicho bloque de entrada se le aplica múltiples veces una transformación invariante llamada función de ronda, la cual es controlada por la llave secreta. La longitud de la llave secreta puede variar según la función de cifrado, no obstante a partir de esta llave se derivan las llaves correspondientes para cada ronda, llamadas llaves de ronda.

Formalmente un cifrador por bloques es una función $E : \{0,1\}^n \times \{0,1\}^k \longrightarrow \{0,1\}^n$ donde $\{0,1\}^k$ es el espacio de llaves y $\{0,1\}^n$ es el espacio de mensajes, n es la longitud de bloque y k la longitud de la llave. Se denota como $E_K(\cdot)$ y su inversa como $E_K^{-1}(\cdot)$. Para toda $K \in \{0,1\}^k$ y $m \in \{0,1\}^n$, se cumple que $E_K^{-1}(E_K(m)) = m$, por lo que cifrador por bloques se comporta como una permutación del conjunto de las cadenas de n bits.

En general un cifrador por bloques permite obtener el cifrado de un bloque de mensaje de longitud fija, no obstante estos tienen problemas con bloques incompletos por no mencionar su más grande limitación, un cifrador por bloques siempre generará el mismo texto cifrado para el mismo patrón de mensaje, sin embargo, esta limitación se resuelve con el uso de los modos de operación (ver capitulo 1.3 para una explicación más detallada).

Existen diversos cifradores por bloques en la actualidad, algunos son: **estándar de cifrado de datos** (DES por sus siglas en ingles *Data Encryption Standard*), **estándar de cifrado avanzado** (AES por sus siglas en inglés *Advanced Encryption Standard*), **algoritmo internacional de cifrado de datos** (IDEA por sus siglas en inglés *International Data Encryption Algorithm*). El cifrador por bloque adoptado como estándar a partir del 2002 por ganar la competencia del Instituto Nacional de Estándares y Tecnología [11] (NIST por sus siglas en inglés *National Institute of Standards and Technology*) es el AES .

1.2.1. Estándar de Cifrado Avanzado

AES es un cifrador por bloques compuesto de una serie de permutaciones y substituciones, las entradas y salidas de este algoritmo constan de bloques de 128 bits. La llave usada en el algoritmo AES es 128, 192 o 256 bits de longitud, cualquier otro tamaño no es permitido en este estándar.

1.2.2. Bytes

Las unidades que procesa el algoritmos de AES son los bytes, estos son una secuencia de 8 bits tratados como una única unidad, tanto las entradas, las salidas y la llave se expresan en su cantidad de bytes. Se utiliza la notación a_i para referirse a un byte específico en una cadena de bytes, donde i puede tomar valores dado los siguientes rangos:

- Tamaño de la llave = 128 bits, $0 \leq i < 16$.
- Tamaño de la llave = 192 bits, $0 \leq i < 24$.
- Tamaño de la llave = 256 bits, $0 \leq i < 32$.

- Tamaño del bloque = 128 bits, $0 \leq i <16$.

Es conveniente la representación de los valores de los bytes usando el sistema hexadecimal, de esta forma es posible representar el valor del byte usando dos dígitos de 4 bits, como se denota en la figura 1.1.

Bits	Hexadecimal
0000	0
0001	1
0010	2
0011	3

Bits	Hexadecimal
0100	4
0101	5
0110	6
0111	7

Bits	Hexadecimal
1000	8
1001	9
1010	a
1011	b

Bits	Hexadecimal
1100	c
1101	d
1110	e
1111	f

Figura 1.1: Representación de los valores hexadecimales.

1.2.3. Matriz de estados

Las operaciones usadas por el algoritmo de AES están realizadas en un arreglo bidimensional o matriz de bytes, esta se conoce como la matriz de estado, la matriz de estados consta en cuatro filas de bytes, cada una contiene 32 bits. Cada byte de la matriz de estados esta representado por S, cada uno tiene dos índices, i y r, los cuales representa la fila y la columna al que pertenece.

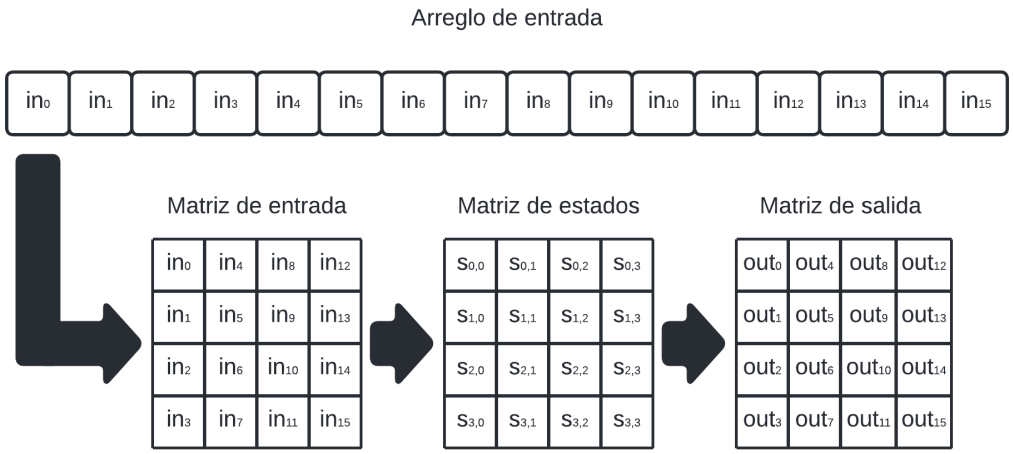

Figura 1.2: Representación del Arreglo de entrada, la matiz de entrada y salida y la matriz de estados de AES .

En la figura 1.2 se aprecia la transición de la matriz de entrada a la matriz de estados y de la matriz de estados a la matriz de salida, los 4 bytes en cada columna de la matriz de estados forman palabras de 32 bits, con ellas se puede expresar la matriz de estados como un arreglo unidimensional de palabras de 32 bits $w = w_0, w_1, w_2, w_3$ donde w_n está conformado por:

$$
\begin{aligned}
w_0 &= S_{0,0}S_{1,0}S_{2,0}S_{3,0};\\
w_1 &= S_{0,1}S_{1,1}S_{2,1}S_{3,1};\\
w_2 &= S_{0,2}S_{1,2}S_{2,2}S_{3,2};\\
w_3 &= S_{0,3}S_{1,3}S_{2,3}S_{3,3}.
\end{aligned}
$$

1.2.4. Transformaciones

AES especifica una función de ronda compuesta de cuatro transformaciones llamadas: Sustitución de bytes, Corrimiento de filas, Mezclado de columnas y Suma de la llave de ronda. El número de iteraciones de la función de ronda depende de la longitud de la llave. La ronda inicial solo contiene la suma de la llave de ronda y la ronda final no contiene el mezclado de columnas. A continuación se explicará cada una de ellas en detalle.

1.2.4.1. Sustitución de bytes

La sustitución de bytes es una transformación no lineal, consiste en la sustitución de los valores de la matriz de estado usando lo que se denomina como la caja S (ver tabla 1.1). Esta es una tabla de correspondencia de valores, los cuales se sustituyen en la matriz de estado.

La tabla 1.1 contiene los valores en hexadecimal para la sustitución, la tabla funciona de acuerdo a los nibles de cada byte, donde donde los primeros cuatro bits corresponden al valor horizontal de sustitución y los siguientes cuatro al valor vertical, por ejemplo si tenemos el valor $\{5, 8\}$ este se sustituiría por el valor $\{6, a\}$.

1.2.4.2. Corrimiento de filas

El corrimiento de filas realiza un corrimiento circular en las últimas tres filas de la matriz de estado, la primer fila no es afectada, la segunda tercera y cuarta reciben corrimientos circulares de uno, dos y 3 bytes respectivamente, como se muestra en la figura 1.3.

1.2.4.3. Mezclado de columnas

El mezclado realiza una multiplicación de cada columna de la matriz de estado por una matriz constante (ecuación 1.1), toda la aritmética es hecha en el campo $\mathbb{F}_2^8$.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
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

Tabla 1.1: Valores de la tabla de sustitución.

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \quad para\ 0 \leq c < Nb. \tag{1.1}$$

1.2.4.4. Suma de la llave de ronda

La suma de llave de ronda es el proceso en el cual a la matriz de estado se le suma la llave de ronda para obtener el nuevo estado. Se denota como:

$$s(x_i)' = k(x_i) \oplus s(x_i).$$

Donde $k(x_i)$ representa la llave de ronda actual y $s(x_i)$ representa el estado actual.

1.2.5. Expansión de llaves

La expansión de llaves es el proceso en el cual se derivan las llaves de ronda necesarias para el cifrado y descifrado de datos. El tamaño de la llave en AES, puede tomar los valores de 128, 192 o 256 bits. El tamaño de la llave esta representado por $Nk = 4$, 6, u 8, los cuales indican el número de palabras de 32 bits que contiene dicha llave. El número

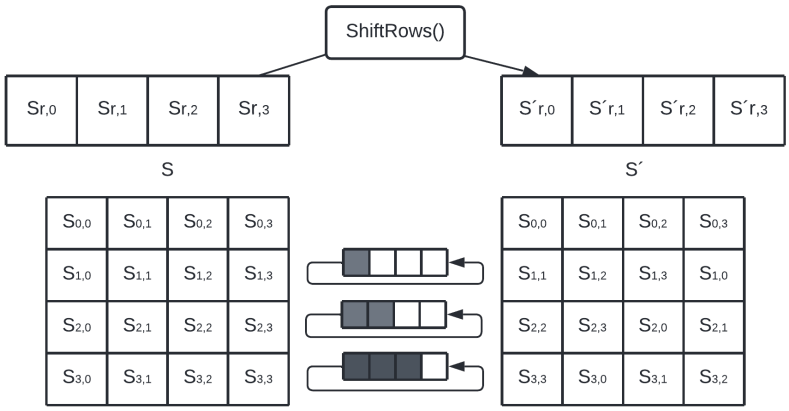

Figura 1.3: Realización del corrimiento de filas.

de rondas de AES depende directamente del tamaño de la llave, el número de rondas está representado por Nr, donde $Nr = 10$ cuando $Nk = 4$, $Nr = 12$ cuando $Nk = 6$ y $Nr = 14$ cuando $Nk = 8$. Las combinaciones posibles que conforman el estándar de AES se muestran en la tabla 1.2.

	Tamaño de la llave (Nk)	Tamaño del bloque (Nb)	Número de rondas (Nr)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Tabla 1.2: Combinaciones posibles del tamaño de la llave con las rondas de AES.

El resultado de la expansión de llaves consiste en un arreglo lineal de palabras de 4 bytes, $W[i]$ donde i puede tomar los valores de $0 \leq i < Nb(Nr + 1)$, $Nb(Nr + 1)$ indica el número total de palabras generados por la expansión de llaves.

La expansión de llaves sigue el pseudocódigo mostrado en el algoritmo 1, para toda palabra $w[i]$, sera igual al XOR de la palabra previa, $w[i-1]$, y la palabra Nk, $w[i-Nk]$, para las palabras que son múltiplos de Nk, es necesario aplicar una transformación a la palabra $w[i-1]$, seguido de realizar un XOR con la constante de ronda $Rcon[i]$, esta transformación consiste en un corrimiento cíclico de los bytes de la palabra, seguido de la aplicación de la sustitución de bytes a la palabra de 32 bits.

Algorithm 1 Pseudo código de la expansión de llaves

```
temp ← 0
i ← 0
while i < Nk do
 |  w[i] = palabra(llave[4 ∗ i], llave[4 ∗ i + 1], llave[4 ∗ i + 2], llave[4 ∗ i + 3])  i + +
end
i ← Nk
while i < Nb ∗ (Nr + 1) do
 |  temp = w[i − 1]  if i%Nk = 0 then
 |  |  temp = SubPalabra(RotPalabra(temp)) ⊕ Rcon[i/Nk]
 |  else
 |  |  if Nk > 6  &  i%Nk = 4 then
 |  |  |  temp = SubPalabra(temp)
 |  |  end
 |  end
 |  w[i] = w[i − Nk] ⊕ temp
 |  i + +
end
```

1.2.6. Descifrado

Es posible invertir el proceso de AES para obtener una función de descifrado, por lo cual este se puede implementar en orden inverso, para ello las transformaciones tienen ligeros cambios, estas transformaciones son: **Sustitución invertida de bytes**, **Corrimiento invertido de filas** y **Mezclado invertido de columnas**. La suma de llave de ronda no tiene un inverso al ser simplemente un XOR de la llave con el estado correspondiente.

1.2.6.1. Sustitución invertida de bytes

La sustitución invertida de bytes es el inverso de la sustitución de bytes, esta nueva transformación aplica una sustitución de valores a cada byte de la matriz de estado de la misma forma que la sustitución de bytes, pero usando una nueva caja S, la caja S inversa también nombrada caja S^{-1} (ver figura 1.3).

1.2.7. Corrimiento invertido de filas

El corrimiento de filas invertido es el inverso del corrimiento de filas, este aplica un corrimiento cíclico a la matriz de estado de igual forma que su contra parte, no obstante la dirección de este es contraria estos son corrimientos a la derecha.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
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

Tabla 1.3: Valores de la tabla de sustitución inversa.

1.2.8. Mezclado invertido de columnas

El mezclado invertido de columnas es la función inversa del mezclado de columnas, en esta transformación al igual que la anterior se opera la matriz de estado usando cada columna como un polinomio en el campo $\mathbb{F}_2^8$ y multiplicándola por la matriz inversa a la usada en el mezclado de columnas. (ver ecuación 1.2).

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \times \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \quad para\ 0 \leq c < Nb. \tag{1.2}$$

1.3. Modos de operación de cifradores por bloque

Los cifradores por bloque solo pueden procesar mensajes de una longitud fija, generalmente corta cómo pueden ser 64 o 128 bits. En las aplicaciones de la vida real, los mensajes a proteger son mucho más grandes que los tamaños mencionados. Para procesar mensajes de longitudes mayores utilizan los modos de operación, estos se definen como:

$$\mathbf{E} : \{0,1\}^k \times \{0,1\}^{iv} \times \{0,1\}^m \longrightarrow \{0,1\}^m.$$

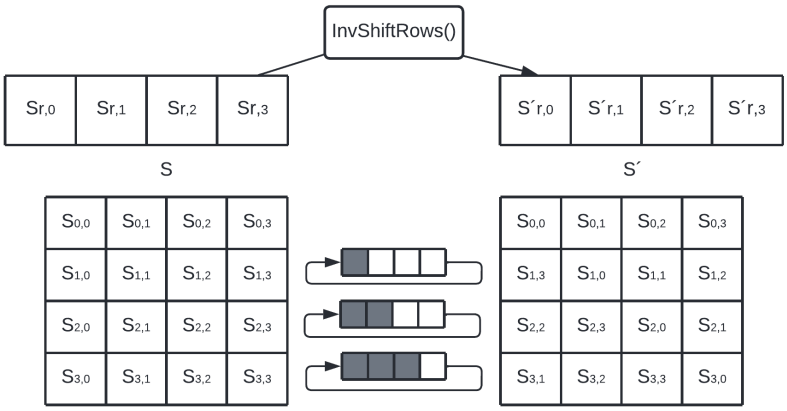

Figura 1.4: Realización del corrimiento invertido de filas.

Donde $\{0,1\}^k$, $\{0,1\}^{iv}$ y $\{0,1\}^m$ representan el espacio de la llave, los vectores de inicialización y el mensaje respectivamente. Los modos de operación de cifradores por bloques permiten cifrar más de un solo bloque de texto usando la misma llave manteniendo la seguridad. Después de seleccionar el nuevo estándar de cifrado avanzado en el 2001 por el NIST [11], se dieron a conocer cinco modos de operación estandarizados para su uso: *Electronic Code Book (ECB)*, *Cipher Block Chaining (CBC)*, *Cipher FeedBack (CFB)*, *Output FeedBack (OFB)* y *Counter mode (CTR)*.

1.3.1. Libro de códigos electrónicos

Libro de códigos electrónicos (ECB por sus siglas en inglés), es un modo de operación en el cual se aplica directamente la función de cifrado a cada bloque del mensaje a cifrar,su principal desventaja es que textos en claros idénticos producirán salidas de cifrado iguales (ver Figura. 1.5).

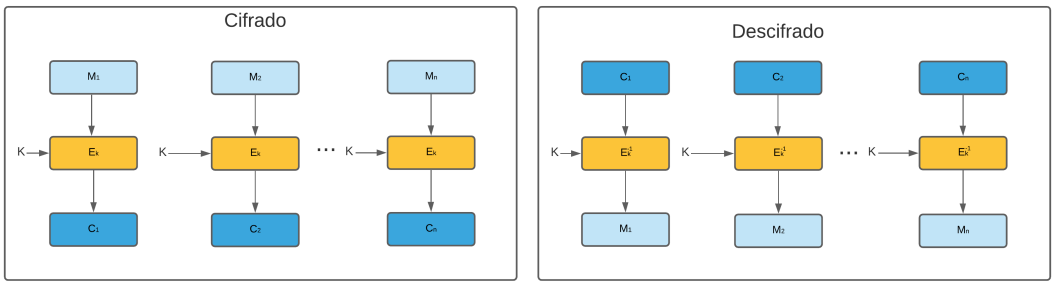

Figura 1.5: Modo de operación ECB.

Esto se debe a que la posición de los bloques no influye en el resultado del cifrado, por lo tanto la autenticación e integridad del mensaje están comprometidas, suponga un mensaje $M = M_1, M_2, M_3, M_4$ donde $M_1 = M_2$ por lo tanto el texto cifrado es $C = E_k(M_1), E_k(M_2), E_k(M_3), E_k(M_4)$ el cual da como resultado $C = C_1, C_1, C_3, C_4$ (ver ecuación 1.3) debido a que $M_1 = M_2$ son iguales, esto produce que la integridad y la autenticación del mensaje se comprometan, revelando patrones del mensaje. Este modo de operación es considerado altamente paralelizable esto se debe a la independencia en el cifrado de los bloques. Ni es recomendado su uso en protocolos.

$$M_2 = M_1; \tag{1.3}$$

$$E_k(M_1), E_k(M_2), E_k(M_3), E_k(M_4) = C_1, C_1, C_3, C_4. \tag{1.4}$$

1.3.2. Modo de encadenamiento de bloques

El modo de encadenamiento de bloques (CBC por sus siglas en inglés), es un modo de operación que implementa confidencialidad en el cual el proceso de cifrado combina el bloque actual del texto en claro con el resultado del bloque del cifrado anterior.

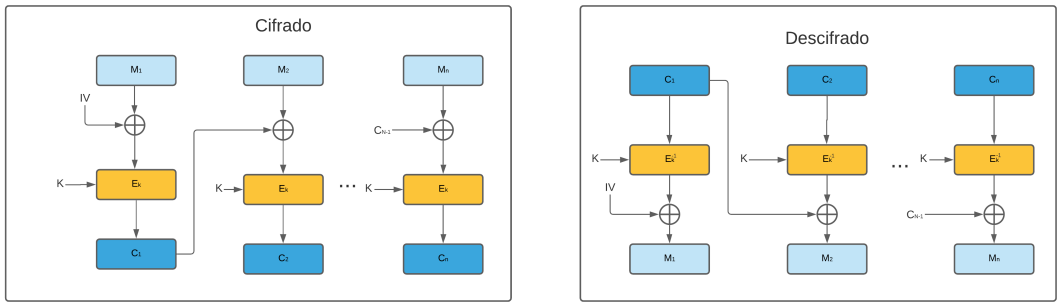

Figura 1.6: Modo de operación CBC.

Este modo de operación requiere el uso de un vector de inicialización (IV) el cual se combina con el primer bloque del texto en claro. El IV no necesita ser secreto pero si impredecible por lo cual generarlo es uno de los puntos más cruciales de este modo de operación. La integridad del vector de inicialización no debe de ser comprometida, además de que debe de ser protegida, en la figura 1.6 se muestra su diagrama a bloques.

En el modo de operación CBC el proceso de cifrado es el siguiente, al bloque de entrada M_0 del texto en claro se realiza un XOR con el vector de inicialización IV, al resultado se aplica la función de cifrado E_k con lo cual se obtiene el siguiente texto cifrado C_1, esta salida se usa para la entrada del siguiente bloque, en la cual se aplica un XOR con M_2 para luego proceder a la función de cifrado E_k y obtener C_3, este proceso se repite para todo el mensaje. En el modo de operación CBC cada bloque de entrada necesita el

resultado del bloque anterior para cifrar de esta forma la salida se vuelve impredecible, no obstante este modo de operación solo puede ser realizado de forma secuencial debido a la dependencia entre bloques.

1.3.3. Retroalimentación de Cifrado

La Retroalimentación de cifrado (CFB por sus siglas en inglés), es un modo de operación que proporciona la confidencialidad usando una retroalimentación sucesiva de los bloques cifrados como la entrada al cifrador por bloques. El bloque cifrado se obtiene al hacer una xor del bloque en claro y la salida del cifrador, en la figura 1.7 se ilustra el esquema del CFB. Es importante resaltar que el mensaje en claro nunca pasa a través del cifrador por lo que este modo de operación no necesita la inversa del cifrador por bloques. No es paralelizable debido a la dependencia entre bloque sucesivos.

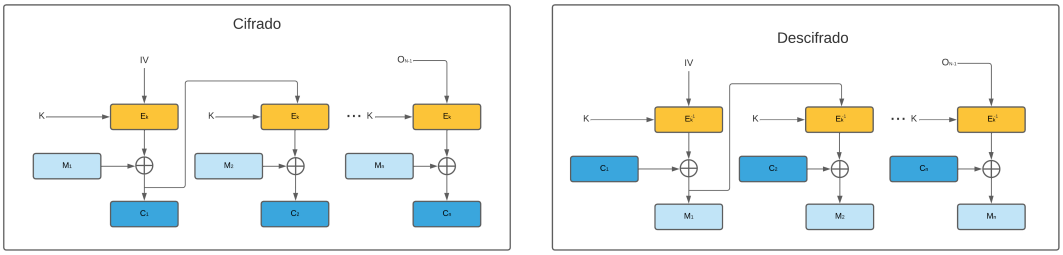

Figura 1.7: Modo de operación CFB.

