UNIVERSIDAD NACIONAL “SIGO XX”

DIRECCIÓN DE POSTGRADO

DIPLOMADO EN REDES Y SEGURIDAD INFORMÁTICA

ESTÁNDAR DE ENCRIPTACIÓN AVANZADA AES RIJNDAEL ACELERADO MEDIANTE UNIDAD DE PROCESAMIENTO GRÁFICO GPU

TRABAJO DE GRADO

AUTOR: DANIEL JIMENEZ JEREZ

La Paz - Bolivia
2018

Índice general

Índice de figuras

Índice de cuadros

Índice de Anexos

Capítulo 1
Introducción

En este capítulo se muestran: el panorama general del problema que se desea solucionar, la importancia teórica y práctica del desarrollo de la solución, el método empleado en el trabajo y el alcance que tiene la solución desarrollada.

1.1. Problemática

El paradigma computacional ha cambiado bastante desde la comercialización de la computadora personal (PC). De acuerdo a la ley de Moore: “el número de componentes de un circuito integrado seguirá doblándose cada año, y en 1975 serán mil veces más complejos que en 1965” [Moore, 1965, p. 2].

Sin embargo actualmente, con la popularidad que alcanzaron los conceptos de Dispositivos Inteligentes (Smart Devices), Dispositivos Portátiles (Wearables) y el Internet de las Cosas (IOT), se observa claramente que los circuitos integrados han llegado a un límite de tamaño tan pequeño que es muy difícil de reducir desde hace algunos años.

Para solventar tal limitación es que los fabricantes de microprocesadores cambiaron el paradigma de desarrollo de hardware de la arquitectura mono-núcleo a la arquitectura multi-núcleo. Posterior a este cambio se pudo observar que el límite del número de procesadores de uso general ha llegado también a un límite difícil de superar debido al calentamiento al que son sometidos los circuitos integrados dentro de dichos procesadores ya que la temperatura es inversamente proporcional al tamaño de los dispositivos y al trabajo de cómputo que se designa a cada circuito; por ello muchas aplicaciones actuales hacen uso de recursos definidos para tareas específicas, por ejemplo

Cuadro 1: Aplicaciones del algoritmo AES

USO	APLICACIONES
Compresiòn de datos	7z, Amanda Backup, PeaZip, PKZIP, RAR, WinZip, UltraISO
Encriptación de archivos	Gpg4win, Ncrypt
Encriptación de particiones de disco duro	NTFS, BTRFS
Encriptación de discos duros	BitLocker, CipherShed, DiskCryptor, FileVault, GBDE, Geli, LibreCrypt, LUKS, Private Disk, VeraCrypt
Seguridad en comunicaciones LAN	CCMP, ITU-T G.hn, IPsec
Seguridad en comunicaciones en Internet	GPG, TLS, SSL
Otras aplicaciones	KeePass Password Safe, Pidgin, Google Allo, Facebook Messenger, WhatsApp

Fuente: Implementaciones AES, Aplicaciones, Wikipedia

para el cómputo de bloques muy grandes de imágenes o videos, se utilizan tarjetas gráficas que procesan volúmenes gigantescos de datos para entregar el resultado nuevamente al procesador central o bien para mostrar el resultado por pantalla u otro dispositivo de salida.

Debido a la carga de tareas que se asigna a la CPU[1], ésta puede llegar a formar colas insostenibles de procesos, por lo tanto, se intentará demostrar la factibilidad de la distribución de tareas a los procesadores de la Unidad de Procesamiento Gráfico (GPU[2]) para incrementar la velocidad de ejecución del algoritmo Estándar de Encriptación Avanzada (AES Rijndael).

1.2. Importancia teórica y práctica

El Estándar de Encriptación Avanzada (AES Rijndael) es utilizado en muchas aplicaciones actuales debido a su característica de patrón abierto para uso público y privado en aplicaciones personales y empresariales.

Cualquier incremento en la velocidad de ejecución de este algoritmo es de gran importancia práctica, ya que, como se muestra en el cuadro 1, los protocolos SSL y TLS trabajan con encriptación AES Rijndael, y es sabido que una gran parte de los servicios brindados en VPN y HTTPS para red local y/o internet transmiten grandes bloques de información enctriptada, por lo cual la ejecución de este proceso debe ser lo más rápida posible a fin de evitar latencia en las comunicaciones.

Por otra parte, en lo que respecta a la teoría de hilos y paralelismo de ejecución de procesos, la investigación del uso de tarjetas gráficas como unidades de procesamiento de datos es todavía un área joven sobre la cual se está comenzando a investigar, con el desarrollo de la tecnología TITAN de NVidia [3], misma que pone a disposición mallas de miles de procesadores Tensor y CUDA; dichos procesadores son núcleos de uso específico y no cuentan con las capacidades de los procesadores de uso general.

Entre los aspectos que caracterizan estas tecnologías de procesamiento están principalmente las operaciones matriciales, las operaciones de bucle independiente y las operaciones de transformación de datos.

1.3. Método de investigación

El método que de enfoque para el desarrollo del proyecto es el método

[1] Unidad Central de Procesamiento (Central Processing Unit)
[2] Graphics Processing Unit
[3] Tecnología NVidia Titan RTX

cuantitativo, ya que los objetivos se demuestran con tablas comparativas de resultados; empírico, ya que se ejecutaron programas de cómputo para llegar a los resultados; racionalista, ya que los resultados se toman en cuenta sin ninguna interpretación previa y positivista, ya que se desea demostrar el la aserción del incremento de velocidad de ejecución del algoritmo AES Rijndael en GPU con respecto a la velocidad de ejecución en CPU.

Capítulo 2
Justificación del problema

En este capítulo se muestran la justificación del problema con una visión desde diferentes ámbitos o enfoques de acuerdo a los paradigmas tecnológico y científico.

2.1. Justificación tecnológica

Como se mencionó anteriormente, de acuerdo al paradigma actual del desarrollo de hardware de microprocesadores, se llegó a un límite difícil de superar con los materiales de fabricación actuales.

Intel propuso el modelo de desarrollo de hardware “Tick-Tock” que consiste en un lanzamiento cada 18 meses (1 año y medio), ambas palabras hacen referencia a:

Tick: Mejoría de una arquitectura anterior
Tock: Lanzamiento de una nueva arquitectura

Pero si se realiza una comparación del último “Tock” que realizó Intel con respecto al microprocesador de la gama i7, se obtiene un resultado claro, y es que en 6 años se redujo el tamaño de transistor de 22nm a 14nm, duplicando los procesadores para llegar a un total de 8 procesadores físicos del CPU Intel i7-3770 del año 2012 al procesador Intel i9-9900 del año 2018. Pero la frecuencia se incrementó tan solo en 2MHz, por lo tanto se puede concluir que el paradigma tiene como objetivo llegar a multiplicar el número de procesadores pero no así la frecuencia de trabajo de los mismos.