1.3.4. Salida retro-alimentada

El modo de operación de salida retro-alimentada, OFB (por sus siglas en inglés), este modo de operación comparte la misma idea que en CFB, no obstante la principal diferencia es la entrada del bloque de cada iteración. En el primer bloque se utiliza el vector de inicialización, mientras que en los siguientes es el resultado del bloque cifrado, antes de realizar el XOR con el texto en claro. Es un análogo a crear una cadena de bits pseudoaleatorios de la misma longitud del mensaje usando el cifrador por bloques, par después ser utilizado para cifrar mediante una XOR.

Al ser un modo de operación donde cada bloque depende del anterior no es posible paralelizarlo, sin embargo es posible realizar todos los bloques cifrados y paralelizar el XOR de salida de cada bloque, en la figura 1.8 se aprecia el esquema del OFB. De igual manera que el CFB no requiere el inverso del cifrador por bloques.

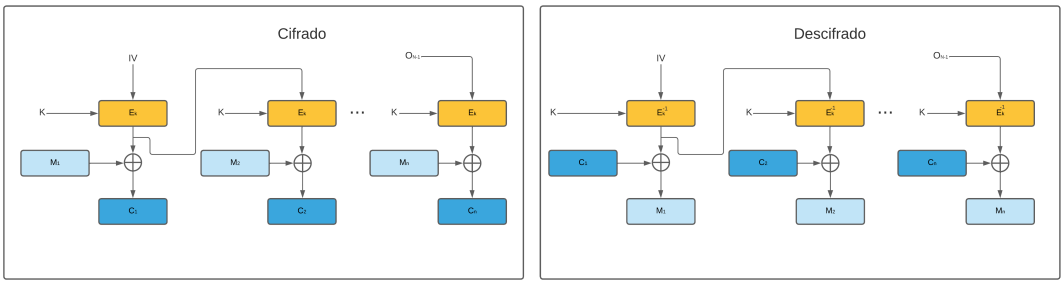

Figura 1.8: Modo de operación OFB.

1.3.5. Modo contador

El modo contador (CTR por sus siglas en inglés) es un modo de operación que permite cifrar datos usando un vector de inicialización y un contador, este modo de operación puede cifrar bloques según la cantidad de bits que se reserven para el contador, comúnmente 8 bits. El contador indica el bloque a cifrar, la entrada a cifrar de cada bloque es el vector de inicialización concatenado con el contador, de esta forma la entrada para cada bloque es distinta. El resultado de este proceso se utiliza para hacer un XOR con el bloque de mensaje correspondiente. Al igual que el OFB, genera una cadena de bits mediante el cifrador por bloques que después se utiliza para cifrar mediante una XOR.

Al no existir dependencia con los bloques anteriores este modo de operación es altamente paralelizable, en la figura 1.9 se ilustra el modo contador.

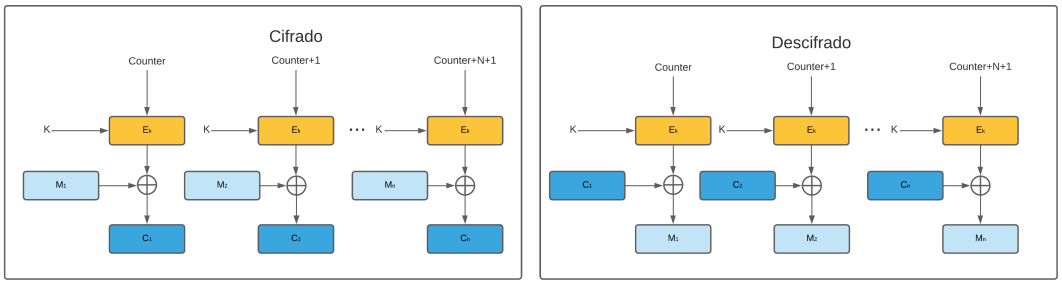

Figura 1.9: Modo de operación CTR.

1.3.6. Desventajas de los modos de operación de cifradores por bloques

Los modos de operación presentados anteriormente: ECB, CBC, CFB, OFB y CTR fueron pensados para el cifrado de mensajes, sin embargo, en implementaciones que requieren la autenticación del mensaje no son una opción viable, no cuentan con dicha

Modo de Operación	Paralelizable	Variabilidad	Confidencialidad	Integridad	Autenticación
Libro de códigos electrónicos	Si	No	Depende implementación	Si	No
Modo de encadenamiento de bloques	No	Si	Si	Si	Depende implementación
Retroalimentación de Cifrado	No	Si	Si	Si	No
Salida retro-alimentada	No	Si	Si	Si	No
Modo contador	Si	Si	Si	Si	No

Tabla 1.4: Comparación de los distintos modos de operación propuestos por el NIST.

característica, es necesario hacer uso de otro tipo de construcciones para poder obtener la autenticación del mensaje que generalmente terminan en una doble lectura del mensaje en caso de seguir usando estos modos de operación tradicionales.

Una desventaja clara de los modos de operación presentados es la falta de paralelismo; CBC, OFB y CFB son modos de operación secuenciales, es decir, requieren procesar el bloque o parte del bloque anterior antes de avanzar al siguiente lo cual genera que en caso de contar con arquitecturas multi-nucleo o multiprocesador no se puedan aprovechar de una forma eficiente, perdiendo desempeño. Los modos que son paralelizables ECB y CTR presentan otros desperfectos; ECB no cuenta con entropía en el sistema, esto puede llegar a filtrar cierto tipo de información sin revelar el mensaje (ver figura 1.10). Por ultimo el CTR a pesar de contar con variabilidad y ser paralelizable no permite obtener la autenticación sin usar otra construcción extra por lo cual el desempeño puede llegar a ser lento contra otras opciones.

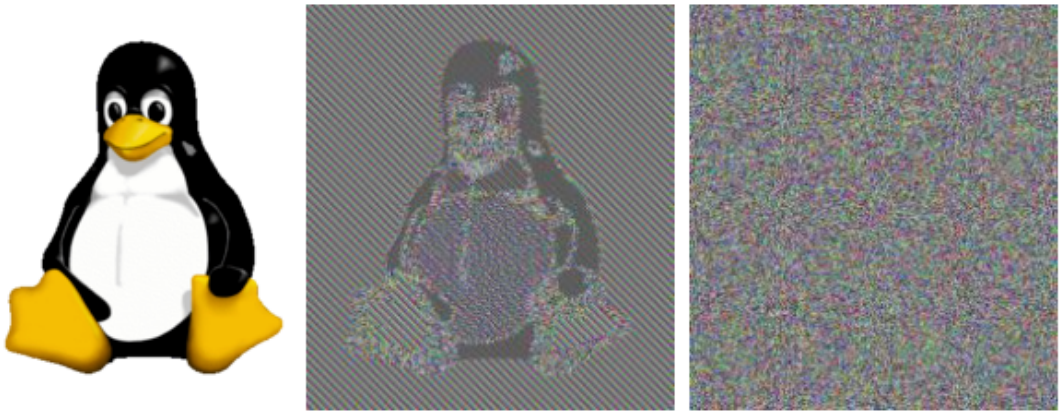

Figura 1.10: Resultado de Cifrar una imagen usando ECB contra otros modos de operación.

Una opción clara a tener en cuenta como respuesta a la necesidad de la autenticación del mensaje, la variabilidad del sistema y de fácil paralelización es el *Offset Codebook*, un modo de operación basado en un cifrador por bloque entonado, los cuales permiten

tener autenticación y cifrado haciendo una única lectura del mensaje, no obstante, el principal problema de estos es la construcción poco eficiente que pueden llegar a tener, sin embargo, *OCB de Acceso Aleatorio* es una alternativa con una construcción elegante a tener en cuenta.

1.4. Cifradores por bloque entonados

Los cifradores por bloques son inherentemente deterministas: dado un mensaje y una llave, siempre se obtendrá la misma salida cifrada. Muchos esquemas de cifrado simétrico, utilizan cifradores por bloques añadiendo un requerimiento que permita diferenciar cada instancia del mismo para evitar ataques futuros. Comúnmente este requerimiento se subsana haciendo cada bloque de entrada dependiente de su posición.

Intentar resolver el problema del determinismo de los cifradores por bloques intentando mantener la misma llave debido al costo que existe de generarla, resulta en muchos intentos de insertar variabilidad al esquema, sin embargo, manipular la entrada antes del cifrado o la salida después del cifrado resulta en diseños poco elegantes. En el articulo [12] se propone una solución al problema denominada *tweakable block cipher* en español conocidos cómo cifradores por bloque entonados, el cual propone un mecanismo de variabilidad, que se define como:

$$\tilde{E} : \{0,1\}^k \times \{0,1\}^t \times \{0,1\}^n \longrightarrow \{0,1\}^n.$$

Esta construcción cuenta con tres entradas, una llave K$\in \{0,1\}^k$, un tweak T$\in \{0,1\}^t$ y un mensaje M$\in \{0,1\}^n$, la cual produce una única salida, el mensaje cifrado C $\in \{0,1\}^n$. Esta nueva entrada denominada *tweak* no es distinta de un *nonce*[3] o un vector de inicialización al ser también público. El principal punto de los cifradores por bloques entonados es la eficiencia, se espera que el tweak pueda cambiar de forma frecuente por lo tanto una de sus características es que sea eficiente. Además cambiar el tweak no debe de afectar o recalcular la llave para el cifrador y el costo del cambio del tweak debe ser menor que el calculo de la llave.

Un cifrador por bloques entonados debe de mantener su seguridad a pesar de que el tweak sea comprometido, el propósito del tweak no es dar más seguridad al cifrador por bloques sino dar la variabilidad que el sistema necesita, por ende el tweak no es secreto.

1.4.1. Construcciones de cifradores por bloque entonados

El principal problema de los cifradores por bloque entonados es la eficiencia, debido a que las operaciones extras para obtener la variabilidad del sistema deben de ser funciones con un costo menor al de actualizar la llave, Phillip Rogaway en [13] introduce dos

[3]Se refiere a un valor que debe ser único para cada llamada al cifrador

construcciones eficientes de cifradores por bloques entonados XE y XEX, se definen como sigue:

Definición 1 (XE) *Sea E_k un cifrador por bloques, $\alpha_1, \cdots, \alpha_n \in \mathbb{F}_2^*$ y $\mathbb{I}_1, \cdots \mathbb{I}_n \subseteq \mathbb{Z}$. Entonces $\tilde{E} = XE[E, \alpha_1^{\mathbb{I}_1}, \cdots, \alpha_k^{\mathbb{I}_k}]$ es el cifrador por bloques entonado representado por la función $\tilde{E} : \mathcal{K} \times (\{0,1\}^n, \times \mathbb{I}_1 \times \cdots \times \mathbb{I}_k) \times \{0,1\}^n \longrightarrow \{0,1\}^n$ y denotado como $\tilde{E}_K^{Ni_1 \cdots i_k}(M) = E_k(M \oplus \Delta)$ donde $\Delta = \alpha_1^{i_1} \cdots \alpha_k^{i_k}$ y $N = E_k(N)$.*

Definición 2 (XEX) *Sea E_k un cifrador por bloques, $\alpha_1, \cdots, \alpha_n \in \mathbb{F}_2^*$ y $\mathbb{I}_1, \cdots \mathbb{I}_n \subseteq \mathbb{Z}$. Entonces $\tilde{E} = XE[E, \alpha_1^{\mathbb{I}_1}, \cdots, \alpha_k^{\mathbb{I}_k}]$ es el cifrador por bloques entonado, representado por la función $\tilde{E} : \mathcal{K} \times (\{0,1\}^n, \times \mathbb{I}_1 \times \cdots \times \mathbb{I}_k) \times \{0,1\}^n \longrightarrow \{0,1\}^n$ y denotado como $\tilde{E}_K^{Ni_1 \cdots i_k}(M) = E_k(M \oplus \Delta) \oplus \Delta$ donde $\Delta = \alpha_1^{i_1} \cdots \alpha_k^{i_k}$ y $N = E_k(N)$.*

Una construcción XE esta enfocada en la autenticación de la fuente y la integridad de datos, esto se debe a su primer XOR con el mensaje, una construcción XEX es enfocada en la privacidad de los datos, si el mensaje cifrado se transporta sin una mascara es susceptible a encontrar patrones, por lo tanto es necesario enmascarar los datos transportados a pesar de que sean cifrados usando un XOR para la salida del mensaje como se muestra en la definición 2.

1.5. Códigos de autenticación de mensajes

En ocasiones solo se requiere verificar la autenticación de la fuente que envía el mensaje además de la integridad del mismo, es decir, no se requiere la privacidad. Para este propósito se utilizan los códigos de autenticación de mensajes MACs (por sus siglas en inglés). Son funciones de la forma $\{0,1\}^m \times \{0,1\}k\{0,1\}^{iv} \rightarrow \{0,1\}^\tau$ donde $\{0,1\}^m$, $\{0,1\}^k$, $\{0,1\}^{iv}$ y $\{0,1\}^\tau$ son los espacios de mensaje, de llave, de IV y de *tag*respectivamente. Se denota como $MAC_K(\cdot)$. Un algoritmo MAC genera una etiqueta de autenticación como $\theta = MAC_K(M)$, la cual se añade al mensaje como $M||\theta$. Cuando se recibe el par M acompañado de su $tag\theta$, quien recibe puede verificar el mensaje calculando$\theta' = MAC_K(M)$ y comparando si $\theta = \theta'$. Existen dos maneras de que la autenticación falle, es decir, $\theta \neq \theta'$. La primera es que las llaves utilizadas para generar θ y θ' sean diferentes, esto es una falla de la autenticación de la fuente emisora o receptora al no contar con la misma llave. La segunda tiene es que el mensaje M recibido haya sido modificado, lo que da lugar a una falla en la integridad de datos. Una función MAC es capaz de detectar el más mínimo cambio en el mensaje original, incluso un único bit. Algunas MACs usadas ampliamente son PMAC [14] y OMAC [15].

1.6. Cifrado autenticado

Una esquema de cifrado autenticado consiste de un algoritmo para el cifrado (se puede tuilizar alguno de los modos de operación descritos anteriormente) y de una función

de autenticación (puede ser una MAC). El esquema de cifrado autenticado recibe tres entradas, una llave, un mensaje y un *nonce*, como resultado entrega lo que será el texto cifrado junto con una etiqueta de autenticación *tag*, el descifrado tiene como entrada una llave, el texto cifrado, un *nonce* y el *tag*. Entrega como resultado el texto descifrado y una bandera que indica si la autenticación fue correcta o no, teóricamente si la autenticación no es correcta no se entrega el mensaje descifrado. Una función de cifrado autenticado es de la forma

$$\mathbf{E} : \{0,1\}^m \times \{0,1\}^K \times \{0,1\}^N \times \{0,1\}^H \longrightarrow \{0,1\}^C \times \{0,1\}^\tau,$$

donde $\{0,1\}^M, \{0,1\}^K, \{0,1\}^N, \{0,1\}^H, \{0,1\}^C$ y $\{0,1\}^\tau$ representan los espacios del mensaje, de la llave, del $Nonce$, de datos asociados, el texto cifrado y el *tag*respectivamente. Para denotar un esquema de cifrado autenticado se usa $\mathbf{E}_K^{N,H}(\cdot,\cdot)$ y $\mathbf{D}_K^{N,H,T}(\cdot,\cdot)$ para el descifrado verificado.

El *tag* es el responsable de asegurar la autenticación e integridad del mensaje, el *tag* puede ser generado de múltiples formas no obstante no todas de ellas proveen la misma seguridad en todos los casos. Un *tag*mal generado puede comprometer toda la seguridad del sistema. En [16] se presentan las formas de construir un algoritmo de cifrado autenticado utilizando un modo de operación de privacidad y una MAC, se dividen en tres tipos principales:

- **Cifrado-y-MAC al texto en claro** en este tipo de algoritmos se cifra el mensaje y se añade la MAC dado el texto en claro. En estos algoritmos es necesario obtener el texto en claro y luego compararlo con el *tag*.
- **MAC-luego-cifrado** en este tipo de algoritmos se añade el MAC al mensaje para luego realizar el proceso de cifrado en conjunto, por lo tanto la MAC está, para comprobar la autenticación en estos algoritmos es necesario primero descifrar el mensaje junto con el *tag*y luego verificarlo.
- **Cifrado-luego-MAC** En estos algoritmos primero se cifra el mensaje para obtener C al cual se le añade el *tag*, éste depende de C. Para verificar la autenticación del mensaje no es necesario descifrar primero el mensaje, solo comprobar el Tag resultante, una vez este coincida, se procede a descifrar el mensaje.

Capítulo 2

GPUs y CPUs

En este capítulo se expone las distintas arquitecturas de las GPU Nvidia y CPU Intel. Las GPU Nvidia son el entorno más estable de trabajo para el ámbito de las GPU. Las CPU de Intel cuentan con el conjunto de instrucciones vectoriales de 512 bits las cuales son imprescindibles para este trabajo de tesis a diferencia de sus contra partes del mercado donde estas instrucciones no están contempladas en los procesadores.

2.1. Unidades Gráficas de Procesamiento

Las unidades de procesamiento gráfico (GPU por sus siglas en ingles) es uno de los principales componentes en los equipos de computo en general, se encuentran en servidores, equipos de sobre mesa, equipos portátiles, teléfonos móviles, etc. Su principal función es la parte del procesamiento gráfico, no obstante esto no las limita para la realización de otro tipo de tarea. Las tarjetas de vídeo se componen de GPU para su funcionamiento, en ellas se encuentran igual una unidad de memoria y un bus de flujo de datos.

Las tarjetas gráficas tienen diversas características, estas dependen principalmente de la arquitectura en las que están basadas.

2.1.1. Historia

Las unidades gráficas de procesamiento han existido desde hace más de 30 años, una de las primeras tarjetas de vídeo comercializadas fue la **Color Graphics Array** o **CGA** de IBM introducida en 1981, esta tarjeta trajo consigo el primer estándar gráfico para la empresa IBM. La **CGA** solo permitía cambiar el color de las letras y el del fondo de la pantalla por las limitaciones de la época. Uno de los mayores inconvenientes que presentaba esta tarjeta es que servía exclusivamente para computadoras IBM limitando la compatibilidad de la misma. La resolución máxima que alcanzaba era de 640x200 píxeles con un máximo de 4 colores, se podían obtener hasta 16 (Ver tabla 2.1) colores distintos pero era necesario limitar la resolución de la tarjeta.

Paleta de colores del CGA	
Negro000000	BlancoFFFFFF
Azul0000AA	Azul intenso5555FF
Verde00AA00	Verde intenso55FF55
Cian00AAAA	Cian intenso55FFFF
RojoAA0000	Rojo intensoFF5555
MagentaAA00AA	Magenta intensoFF55FF
MarronAA5500	AmarilloFFFF55
Gris oscuro555555	Gris claroAAAAAA

Tabla 2.1: Paleta de colores de CGA.

Después de la llegada de CGA la investigación y desarrollo de las tarjetas de vídeo aumento, apareciendo en el mercado nuevas opciones tales cómo la **Hercules Graphics Card (HGC)**, **Enhanced Graphics Adapter (EGA)** y la **Extended Graphics Array (XGA)**. Dichas tarjetas aportaban mejores prestaciones.

Con la llegada de nuevas tecnologías, fue necesario un estándar para poder trabajar con ellas por lo cual se formó la **Asociación de Estándares de Electrónicos de Vídeo** (VESA por sus siglas en inglés). En 1992 la empresa *Silicon Graphics* popularizó el uso de tecnologías para 3D al mismo tiempo que abrió la interfaz de programación para su hardware usando OpenGL con el objetivo de convertirlo en el estándar para aplicaciones 3D.

Nvidia es una empresa fundada en 1993 por Jensen Huang, Chris Malachowsky y Curtis Priem, con el objetivo de diseñar y fabricar nuevos chips para tarjetas de vídeo, Nvidia fue una de las primeras empresas en comercializar de forma general las tarjetas de vídeo para un ámbito profesional y personal, el primer producto diseñado por ellos fue la **NVIDIA NV1** y fue comercializado bajo el nombre de *Diamond Edge 3D*, no fue hasta la salida de la Nvidia GeForce 256 que Nvidia generó una relevancia en el mercado, era la primer tarjeta que podía realizar operaciones de transformación e iluminación directamente en el procesador gráfico es decir no dependía de la unidad central de procesamiento, por lo mismo se desprendía una carga de dicha unidad y se mejoraba las posibilidades de desarrollar aplicaciones con un enfoque más visual.

2.1.2. Arquitectura

Nvidia como distribuidor y fabricante de GPUs marcó un antes y después en las tarjetas de vídeo durante el año 2006, Nvida comenzó el uso de arquitecturas que representaban un avance significativo en el paralelismo (ver Tabla 2.2),durante el 2006 Nvidia rediseñó toda la arquitectura que usaban sus tarjetas de vídeo junto a sus GPUs esta primer arquitectura usada se denomino cómo Tesla.

Arquitectura	Año de lanzamiento	Litografía	Die (Tamaño del procesador)	Rango de Núcleos
Tesla	2006	90 nm	G80	
Fermi	2010	40 nm	GF100	448-1792
Kepler	2012	28 nm	GK104	2496-3072
Maxwell	2014	28 nm	GM204	1024-4096
Pascal	2016	16 nm	GP102	2048-3584
Turing	2018	12 nm	TU102	2560
Ampere	2020	7 nm	GA102	3584-10752

Tabla 2.2: Arquitecturas de Nvidia.

2.1.2.1. Antes de Tesla

Antes de la llegada de la arquitectura Tesla las tarjetas de vídeo de Nvidia no tenían una arquitectura establecida para su fabricación, el diseño de las GPUs iba de la mano de tres secciones o capas las cuales podían contar con distintas cantidades de unidades, estas secciones eran: *Vertex Shader*, *Fragment Shader* y *Fragment Crossbar*.

El *Vertex Shader* era la parte encargada del calculo de los vértices, la cantidad de vértices que se podían procesar estaba limitado a las unidades que este tuviera.

El *Fragment Shader* era la parte encargada de procesar fragmentos de la imagen para su calculo en paralelo, sin embargo, la potencia de esta estaba fuertemente limitada a las unidades por bloque que contenían las tarjetas de vídeo.

El *Fragment Crossbar* era el canal encargado de la unión de los fragmentos generados por el *Fragment Shader* para poder transmitir la imagen deseada.

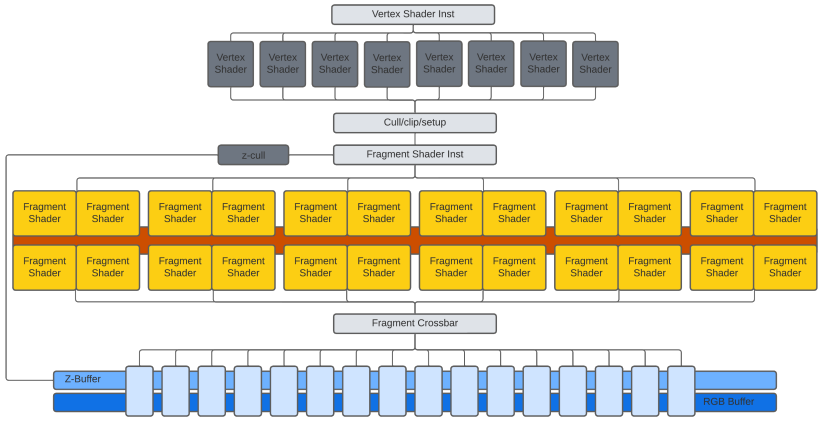

Figura 2.1: Diagrama de la arquitectura de la Geforce 7900 GTX de Nvidia

La Nvidia Geforce 7900 GTX, fue una tarjeta de sobre mesa que llevaba consigo la arquitectura por capas anteriormente descrita (ver Figura 2.1), en ella se puede apreciar sus 8 unidades de *vertex shader*, 24 unidaddes de *fragment shader* y sus 16 unidades de

fragment crossbar.

El principal problema de este diseño de construcción se presenta cuando una imagen necesita más unidades de las que contiene alguna de las secciones, debido a la dependencia en la comunicación que existía entre cada capa, por lo cual el proceso se volvía complicado y lento.

La dependencia entre capas genera que una no pueda avanzar hasta que la otra haya terminado el proceso actual, por lo cual se generaba lapsos de tiempos muertos cuando un proceso requería más una capa que las otras, de la misma manera encontrar el punto de equilibrio entre la cantidad de unidades en cada capa era una tarea sumamente complicada, principalmente por el espacio limitado del GPU.

Todos estos problemas se resolvieron con la llegada de la siguiente arquitectura de Nvidia, la arquitectura *Tesla*.

2.1.2.2. Tesla

La llegada de la arquitectura **Tesla** introdujo nuevos conceptos en el diseño de la construcción de las arquitecturas (ver Figura 2.2), estos son: *Stream multiprocessor*, *CUDA core*, *Warp*, *Special Function Unit*, *Multi Thread Issue*, *Register File*, *Shared Memory*.

- El *Stream multiprocessor (SM)* fue la solución a los problemas de ejecución y tiempos muertos de las arquitecturas anteriores, los SM reemplazaban a todas las unidades anteriores debido a la capacidad de procesar vértices, generar fragmentos y la unión de fragmentos, usando un único kernel, por lo cual la carga de trabajo se equilibra por SM dependiendo de la necesidades de cada momento.

- Los *CUDA core* son los nuevos núcleos usados por las tarjetas de vídeo, anteriormente eran denominados *Unit Shader*, no obstante estos núcleos tenían otro tipo de enfoque, los CUDA core tienen la capacidad de manejar múltiples hilos por núcleo, no obstante cada uno solo puede manejar datos de 32 bits.

- Los *Warps* son bloques de 32 hilos los cuales pueden ejecutar la misma instrucción de forma simultanea a distintos datos, estos bloques hicieron que cuda se renombrara de ser una arquitectura *Single Instruccion Multiple Data* a *Single Instruccion Multiple Thread*.

- Los *Special Function Unit (SFU)* son las unidades encargadas de los cálculos complejos de la arquitectura, los cuales no puede realizar de forma independiente cada núcleo CUDA, estos pueden ser raíces cuadradas, inversas, senos, cosenos, tangente, etc.

- El *Multi Thread Issue (MTI)* es la región encargada de la planificación de los hilos, en ella se establece en que tiempo y momento cada hilo sera usado en cada núcleo CUDA, todo esto para evitar conflictos y tiempos muertos de los núcleos.

- El *Register File* es el bloque que contiene los datos asociados a la región a procesar, en este bloque se contiene el estado de cada hilo de cada *warp*.
- A diferencia del anteriormente mencionado *Register File* el *Shared Memory* se encarga de mantener valores exclusivos para cada *warp*, el *Shared Memory* se encuentra ubicado con un acceso directo a los hilos del sistema por lo tanto su tamaño es reducido a cambio de una gran velocidad de lectura o escritura.

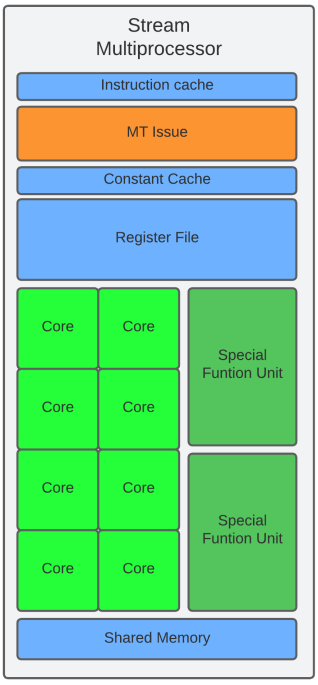

Figura 2.2: Diagrama del *Stream Multiprocessor* de la arquitectura Tesla de Nvidia.

Tesla fue la arquitectura que resolvió el problema de complejidad de las arquitecturas anteriores, en esta arquitectura Nvidia eliminaba la distinción entre capas, se usó los *Stream multiprocessor* en toda la arquitectura debido a que esto permitía equilibrar de forma más accesible el uso del sistema en todo momento. A partir de esta arquitectura Nvidia utilizó los núcleos CUDA, descontinuando por completo a sus antecesores, los *Unit Shader*, incluyo múltiples bloques de *warps*, una *shared memory* de 16KB y un *Register file* de 4KB

Esta arquitectura contaba con dos *Special Function Unit*, estas unidades podían ejecutar una instrucción compleja usando un único ciclo de reloj no obstante debido a que la arquitectura contaba únicamente con dos, era necesario tener una cola de espera. Con esta arquitectura Nvidia introdujo el lenguaje de programación para GPU denominado CUDA basado en el lenguaje de programación C y su compilador.

2.1.2.3. Fermi

Con el éxito de la arquitectura Tesla, Nvidia opto por volverla la base para las arquitecturas futuras. La arquitectura **Fermi** fue la sucesora directa de Tesla, en esta nueva arquitectura no se puede apreciar un cambio o reestructuración radical como Tesla no obstante el principal aporte de Fermi fue en su nueva litografía a 40 nm en comparación a los 90 nm de Tesla, gracias a ello Fermi cuadruplico todo en el interior de la GPU, se añadieron la existencia de medios warp (bloques de 16 hilos) de manera simultanea, las SFU se duplicaron lo que derivo en el soporte de software de cálculos decimales de 64 bits, además de que los núcleos CUDA ahora podían resolver instrucciones de 32 bits por cada ciclo de reloj. Por ultimo se añadió un motor *Polymorph* el cual se encargaba de la obtención de vértices de los objetos y soporte para funciones de c++ cómo objetos y métodos virtuales.

2.1.2.4. Kepler

Kepler fue la primer arquitectura en presentar una mejora no solo en el rendimiento sino en las temperaturas, la arquitectura anterior Fermi, tenia grandes problemas de sobre calentamiento al igual que de consumo, al poco tiempo de uso alcanzaban temperaturas extremadamente altas, por lo tanto, kepler bajo la velocidad de reloj base de la tarjeta e unifico todo en un solo reloj, con ello solucionaron los problemas de consumo y sobrecalentamiento, además, al ser una arquitectura a 28nm les permitió añadir más núcleos lo que resulto en un mayor rendimiento.

2.1.2.5. Maxwell

A pesar de la mejora presentada en Kepler la siguiente arquitectura de Nvidia tuvo un reto mayor, principalmente porque no podían confiar en la reducción de tamaño de los transistores para mejorar el rendimiento, seguían estancados en los 28nm, no obstante se opto por quitar el enfoque que se había tenido en kepler y volver a la forma de trabajo de Fermi pero persistiendo las velocidades de los relojes de Kepler, en **Maxwell** se usaron un enfoque por medios warps, esto logro que se pudieran repartir de mejor manera en el procesador y que pudieran reducir su tamaño, por lo cual pudieron incluir más hilos en el sistema y se mejoro el pipeline del mismo por lo cual las velocidades y el rendimiento se vio altamente mejorado.

2.1.2.6. Pascal

La arquitectura **Pascal** era una copia de su predecesora, no había cambios cómo tal en la arquitectura tampoco cambios en los hilos o instrucciones, el único cambio importante fue el proceso de fabricación a 16nm, el cual fue más que suficiente para mejorar el rendimiento, la cantidad de SM por bloque era de 16 bloques por sección del chip gráfico,

junto a esto llego la esperada memoria GDDR5X por lo cual aumento el ancho de banda máximo lo que hizo que sobre saliera esta arquitectura frente a las pasadas, llegando a un máximo de 256 bits.

2.1.2.7. Volta

Volta como arquitectura trajo consigo un cambio muy importante para la empresa de Nvidia, en esta arquitectura se anunciaria por primera vez los *Tensor Cores 2.0*; núcleos capaces de realizar operaciones de coma flotante de 64 bits. Específicamente sirven para realizar multiplicaciones matriciales de coma flotante. Estos núcleos también tuvieron un detalle importante en las redes neuronales, debido a que con su llegada se acelero el tiempo de entrenamiento de estas. La arquitectura **Volta** fue una mejora significativa, construida a 12 nm y con nuevo núcleos marco un antes y después.

2.1.2.8. Turing

Turing es la arquitectura que tuvo el mayor cambio arquitectónico en los últimos años, no solo por sus 12nm, si no por los cambios de hardware que introdujo, en esta generación se añadió el hardware dedicado para *Ray tracing*, los *Tensor Cores 2.0* y los núcleos *Ray Tracing*.

- Los *Tensor Cores 2.0* son núcleos enfocados en instrucciones de 64 bits, por lo cual estos podían realizar instrucciones de coma flotante de forma independiente.
- Los núcleos *Ray Tracing* al igual que los *Tensor Cores* están enfocados en instrucciones de 64 bits, con una ligera diferencia, los núcleos *Ray Tracing* están pensados para el calculo en tiempo real de vértices en la imagen.

Turing esta basada en las primeras arquitecturas de Nvidia, la era antes de Tesla, en la cual todo estada dividido por capas, esto se debe a la inclusión de los núcleos Ray Tracing y los Tensor Cores, los cuales se encuentran en capas distintas del kernel.

Una de las características más importantes de esta arquitectura son los núcleos CUDA escalables, con esta nueva versión de los núcleos CUDA estos pueden implementar instrucciones de enteros o punto flotante según necesiten, otra característica importante son los punteros independientes, cada unidad warps ahora contiene sus propias direcciones de memoria por lo cual no hay posibilidad de conflicto y la comunicación entre ellos es más clara y simple.

2.1.2.9. Ampere

La arquitectura **Ampere** lanzada el 14 de mayo del 2020 realizo cambios importantes para Nvidia, teniendo diferencias notorias entre las versiones de centros de datos y de consumidores, las diferencias a remarcar son:

- Dos procesos distintos de fabricación
- Una diferencia notable en la capacidad de computo.

Nvidia contaba con dos procesos de fabricación distintos según el publico objetivo, es decir uso profesional y consumidor, siendo estos de: 7nm y 8 nm respectivamente. La capacidad de computo difiere a favor de las versiones del consumidor teniendo 8.6 en contra de los 8 de las versiones profesionales, este numero indica la versión en la que trabajan los SM, versiones más recientes soportan nuevas instrucciones. La nomenclatura para los Die de esta generación es GA10X. **Ampere** introdujo los *Tensor Cores 3.0* y los núcleos *Ray Tracing 2.0* llegando a tener eficiencias de aproximadamente el 30 % más que sus antecesores.

2.2. Unidades Centrales de Procesamiento

2.2.1. Historia

Las Unidades Centrales de Procesamiento surgen por la evolución en los componentes electrónicos, los avances en la reducción de tamaño de transistores permitieron la llegada de microprocesadores, pequeños componentes capaces de realizar tareas simples, pero no fue hasta la llegada de la arquitectura de von Neumann [17] en la cual se describiría por primera vez cómo debería de ser el funcionamiento de una Unidad Central de Procesamiento, esta arquitectura describe los componentes que debe de tener (ver Figura 2.3) los cuales son: una Unidad de Control, una Unidad Aritmética Lógica (ALU por sus siglas en ingles), una Unidad de Registros, una Memoria Principal y los sistemas de entrada y salida.

- La **unidad de control** es la encargada del orden dentro del sistema, tiene la capacidad de decidir en que momento se ejecuta cada tarea, además puede decidir la prioridad de las mismas.
- La **ALU** es el componente encargado de todos los cálculos aritméticos del sistema, estos son, sumas, restas, sumas con acarreo, restas con acarreo, etc. La ALU esta compuesta de múltiples compuertas lógicas las cuales le permiten realizar los cálculos mencionados anteriormente.
- El **contador de programa** es el componente que indica las instrucciones a realizar dado el contador, este puede avanzar o regresar según lo indique la instrucción.
- Los **registros** son las instrucciones del programa a ejecutar estas pueden variar según el programa, el registro es el acceso inmediato de información con el que cuenta la unidad central de procesamiento.

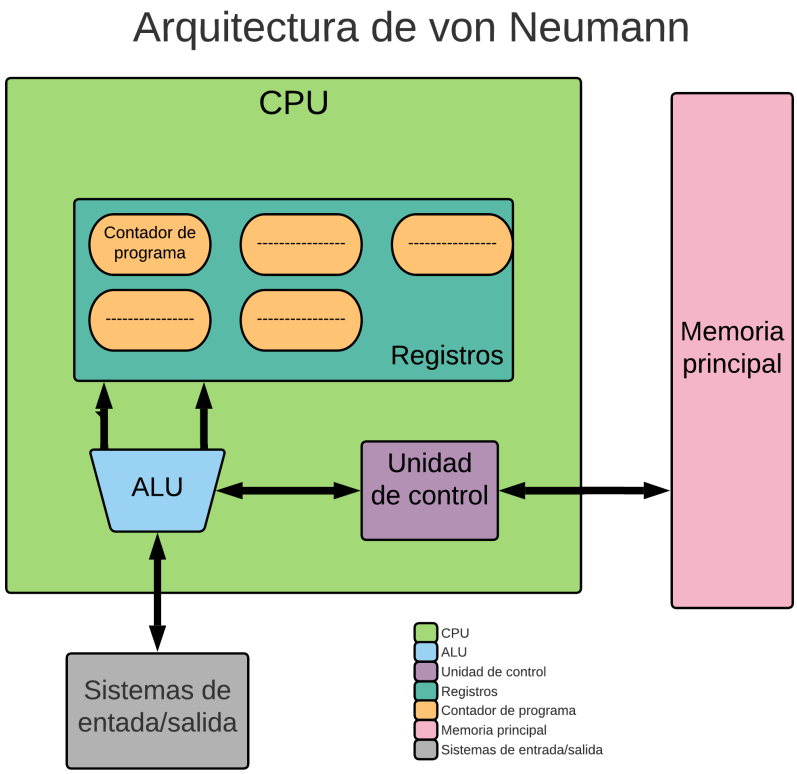

Figura 2.3: Diagrama de la arquitectura de Jhon Von Neumann.

- La **memoria principal** es el lugar donde se guardan los datos del programa a ejecutar o en ejecución, además de los valores requeridos para el funcionamiento del programa.
- Los **dispositivos de entrada y salida** es todo componente externo a la unidad central de procesamiento que permite la interacción del usuario con esta.

Hasta los primeros años de la década de 1970 los diferentes componentes electrónicos que conformaban un procesador no podían ser un único circuito integrado, para poder realizar un procesador era necesario tener múltiples chips dedicados de forma individual a cada tarea, estos comúnmente eran la ALU, la unidad de control y los registros. En 1971 la compañía Intel conseguiría crear el primer microprocesador del mundo en un único chip, el Intel 4004 , este procesador era de 4 bits y tenia cómo propósito ser un procesador comercial, impulsó el desarrollo de la calculadora.

Con el paso de los años los procesadores de Intel fueron ganando publico en el mercado, los modelos siguientes como el Intel 8008 reafirmo la posición de Intel como fabricante y distribuidor de procesadores.

El Intel 8086 es considerado sino el mayor logro de Intel de la década de 1970, uno de los mayores logros de la compañía hasta hoy en día, esto se debe a que este procesador presento por primera vez la arquitectura x86, una arquitectura con memoria segmentada para poder procesar información a más de 16 bits, después de su llegada se convirtió en el estándar de fabricación para los procesadores.

2.2.2. Arquitectura

Las unidades centrales de procesamiento (CPU por sus siglas en ingles), son el componente principal de todo equipo de computo, después de la llegada del Intel 8086 las arquitecturas y el proceso de fabricación presento un cambio drástico, a partir de este punto, todo procesador siguiente podía ejecutar programas de procesadores anteriores, la retro-compatibilidad se volvió un punto importante para toda arquitectura futura.

La llegada de los modelos de procesadores de Intel i3, i5, i7 eh i9, marco el siguiente salto generacional en el proceso y arquitecturas de procesadores, estos chips están divididos generacionalmente en cada generación se usa una arquitectura diferente que busca mejorar a la anterior. En la Tabla 2.3 se pueden apreciar las cinco arquitecturas más recientes de los modelos i3, i5, i7 eh i9, en ellas se puede ver cómo comparten ciertos conjuntos de instrucciones lo que permite la compatibilidad de programas entre los modelos más antiguos con los modelos más recientes, cabe resaltar que la litografía [1] para los procesadores Intel ha sido la misma desde hace seis generaciones.

Arquitectura	Año de lanzamiento	Litografía	Conjutnos de instrucciones	Generación
Skylake	2015	14nm	MMX, SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2, ADX, AVX, AVX2, AVX-512 (SkyLake-SP, SkyLake-W & SkyLake-X), TSX, FMA3 AES-NI, CLMUL, RDRAND, MPX, TXT, SGX VT-x, VT-d	Sexta
Kaby Lake	2016	14nm	MMX, AES-NI, CLMUL, FMA3, RDRAND SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2 AVX, AVX2, TXT, TSX, SGX VT-x, VT-d	Séptima
Coffee Lake	2017	14nm	MMX, AES-NI, CLMUL, FMA3, RDRAND SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2 AVX, AVX2, TXT, TSX, SGX VT-x, VT-d	Octava
Coffee Lake Refresh	2018	14nm	MMX, AES-NI, CLMUL, FMA3, RDRAND SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2 AVX, AVX2, TXT, TSX, SGX VT-x, VT-d	Novena
Comet Lake	2019	14nm	MMX, AES-NI, CLMUL, FMA3, RDRAND SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.1, SSE4.2 AVX, AVX2, TXT, TSX, SGX VT-x, VT-d	Decima
Rocket Lake	2021	14nm	AES-NI , CLMUL , RDRAND , SHA , TXT MMX , SSE , SSE2 , SSE3 , SSSE3 , SSE4 , SSE4.1 , SSE4.2 AVX , AVX2 , AVX-512 , FMA3 VT-x , VT-d,VAES	Undecima

Tabla 2.3: Generaciones de arquitecturas Intel.

[1] Se refiere a la tecnología utilizada por los fabricantes de semiconductores y se reporta en nanómetros

Cada arquitectura de las CPUs ejecuta de distinta forma la misma instrucción por lo que los ciclos de reloj que se utilizan en finalizar cada operación son distintos. Esto se debe a la latencia, el tiempo en el que la CPU tarda en realizar una instrucción. La latencia depende directamente del diseño de la instrucción dentro del procesador, por lo tanto según la arquitectura los tiempos obtenidos en ejecuciones del mismo programa serán distintos.

Otro aspecto que varía entre generaciones es el tamaño y la cantidad de registros con que cuenta el CPU, en general en generaciones anteriores a *Comet Lake* el tamaño máximo de registros es de 256 bits mientras que algunos *Comet Lake* y todos los *Rocket Lake* contienen registros de 512 bits. Para el desarrollo de este trabajo nos enfocaremos en los procesadores de 11° generación ya que son los que contienen el conjunto de instrucciones de AES vectorizado VAES en un tamaño de registros de 512 bits.

2.2.3. Skylake

Skylake lanzada en agosto del 2015, es la sexta arquitectura de procesadores Intel,es la sucesora de la arquitectura *Broadwell*, su principal diferencia radica en el consumo energético llegando en versiones ligeras a 4.5W, esta arquitectura tenia cómo objetivo ser eficiente en sistemas embebidos como tabletas y computadoras portátiles.