Cuadro 2: Comparación procesadores Intel i7

Característica	**I9-9900K**	**I7-3770**
Núcleos	8	4
Hilos	16	8
Serie	Coffee Lake	Ivy Bridge
Socket	FCLGA1151	FCLGA1155
Fecha de lanzamiento	4º trimestre de 2018	2º trimestre de 2012
Cache	16 MB	8 MB
Set de instrucciones	SSE4.1, SSE4.2, AVX2	SSE4.1/4.2, AVX
Litografía	14 nm	22 nm
Velocidad de bus	8 GT/s DMI3	5 GT/s DMI
Solución térmica	PCG 2015D (130W)	2011D
Máximo tamaño de memoria	64 GB	32 GB
Tipo de memoria	DDR4-2666	DDR3 1333/1600
Ancho de banda de memoria	41.6 GB/s	25.6 GB/s
Frecuencia base para gráficos	350 MHz	650 MHz
Frecuencia máxima para gráficos	1.20 GHz	1.15 GHz
Configuración de PCI Express	1x16, 2x8, 1x8+2x4	1x16, 2x8, 1x8 & 2x4

Fuente: UserBenchmark, 2018

Por lo tanto se justifica el uso de la Unidad de Procesamiento Gráfico para la distribución de tareas repetitivas aptas para un enfoque de ejecución en paralelo, ya que de acuerdo a la tabla comparativa anterior se observa que el paradigma de desarrollo de hardware por parte de los fabricantes es hacia un ecosistema de procesadores en paralelo en lugar de un bajo número de procesadores trabajando a frecuencias altas.

2.2. Justificación científica

En lo referente al ámbito científico, la utilidad de esta investigación radica en la profundización del estudio acerca de los métodos de ejecución de aplicaciones en múltiples hilos utilizando la Unidad de Procesamiento Gráfico (GPU), con la finalidad de llegar a utilizar las mallas de procesadores disponibles en las tarjetas gráficas, este estudio a la fecha de realización de la presente investigación, aún no se ha profundizado, ni es de aplicación general.

Por tanto, al concluir esta investigación, se habrán implantado los conocimientos necesarios para lograr aplicar métodos de ejecución de tareas en paralelo en la GPU utilizados en otros métodos de encriptación y otros procesos no necesariamente relacionados con criptografía.

Capítulo 3
Problema de investigación

En este capítulo se muestra en detalle la problemática de la investigación.

3.1. Determinación del problema

En el contexto de la computación, los procesos a ejecutarse ingresan a una cola de espera que dependiendo del trabajo del procesador, estos procesos pueden tardar un tiempo considerable en ser atendidos, e inclusive desecharse por la excesiva carga de trabajo de la CPU. En las comunicaciones encriptadas, dada la lógica inherente de los procesos de encriptación, los tiempos de ejecución pueden llegar a incrementarse hasta un punto insostenible para un solo procesador. Por lo cual los fabricantes vieron necesario el incremento del número de procesadores a fin de atender los procesos en la cola y evitar desechar tareas por los tiempos de ejecución.

Desde otro punto de vista, se cuenta con otra alternativa de procesamiento descentralizado mediante la delegación de trabajos a la GPU. Por tanto se verificará la reducción del tiempo de ejecución del algoritmo de encriptación AES Rijndael mediante la delegación de procesos a la GPU haciendo uso del entorno de programación Python, que cuenta con compatibilidad nativa para la ejecución de tareas en matrices de procesadores CUDA mediante la librería Numba[1].

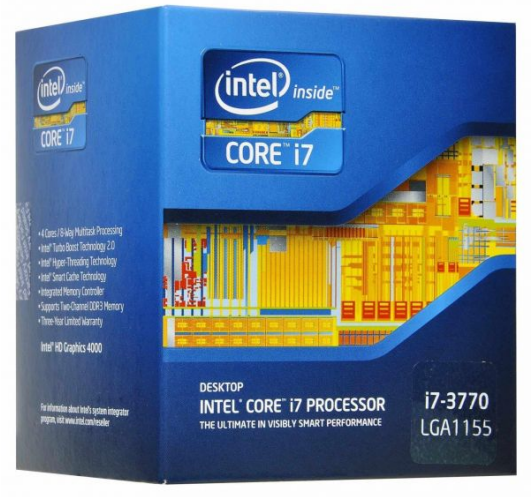
[1] NumPy+Mamba=Numba: Array-oriented Python Compiler for NumPy

3.2. Límites y alcances

El algoritmo implementado y modificado para la obtención de resultados en CPU y GPU, cumple con la definición de la Publicación de Estándares de Procesamiento de Información Federal 197[2]. [Information, 2001]. Dicha publicación fué aprobada por el Instituto Nacional de Estándares y Tecnología[3] después de la verificación en la Reforma de Administración de Tecnologías de la Información[4] de 1996.

El relleno de datos de 128 bits cumple con el Estándar de Criptografía de Llave Pública PKCS #7 plasmado en el reporte RFC 2315 [Kaliski, 1998].

Figura 1: Microprocesador Intel i7-3770



Fuente: Intel® Core™ i7-3770 Processor, ark.intel.com

En cuanto al hardware, se realiza la comparación de tiempo de ejecución en el microprocesador Intel i7-3770[5] con respecto a la tarjeta gráfica NVidia GTX 650 Ti[6]. Lo que representa una comparativa de trabajo simultáneo de 8 núcleos trabajando a 3.4GHz versus 768 núcleos trabajando a 928MHz para las operaciones pasibles a paralelismo.

[2] Federal Information Processing Standards Publications - FIPS 197
[3] National Institute of Standards and Technology - NIST
[4] Information Technology Management Reform
[5] Intel® Core™ i7-3770 Processor, ark.intel.com
[6] NVidia GeForce GTX-650Ti, www.geforce.com

El algoritmo utilizado modificado para esta investigación fué escrito por Pablo Caro y es de código abierto, se puede encontrar en la plataforma Github[7].

Figura 2: Tárjeta Gráfica NVidia GeForce GTX-650Ti

Fuente: NVidia GeForce GTX-650Ti, www.geforce.com

Se realizaron modificaciones a este algoritmo mediante la librería Numba que cuenta con decoradores predefinidos que compilan el código a ser paralelizado previa ejecución del programa. Esta librería genera kernels compatibles con plataformas CUDA de distintas arquitecturas.

El sistema operativo utilizado para la investigación es Arch Linux con el Kernel versión 4.18.10.

El driver de NVidia que se utilizó es de la versión 410.57-2 con el manejador de procesadores CUDA versión 10.0.130-2.

La versión de Python utilizada fué 3.6.7, con las librerías externas Numba v0.41 y Numpy v1.15.1.

[7] Python-AES, Pablo Caro

Capítulo 4
Objetivos

En este capítulo se detalla el objetivo principal y los objetivos específicos.

4.1. Objetivo general

Demostrar el incremento de velocidad para la ejecución del algoritmo AES Rijndael en la Unidad de Procesamiento Gráfico (GPU) con respecto a la ejecución del algoritmo en la Unidad Central de Procesamiento (CPU).