Esta arquitectura contaba con cinco distintas variantes en cada modelo, estas eran S, X, H, U y Y. Las variantes K y X permitían el cambio de los multiplicadores esto permite llegar a frecuencias mayores usando un mayor consumo energético, en cambio las versiones H, U e Y eran las variantes para sistemas embebidos consumiendo menor cantidad de energía y bloqueando distintas funciones a solo valores iniciales, las variantes S y X utilizaban un conector LGA 2066 debido a su proceso distinto de ensamblaje.

Esta arquitectura incluía el uso de instrucciones vectoriales de 128 bits, 256 bits y 512 bits, *AVX*, *AVX2* y *AVX512* respectivamente, no obstante, no todas las versiones incluían todos los conjuntos de instrucciones, a pesar de ofrecer las versiones de 512 bits estas solo estaban disponibles para la linea Xeon y las *AVX2* tenían un gran problema, el desempeño de estas en esta arquitectura se considera pobre llegando a ser de 4-5 veces más lenta que el conjunto de instrucciones de 128 bits.

Entre las características físicas que resaltaban en este procesador esta la inclusión de cinco ALU, cuatro núcleos físicos para las versiones de entrada y hasta 18 núcleos para las series X y la memoria cache L4 pasa de 64MB a 128MB. En la figura 2.4 se puede apreciar la arquitectura completa del Intel i7-6700K, uno de los modelos más representativos de la familia de procesadores *Skylake*.

2.2.4. Kaby Lake

Lanzada el 30 de agosto del 2016, *Kaby lake* es la séptima arquitectura de la linea Intel, esta se compone de un nuevo proceso de fabricación denominado 14nm+, proceso basado

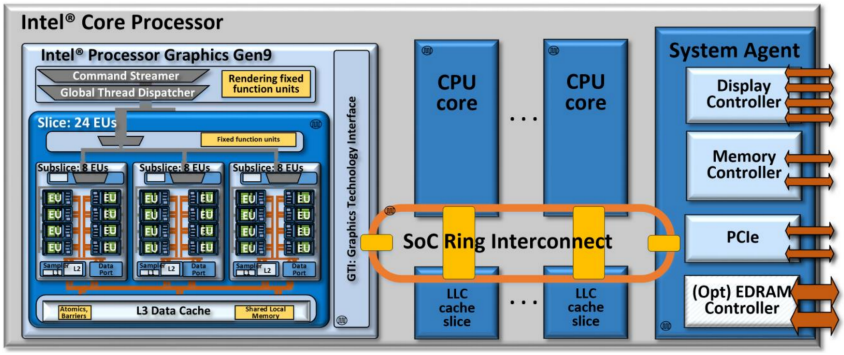

Figura 2.4: Diagrama de la arquitectura del Intel i7-6700K [1].

en 14nm al igual que su predecesora *Skylake*, con diferencias en las frecuencias en las que esta puede llegar, logrando frecuencias mayores en el mismo tamaño de fabricación.

En esta arquitectura se mantuvo el mismo conjunto de instrucciones vectoriales con los mismos costos de instrucciones por ciclo de reloj (IPC), no obstante debido a sus frecuencias mayores obtenía mejores rendimientos que sus predecesores.

Esta arquitectura presentaba más mejoras aparte de las frecuencias, sin embargo, dichas mejoras no eran en la arquitectura del procesador si no en los puertos de entrada y salida que podía soportar el chip, por la parte de las unidades gráficas de procesamiento implemento soporte nativo para salidas Displayport 1.4, codificación y decodificación por hardware *H264*, *HVEC* y *VP9* de 8 bits, soporte para *OpenGL* 4.6 y *OpenCL* 3.0, para los puertos permitió la conexión *PCI Express* 3.0 y el nuevo producto de Intel la *Intel Optane Memory*.

Esta arquitectura tuvo un segundo lanzamiento *Kaby Lake Refresh* esta segunda versión fue usada para dispositivos móviles como una nueva arquitectura de octava generación y su principal mejora fue el consumo energético,

2.2.5. Coffee Lake

Coffee Lake es la octava generación de procesadores de la linea Intel, fabricada usando en proceso 14nm++ de Intel. Lanzada el 5 de octubre de 2017 esta arquitectura esta hecha a 14nm igual que su predecesora *Kaby Lake*, no obstante cuenta con mejoras en los modelos de sus procesadores de la linea de sobre mesa, incluyo por primera vez seis núcleos físicos en los modelos i5 eh i7 y cuatro núcleos para el i3, añadió la tecnología de *Hyperthreading* a los modelos i5, i7 eh i9 con el limitador desbloqueado, esta tecnología permite que existan dos hilos por núcleo físico en el chip. Otra mejora presentada es la frecuencia máxima que alcanza estos procesadores siendo hasta 400 MHz mayor en los modelos i5 eh i7 que sus respectivas versiones pasadas,

A pesar de las mejoras en la cantidad de núcleos eh hilos eh incluso el aumento en las

frecuencias, el conjunto de instrucciones de esta arquitectura es el mismo que la arquitectura *Kaby Lake*, por lo cual el costo en IPC sigue siendo el mismo.

2.2.6. Coffee Lake Refresh

La novena generación denominada *Coffee Lake Refresh* es una versión con más núcleos físicos (ver Figrua 2.5) junto con mayores frecuencias que la mencionada *Coffee Lake*, no teniendo ninguna otra mejora notable ni otro cambio, *Coffee Lake Refresh* al igual que su antecesora esta fabricada a 14 nanómetros pero en un proceso denominado 14nm+++.

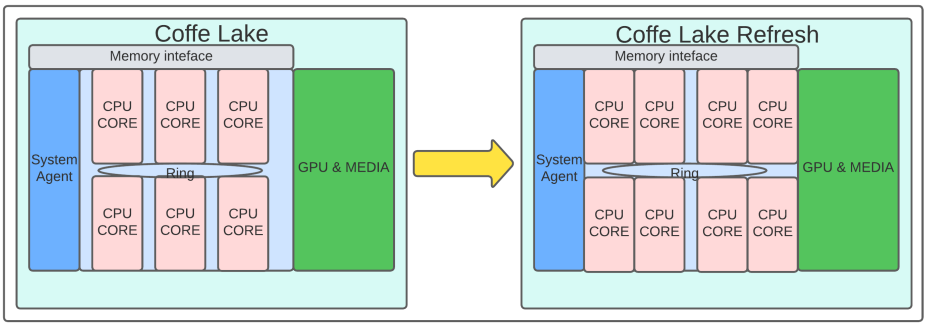

Figura 2.5: Comparación de las arquitecturas *Coffe Lake* y *Coffe Lake Refresh* de Intel.

2.2.7. Comet Lake

La décima generación de procesadores Intel, *Comet Lake*, con un proceso de fabricación 14nm++++, el cual esta basado en 14nm, con mejoras solo en la cantidad física de núcleos llegando a ser hasta 10 para los procesadores i9, incluyo por primera vez el *Hyperthreading* para la versión i3, convirtiéndolo en un procesador de 4 núcleos físicos y 8 hilos.

Las frecuencias de los procesadores Intel usando la arquitectura *Comet Lake* alcanzaban hasta 5.3 Ghz usando el *Turbo Boost* 3.0, esto es 300 Mhz más que su antecesor *Coffee Lake Refresh*

No obstante al seguir fabricado sobre un proceso de 14nm esta arquitectura presenta la misma desventaja que todas las mencionadas anteriormente, la misma cantidad de IPC por lo cual tiene el mismo conjunto de instrucciones del procesador.

La versión de dispositivos móviles de la décima generación de Intel se denomino *Ice Lake* el primer proceso de Intel a 10nm, esta arquitectura a pesar de ser para dispositivos móviles presentaba nuevos avances en el IPC junto con las instrucciones vectoriales del procesador, este nuevo conjunto de instrucciones eran los AVX-512 y VAES, instrucciones que permitían procesar más conjuntos de datos por bloque, no obstante esta arquitectura no salio a relucir mucho debido a su poco desempeño principalmente por la limitación

de la potencia del procesador, esto se debe a que la arquitectura estaba pensada para dispositivos móviles por lo cual su frecuencia turbo máxima de 4.1GHz solo la podía alcanzar usando un único núcleo, la cantidad de núcleos por procesador varia de los 2 a 4 dependiendo el modelo.

2.2.8. Rocket Lake

La onceava generación de Intel y la principal de este trabajo de investigación, diseñada en un proceso litográfico a 14nm al igual que su antecesora *Comet Lake*, no obstante con un enfoque distinto, esta arquitectura esta basada en la micro arquitectura *Cypress Cove*, una variación de la arquitectura usada en la versión de procesadores para dispositivos móviles de la décima generación de Intel *Ice Lake* la cual esta construida a 10nm.

Modelos	Modelos de procesadores de escritorio					
Modelos	Frecuencia base (Ghz)	Frecuencia turbo mono núcleo (Ghz)	Frecuencia turbo Todos los núcleos (Ghz)	Número de núcleos/hilos	Litografía (nm)	TDP (W)
i9-11900K	3.5	5.3	4.7	8 / 16	14	125
i9-11900	2.5	4.6	4.6	8 / 16	14	65
i9-11900T	1.5	3.7	3.7	8 / 16	14	35
i7-11700K	3.6	5	4.6	8 / 16	14	125
i7-11700	2.5	4.9	4.4	8 / 16	14	65
i7-11700T	1.4	4.6	3.6	8 / 16	14	35
i5-11600K	3.9	4.9	4.6	6 / 12	14	125
i5-11600	2.8	4.8	4.3	6 / 12	14	65
i5-11600T	1.7	4.1	3.5	6 / 12	14	35
i5-11400	2.6	4.4	4.2	6 / 12	14	65
i5-11400T	1.3	3.7	3.3	6 / 12	14	35

Tabla 2.4: Características de algunos modelos de procesadores de escritorios de Intel *Rocket Lake*

Esta arquitectura trajo cambios importantes para el conjunto de instrucciones de procesadores de escritorio, incluyo en estos procesadores las instrucciones *AVX-512* y *VAES*, Las instrucciones *AVX-512* permiten procesar bloques de datos de hasta un máximo de 512 bits por instrucción mientras que las instrucciones *VAES* procesan bloques de información del cifrador por bloques AES de forma óptima. *Rocket Lake* mejoro de forma drástica las latencias y ciclos de reloj usados por cada instrucción del procesador, esto le permitió tener un mejor desempeño que su antecesor *Comet Lake* de hasta un 19 % por instrucción, logrando unas frecuencias máximas de 5.3Ghz en su modelo tope de gama (ver Tabla 2.4).

En la tabla 2.5 se indica el costo de la instrucción de una ronda de AES para la arquitectura *Rocket Lake* y algunas otras de la familia Intel, en la tabla se puede apreciar que las instrucciones de 128 y 256 bits, tienen un menor costo comparado con las de 512 bits, no obstante las instrucciones de 512 bits permiten procesar del doble al cuádruple

	Instrucción		
Arquitectura	_mm_aesenc_epi128 Throughput (CPI)	_mm256_aesenc_epi128 Throughput (CPI)	_mm512_aesenc_epi128 Throughput (CPI)
Skylake	0.5	0.5	-
Icelake	0.5	0.5	1
Rocket Lake	0.5	0.5	1
Tiger Lake	0.5	0.5	1

Tabla 2.5: Costo de instrucciones de 128, 256 y 512 bits para una ronda de AES en algunas arquitecturas, el costo esta en ciclos por instrucción (CPI).

de información en un mismo bloque dependiendo de la instrucción por un ligero aumento en el costo de la instrucción, esto las vuelve una opción viable para algunos problemas.

Rocket Lake no solo introdujo nuevos conjuntos de instrucciones, también elimino conjuntos de instrucciones para darle espacio a las nuevas, el conjunto más notable que se elimino fue el de SGX, un conjunto de instrucciones enfocados en los procesos criptográficos del sistema tales cómo cifrar o descifrar datos, este conjunto se reemplazo por el *VAES*, estos cambios solo aplican para las versiones móviles y de escritorio, los procesadores enfocados en estaciones de trabajo mantienen todavía el SGX. En la figura 2.6 se puede apreciar un diagrama general de los puertos con los que cuenta los procesadores de escritorio de la onceava generación de Intel.

Tiger Lake

Los procesadores para dispositivos móviles de la onceava generación de Intel están construidos en un proceso litográfico distinto al de las versiones de escritorio, este proceso es a 10nm, por lo tanto se denominaron como una nueva arquitectura *Tiger Lake*, siendo esta la arquitectura para procesadores móviles de la onceava generación.

Esta arquitectura comparte las mismas características que sus versiones de sobre mesa, no obstante las versiones de dispositivos móviles tienen frecuencias máximas menores que sus versiones de escritorio. En la tabla 2.6 se muestran las características de algunos procesadores de la onceava generación de Intel para dispositivos móviles, las frecuencias base de estos procesadores son mucho menores que sus versiones de escritorio, esto se debe a la necesidad de economizar el consumo energético para los dispositivos móviles, los cuales dependen de una batería.

2.3. Extensiones Vectoriales Avanzadas

Las Extensiones Vectoriales Avanzadas comúnmente denominadas *AVX* por sus siglas en ingles, son un conjunto de instrucciones para procesadores propuestas por Intel en marzo del 2008.

Las *AVX* utilizan se componen de distintos tamaños de registros, estos son 128, 256 y

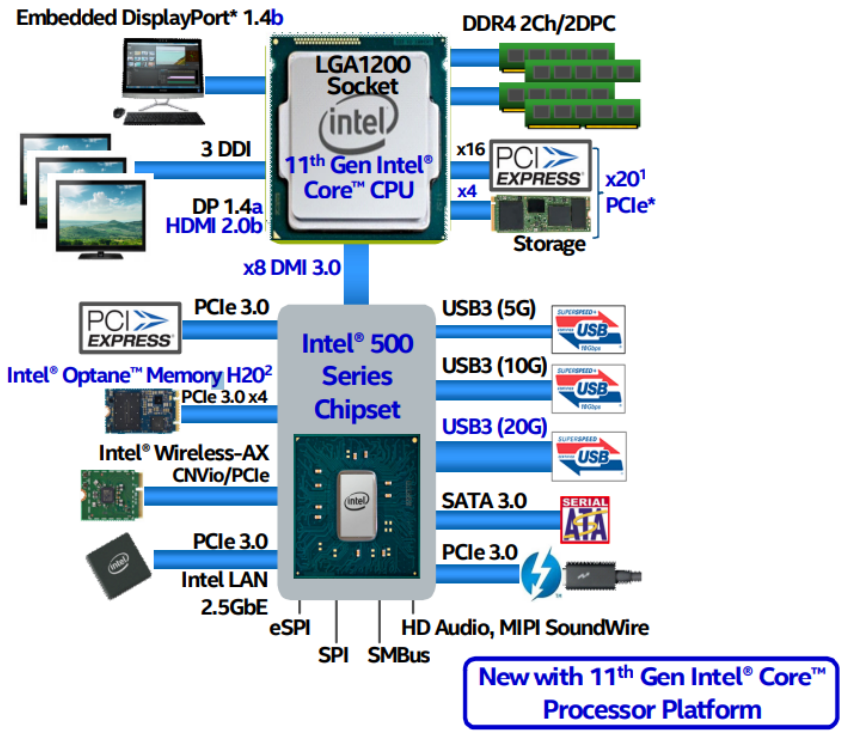

Figura 2.6: Diagrama general de los procesadores de la onceava generación de Intel *Rocket Lake*[2].

512 bits. Los XMM o registros de 128 bits permiten realizar operaciones de un tamaño máximo de 128 bits los procesadores cuentan con hasta 32 registros XMM dependiendo el modelo. Los YMM o registros de 256 bits son registros que permiten realizar operaciones de hasta 256 bits, en ellos se pueden procesar al mismo tiempo hasta dos bloques de 128 bits para obtener mayores velocidades, cada registro YMM del sistema puede realizar múltiples operaciones al mismo tiempo los procesadores cuentan con hasta 32 registros YMM dependiendo del modelo. Los ZMM o registros de 512 bits permiten realizar operaciones de un tamaño máximo de 512 bits estas pueden dividirse en múltiples tamaños, los registros ZMM se añadieron por primera vez en la arquitectura *Skylake* de Intel en los procesadores destinados a servidores, los procesadores cuentan con hasta 32 registros ZMM dependiendo el modelo.

Todas las instrucciones AVX pueden realizar operaciones simultaneas, dependiendo el tamaño del registro y la operación a realizar, estas operaciones se dividen en tamaños, 8, 16, 32, 64, 128, 256 y 512 bits. Por ejemplo se puede realizar cuatro sumas simultaneas de 32 bits en un registro XMM (ver Figura 2.7).

Las *AVX* son consideradas instrucciones vectoriales debido a la capacidad para poder realizar operaciones simultaneas, no obstante, no son el único conjunto de instruccio-

Modelos de procesadores para dispositivos moviles						
Modelos	Frecuencia base (Ghz)	Frecuencia turbo mono núcleo (Ghz)	Frecuencia turbo todos los núcleos (Ghz)	Número de núcleos/hilos	Litografía (nm)	TWD (W)
i9-11980HK	3.3	5	4.5	8/16	10	45-65
i9-11950H	2.6	5	4.5	8/16	10	35-45
i9-11900H	2.5	4.9	4.4	8/16	10	35-45
i7-11850H	2.5	4.8	4.3	8/16	10	35-45
i7-11800H	2.3	4.6	4.2	8/16	10	35-45
i7-11600H	2.9	4.6	-	6/12	10	35-45
i5-11500H	2.9	4.6	4.2	6/12	10	35-45
i5-11400H	2.7	4.5	4.1	6/12	10	35-45
i5-11260H	2.6	4.4	4	6/12	10	35-45

Tabla 2.6: Características de algunos modelos de procesadores de escritorios de Intel *Tiger Lake*.

nes capaz de hacerlo, las *Streaming SIMD Extension (SSE)* son otro de los conjuntos de instrucciones de Intel capaces de realizar operaciones simultaneas usando registros, sin embargo, estas instrucciones están limitados a 128 bits y utilizan únicamente dos operandos de la forma $a \longleftarrow a + b$, lo cual genera múltiples inconvenientes en algunas implementaciones. Las *AVX* permiten usar tres operandos en las instrucciones de la forma $c \longleftarrow a + b$, con ello se solucionan múltiples problemas de las *SSE*.

Existen tres conjuntos de instrucciones *AVX*, los cuales son *AVX*, *AVX2* y *AVX-512*, cada uno de ellos contiene distintas instrucciones vectoriales, estos se encuentran en los procesadores dependiendo el modelo.

- El conjunto *AVX* fue el primero en introducir las instrucciones de 256 bits, no obstante no todas las instrucciones se podían realizar en 256 bits.

- El conjunto *AVX2* amplio la cantidad de instrucciones posible a realizarse en 256 bits. El conjunto *AVX-512* introdujo las primeras instrucciones capaces de realizarse en 512 bits.

Es importante tener en cuenta que no todas las instrucciones que existen en en un tamaño de bits existirán en otro, por lo tanto es posible que existan instrucciones de 128 bits que no existan en 256 y 512 bits y viceversa.

2.4. AES-NI y VAES

Las *AES-NI* o Instrucciones Nativas de AES, son un conjunto de seis instrucciones que permiten la implementación del cifrador por bloques *AES* de forma eficiente, estas instrucciones son:

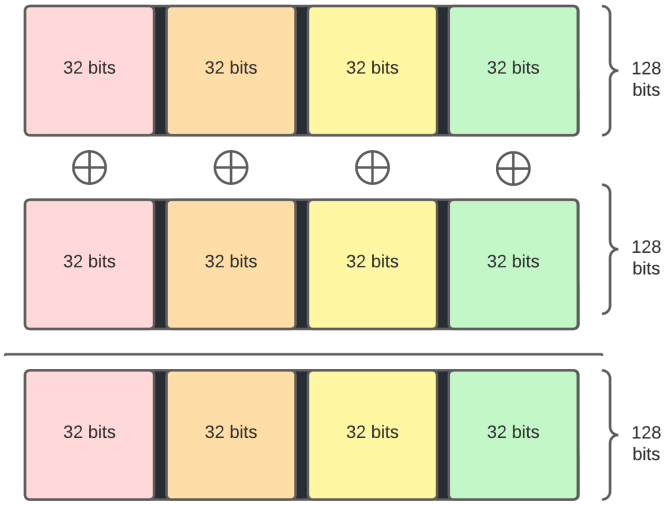

Figura 2.7: Ejemplo de suma de 128 bits usando registros de 32 bits .

- **AESENC** esta instrucción realiza una ronda completa del cifrado de *AES*, sustitución de bytes, corrimiento de filas, mezclado de columnas y la suma de la llave de ronda.
- **AESENCLAST** esta instrucción realiza la ultima ronda del cifrado de *AES*, utiliza la sustitución de bytes, el corrimiento de filas y la suma de la llave de ronda.
- **AESDEC** esta instrucción realiza una ronda completa del descifrado de *AES*, sustitución invertida de bytes, corrimiento invertido de filas, mezclado invertido de columnas y la suma de la llave de ronda.
- **AESDECLAST** esta instrucción realiza la ultima ronda del descifrado de *AES*, utiliza la sustitución investida de bytes, el corrimiento invertido de filas y la suma de la llave de ronda.
- **AESKEYGENASSIST** esta instrucción es usada para la generación de llaves.
- **AESIMC** esta instrucción se utiliza para convertir las llaves del cifrado de *AES* a su forma inversa equivalente para el descifrado.

Estas instrucciones existen para múltiples tamaños de bloques, estos son 128, 256 y 512 bits, dependiendo del tamaño de los registros del procesador. Están acompañadas de un componente físico en el procesador, para así tener un desempeño eficiente, los procesadores que cuentan con este componente tienen expresado en las características la etiqueta de *VAES*.

El componente físico *VAES* es el que permite un alto desempeño del cifrador por bloques *AES* en los procesadores, al ser el estándar adoptado en el 2001, la mayoría de procesadores en el mercado cuentan con un componente *VAES*, en el caso de los procesadores Intel todo procesador móvil, de sobre mesa o para estación de trabajo a partir de la arquitectura *Skylake* cuenta con un componente *VAES*. Las instrucciones de *AES-NI* se encuentran como un subconjunto de las *AVX*, teniendo identificadores específicos dado el modelo de los procesadores.

2.5. Pipeline

El *pipeline* permite la ejecución de múltiples instrucciones divididas en etapas, este concepto se puede ver reflejado como una linea de ensamblaje donde cada etapa de la linea de ensamblaje debe pasar por la etapa anterior para avanzar, es decir es necesario el resultado de la etapa anterior para la siguiente.

En la figura 2.8 se puede apreciar un ejemplo de un *pipeline* de cuatro etapas, en esta figura se muestra como cuatro instrucciones se dividen en distintos segmentos para abarcar cada etapa, cuando el *pipeline* esta lleno después de cada iteración se producirá un resultado.

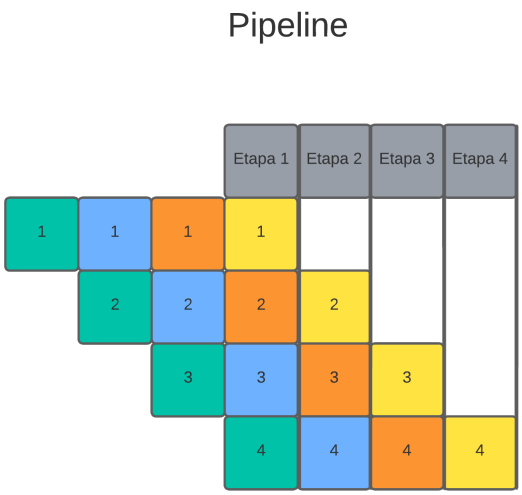

Figura 2.8: Ejemplo de un *pipeline* de cuatro etapas.

La cantidad de etapas del pipeline de un procesador depende de la arquitectura de este y los registros del mismo, es decir no todos los procesadores tendrán el mismo tamaño de pipeline y tampoco tendrán el mismo tamaño de registros, en las arquitecturas de Intel los registros son compartidos, es decir los registros ZMM son los mismos que los XMM con la diferencia que los XMM limitan su tamaño a 128 bits, por lo tanto, el pipeline de la

onceava generación de Intel se puede llenar de distintas formas pero para este trabajo nos enfocaremos en dos principalmente, usando únicamente registros de 128 bits (ver figura 2.9) o usando únicamente registros de 512 bits (ver figura 2.10).

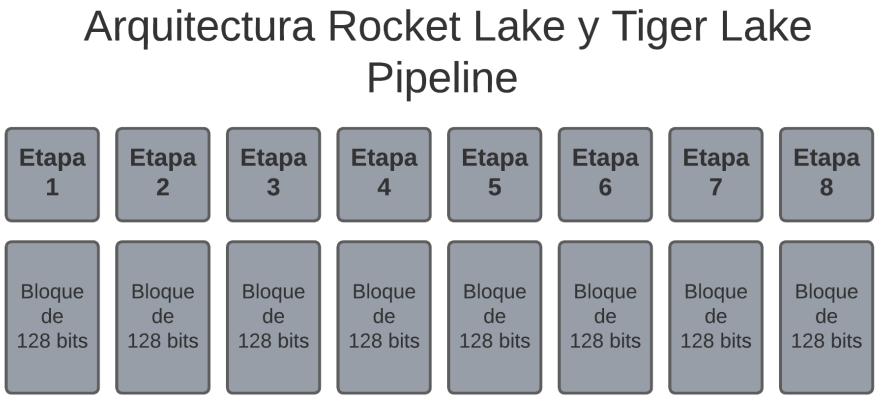

Figura 2.9: Ejemplo del *pipeline* de la onceava generación de Intel usando bloques de 128 bits.

Para las arquitecturas de la onceava generación de Intel, *Rocket Lake* y *Tiger Lake*, el *pipeline* se divide en ocho etapas, por lo cual es necesario tener por lo menos 8 instrucciones a procesar para obtener una desempeño considerable.

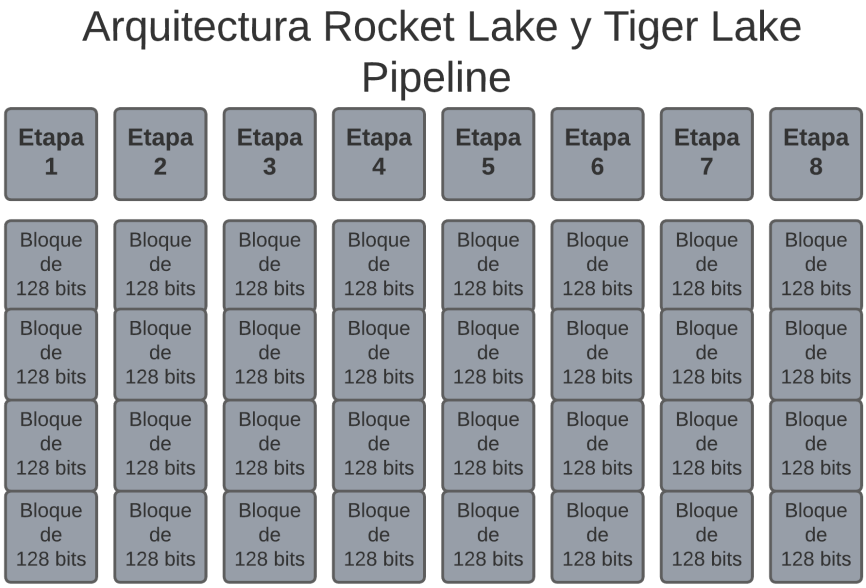

Figura 2.10: Ejemplo del *pipeline* de la onceava generación de Intel usando bloques de 512 bits.

Capítulo 3

OCB

En este capítulo se describe detalladamente el modo de operación OCB junto con sus distintas versiones, se comparan estas versiones en busca de los defectos y fortalezas de cada una, se explica la construcción genérica del OCB además de las posibles funciones con las cuales se puede llegar a calcular el Δ.

3.1. Definición

Offset Codebook (OCB) es un esquema de cifrado autenticado, proveé simultáneamente confidencialidad, integridad y autenticación de los datos. OCB fue propuesto en [18] por Phillip Rogaway.

Se basa en el uso del cifrador por bloques entonado **XEX**, que se define como $E_K(M_i \oplus \Delta_i) \oplus \Delta_i$ para $i = 0....m-1$. Es un requisito indispensable que todas los valores Δ_i sean diferentes entre sí, por lo que deben ser generadas mediante el uso de una función de periodo máximo.

Una de las principales características que tiene OCB es la paralelización, esta depende directamente del cálculo de la función mascara que se utilice para generar las Δ_i del esquema, una función totalmente paralelizable permite una versión totalmente paralelizable del OCB.

En [3] se estudian distintas funciones para generar la mascara Δ de OCB, además de proponer una nueva función mascara óptima. En la tabla 3.1 se en listan las funciones estudiadas en [3].

3.2. Estructura genérica

La estructura de los esquemas de OCB es muy similar entre sus diferentes versiones, la diferencia principal es la forma en que se calcula el valor de enmascaramiento usualmente denotado como Δ. En [3] se presenta la generalización de OCB *Generic Offset Codebook*

Familia Hash	Secuenciales (operaciones)	Directo (operaciones)	Tamaño de la llave (bits)	Tamaño del dominio (base 2)	MDP
xtimes	$1 \ll, 1\oplus_?$	$O(lg(i))\odot$	128	128	2^{-128}
gray	1 ntz(i), 1 $\oplus$	$1\oplus, 1 \ll, 1\odot$	128	128	2^{-128}
mtrx	$2 \ll, 1\oplus$	$O(lg(i))\odot$	128	128	2^{-128}
mlin	wt$(i)\oplus$	wt$(i)\oplus$	$128r$	r	2^{-128}
clh	wt$(i)\ll$, wt$(i)\oplus$	wt$(i)\ll$, wt$(i)\oplus$	128	127	2^{-128}
sts	$1\|\|, 1\oplus, 3 \ll$	$1\|\|, 1\oplus, 3 \ll$	128	6	2^{-128}
Icube	$1\oplus, 2\odot$	$1\oplus, 2\odot$	128	128	2^{-128}
Iinv	$1\oplus, 1\odot$	$1\oplus, 1\odot$	128	128	2^{-128}
aes4	4 AESRD	4 AESRD	512	128	2^{-113}
aes2	2 AESRD	2 AESRD	256	32	2^{-113}
aes1	1 AESRD	1 AESRD	128	8	2^{-96}

Tabla 3.1: Distintas funciones posibles para el cálculo de Δ, $\oplus$, $\odot$, $\ll$, $\oplus_?$ denotan suma, multiplicación, corrimientos, concatenación y la suma condicional sobre el campo; AESRD denota la función de ronda de AES, wt y ntz definen el peso de hamming y el número de ceros finales [3].

(GOCB) la cual es una generalización de OCB y solo varía la función para generar los Δ. En las figuras 3.1 y 3.2 se ilustran las versiones para bloques completos he incompletos del *GOCB* respectivamente, se diferencían en la forma en que procesa el último bloque de mensaje y el bloque del checksum. Todo mensaje completo del GOCB utiliza el Δ_1 y Δ_3 para los bloques y la sumatoria respectivamente, si el mensaje es incompleto es necesario utilizar Δ_1, Δ_2 y Δ_4 para los bloques completos, para el bloque incompleto y para la sumatoria respectivamente.

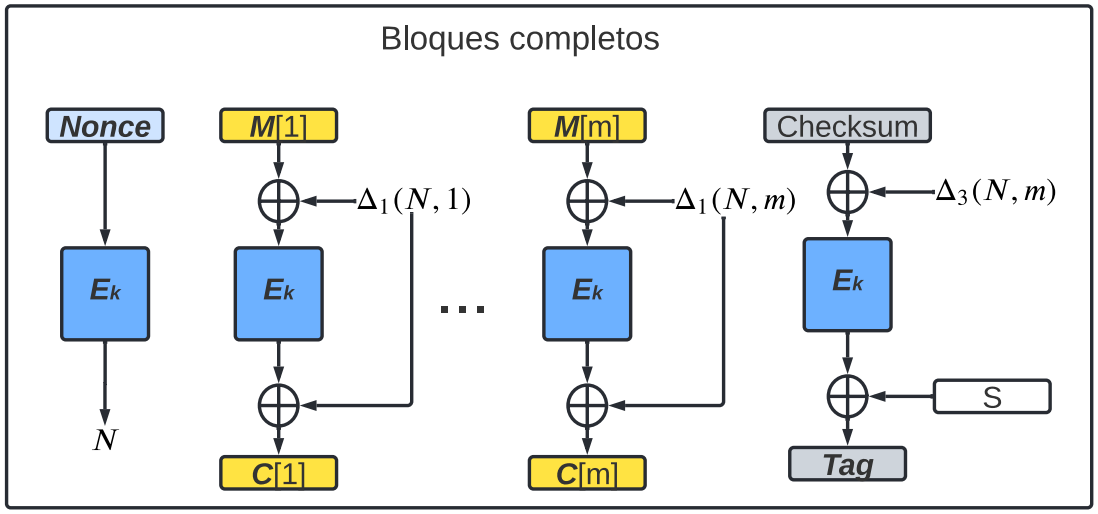

Figura 3.1: Esquema del cifrado del mensaje para bloques completos del GOCB [3].

Los Δ_1, Δ_2, Δ_3, Δ_4 dependen directamente de la función usada para calcular Δ, cada

uno de estos tendrá un proceso distinto dependiendo de la función elegida, en la tabla 3.1 se muestran las distintas funciones conocidas para calcular Δ, esta tabla contiene el tamaño del dominio y la máxima probabilidad diferencial.

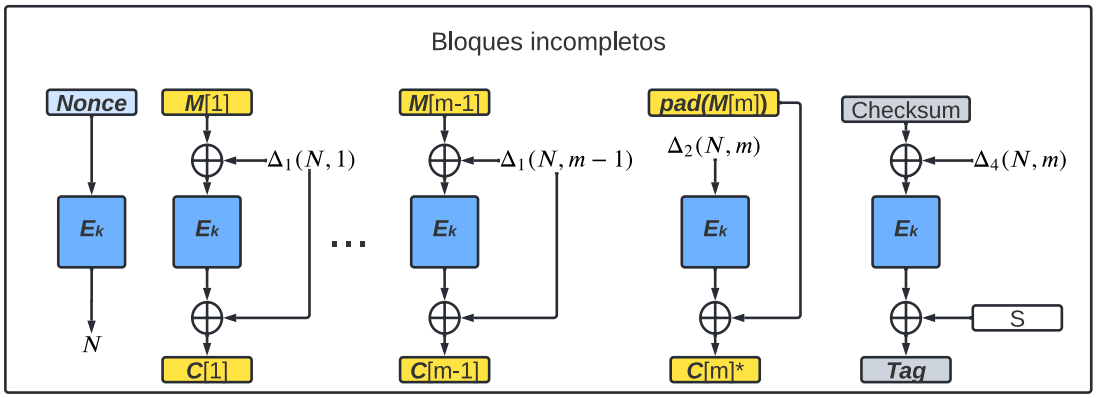

Figura 3.2: Esquema del cifrado del mensaje para bloques incompletos del GOCB [3].

Δ_5 y Δ_6 son exclusivos para el cálculo de los datos asociados, para bloques completos e incompletos respectivamente (Ver figura 3.3).

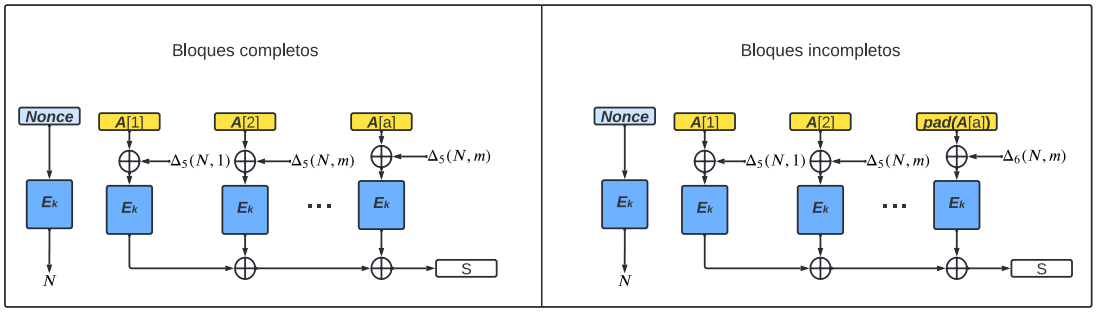

Figura 3.3: Procesamiento de los datos asociados del GOCB [3] .

3.3. OCB 1

Offsett Codebook 1 o OCB1 fue la primer versión propuesta del OCB, esta versión fue diseñada en respuesta a los modos de operación propuestos por el NIST en el 2001. Tenía como objetivo dar una alternativa que ofreciera autenticación, integridad y confidencialidad con un costo mínimo adicional comparado con los modos de operación estandarizados que solo proveen privacidad.

Esta versión de OCB1 fue pensada para ser totalmente paralelizable cuenta con las siguientes características adicionales.

- **Tamaño de mensajes arbitrario + mínimo tamaño del mensaje cifrado**: Para cualquier mensaje $M \in \{0,1\}^*$ puede ser cifrado; en particular $|M|$ no necesita ser un múltiplo del tamaño n, esto resulta en mensajes cifrados lo más costos posibles.
- **Mínimos requerimientos del nonce**: Al igual que los modos de operación estandarizados por el NIST, OCB1 requiere un vector de inicialización. La entidad encargada de realzar el cifrado elige un nuevo *nonce* para cada mensaje, con la única restricción de que un *nonce* no puede ser usado dos veces.
- **Número casi óptimo de llamadas al cifrador por bloques**: El OCB1 utiliza $\lceil |M|/n \rceil + 2$ llamadas al cifrador por bloques para cifrar y autenticar un mensaje no vació M.
- **Uso de una única llave**: El OCB1 usa la misma llave para todos los cifradores por bloques, por lo cual solo es necesario calcular una única vez todo el conjunto de llaves.
- **Cálculo eficiente de la máscara Δ**: Al igual que otros modos de operación OCB1 requiere el uso de una secuencia de Δ, estos son calculados particularmente en una forma muy eficiente, cada Δ requiere muy pocos ciclos de reloj y sin aritmética de precisión extendida.

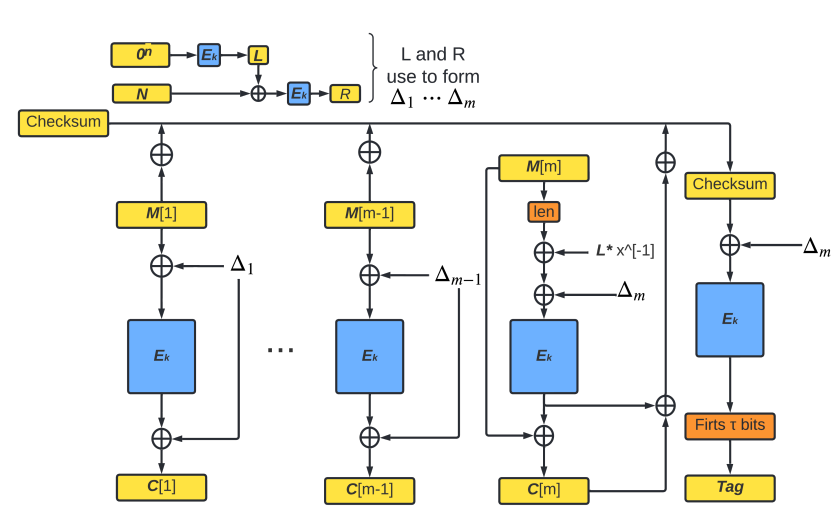

Figura 3.4: Esquema del OCB1. .

En la figura 3.4 se aprecia el esquema del OCB, esta basado en una construcción **XEX** de los cifradores por bloques entonados.

El esquema de OCB1 a pesar de presentar muchas ventajas tiene ciertos desperfectos, los principales son:

- **El ultimo bloque del Checksum**: El primer problema es la realización del *Checksum* del texto claro, donde es necesario esperar a la obtención del cifrado del ultimo bloque para poder realizar el *Checksum*, esto afecta directamente al rendimiento del esquema en el paralelismo.
- **Sin soporte a datos asociados**: El segundo problema es que este esquema no da soporte para mensajes con datos asociados, por lo cual esa parte es necesaria realizarla con otra función de cifrado.
- **Cifrado resultante más grande**: El tercer y ultimo problema es la obtención de un cifrado de mayor tamaño que el mensaje original, este esquema añade los bits necesarios para que el resultado del cifrado siempre sea un múltiplo del tamaño n en el esquema.

3.4. OCB 2

Offsett Codebook 2 o OCB2 fue propuesto como una mejora al OCB1 en [13], en este modo de operación se contempla el manejo de datos asociados utilizando la construcción **XE** propuesta en el mismo articulo. En el proceso de cifrado del mensaje OCB2 presentaba una construcción bastante similar al OCB1 cambiando exclusivamente el cálculo del último bloque. Sin embargo, por este cambio al último bloque el OCB2 fue atacado logrando recuperar el texto en claro. El principal uso de este ataque se basa en un mensaje lo suficiente largo de 0 y el atacante debe de tener acceso a los oráculos de cifrado y descifrado. El proceso completo se encuentra en [19].

3.5. OCB 3

Offsett Codebook 3 o OCB3 es la versión más reciente propuesta por Philid Rogaway, en esta versión se resuelven los problemas del tamaño del cifrado, junto con el proceso del *Checksum*. Este proceso esta detallado en el artículo [20]. Este modo de operación tiene como objetivo ofrecer un rendimiento similar al OCB1 sin comprometer la confidencialidad del mensaje a diferencia del OCB2. Al igual que las versiones anteriores el OCB3 esta basado usando en las dos construcciones de cifradores por bloques entonados, **XE** y **XEX**, para los datos asociados y el mensaje respectivamente.

La construcción **XE** es usada exclusivamente para los datos asociados ya que está basada en PMAC. Esta nueva construcción usando XE tiene como resultado PMAC1 la cual es por mucho más simple y eficiente que PMAC. PMAC1 no necesita el *código Gray* solo es requerido el cálculo *xtimes* estos cálculos son eficientes en el computo.

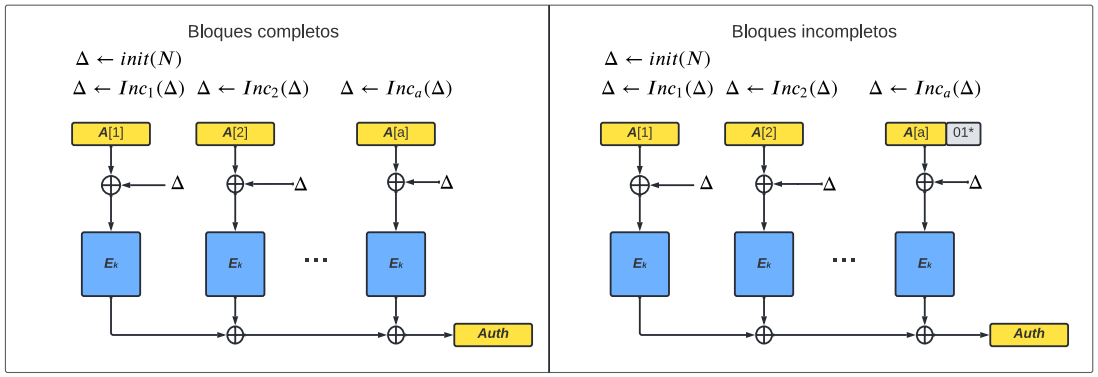

Figura 3.5: Esquema de los datos asociados del OCB3 (PMAC1).