4.2. Objetivos específicos

- Paralelizar los procesos del algoritmo AES Rijndael haciendo uso del entorno Python enfocado hacia la tecnología de malla de procesadores CUDA
- Mediante la implementación de un programa destinado a la ejecución en la GPU, obtener la diferencia de tiempo de ejecución del algoritmo AES utilizando solo la CPU con respecto a la ejecución del algoritmo con la asistencia de la GPU
- Liberar a la CPU de trabajo computacional para de forma, ejecutar otros procesos en la cola designando tareas pasibles a paralelización y cálculos matriciales a la GPU

Capítulo 5
Marco teórico

En este capítulo se muestran los conceptos referentes a computación paralela en CPU y GPU, definición y estructura del algoritmo AES Rijndael, operaciones utilizadas para el cifrado y descifrado, muestras y resultados obtenidos de la ejecución del algoritmo AES Rijndael en CPU y GPU.

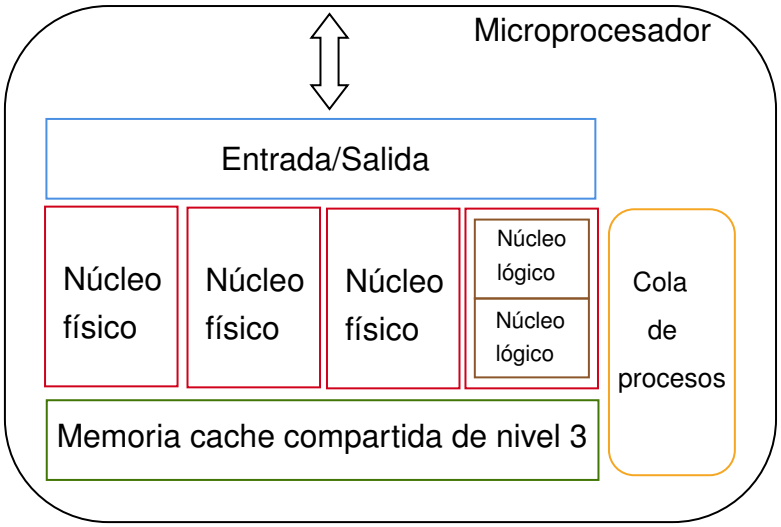
5.1. Antecedentes

5.1.1. Computación paralela

La computación paralela es una rama de la informática que se encarga del estudio de la ejecución de una tarea dividida en sub-procesos o varias tareas independientes de forma simultánea en forma de hilos de ejecución en un grupo de procesadores llamados también procesadores multinúcleo, que luego de realizar dichas tareas sincronizan sus resultados a fin de mantener la integridad de los datos.

Los microprocesadores actuales contienen comúnmente dos tipos de núcleos, los núcleos físicos y los núcleos lógicos. Cada zócalo de una tarjeta madre contiene un microprocesador, este contiene uno o más núcleos físicos; un núcleo físico es aquel que se encuentra físicamente dentro del circuito integrado del microprocesador, mientras que un núcleo lógico es una división virtual en 2 o más partes de un núcleo físico. Las tareas son asignadas a los núcleos físicos; estas tareas pueden dividirse en tareas más pequeñas a fin de resolver un gran problema en partes pequeñas que al final serán unidas para generar la solución, estas partes pequeñas son llamadas “hilos” y son las que se ejecutan en los núcleo lógicos.

Figura 3: Disposición de un microprocesadores multinúcleo



Fuente: Elaboración propia

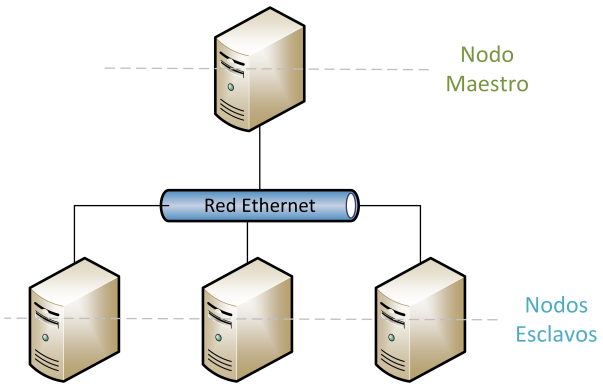
Las técnicas principales para lograr estas mejoras de rendimiento (mayor frecuencia de reloj y arquitecturas cada vez más inteligentes y complejas) están golpeando la llamada "Power Wall". La industria informática ha aceptado que los futuros aumentos en rendimiento deben provenir en gran parte del incremento del número de procesadores (o núcleos) en una matriz, en vez de hacer más rápido un solo núcleo. [Adve y col., 2008, p. 6]

El incremento de la frecuencia en los microprocesador acarrea consigo el consumo de energía y la disminución del espacio entre los transistores dentro de cada núcleo, lo que provoca un incremento considerable de la temperatura dentro del microprocesador; por tanto, para mantener el microprocesador en funcionamiento evitando su deterioro por las temperaturas elevadas es necesario buscar fuentes más óptimas de enfriado como los tubos de conducción de gas o líquido, que incrementan aún más el consumo de energía y que son costosos para una PC de escritorio.

$$T_m = T_a \cdot [(1 - F_m) + \frac{F_m}{A_m}] \tag{5.1}$$

Donde:

Figura 4: Clúster de alto rendimiento



Fuente: Jimenez y Medina, 2014, p. 2

$F_m =$ Fracción de tiempo que el sistema utiliza el subsistema mejorado

$A_m =$ Factor de mejora que se ha introducido en el subsistema mejorado

$T_a =$ Tiempo de ejecución antiguo

$T_m =$ Tiempo de ejecución mejorado

Por tales motivos Gene Amdahl formuló la ecuación 5.1 que establece que:

La mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente

Despejando la ecuación 5.1 se obtiene la aceleración del programa completo una vez que se haya paralelizado uno o más algoritmos del programa.

$$A = \frac{1}{(1 - F_m) + \frac{F_m}{A_m}} \tag{5.2}$$

Donde:

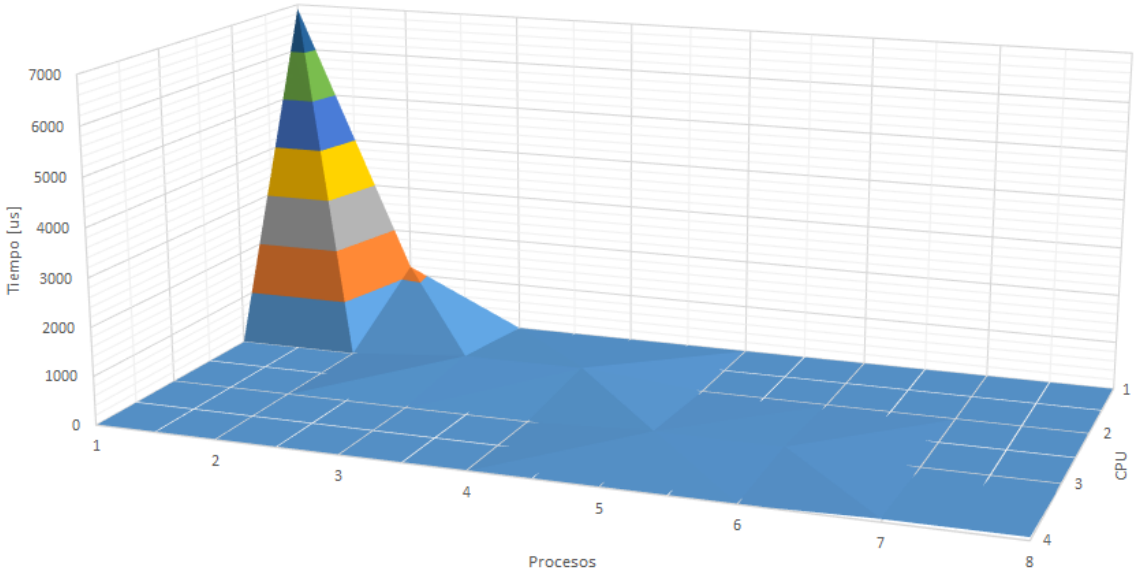
$A =$ Aceleración o ganancia en velocidad conseguida en el sistema completo debido a la mejora de uno de sus subsistemas