La figura 3.5 indica el esquema de los datos asociados para el OCB3, al ser una construcción **XE** es posible notar que el proceso por bloque consta de $E_k(|A_n| \oplus \Delta_n)$, es decir un XOR con el bloque correspondiente de los datos asociados seguido del proceso de cifrado. El *Auth* se compone de las sumas de todos los resultados de los bloques cifrados. El proceso de cifrado del OCB3 es similar al proceso del OCB1, exceptuando el último bloque, a diferencia del OCB1, el proceso del último bloque depende si el mensaje es completo o no, en caso de que el mensaje sea completo, el proceso del ultimo bloque es idéntico al de los bloques anteriores, en caso contrario, es necesario realizar un proceso distinto, este se muestra en la figura 3.6. Este proceso al igual que sus versiones anteriores sigue basado en la construcción **XEX**.

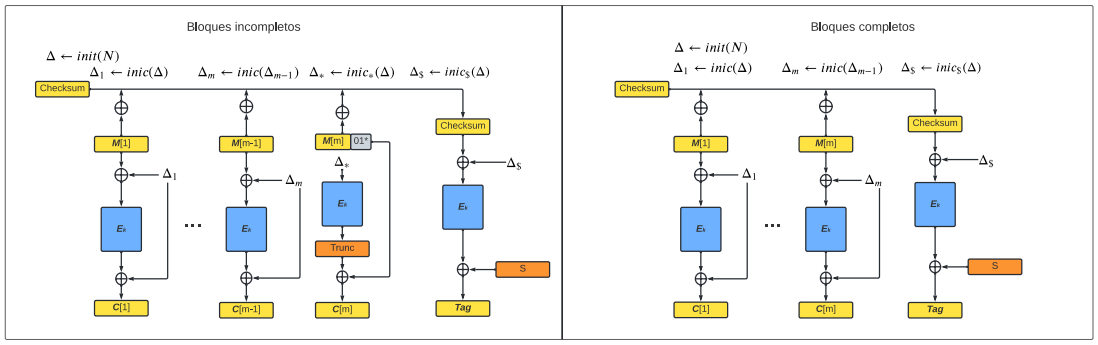

Figura 3.6: Esquema del cifrado del mensaje del OCB3.

El COB3 presenta soluciones a las desventajas expuestas del OCB1 estas son:

- *El último bloque del Checksum*: A diferencia del OCB1, el OCB3 no necesita esperar la realización del último bloque para realizar el *Checksum* por lo cual su rendimiento no se mira afectado.

- *Soporte de datos asociados*: El OCB3 incluye soporte para los datos asociados siguiendo el esquema propuesto en OCB2 usando la construcción **XE** (ver figura 3.5)

- *Cifrado resultante del mismo tamaño*: El OCB3 a diferencia del OCB1 y OCB2 tiene como resultado un cifrado del mismo tamaño al mensaje ingresado esto permite no depender de un segundo espacio de almacenamiento.

3.6. OCB de Acceso Aleatorio

OCB de Acceso Aleatorio conocido en ingles como *OCB Random Access* es la versión más reciente del modo de operación OCB, esta se basa en cifrar o descifrar cualquier bloque de forma aleatoria. Esta versión fue propuesta en [3], en este articulo de describe las desventajas y ventajas de las actuales δ para el OCB, la mayoría de estas presentan ventajas únicamente en el cálculo secuencial en el cual son óptimos. La principal desventaja de estas Δ se presenta cuando se requiere un bloque en concreto, para estos casos es necesario calcular todas las Δ anteriores a la de ese bloque independientemente si se requieren los bloques anteriores o no, lo cual representa un gasto excesivo en recursos y tiempo cuando se requiere un bloque aleatorio.

En [3] se propone una solución a este problema, una Δ que no depende de las anteriores para el bloque a calcular y que mantenga la seguridad del sistema. La solución propuesta es utilizar la función de ronda del cifrador por bloques *AES* para el cálculo de Δ, esta solución tiene dos versiones: usando una ronda de *AES* y usando dos rondas de *AES*, cada una con sus propias ventajas y desventajas.

Utilizar una ronda de *AES* tiene su principal mejora en el rendimiento, esto se debe a que siempre la cantidad de instrucciones serán menores a la de dos rondas de *AES*, no obstante, esta versión requiere para aprovechar su máximo rendimiento precomputar todos los **Nonce** necesarios del mensaje a usar, debido a que utilizando una única ronda de AES es necesario actualizar el **Nonce** cada 2^6 bloques, por lo cual el consumo en memoria es considerablemente más alto. Cuando se utilizan dos rondas el rendimiento es inferior pero al mismo tiempo el consumo de memoria es sumamente menor, esto se debe que el **Nonce** es necesario actualizarlo cada 2^{30} bloques.

El rendimiento general del OCB de Acceso Aleatorio depende directamente de la implementación de *AES*, una implementación más óptima permitirá mejores resultados. A diferencia de las versiones anteriores este OCB no tiene un esquema propio, utiliza los anteriores esquemas mencionados en las figuras 3.1 y 3.2 para el proceso del mensaje, y los esquemas de las figuras 3.1 y 3.2 para los datos asociados.

3.7. Comparación

Cada esquema de OCB tiene sus propias ventajas y desventajas, definir que esquema es mejor que otro es complicado, debido a que estos dependen de las circunstancias en la que sean comparados, buscando una comparación lo más justa posible, es necesario identificar el objetivo de este esquema el cual es ser un esquema eficiente para el cifrado de datos permitiendo la paralelización al mismo tiempo proporcionando integridad, confidencialidad y autenticación de los datos.

- **OCB1** Es el esquema más antiguo de todos y por lo tanto es el que presenta las principales desventajas, no permite un rendimiento óptimo por la dependencia al último bloque calculado, el cifrado resultante siempre sera más grande por lo que es algo a tomar en cuenta en la memoria, al no contar con soporte para datos asociados es necesario usar otro algoritmo para dichos datos, en general a pesar de ser el primero esta versión presenta múltiples desperfectos.
- **OCB2** Busco corregir los errores del OCB1, no obstante, a pesar de incluir el soporte de los datos asociados, este esquema no corrigió el problema del último bloque, además de ser el único de los mencionados en ser vulnerado por lo cual es considerado inseguro.
- **OCB3** Esta versión corrige todos los errores del OCB1 y OCB2 no obstante presenta nuevos problemas si se habla de paralelismo a gran escala, principalmente por el cálculo de la Δ la cual para este esquema es secuencial, es necesario calcular siempre la Δ anterior para obtener la actual, en caso de necesitar un bloque en especifico el rendimiento se mira altamente afectado por esto.
- **OCB de Acceso Aleatorio** Esta versión de OCB sigue el esquema del OCB3 no obstante el cálculo de Δ es el cambio, en esta versión todos los bloques tienen el mismo costo de obtención por lo cual no hay penalización de necesitar un bloque en especifico. En esta versión se utiliza el cifrado por bloques AES para el cálculo de Δ, por lo cual el rendimiento depende directamente de la forma en cómo se implemente AES, una mala implementación presentara resultados deplorables mientras que una implementación eficiente permite rendimientos óptimos en este esquema.

Version de OCB	Atacado	Δ	Datos Asociados	Cifrado de mensaje	Dependencia de los bloques	Calculo de Δ
OCB1	No	xtimes	No	Si	Al último bloque	Secuencial
OCB2	Si	xtimes	Si	Si	Al último bloque	Secuencial
OCB3	No	xtimes	Si	Si	Ninguna	Secuencial
OCBRA	No	1 o 2 rondas de AES	Si	Si	Ninguna	Dependiente de la posición

Tabla 3.2: Comparación de las distintas versiones del OCB.

En la tabla 3.2 se presentan las diferencias expuestas del OCB. Con los puntos anteriormente presentados la mejor opción a utilizar esta entre el OCB3 y el OCB de Acceso

Aleatorio, esto depende de las circunstancias y de las características del equipo donde se implemente.

Capítulo 4

Estrategias de programación

En este capítulo se expone las distintas versiones implementadas del OCB, se explican las diferentes mejoras realizadas a cada una de estas, el cómo se implementaron y los lenguajes en los que están realizados, la cantidad de bloques necesarios de precomputar y la memoria requerida en algunas versiones.

4.1. GPUs

Las implementación de GPUs se dividen en dos partes fundamentales: la implementación de *AES* y la implementación de OCB. La implementación de *AES* es considerada la parte más pequeña pero la más importante, de esta implementación depende gran parte del desempeño debido al uso constante de este cifrador por bloques en todo el algoritmo. Si *AES* esta implementado de forma ineficiente el rendimiento del OCB será ineficiente. La implementación de OCB no requiere una alta optimización, en cambio esta implementación busca una alta sincronización para poder realizar el *Checksum* al momento de realizar el cifrado aprovechando en todo momento los múltiples hilos del programa.

4.1.1. Implementación de AES

Las implementaciones de *AES* para GPUs sean estudiados en múltiples artículos, tales como: *Bitsliced High-Performance AES-ECB on GPUs*[21], *GPU Accelerated AES Algorithm*[22] y *Implementation and Analysis of AES Encryption on GPU*[6], en cada uno de ellos explican los resultados obtenidos, no obstante la mayoría de las implementaciones de los artículos mencionados anteriormente se basan en *Acceleration of AES encryption on CUDA GPU* para realizar sus propias versiones y en este trabajo de tesis no es la excepción.

Las GPUs tiene como una de sus principales ventajas la cantidad de hilos con las cuales estas pueden trabajar al mismo tiempo, debido a esto y los resultados obtenidos en el artículo *Acceleration of AES encryption on* se opto por una construcción que utilice un

hilo por cada bloque de *AES* a procesa (ver figura 4.1), es decir cada hilo procesa a la vez 128 bits o 16 bytes. Al ser procesados por un único hilo, no es necesario sincronizar cada uno de estos, esperando los resultados de uno anterior, por lo cual el desempeño obtenido es mejor. El *AES* implementado es el AES-128.

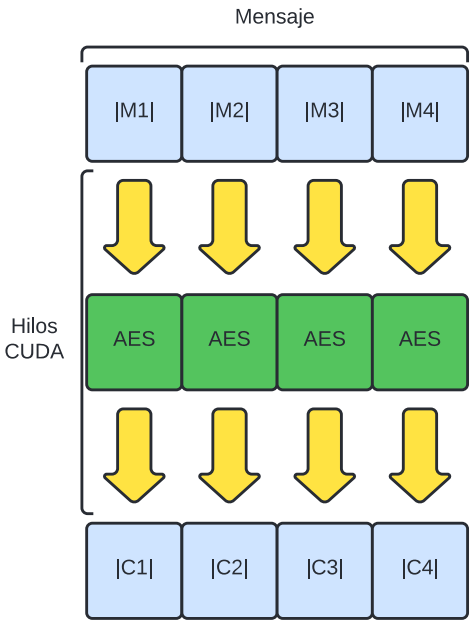

Figura 4.1: Esquema del proceso de AES en CUDA .

Cada bloque de *AES* es tratado como la unión de cuatro palabras de 32 bits, esto permite la optimización de las *T-Boxes* o tablas T, esta implementación de AES permite realizar cálculos más eficientes utilizando un conjunto de tablas de consultas denominadas *T-Boxes* y un XOR, de esta forma no es necesario realizar el calculo de las transformaciones de *SubBytes* y *MixColumns*, por lo cual el desempeño se mira altamente beneficiado.

Para que esta optimización pueda realizarse es necesario tener un total de cuatro *T-Boxes* denominadas: T1, T2, T3 y T4, cada tabla cuenta con 256 posiciones de 32 bits cada una. Para evitar un uso excesivo de memoria estas tablas están ubicadas originalmente en la memoria constante para un acceso rápido, no obstante la memoria constante no es la más rápida de todas, estas tablas son copiadas a memoria compartida un apartado de memoria más rápido que la memoria constante, de esta forma todos los hilos de un mismo bloque acceden a la misma memoria compartida por lo cual la cantidad de memoria necesaria es reducida en gran escala.

4.1.2. Implementaciones de OCB

Se implementaron dos versiones de OCB para GPUs, OCB3-CUDA y OCBRA-2-CUDA, estas versiones comparten las optimizaciones realizadas: inicialización eficiente, *Checksum* paralelo y división del mensaje.

La inicialización eficiente se consigue en cinco tiempos, un tiempo se denomina a la ejecución de un hilo sobre una instrucción, CUDA permite la ejecución de multiples hilos por bloques, cada hilo tiene la tarea de asignar un valor a cada una de las *T-Boxes* mencionadas anteriormente, además de inicializar la *S-Box* necesaria para el *AES*, para poder realizar esto de forma eficiente es necesario contar con un mínimo de 256 hilos por bloque debido a que todas las tablas cuentan con 256 posiciones.

El *Checksum* es el resultado de la sumatoria de todos los bloques del mensaje, realizar dicha suma en distintos llamados en las GPUs generaría un costo excesivo en las ejecuciones del kernel por no mencionar el tiempo de ejecución y cómo terminaría afectado, por lo cual se opto por realizarlo, al momento de procesar cada bloque, esto es posible con el uso de funciones atómicas, estas funciones permiten realizar operaciones básicas sin generar colisiones y con un mínimo aumento en los tiempos de ejecución, debido a esto el *Checksum* es calculado de forma óptima y puede ser procesado cuando todos los demás bloques han terminado.

La división del mensaje esta relacionada a cómo cada bloque de *AES* es procesado, cómo se menciono anteriormente cada bloque de AES es procesado por un único hilo, no obstante es muy común que en mensajes grandes existan más bloques que hilos disponibles, es posible indicarle al GPU que use más hilos de los que realmente este tiene, no obstante esto genera tiempo muertos y no hay un orden de acceso a la memoria, por lo cual se obtienen resultados poco favorables, la solución es basada en el algoritmo divide y vencerás, cuando existe un mensaje de un tamaño superior a la cantidad de hilos disponibles por la GPU, se divide el mensaje en segmentos dado la cantidad de hilos definidos por la GPU, cuando un hilo termina un bloque del mensaje este pasa al siguiente bloque correspondiente en un nuevo segmento, de esta forma se procesan todos los bloques de un mensaje más grande de la capacidad de hilos del sistema(ver figura 4.2).

A pesar de que las implementaciones OCB3-CUDA y OCBRA-2-CUDA comparten las mismas optimizaciones estas tienen pequeñas características que las diferencian:

- **OCB3-CUDA**: En esta versión del OCB es necesario precomputar todas los Δ, en consecuencia la memoria necesaria se duplica, es necesario una delta por cada bloque de mensaje. Al necesitar más memoria el tiempo de transferencia de los datos a la GPU sera altamente afectado.

- **OCBRA-2-CUDA**: En esta versión del OCB no es necesario precomputar ningún valor, cada Δ es independiente de la otra por lo cual se puede computar al momento de procesar el bloque de mensaje correspondiente, no obstante es necesario tener

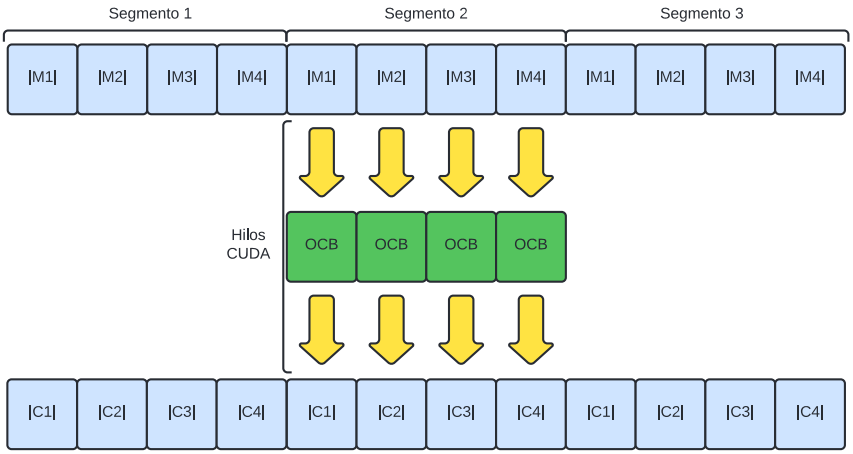

Figura 4.2: Esquema del proceso de la implementación de OCB en CUDA .

espacio suficiente en la GPU, sin embargo el tiempo de transferencia no se mira afectado porque el calculo se realiza en la GPU.

4.2. CPUs

Las implementaciones de CPUs se dividen en tres tipos 128, 256 y 512 bits, cada uno utilizando las *AVX* correspondientes, no obstante estas implementaciones comparten dos características importantes: el cifrador por bloques *AES* y la estructura del OCB. Estas características influyen directamente en el rendimiento del código y todas las versiones utilizan las mismas optimizaciones, el principal objetivo de las versiones en CPUs es aprovechar el pipeline del procesador.

4.2.1. Implementación de AES

Para la implementación de AES en CPUs se realizaron tres versiones distintas: 128, 256 y 512 bits, todas las versiones están basadas en AES-128, estas versiones buscan aprovechar el pipeline del procesador, esto se consigue realizando una cierta cantidad de instrucciones con la misma latencia y capacidad de las instrucciones, en el caso de los procesadores de onceava generación de Intel el Pipeline propuesto es de ocho, por lo cual realizar cualquier cantidad de instrucciones con la misma latencia y capacidad en múltiplos de ocho permite aprovechar el pipeline. Cada versión de las implementaciones de *AES* utiliza esta mejora, lo que diferencia a cada uno es la cantidad de bloques que se procesa por instrucción, en el caso de 128 bits solo se procesa un bloque por instrucción dando un total de ocho bloques por pipeline, en el caso de las de 256 bits son dos bloques por instrucción permitiendo 16 bloques en cada pipeline y por último las instrucciones

de 512 bits permiten procesar cuatro bloques por instrucción permitiendo un total de 32 bloques por pipeline.

4.2.2. Implementaciones de OCB

Las implementaciones de OCB en CPUs aprovechan las distintas versiones de *AES* para obtener un mejor rendimiento, no obstante las implementaciones realizadas de OCB son: OCB3-128, OCB3-256, OCB3-512, OCBRA-1-512 y OCBRA-2-512, donde las primeras tres versiones mencionadas hacen referencia al OCB3 y los tamaños de instrucciones que usan y las últimas dos versiones hacen referencia a las versiones del OCB de Acceso Aleatorio, la cantidad de rondas de AES que utilizan y el tamaño de las instrucciones respectivamente. El pipeline es la principal herramienta de optimización para estas versiones, por ello todas hacen uso de este, cada versión tiene como objetivo procesar ocho instrucciones iguales. Hay tres puntos a tener en cuenta para poder optimizar las implementaciones para CPU: instrucciones de carga y guardado, latencias de instrucciones y la memoria.

- **Instrucciones de carga y guardado**: Estas son las principales instrucciones que hay que evitar usar, debido que son consideradas las más lentas, estas instrucciones transforman los datos a variables de 128, 256 o 512 bits según se necesite y a pesar de que existan, existen formas más eficientes para asignar valores a las variables.
- **Latencias de instrucciones**: La latencia es algo a tomar en cuenta cuando se utilizan instrucciones vectoriales, usar instrucciones de distintas latencias pueden generar huecos en los ciclos de reloj del procesador generando tiempos más lentos, la solución más simple es implementar siempre la misma instrucción el número de veces el tamaño de pipeline antes de cambiar a una nueva, de esta forma se llena el pipeline aprovechando siempre las mismas latencias.
- **La memoria**: Este es el punto más crucial de todo, el mayor costo para las implementaciones esta en la memoria, las llamadas a esta tanto para guardar o obtener datos siempre es lo más complicado, cómo se menciono antes esta es la razón por la cual no se usan instrucciones de carga y guardado, la forma óptima de evitar esto es alinear la memoria dado la cantidad de bytes por instrucción que estemos usando, esto permite que las variables de las instrucciones vectoriales puedan apuntar a una dirección de memoria y obtenerlos por los segmentos en los cuales fueron divididos, de esta forma se evita el uso de instrucciones y se obtiene un mejor rendimiento.

Las cinco versiones del OCB comparten cada una de las mejoras presentadas, no obstante cada uno presenta sus propias diferencias ante las otras versiones estas son:

- **OCB3-128**: Esta versión solo procesa un máximo de ocho bloques, requiere precomputar ocho Δ para poder procesar el pipeline, la aceleración obtenida requiere un mensaje de mínimo 1024 bits para poder obtener un rendimiento óptimo.

- **OCB3-256**: Esta versión puede procesar hasta 16 bloques del mensaje completo, debido a lo mismo requiere precomputar 16 Δ, las cuales solo pueden ser procesadas en bloques de 128 bits debido a su dependencia con la anterior, afectando el rendimiento, para iniciar a obtener una aceleración considerable es requerido un mensaje de mínimo 2048 bits.

- **OCB3-512**: Esta versión del OCB3 es la que permite procesar más bloques, llegando a un total de 32, al igual que la versión de OCB3-256 requiere calcular los Δ de forma secuencial en 128 bits, es requerido un mensaje de 4096 bits para obtener las primeras aceleraciones considerables.

- **OCBRA-1-512**: Esta versión del OCB procesa 32 bloques, la principal diferencia con las otras versiones es la independencia del Δ, estos pueden ser calculados de la misma forma que los bloques principales es decir de 32 bloques a la vez sin embargo, esta versión requiere el calculo de un nuevo *nonce* cada 2^6 bloques, por lo cual es necesario computar la cantidad de *nonce* necesarios según el tamaño del mensaje, es requerido un mensaje de mínimo de 4096 bits para obtener un buen rendimiento.