$A_m =$ Factor de mejora que se ha introducido en el subsistema mejorado

$F_m =$ Fracción de tiempo que el sistema utiliza el subsistema mejorado

En base a este principio se desarrollaron tecnologías de matrices de núcleos de cómputo tomando como elementos principales a los procesadores existentes y acomodándolos de tal forma que se pueda administrar la ejecución de tareas en cada procesador de manera individual y la sincronización de resultados al final del proceso. Estos arreglos reciben el nombre de clústers. Los clústers de alto rendimiento son un tipo de clústers utilizados con el propósito de ejecutar tareas exhaustivas divididas en tareas pequeñas ejecutadas en cada computador de acuerdo a la gestión realizada por el llamado nodo maestro. [Jimenez y Medina, 2014]

Figura 5: Comparación de tiempos de proceso en múltiples CPUs



Fuente: Jimenez y Medina, 2014, p. 7

5.1.2. Unidad de procesamiento gráfico (GPU)

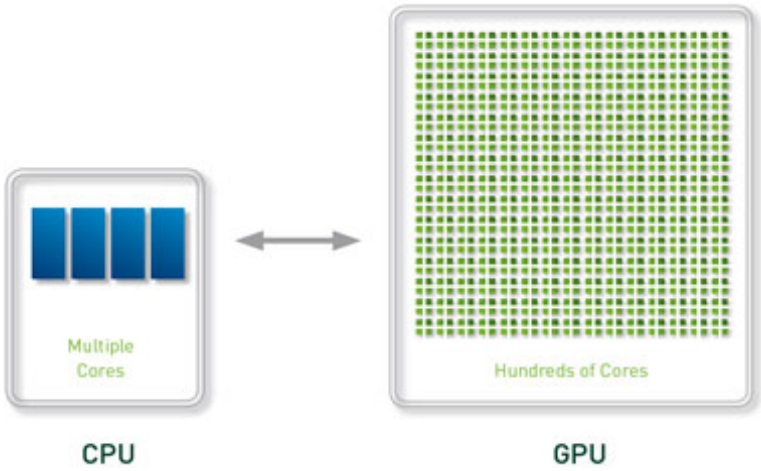
Esta unidad actúa como un co-procesador que se encarga de las operaciones matriciales o de coma flotante, por lo general los procesos gráficos de transformación o renderización son distribuidos a la o las GPUs desde el procesador central o CPU.

Dado el estudio generado sobre las plataformas GPU, los fabricantes pusieron a

disposición de los usuarios herramientas de desarrollo para utilizar las GPU como ayuda en cálculos de álgebra dispersa, tensores en dinámica de fluidos, minería de datos, inteligencia artificial, deep learning, etc, con lo cual la denominación de las GPU abiertas a otro tipo de uso más que el simple uso gráfico cambió a GPGPU[1].

Estas tarjetas están desarrolladas en base al paralelismo de núcleos de frecuencia baja con un esquema de operaciones limitado.

Figura 6: Cantidad de núcleos en CPU vs GPU



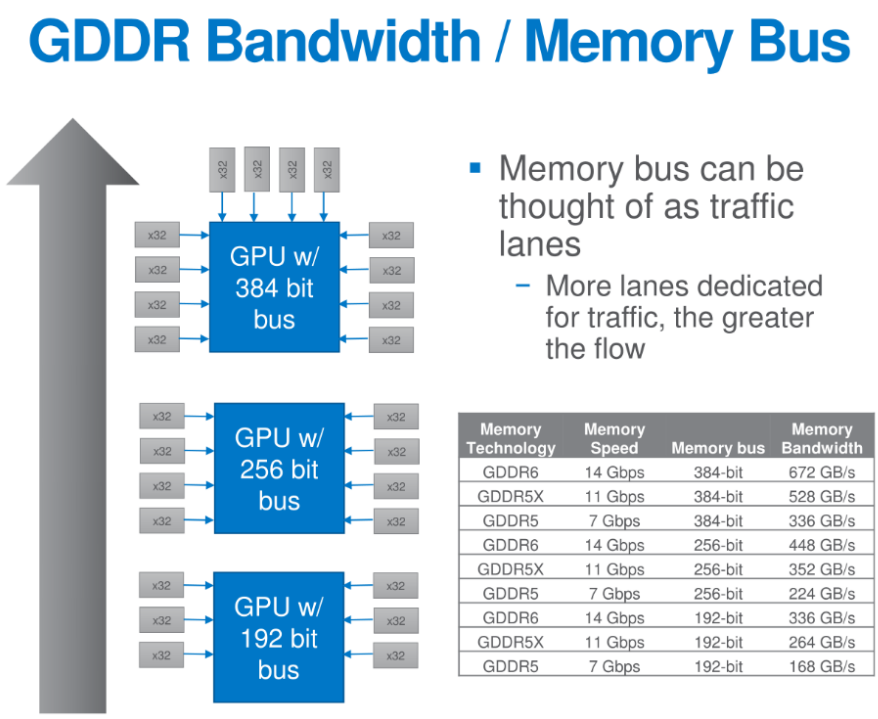
Fuente: SuperComputing Applications and Innovation, 2012

El obstáculo principal para el desarrollo de aplicaciones orientadas hacia la GPU es que las arquitecturas de las tarjetas gráficas son demasiado variables, a pesar de la existencia de librerías o APIs genéricas como OpenGL, muchas funcionalidades dentro de los métodos o clases son variables entre fabricantes e incluso entre modelos de dispositivos de un mismo fabricante. Las librerías genéricas utilizan un núcleo basado en el esquema de Conductos de Renderización[2], con los que se pueden tratar vectores, mapas de bits y elementos definidos pixel-pixel.

[1] Unidad de Procesamiento Gráfico de Uso General (General Purpose Graphics Processing Unit)
[2] Rendering Pipeline

Otro factor importante que impide hacer un uso adecuado de estos dispositivos es el límite físico con el que actualmente cuenta la conexión de memoria RAM de la GPU con el bus de la CPU para la transferencia de datos. Al cuarto trimestre de 2018 ya se cuenta con la tecnología GDDR6[3] que ofrece un ancho de banda de hasta 16Gbps frente a los 10Gbps de su predecesor GDDR5X, cabe mencionar que se lograron estos anchos de banda gracias al cambio de modo half-duplex o transferencia en ambos sentidos pero solo uno a la vez, por el modo full-duplex que transfiere los datos en ambos sentidos al mismo tiempo.

Figura 7: Comparación de tecnologías GDDR6 vs GDDR5



Memory Technology	Memory Speed	Memory bus	Memory Bandwidth
GDDR6	14 Gbps	384-bit	672 GB/s
GDDR5X	11 Gbps	384-bit	528 GB/s
GDDR5	7 Gbps	384-bit	336 GB/s
GDDR6	14 Gbps	256-bit	448 GB/s
GDDR5X	11 Gbps	256-bit	352 GB/s
GDDR5	7 Gbps	256-bit	224 GB/s
GDDR6	14 Gbps	192-bit	336 GB/s
GDDR5X	11 Gbps	192-bit	264 GB/s
GDDR5	7 Gbps	192-bit	168 GB/s

Fuente: ExtremeTech, 2018

Pero AMD ya se encuentra desarrollando tarjetas madres con conectores PCI-E[4] 5.0 que incrementarán la velocidad de transferencia hasta los 32Gbps que conjuntamente con el almacenamiento SSD[5] lograrán impulsar el desarrollo de aplicaciones de uso general en las GPUs.

[3] Tasa Doble de transferencia de Datos (Double Data Rate)
[4] Componente Periférico de Interconexión Expresa (Peripheral Component Interconnect Express)
[5] Solid State Drive

Cuadro 3: Comparación de tecnologías PCI-E

	RAW Bitrate	**Link BW**	**BW/Lane/Way**	**Total BW X16**
PCIe 1.x	2.5 GT/s	2 Gb/s	250 MB/s	8 GB/s
PCIe 2.x	5.0 GT/s	4 Gb/s	500 MB/s	16 GB/s
PCIe 3.x	8.0 GT/s	8 Gb/s	$\sim$1 GB/s	$\sim$32 GB/s
PCIe 4.x	16 GT/s	16 Gb/s	$\sim$2 GB/s	$\sim$64 GB/s
PCIe 5.x	32 GT/s	32 Gb/s	$\sim$4 GB/s	$\sim$128 GB/s

Fuente: Smith, 2018

5.2. Algoritmo Estándar de Encriptación Avanzada Rijndael

El Estándar de Encriptación Avanzada fue de desarrollado mediante un concurso en 1997, por los criptógrafos Vincent Rijmen e Joan Daemen en el año 2001, como la sustitución al algoritmo DES[6] que había sido crackeado mediante la máquina DES Cracker construida por la ONG Electronic Frontier Foundation, con una inversión de 250 mil dólares. Este estándar fue aprobado y es utilizado por entes reguladores como la NSA[7] y se estandariza mediante la norma ISO/IEC 18033 [ISO/IEC JTC 1, Information technology, Subcommittee SC 27, Security techniques, 2015].

El algoritmo AES Rijndael es un algoritmo de llave simétrica, lo cual indica que se utiliza una misma llave para cifrar y descifrar los mensajes en el lado del emisor y del receptor. Por tal razón, toda la seguridad recae en proteger la clave secreta, por tal razón el abanico de claves posibles debe ser de una cantidad tan grande que el intruso deba realizar pruebas, por inclusive años, para poder descifrar el mensaje. Para el caso de DES, la clave es de 56 bits por lo que la cantidad de claves será igual a: $2^{56} = 7{,}2 \times 10^{16}$ posibles claves; un computador actual puede lograr descifrar un mensaje mediante el cálculo de la llave secreta en un tiempo

[6] Estándar de Encriptación de Datos(Data Encryption Standard)
[7] Agencia de Seguridad Nacional(National Security Agency)

de tan solo segundos.

Cuadro 4: Comparación de tecnologías PCI-E

Tamaño de Clave	Combinaciones Posibles
1 bit	2
2 bit	4
4 bit	16
8 bit	256
16 bit	65536
32 bit	$4{,}2 \times 10^9$
56 bit (DES)	$7{,}2 \times 10^{16}$
64 bit	$1{,}8 \times 10^{19}$
128 bit (AES)	$3{,}4 \times 10^{38}$
192 bit (AES)	$6{,}2 \times 10^{57}$
256 bit (AES)	$1{,}1 \times 10^{77}$

Fuente: DataQUBO, 2013

El algoritmo AES Rijndael trabaja con mensajes divididos en bloques de 128 bits y llaves de logitud de 128, 192 y 256 bits. Por lo tanto con una llave de 128 bits el atacante necesitaría generar: $2^{56} = 3{,}4 \times 10^{38}$ llaves, tarea que en la actualidad, aún con computadoras tan potentes, el trabajo tardaría millones de años.

Suponiendo la super-computadora Summit de IBM [IBM, 2018], designada para descifrar un mensaje, trabajando a $143{,}5PFlops$[8] o $143{,}5 \times 10^{15}Flops$ y sabiendo que la cantidad de segundos en un año es de: $365x24x60x60 = 31536000$. Se calcula la cantidad de años necesarios para crackear AES con una longitud de clave de 128 bits.

[8] Operaciones de Punto Flotante por Segundo(Floating point Operations Per Second)

$$
\begin{aligned}
t &= \frac{3{,}4 \times 10^{38}}{143{,}5 \times 10^{15} x 31536000} \\
t &= \frac{23{,}69 \times 10^{15}}{315{,}36} \\
t &= 75{,}13 \times 10^{12} años
\end{aligned}
\tag{5.3}
$$

Es decir, con la última tecnología disponible actualmente se tomaría un tiempo de 75.13 billones de años en generar las llaves secretas necesarias para descifrar un mensaje. Suponiendo que solo fuese necesario generar la mitad de las llaves para encontrar la correcta, el proceso tardaría mas de 32 billones de años. Por lo tanto una llave secreta de 128 bits utilizada para cifrar un mensaje con el algoritmo AES Rijndael es suficiente seguridad para la actualidad y para unos años más en el futuro.

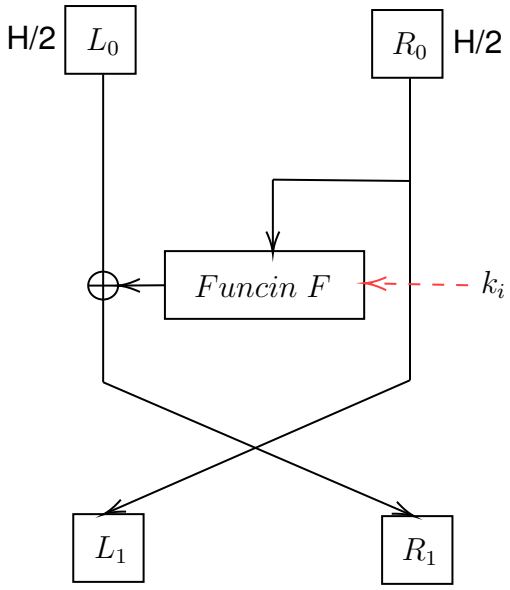
Este algoritmo fue seleccionado ganador de entre muchos otros participantes del concurso de 1997 considerando sus tres axiomas importantes:

- Resistencia contra todos los ataques conocidos hasta la fecha.
- Compatibilidad con un gran número de plataformas de hardware.
- Velocidad y sencillez en el diseño.