- **OCBRA-2-512**: Esta versión al igual que todas las de 512 bits procesa 32 bloques. Los Δ necesarios pueden ser calculados de forma paralela al igual que los 32 bloques, de esta forma se obtiene un calculo eficiente, por último esta versión solo requiere cambiar el nonce cada 2^{31} bloques. Esta versión requiere un mensaje de 4096 bits cómo mínimo para iniciar a obtener un buen rendimientos.

Capítulo 5

Resultados

5.1. Estrategias básicas de implementación

Se implementaron las siguientes versiones del OCB3 y OCBRA, **OCB3-CUDA**, **OCBRA-2-CUDA**, **OCB3-128**, **OCB3-256**, **OCB3-512**, **OCBRA-1-512** y **OCBRA-2-512**, todas las implementaciones fueron pensadas para mensajes de al menos 4096 bytes. En las implementaciones de CPU se utilizó el conjunto de instrucciones AES-NI en sus versiones de 128, 256 y 512 bits utilizando el lenguaje de programación C. Las versiones de GPUs se realizaron en el lenguaje de programación CUDA. Estas implementaciones se explican a detalle en el capítulo 4.

El nuevo conjunto de instrucciones AES-NI512 permite el procesamiento de cuatro bloques de *AES* por instrucción por un costo similar al procesamiento de un único bloque usando AES-NI128, teóricamente llenando el pipeline del procesador se podrían procesar simultáneamente hasta 32 bloques de *AES* en un pipeline de 8 bloques, en contra de los 8 bloques de *AES* usando AES-NI128.

Para los modos de operación que utilizan *AES* como su cifrador por bloques esto permite obtener mejores aceleraciones, tales cómo el CTR o el ECB e incluso el mismo OCB permite obtener mejoras notables comparados a otras implementaciones.

CUDA permite el procesamiento de múltiples bloques de *AES* simultáneamente, el número máximo de bloques depende directamente de las características de la tarjeta en la que se realicen los experimentos. CUDA permite a llegar cantidades abrumadoras de bloques de AES procesados al mismo tiempo, sin embargo, es requerido transferir dicha información a la GPU generando un costo extra en tiempo, no obstante para mensajes extremadamente grandes puede ser una opción a considerar.

CUDA y AES-NI permiten realizar múltiples bloques de *AES* al mismo tiempo, debido a esto se tiene como objetivo procesar la mayor cantidad de bloques de *AES* para cada versión de OCB implementada, utilizando mensajes de tamaños considerablemente grandes.

Es posible procesar de dos formas distintas un mensaje usando *AES* estas son: usando

el mismo *AES* y usando distintos *AES*

- **Mismo mensaje distintos *AES***: Esta versión esta pensada para utilizar distintas llaves de *AES* para cifrar los bloques del mensaje, es decir, es similar a cifrar múltiples mensajes distintos donde cada uno tiene una llave única de *AES*, no obstante es poco ortodoxo realizar esta acción, no es común tener la necesidad de cifrar muchos mensajes a la vez.
- **Mismo mensaje mismo *AES***: En esta versión cada bloque del mensaje es cifrado usando la misma llave, de esta forma todos los recursos se centran únicamente en realizar la misma acción a distintas partes del mensaje por lo cual se pueden obtener mejoras en el rendimiento.

Las pruebas realizadas en los experimentos han seguido estrategia del **Mismo mensaje mismo *AES***, a pesar de que la otra versión podría tener resultados interesantes, tanto el modo de operación cómo el enfoque de esta tesis es exclusivamente para un único mensaje de gran tamaño.

5.2. Especificaciones técnicas de las plataformas de prueba CPU y GPU

Para realizar las pruebas se utilizaron dos equipos de computo distintos con las siguientes especificaciones:

Equipo 1 CPU (M1)

- **CPU**: Intel i7-11800H 2.3GHz 24MB de cache.
- **GPU**: NVIDIA RTX A2000 Laptop 4GB de memoria VRAM.
- **Memoria volátil**:32GB a 3200 MHz DDR4.
- **Memoria**: SSD M.2 512GB a 6900MB de lectura y 5000MB de escritura.
- **versión de C++**: 14.
- **Compilación**: g++ ocb_version.cpp -o ocb_version -O3 -march=native.
- **Sistema operativo**: GNU/Liux Ubuntu 20.04 LTS.

Equipo 2 GPU (M2)

- **CPU**: Intel Xeon Gold 6230R 2.1GHz 35.75MB de cache.
- **GPU**: NVIDIA Quadro P2200 5GB de memoria VRAM.

- **Memoria volátil**: 32GB a 3200 MHz DDR4.
- **Versión de CUDA**: 11.2.
- **Controlador**: Versión del controlador 460.91.03.
- **Compilación**: nvcc ocb_version.cu -o ocb_version -O3 -gencode arch=compute_61,code=sm_61.
- **Sistema operativo**: GNU/Liux Debian 11 Bullseye.

Los equipos M1 y M2 fueron utilizados para las pruebas de CPU y GPU respectivamente. M1 cuenta con soporte para todos los tamaños de AES-NI y AVX, M2 cuenta con una GPU capaz de soportar las instrucciones realizadas. Los compiladores usados fueron GCC y NVCC con sus respectivas indicaciones para el uso de las instrucciones, ademas se añadio la bandera -O3 para una optimización por parte del compilador para ambos casos. En el caso de M1 las tecnologías de *Hyper-Threading* y *Turbo-Boost* de Intel fueron deshabilitadas.

5.3. Medición del tiempo

Para la evaluación del tiempo se reutilizo el código proporcionado por Ted Krovetz en [20], este consta en ejecutar repetidas veces el código y calcula los ciclos por byte y el tiempo dada la frecuencia del procesador que se este usando.

Para las versiones de CPU se busco reducir la latencia que implica cargar la memoria cache en los primeros tamaños de mensajes, esto se realiza con un proceso de calentamiento, es decir, se ejecuta el algoritmo unas cuantas veces antes de medir el tiempo con el fin de tener todos los datos en cache. Cabe resaltar que esto solo beneficia a los mensajes más pequeños, para mensajes de mayor tamaño la latencia de la cache es algo que afectará el rendimiento del programa.

Para medir el tiempo en GPUs se realizo un proceso similar basado en ejecutar múltiples veces el programa y obtener los tiempos, la principal diferencia de estos radica en utilizar *events* de CUDA, objetos que permiten obtener el tiempo de ejecución de las distintas etapas de un programa de CUDA sin afectar el rendimiento del mismo.

5.4. Tiempos de ejecución en GPUs

Los tiempos presentados en esta sección son el resultado de múltiples ejecuciones de los programas **OCB3-CUDA** y **OCBRA-2-CUDA**. En la tabla 5.1 se puede apreciar el tiempo total necesario para procesar diversos tamaños de mensajes, en esta tabla se muestra el aumento del tiempo respecto al tamaño del mensaje. Los tamaños de los mensajes para las pruebas son de 4KB, 8KB, 16KB, 1MB, 2MB, 4MB, 16MB, 67MB, 1GB

	Tiempo total del las implementaciones en milisegundos del COB3 y OCBRA en GPU									
Tamaño de mensaje	4KB	8KB	16KB	1MB	2MB	4MB	16MB	67MB	1GB	1.5GB
OCB3-CUDA	0.51	0.53	0.51	1.91	3.07	5.14	18.01	72.06	987.76	1475.55
OCBRA-2-CUDA	0.46	0.46	0.46	1.34	2.57	3.10	11.93	42.36	686.04	1068.07

Tabla 5.1: Resultados del tiempo total en milisegundos para cifrar mensajes utilizando OCB3 y OCB de Acceso Aleatorio en la GPU Nvidia Quadro P2200

y 1.5GB. Estos diversos tamaños de mensajes permiten apreciar el cambio del desempeño de cada algoritmo.

En la tabla 5.2 se muestran el tiempo de transferencia de los datos hacia la GPU, en ella se puede percibir el aumento del tiempo conforme los mensajes son más grandes, haciendo que sea un punto a considerar para futuras implementaciones. Este incremento de tiempo se debe al precomputo necesario para cada versión, es necesario almacenar los datos temporalmente antes de enviarlos a la GPU, en el caso del OCB3 es necesario precomputar cada uno de los Δ por lo cual el tamaño se duplica esto se mira afectado en los mensajes de mayor tamaño. En la tabla 5.2 se exponen los tiempos requeridos para el precomputo de cada una de las versiones del OCB, es posible notar un incremento en el tiempo para el OCB3, esto se debe al calculo de los Δ mencionado anteriormente.

	Tiempo de transferencia de datos en milisegundos a la GPU									
Tamaño de mensaje	4KB	8KB	16KB	1MB	2MB	4MB	16MB	67MB	1GB	1.5GB
OCB3-CUDA	0.34	0.35	0.34	0.97	1.34	1.88	5.37	20.98	343.19	516.05
OCBRA-2-CUDA	0.35	0.35	0.35	0.66	1.06	1.26	3.51	12.49	223.61	373.16

Tabla 5.2: Resultados del tiempo de transferencia de datos en milisegundos para la GPU Nvidia Quadro P2200 para distintos tamaños de mensajes

En las figuras 6.1, 6.2, 6.3 y 6.4 se expone la comparación de los tiempos de transferencia usando gráficas de barras. En la figura 6.1 es posible apreciar que el cambio para mensajes pequeños, es decir, 4 KB, 8 KB y 16 KB es mínimo. La diferencia de tiempos de la transferencia entre el **OCB3-CUDA** y el **OCBRA-2-CUDA** es casi inexistente en algunos casos, teniendo diferencias de 0.01 hasta 0.02 milisegundos. No obstante conforme el tamaño del mensaje incrementa de la misma manera aumenta el tiempo de transferencia para los mensajes de 1MB 2Mb y 4MB (ver figura 6.2) la diferencia ya es considerablemente notoria. El **OCBRA-2-CUDA** tiene una ventaja a considerar logrando hasta 0.6 milisegundos de diferencia contra el **OCB3-CUDA**, esto se debe principalmente a los valores almacenados de precomputo. Cuando los mensajes se vuelven considerablemente grandes es decir 16MB, 67MB, 1GB y 1.5GB la diferencia se vuelve muy grande (ver figuras 6.3 y 6.4), teniendo a **OCBRA-2-CUDA** con una ventaja de hasta 140 milisegundos para mensajes de 1.5GB, este número es considerablemente alto debido a que representa un aumento en la velocidad del 27 % en la transferencia de datos en contra de la versión **OCB3-CUDA**.

	Tiempo del preprocesamiento de GPU en milisegundos									
Tamaño de mensaje	4KB	8KB	16KB	1MB	2MB	4MB	16MB	67MB	1GB	1.5GB
OCB3-CUDA	0.00465	0.00505	0.00494	0.04797	0.09082	0.1729	0.70647	9.67631	153.7912	231.0173
OCBRA-2-CUDA	0.00354	0.00333	0.00341	0.00389	0.00407	0.00413	0.00403	0.00395	0.00415	0.00428

Tabla 5.3: Resultados del tiempo del preprocesamiento del OCB3 y OCB de Acceso Aleatorio para la GPU Nvidia Quadro P2200.

La diferencia de los tiempos de transferencia entre las versiones de **OCB3-CUDA** y **OCBRA-2-CUDA** se debe al precomputo de cada uno de estos. Como es posible apreciar en la figura 6.5 se muestra una gráfica que expone la evolución del incremento del tiempo de precomputo, es notable que **OCBRA-2-CUDA** mantiene su tiempo de precomputo a diferencia de la versión **OCB3-CUDA** el cual incrementa de manera considerable (tabla 5.3. El cálculo del Δ es secuencial debido a esto es poco factible realizar el calculo de estos valores en CUDA, a pesar de existir maneras en que se puede llegar a calcular usando CUDA, estas versiones serían ineficientes principalmente por la necesidad de calcular siempre todos los valores pasados para obtener el actual.

En los tiempos del cifrado de los datos para las versiones de **OCB3-CUDA** y **OCBRA-2-CUDA** se puede apreciar que para los mensajes pequeños, 4, 8 y 16KB la diferencia es mínima en tiempo, no obstante al igual que el tiempo de transferencia y el tiempo de precomputo conforme el mensaje tiene un tamaño mayor la diferencia del tiempo aumenta. **OCBRA-2-CUDA** tiene una ventaja considerable en tiempo total contra el **OCB3-CUDA**, llegando a ser hasta un 38 % más rápido.

5.5. Tiempos de ejecución en CPUs

Los tiempos presentados en esta sección son el resultado de múltiples ejecuciones de los programas **OCB3-128**, **OCB3-256**, **OCB3-512**, **OCBRA-1-512** y **OCBRA-2-512**, la cantidad de ejecuciones esta relacionada directamente a la frecuencia base del procesador utilizado. Todas las pruebas fueron ejecutadas en el equipo **M1** el cual cuenta con un Intel i7-11800H. Se realizaron pruebas de cifrado de distintos tamaños de mensajes 4KB, 8KB, 16KB, 1MB, 2MB, 4MB, 16MB, 67MB, 1GB y 1.5GB. En la tabla 5.4 se muestra el tiempo promedio para cifrar los distintos tamaños de mensaje, en esta tabla se puede ver el incremento del tiempo dado el aumento en el tamaño del mensaje.

En la figura 6.6 se puede apreciar la comparación de rendimientos en mensajes pequeños 4 KB, 8 KB y 16 KB respectivamente, se puede apreciar una diferencia considerable entre las versiones de 128 y 256 bits contra las versiones de 512 bits. Las versiones del **OCB3-128**, **OCB3-256** y **OCB3-512**, tienen una dependencia secuencial del calculo del Δ, cada uno de estos requiere calcular 8 Δ, 16 Δ y 32 Δ de forma secuencial antes de procesar los bloques del OCB correspondientes, debido a esto podemos ver que son ligeramente más lentos en los mensajes cortos que las versiones de OCBRA de 1 y 2 rondas de AES respectivamente en 512 bits, esta relación se mantiene también con mensajes de 1 MB, 2

	Tiempo en milisegundos del cifrado en CPU									
Tamaño de mensaje	4KB	8KB	16KB	1MB	2MB	4MB	16MB	67MB	1GB	1.5GB
OCB3-128	0.0007208	0.0013954	0.0027468	0.2127537	0.4257654	0.8498692	3.4887611	15.3434390	243.0166000	303.1049798
OCB3-256	0.0006926	0.0012789	0.0024326	0.1507995	0.3029211	0.6136949	2.4856108	10.9197632	175.9556930	263.4645614
OCB3-512	0.0005483	0.0009702	0.0018162	0.1095722	0.2207153	0.4416348	1.8029088	8.7292254	144.8310634	216.9126408
OCBRA-1-512	0.0005029	0.0008336	0.0015140	0.0971545	0.1933196	0.3871236	1.6104974	9.3800303	175.2986591	264.2564697
OCBRA-2-512	0.0005087	0.0008492	0.0015301	0.0873818	0.1769009	0.3540615	1.4517707	7.5836564	127.8253620	192.2395644

Tabla 5.4: Resultados del tiempo total del cifrado en milisegundos en el CPU I7-11800H

Versión	**Bloques**	**Costo total en CPB**	**Costo por bloque en CPB**
OCB3-128	8	60	7.5
OCB3-256	16	88	5.5
OCB3-512	32	160	5
OCBRA-1-512	32	110-180	3.4-5.625
OCBRA-2-512	32	116	3.625

Tabla 5.5: Costo en CPB para procesar un bloque de mensaje en CPU para cada versión implementada del OCB3 y OCBRA

MB y 4 MB (ver figura 6.7).

El desempeño obtenido en mensajes cortos muestra que tanto **OCBRA-1-512** y **OCBRA-2-512** son las mejores opciones, no obstante el aumento del incremento del tiempo con respecto al tamaño de los mensajes no se mantiene para el **OCBRA-1-512**. En la figura 6.8 se puede apreciar que conforme el mensaje incrementa su tamaño el **OCBRA-1-512** deja de ser una opción óptima, esto se debe al precomputo necesario para la obtención de un nuevo Δ cada 64 bloques. Para 16 MB y 67 MB es posible notar el incremento del tiempo, en los mensajes de 1 GB y 1.5 GB (ver figura 6.9) el desempeño del **OCBRA-1-512** es equiparable con el **OCB3-256**, por lo cual deja de ser una opción a tener en cuenta.

En la tabla 5.6 se muestran los **ciclos por byte (CPB)** para cada tamaño de mensaje. La cantidad de CPB disminuye conforme el mensaje aumenta su tamaño, sin embargo, el CPB tienden a subir pasando los 16 MB. Lo anterior se debe a que los mensajes subsecuentes superan el tamaño de la memoria cache del procesador por lo cual es requerido un mínimo ciclos para el traslado de la memoria RAM a la memoria cache del procesador. A pesar del incremento obtenido el **OCBRA-2-512** es la versión que requiere menos CPB por lo tanto utiliza la menor cantidad de tiempo, llegando a ser hasta dos veces más rápido que la versión de **OCB3-128** en ciertos tamaños de mensajes. Este incremento era esperado, en la tabla 5.5 se muestra el costo teórico de procesar cada bloque para las distintas versiones del OCB, donde podemos apreciar que el **OCB3-128** es aproximadamente 2,06 veces más caro que el **OCBRA-2-512**. Estos costos fueron calculados usando la información proporcionada por Intel en el manual de Intrinsics y comprobando la cantidad de llamadas realizaras a cada función. El **OCBRA-1-512** queda descartado porque su costo varia dado el tamaño del mensaje, llegando a tener desempeños similares

	Ciclos por byte usados en CPU									
Tamaño de mensaje	4KB	8KB	16KB	1MB	2MB	4MB	16MB	67MB	1GB	1.5GB
OCB3-128	0.4047200	0.3917800	0.3856000	0.3833700	0.3851300	0.3850500	0.3912000	0.4334900	0.4332000	0.4328400
OCB3-256	0.3889000	0.3590700	0.3415000	0.3306900	0.3378700	0.3358400	0.3407500	0.3742500	0.3769000	0.3762300
OCB3-512	0.3078800	0.2724100	0.2549600	0.2403400	0.2420600	0.2421800	0.2471600	0.2991700	0.3102300	0.3097600
OCBRA-1-512	0.2823800	0.2340400	0.2125400	0.2131000	0.2120200	0.2122800	0.2207800	0.3214800	0.3755000	0.3773700
OCBRA-2-512	0.2856400	0.2384200	0.2148000	0.1916700	0.1940100	0.1941500	0.1990200	0.2599100	0.2738100	0.2745200

Tabla 5.6: Resultados de los ciclos por bytes en el CPU I7-11800H

	Ciclos por byte usados en CPU									
Tamaño de mensaje	4KB	8KB	16KB	1MB	2MB	4MB	16MB	67MB	1GB	1.5GB
OCB3-128	0.41735	0.40500	0.39943	0.40566	0.40465	0.40220	0.41088	0.41807	0.40648	0.40602
OCB3-256	0.39080	0.36042	0.34179	0.32795	0.32881	0.32834	0.33535	0.33959	0.33730	0.33691
OCB3-512	0.30978	0.27490	0.25667	0.23912	0.23898	0.24059	0.24834	0.25132	0.25201	0.25123
OCBRA-1-512	0.28367	0.23474	0.21112	0.21043	0.20906	0.20998	0.23556	0.23847	0.27046	0.26324
OCBRA-2-512	0.28604	(0.23871	0.21512	0.19271	0.19225	0.19346	0.20111	0.20204	0.20234	0.20239

Tabla 5.7: Resultados de los ciclos por bytes en el CPU i7-1065G7

que el **OCB3-256**. Teniendo una ventaja teórica de un poco más de 2 es posible esperar un desempeño en el tiempo similar, no obstante una vez pasado el tamaño de la cache del procesador esta ventaja es reducida.

.

Capítulo 6

Conclusiones

6.1. Conclusión

Con la inclusión de las nuevas instrucciones AES-NI512 para los procesadores actuales y el desarrollo de nuevas GPUs con mayor capacidad de procesamiento, se puede explorar nuevos métodos de paralelismo que favorecen a los algoritmos con un amplio grado de paralelismo. Lo anterior se ha demostrado en los resultados de esta tesis. Dado los resultados obtenidos se puede apreciar el que el desempeño del *OCB de Acceso Aleatorio* es superior al *OCB3* tanto en sus versiones de CPU y GPU. En el caso de las CPU es necesario utilizar las instrucciones de 512 bits para obtener un resultado favorable mientras que en las versiones de GPU es necesario tener un mejor manejo de la memoria para poder obtener resultados favorables. El *OCB de Acceso Aleatorio* permite un paralelismo a gran escala debido a la independencia de sus bloques permitiendo procesar siempre la cantidad máxima de bloques que permita el sistema donde se implemente (GPU o CPU), es debido a esta ventaja que cuando se procesan mensajes grandes se puede apreciar como la latencia del algoritmo se limita al traslado del mensaje de la memoria RAM a la memoria cache, mientras que el *OCB3* requiere un precomputo constante de los valores de Δ para hacer el xor bloques con los bloques de mensajes.

Realizar una implementación óptima es un proceso complicado, los criterios para ella en CPU y GPU son distintos. En CPU es necesario tener en cuenta tres puntos importantes: el costo de las instrucciones, el pipeline del procesador y la arquitectura del procesador. Conocer la arquitectura te permite saber tanto el costo de las instrucciones cómo el tamaño del pipeline, en algunos casos esta información no es proporcionada por los fabricantes por lo cual es necesario hacer múltiples pruebas de rendimiento para obtener un aproximado de dicho pipeline. En GPU es necesario saber el comportamiento de la memoria, poder ordenar el problema para procesarlo en paralelo y conocer el tamaño máximo de núcleos que tiene la GPU.