5.2.1. Estructura del algoritmo

Los cifradores simétricos están desarrollados con una estructura de tipo Feistel, esta estructura tiene dos partes, la izquierda y la derecha.

Figura 8: Bloque de información H para el modelo de cifrador Feistel



Fuente: Muñoz, 2004

La diferencia es que Rijndael define las vueltas de transformaciones en tres funciones invertibles llamadas capas:

- La mezcla lineal garantiza alta difusión de la información dado el número de vueltas.
- La capa no lineal hace que el comportamiento del sistema no pueda ser representado como la suma de los comportamientos de sus sub-sistemas.
- La capa de adición de clave crea estados intermedios para hacer difusa la matriz de estado.

Este algoritmo es conformado por rondas en las que se ejecutan 4 funciones matemáticas en un orden establecido. El resultado de cada ronda es llamado *Estado* que es una matriz de 4 filas por N_b columnas, donde:

$$N_b = \frac{Tamaño\ de\ bloque\ utilizado\ en\ bits}{32} \tag{5.4}$$

De manera similar, la clave inicial se representa mediante una matriz de 4 filas y N_k columnas, donde:

$$N_k = \frac{Tamaño\ de\ la\ clave\ en\ bits}{32} \tag{5.5}$$

Las matrices se acomodan de tal forma que cada palabra (4 bytes = 32 bits) es representada en una columna de izquierda a derecha.

Por ejemplo la frase: “mensaje secreto.” se convierte de ASCII a su representación hexadecimal, cuyo resultado es:

6d 65 6e 73 61 6a 65 20 73 65 63 72 65 74 6f 2e

Que se acomoda en una matriz de estado como se muestra a continuación:

6d	61	73	65
65	6a	65	74
6e	65	63	6f
73	20	72	2e

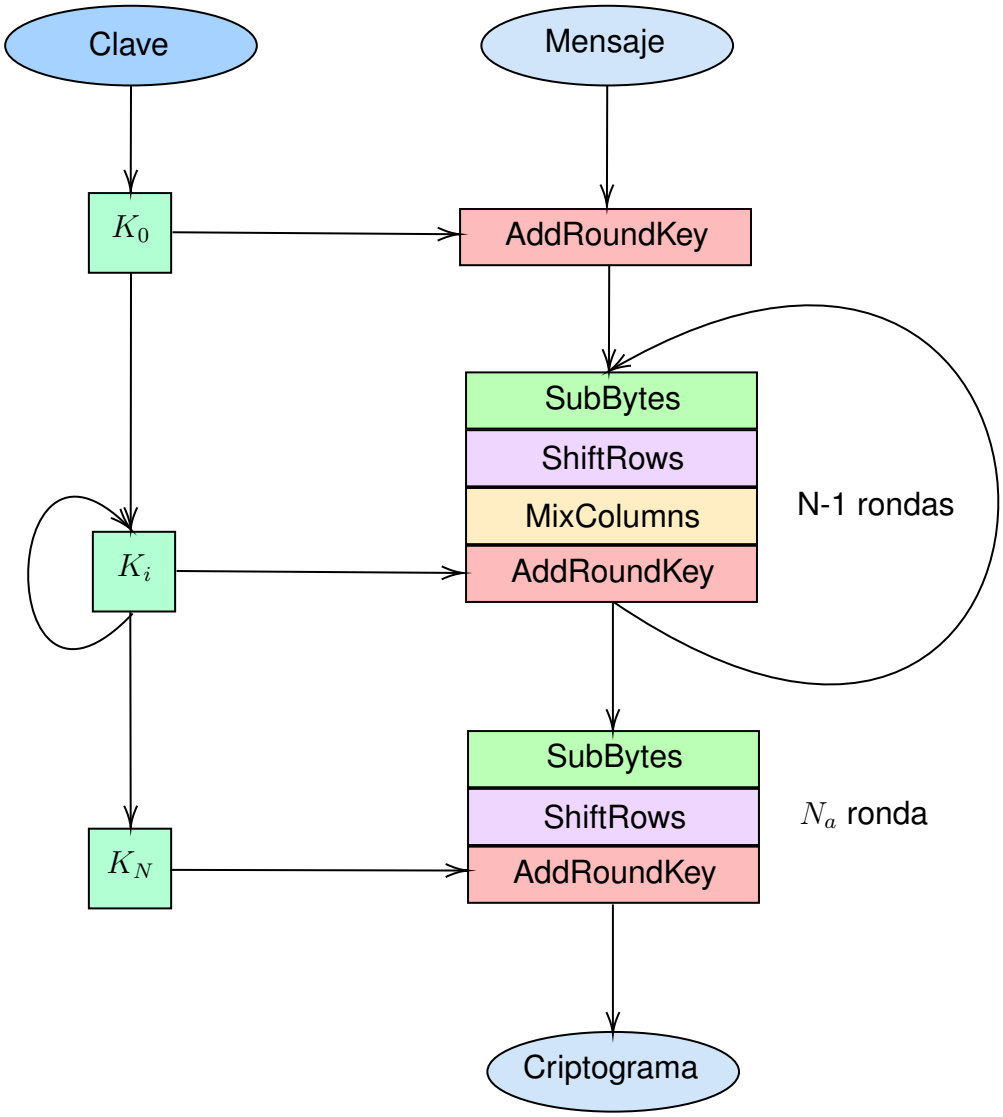
Cabe recalcar que el mensaje es de una longitud de 128 bits; en caso de no ser así, el mensaje es dividido en paquetes de 128 bits y se acomoda un relleno o padding de acuerdo a la norma PKCS#7 en el caso de AES Rijndael.

De forma análoga, se realiza el mismo proceso para la clave secreta inicial que puede ser de 128, 192 o 256 bits. El número de rondas para llegar a calcular el criptograma varía de acuerdo al tamaño de la clave:

- Para una clave de cifrado de 128 bits el algoritmo realiza 10 rondas
- Para una clave de cifrado de 192 bits el algoritmo realiza 12 rondas
- Para una clave de cifrado de 256 bits el algoritmo realiza 14 ronda

5.2.1.1. Operaciones para el proceso de cifrado

Figura 9: Algoritmo AES Rijndael



Fuente: Elaboración propia

Donde:

N: Número de rondas

i: Ronda actual

N_a**:** Enésima ronda

1. **SubBytes**

   Aporta la propiedad de no linealidad, lo que evita ataques de interpolación, criptoanálisis diferencial y criptoanálisis lineal.

   Las propiedades de esta operación son:

   - Es invertible.
   - Minimiza la relación lineal entre la entrada y la salida.
   - Incrementa la complejidad por sus expresiones en el Campo de Galois en GF(2^8).
   - Sencillez de diseño.

   Es la sustitución byte a byte de la matriz de estado mediante la tabla S-Box que se construye mediante dos transformaciones consecutivas. Cada byte se considera un elemento del Campo de Galois $GF(2^8)$ de cuerpos finitos; esto quiere decir que para cada valor existe un inverso aditivo y multiplicativo que elimina los problemas de redondeo y desbordamiento. Cabe mencionar que en la matemática modular se trabaja únicamente con números primos. Por tanto cada byte genera un polinomio irreducible $m(x) = x^8+x^4+x^3+x+1$, que es sustituido por su inverso multiplicativo. Una vez realizada la primera transformación se aplica la transformación afín en GF(2):

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

   De acuerdo a esta lógica se calcula la tabla S-Box para cada valor de los

256 que pueden componer un byte (8 bits).

Cuadro 5: Tabla S-Box

HEX		y															
		00	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0A**	**0B**	**0C**	**0D**	**0E**	**0F**
x	**00**	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	10	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	20	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	30	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	40	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	50	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	60	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	70	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	80	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	90	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	A0	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B0	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	C0	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	D0	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E0	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F0	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Fuente: Muñoz, 2004

Por ejemplo la transformación SubBytes sobre el hexadecimal A2:

$$SubBytes(A2) = 3A$$

2. ShiftRows

En esta transformación se rotan hacia la izquierda las filas de la matriz de estado de manera incremental; es decir, la primera fila no rota, la segunda rota 1 vez, la tercera 2 veces y la cuarta 3 veces. El número de rotaciones se mantiene constante para la matriz de estado conformada por el mensaje pero varía para la clave ya que esta puede ser de 128, 192 o 256 bits como se mencionó anteriormente, para ello se aplica la siguiente regla, tomando C_0 como la primera fila de la matriz de estado, C_1 como la segunda, C_2 como la tercera u C_3 como la cuarta, se mantiene C_0 sin rotar, las demás filas rotan de acuerdo a la tabla 6.

Cuadro 6: Rotaciones de las filas de la matriz de estado

Tamaño de bloque	**C1**	**C2**	**C3**
128 bits ($N_b = 4$)	1	2	3
192 bits ($N_b = 6$)	1	2	3
256 bits ($N_b = 8$)	1	3	4

Fuente: Muñoz, 2004

Los números de rotaciones fueron elegidos con la finalidad de evitar los ataques de truncated differentials y de square.

Un ejemplo gráfico de la rotación de la matriz de estado se puede observar en la table 7.

Cuadro 7: ShiftRows

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$		$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$	=>	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$	$S_{1,0}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$		$S_{2,2}$	$S_{2,3}$	$S_{2,0}$	$S_{2,1}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$		$S_{3,3}$	$S_{3,0}$	$S_{3,1}$	$S_{3,2}$

Fuente: Muñoz, 2004

Como ejemplo, a continuación se realiza la transformación ShiftRows sobre una matriz de estado:

C3	67	92	67	Rota 0 Bytes	C3	67	92	67
0C	00	CB	3B	Rota 1 Byte	00	CB	3B	0C
D7	6B	4C	A0	Rota 2 Bytes	4C	A0	D7	6B
C9	6E	29	1A	Rota 3 Bytes	1A	C9	6E	29

3. **MixColumns**

Esta operación cuenta con las siguientes propiedades:

- Es invertible.

- Es lineal en el Campo de Galois GF(2).
- Incluye un grado de difusión en la matriz de estado.
- Compatibilidad y velocidad con procesadores de 8 bits.
- Sencillez de diseño.

En esta transformación se considera a las columnas de la matriz de estado como polinomios con coeficientes pertenecientes al Campo de Galois GF(2^8); es decir, estos son también polinomios. La transformación se la realiza multiplicando cada columna de la matriz de estado en módulo $x^4 + 1$ por el polinomio irreducible de Rijndael $m(x) = x^8 + x^4 + x^3 + x + 1$.

Cuadro 8: Polinomio irreducible de Rijndael

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

Fuente: Muñoz, 2004

Por ejemplo la aplicación de la transformación MixColumns sobre la primera columna de la matriz de estado calculada en el paso anterior llega a ser:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} C3 \\ 00 \\ 4C \\ 1A \end{bmatrix} = \begin{bmatrix} CB \\ 0D \\ 75 \\ 26 \end{bmatrix}$$

4. **AddRoundKey**

Esta transformación consiste en realizar la operación XOR u OR-Exclusivo entre la matriz de estado y la sub-clave de la ronda.

Por ejemplo la aplicación de esta operación entre una matriz de estado y una sub-clave llega a ser:

$$\begin{bmatrix} CB & E1 & CB & 98 \\ 0D & D8 & E8 & EB \\ 75 & B7 & AE & C6 \\ 26 & 4B & 9D & 9C \end{bmatrix} \oplus \begin{bmatrix} 9B & FE & DE & BC \\ FE & DE & EF & 86 \\ EE & 8A & B8 & CC \\ DC & B9 & 81 & F2 \end{bmatrix} = \begin{bmatrix} 50 & 1F & 15 & 24 \\ F3 & 06 & 07 & 6D \\ 9B & 3D & 16 & 0A \\ FA & F2 & 1C & 6E \end{bmatrix}$$

5.2.1.2. Expansión de clave

Esta operación permite derivar sub-claves a partir de la clave inicial de cifrado, una para cada vuelta. Mediante esta operación se evitan los ataques de criptoanálisis, ataques de comprensión de función hash y ataques por clave relacionada. Al mismo tiempo esta operación elimina la simetria en cada ronda y entre las rondas del algoritmo, con lo cual se elimina también la posibilidad de claves débiles, error conocido en algoritmos como DES o IDEA.

El número de sub-claves a calcular es igual a N, ya que la primera operación a realizar es siempre un OR exclusivo entre la clave inicial y la primera matriz de estado, en este caso del texto en claro acomodado en la matriz de dimensión 4x4.

Se toma la clave inicial como la primera sub-clave K_0, por ejemplo:

63 6C 61 76 65 20 64 65 20 31 32 38 62 69 74 73

1. **RotWord**

   Se selecciona la última columna o palabra de la sub-clave y se rota el byte superior de manera vertical.

63	65	20	62		69
6C	20	31	69	=>	74
61	64	32	74		73
76	65	38	73		62

2. **SubBytes**

   Se sustituyen los valores de acuerdo a la tabla de sustitución S-Box de Rijndael.

$$\begin{bmatrix} 69 \\ 74 \\ 73 \\ 62 \end{bmatrix} => \begin{bmatrix} F9 \\ 92 \\ 8F \\ AA \end{bmatrix}$$

3. **XOR [i-3]**

   Se realiza la operación de OR exclusivo con la columna que se encuentra 3 posiciones atrás en la matriz de estado de la sub-clave.

$$\begin{bmatrix} 63 & 65 & 20 & 62 \\ 6C & 20 & 31 & 69 \\ 61 & 64 & 32 & 74 \\ 76 & 65 & 38 & 73 \end{bmatrix} => \begin{bmatrix} F9 \\ 92 \\ 8F \\ AA \end{bmatrix} \oplus \begin{bmatrix} 63 \\ 6C \\ 61 \\ 76 \end{bmatrix} = \begin{bmatrix} 9A \\ FE \\ EE \\ DC \end{bmatrix}$$

4. **XOR [RCON]**

   En este paso se utiliza un vector constante conocido como RCON.

$$\begin{bmatrix} 01 & 02 & 04 & 08 & 10 & 20 & 40 & 80 & 1B & 36 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \end{bmatrix}$$