En general las implementaciones para procesadores actuales del *OCB de Acceso Aleatorio* son una mejora en contra de las implementaciones del *OCB3* permitiendo procesar

una mayor cantidad de datos en menor tiempo como se muestra en las tablas 5.6 donde están los ciclos por byte necesarios una menor cantidad de ciclos por byte indica un menor tiempo total, para ver esta mejora es necesario un mensaje de al menos 4KB. Es posible ver un incremento de los ciclos por byte conforme el tamaño del mensaje aumenta sin embargo, este incremento depende directamente de la arquitectura del procesador y cómo realice el manejo de la memoria cache. Las GPU permitieron demostrar el gran paralelismo que existe para el *OCB de Acceso Aleatorio* permitiendo resultados bastantes convincentes, estos resultados pueden mejorar con el uso de GPU de generaciones más recientes, incluso es factible pensar que se puede superar a las CPU no obstante, el precio de estas nuevas GPU supera por mucho el precio de los procesadores actualmente en el mercado, llegando a ser entre 5 a 6 veces mayor. Con las nuevas arquitecturas de GPU y mayor cantidad de núcleos, incluso es posible ver a futuro un desempeño mejor usando implementaciones más eficientes de AES. Hoy en día la opción más viable por el desempeño obtenido y el precio del equipo las CPU tienen una clara ventaja.

6.2. Trabajo a futuro

cómo trabajo a futuro se podrida investigar los siguientes aspectos:

- Proponer una nueva forma de calcular Δ, una Δ más óptima permitiría la obtención de mayores velocidades usando instrucciones de 512 bits.
- Proponer una construcción genérica de modos de operación de acceso aleatorio permitiendo así transformar cualquier modo de operación actual a uno de acceso aleatorio.
- Experimentar con nuevos conjuntos de instrucciones para el futuro, así cómo llego AES-NI512 es solo cuestión de tiempo para que AES-NI1024 sea implementado en los procesadores.
- Experimentar con otros modelos de GPU más actuales, esto permitirá ver el comportamiento de un modo de operación de acceso aleatorio en arquitecturas de GPU recientes.
- Implementar el OCB de acceso aleatorio en servicios de transmisiones en tiempo real.
- Comprobar el desempeño del OCB de acceso aleatorio en otras arquitecturas de CPU usando mensajes de al menos 1GB para comprobar el desempeño de la memoria cache.

Apéndice

Tablas de tiempo de GPU y CPU

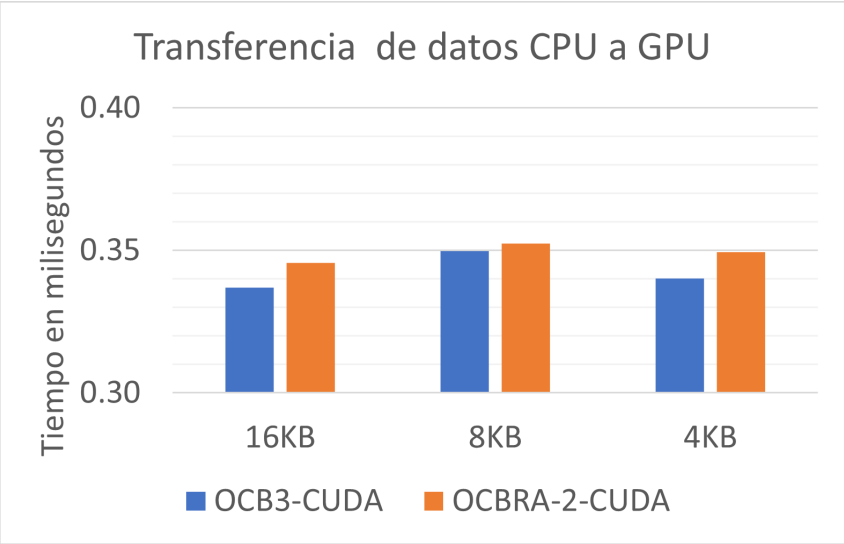

Figura 6.1: Comparación del tiempo de transferencia de datos para mensajes de 4KB, 8KB y 16KB usando la Nvidia Quadro P2200

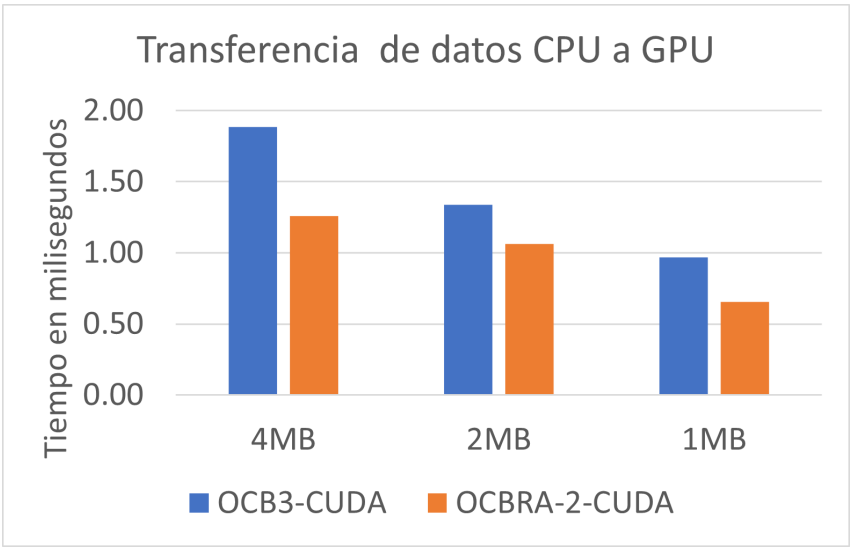

Figura 6.2: Comparación del tiempo de transferencia de datos para mensajes de 1MB, 2MB y 4MB usando la Nvidia Quadro P2200

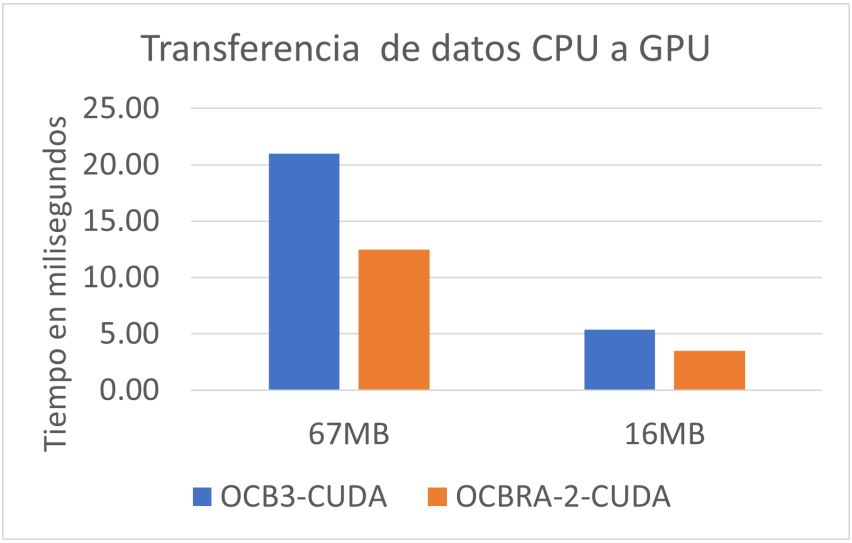

Figura 6.3: Comparación del tiempo de transferencia de datos para mensajes de 16MB y 67MB usando la Nvidia Quadro P2200

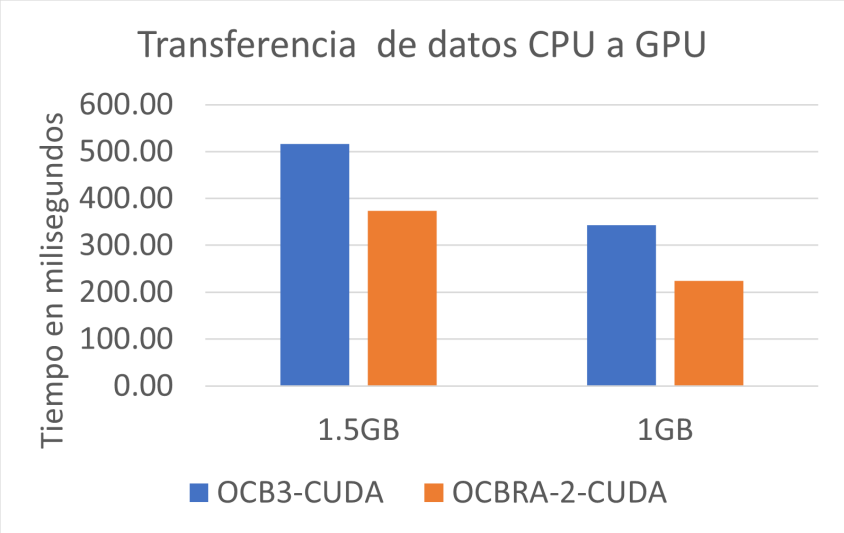

Figura 6.4: Comparación del tiempo de transferencia de datos para mensajes de 1GB y 1.5GB usando la Nvidia Quadro P2200

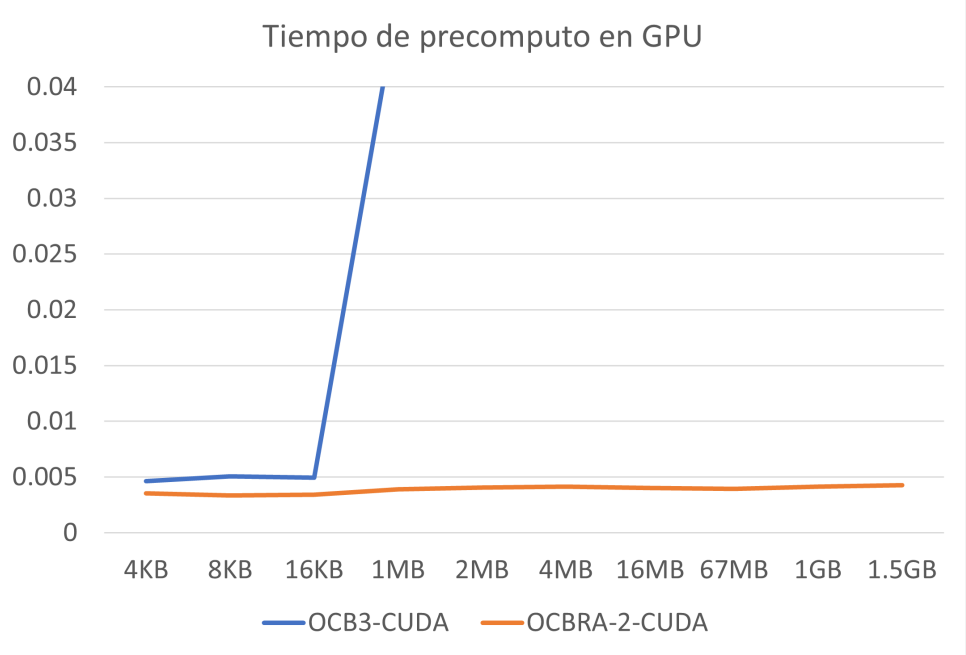

Figura 6.5: Aumento del tiempo de precomputo para GPU

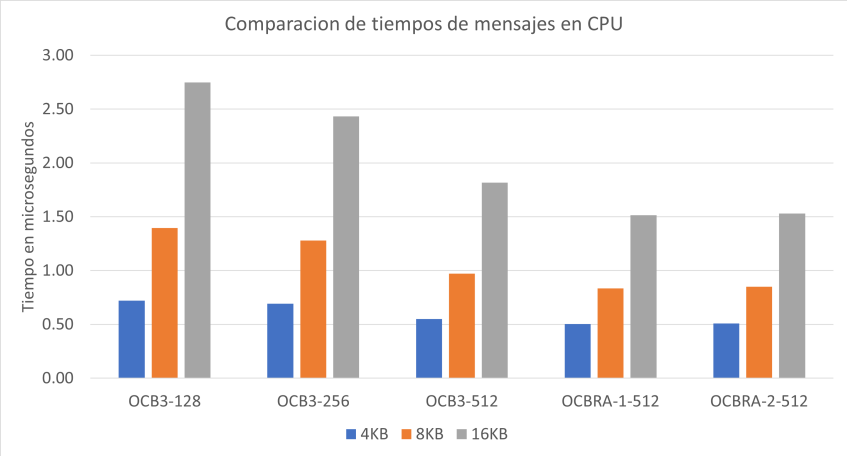

Figura 6.6: Comparación del tiempo para cifrar datos para mensajes de 4KB, 8KB y 16KB usando el CPU I7-11800H

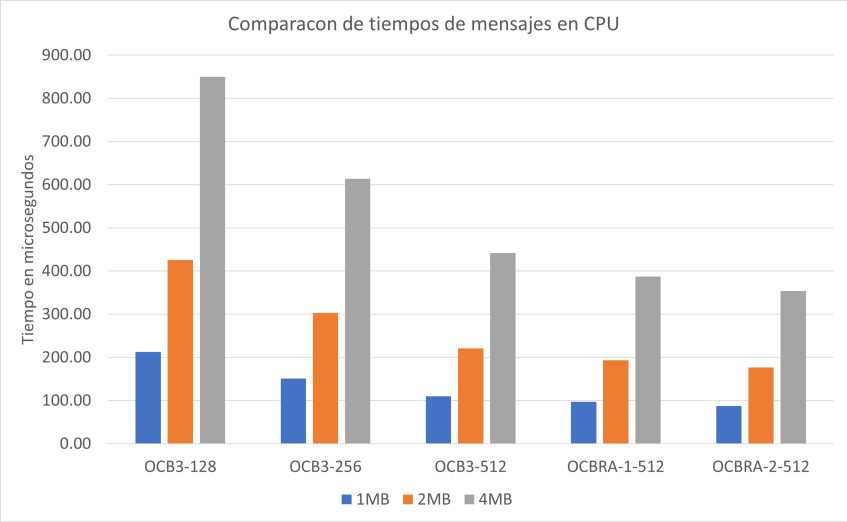

Figura 6.7: Comparación del tiempo para cifrar datos para mensajes de 1MB, 2MB y 4MB usando el CPU I7-11800H

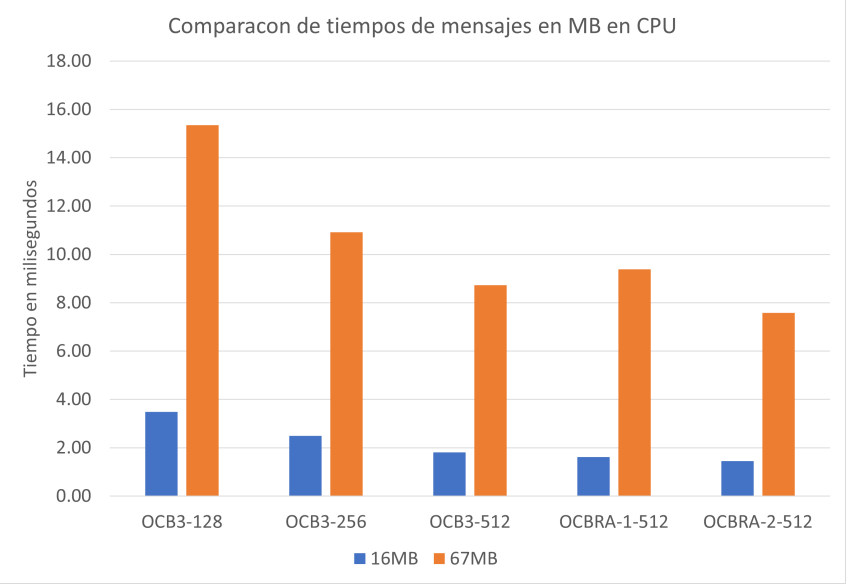

Figura 6.8: Comparación del tiempo para cifrar datos para mensajes de 16MB y 67MB usando el CPU I7-11800H

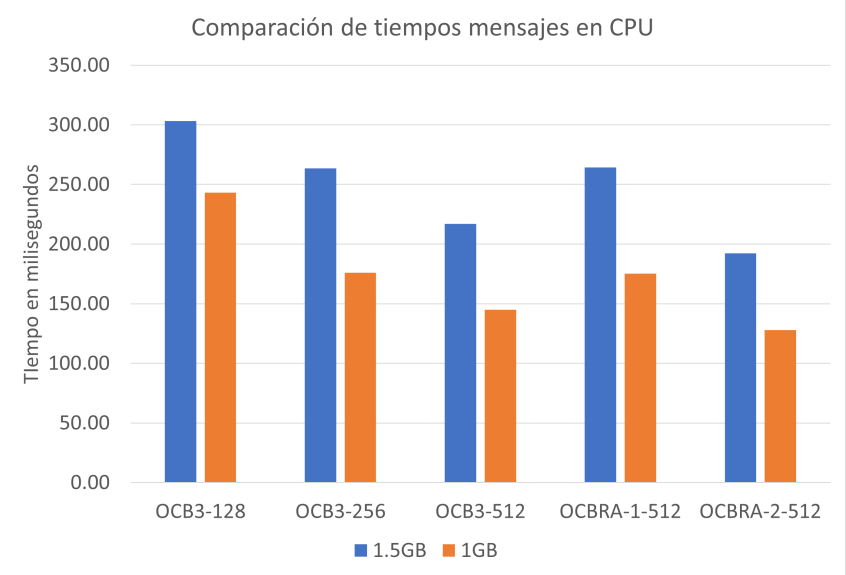

Figura 6.9: Comparación del tiempo para cifrar datos para mensajes de 1GB y 1.5GB usando el CPU I7-11800H

Bibliografía

[1] Intel Corporation. The compute architecture of intel processor graphics gen 9. Technical report, Intel Corporation, 2015.

[2] Intel Inc. 11th generation intel core processor desktop, 2022-05-01 2022.

[3] Ashwin Jha, Cuauhtemoc Mancillas-López, Mridul Nandi, and Sourav Gupta. On random read access in ocb. *IEEE Transactions on Information Theory*, PP:1–1, 06 2019.

[4] Phillip Rogaway, Mihir Bellare, John Black, and Ted Krovetz. OCB mode. *IACR Cryptol. ePrint Arch.*, page 26, 2001.

[5] Keisuke Iwai, Naoki Nishikawa, and Takakazu Kurokawa. Acceleration of AES encryption on CUDA GPU. *Int. J. Netw. Comput.*, 2(1):131–145, 2012.

[6] Qinjian Li, Chengwen Zhong, Kaiyong Zhao, Xinxin Mei, and Xiaowen Chu. Implementation and analysis of AES encryption on GPU. In Geyong Min, Jia Hu, Lei (Chris) Liu, Laurence Tianruo Yang, Seetharami Seelam, and Laurent Lefèvre, editors, *14th IEEE International Conference on High Performance Computing and Communication & 9th IEEE International Conference on Embedded Software and Systems, HPCC-ICESS 2012, Liverpool, United Kingdom, June 25-27, 2012*, pages 843–848. IEEE Computer Society, 2012.

[7] Dobre Blazhevski, Adrijan Bozhinovski, Biljana Stojcevska, and Veno Pachovski. Modes of operation of the aes algorithm. 04 2013.

[8] Rone Kwei Lim, Linda R. Petzold, and Çetin Kaya Koç. Bitsliced high-performance AES-ECB on gpus. In Peter Y. A. Ryan, David Naccache, and Jean-Jacques Quisquater, editors, *The New Codebreakers - Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*, volume 9100 of *Lecture Notes in Computer Science*, pages 125–133. Springer, 2016.

[9] Canhui Wang and Xiaowen Chu. GPU accelerated AES algorithm. *CoRR*, abs/1902.05234, 2019.

[10] Nir Drucker, Shay Gueron, and Vlad Krasnov. Making AES great again: the forthcoming vectorized AES instruction. *IACR Cryptol. ePrint Arch.*, page 392, 2018.

[11] Morris Dworkin, Elaine Barker, James Nechvatal, James Foti, Lawrence Bassham, E. Roback, and James Dray. Advanced encryption standard (aes), 2001-11-26 2001.

[12] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. *J. Cryptology*, 24:588–613, 2011.

[13] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes ocb and pmac. In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004*, pages 16–31, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[14] John Black and Phillip Rogaway. A block-cipher mode of operation for parallelizable message authentication. In Lars R. Knudsen, editor, *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*, pages 384–397. Springer, 2002.

[15] Tetsu Iwata and Kaoru Kurosawa. OMAC: one-key CBC MAC. In Thomas Johansson, editor, *Fast Software Encryption, 10th International Workshop, FSE 2003, Lund, Sweden, February 24-26, 2003, Revised Papers*, volume 2887 of *Lecture Notes in Computer Science*, pages 129–153. Springer, 2003.

[16] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology*, 21:469–491, 10 2008.

[17] John von Neumann. First draft of a report on the EDVAC. *IEEE Ann. Hist. Comput.*, 15(4):27–75, 1993.

[18] Phillip Rogaway, Mihir Bellare, and John Black. Ocb: A block-cipher mode of operation for efficient authenticated encryption. *ACM Trans. Inf. Syst. Secur.*, 6(3):365–403, aug 2003.

[19] Tetsu Iwata. Plaintext recovery attack of ocb2. Cryptology ePrint Archive, Paper 2018/1090, 2018. https://eprint.iacr.org/2018/1090.

[20] Ted Krovetz and Phillip Rogaway. The software performance of authenticated-encryption modes. In Antoine Joux, editor, *Fast Software Encryption*, pages 306–327, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[21] Rone Kwei Lim, Linda Ruth Petzold, and Çetin Kaya Koç. Bitsliced high-performance aes-ecb on gpus. In *LNCS Essays on The New Codebreakers - Volume 9100*, page 125–133, Berlin, Heidelberg, 2015. Springer-Verlag.

[22] Canhui Wang and Xiaowen Chu. Gpu accelerated aes algorithm. *arXiv preprint arXiv:1902.05234*, 2019.