El último resultado se opera en OR exclusivo con la llave de la ronda "i" que se esté calculando. Por ejemplo, para la primera ronda de acuerdo al resultado del paso anterior:

$$\begin{bmatrix} 9A \\ FE \\ EE \\ DC \end{bmatrix} \oplus \begin{bmatrix} 01 \\ 00 \\ 00 \\ 00 \end{bmatrix} = \begin{bmatrix} 9B \\ FE \\ EE \\ DC \end{bmatrix}$$

Con este último paso se habrá calculado la primera palabra de la primera sub-clave. Para las otras 3 palabras tan solo se realiza una operación XOR entre el último resultado y el que ocupa tres posiciones atrás, es decir, XOR [i-3]:

63	65	20	62	=>	9B	$\oplus$	65	=	FE
6C	20	31	69		FE		20		DE
61	64	32	74		EE		64		8A
76	65	38	73		DC		65		B9

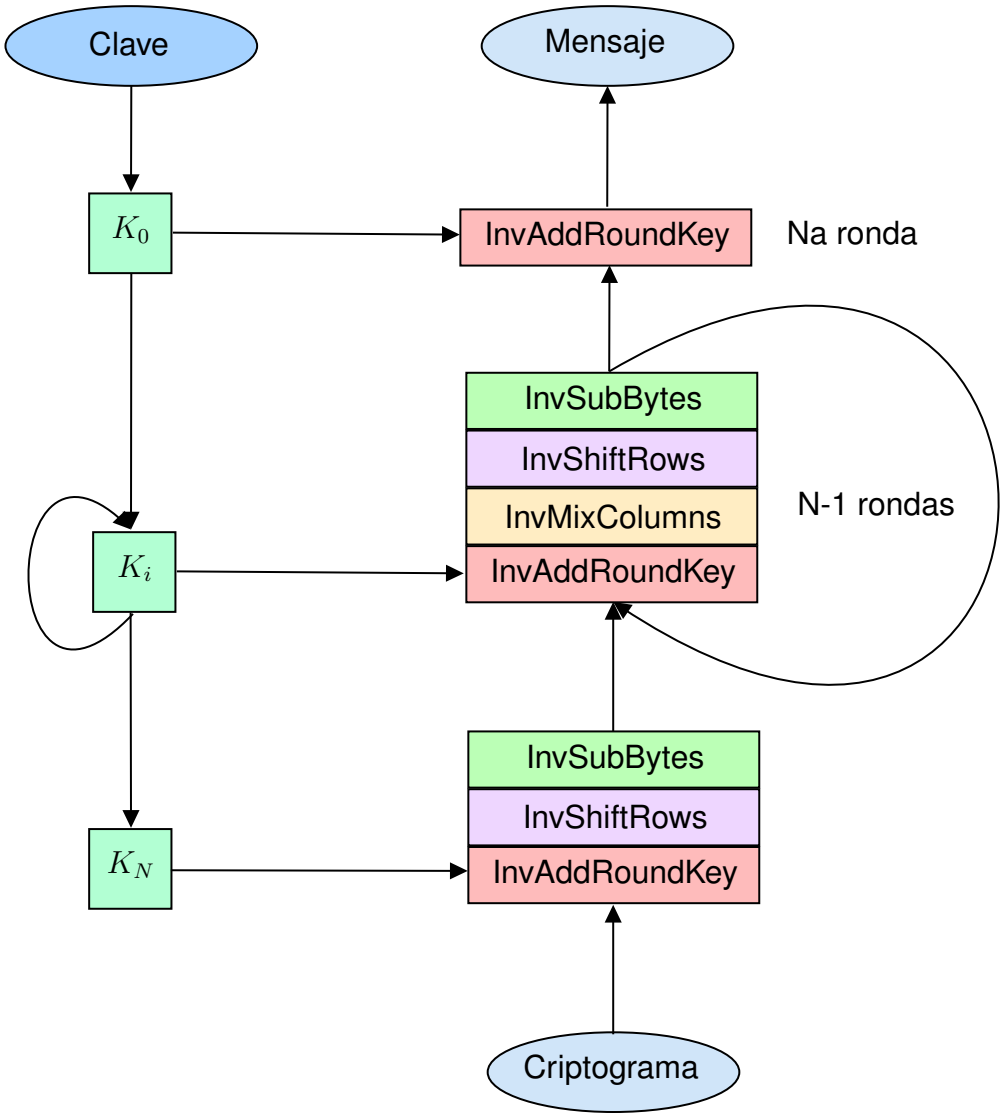
Este último paso se repite hasta formar por completo la subclave:

$$\begin{bmatrix} 9B & FE & DE & BC \\ FE & DE & EF & 86 \\ EE & 8A & B8 & CC \\ DC & B9 & 81 & F2 \end{bmatrix}$$

5.2.1.3. Operaciones para el proceso de descifrado

El proceso de descifrado es el mismo que el cifrado pero en orden contrario y con las funciones inversas:

Figura 10: Algoritmo AES Rijndael



Fuente: Elaboración propia

1. **InvSubBytes**

   Al igual que en la transformación SubBytes, se realiza la sustitución de la

matriz de estado byte a byte, pero en este caso se utiliza la tabla inversa de S-Box.

Cuadro 9: Tabla S-Box inversa

HEX		y															
		00	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0A**	**0B**	**0C**	**0D**	**0E**	**0F**
x	**00**	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
	10	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
	20	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
	30	08	2E	A1	66	28	0D	24	B2	76	5B	A2	49	6D	8B	D1	25
	40	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
	50	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
	60	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
	70	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
	80	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
	90	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6A
	A0	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
	B0	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
	C0	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
	D0	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
	E0	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
	F0	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Fuente: Muñoz, 2004

2. InvShiftRows

Es la transformación contraria a ShiftRows, por tanto consiste en rotar los bytes de la matriz de estado hacia la derecha en lugar de hacia la izquierda; es decir, en sentido contrario a ShiftRows.

C3	67	92	67	Rota 0 Bytes	C3	67	92	67
00	CB	3B	0C	Rota 1 Byte	0C	00	CB	3B
4C	A0	D7	6B	Rota 2 Bytes	D7	6B	4C	A0
1A	C9	6E	29	Rota 3 Bytes	C9	6E	29	1A

3. InvMixColumns

La transformación inversa a MixColumns se la realiza multiplicando cada columna de la matriz de estado en módulo $x^4 + 1$ por el polinomio $m'(x) = (0B)x^3 + (0D)x^2 + (09)x + 0E$.

La relación de los polinomios $m(x)$ y $m'(x)$ es tal que:

$$m(x) \otimes m'(x) = 01 \tag{5.6}$$

4. **InvAddRoundKey**

   Esta operación es la misma que AddRoundKey con la única diferencia de que se utiliza primero la última sub-clave generada en la expansión y por consiguiente se utiliza al final la clave utilizada para el cifrado.

5.3. Paralelización del algoritmo AES Rijndael

Para la ejecución del algoritmo se utilizó el script en el entorno Python que se muestra en el Anexo 1, mismo que fue desarrollado por Pablo Caro en Septiembre del año 2013.

Este algoritmo fue modificado para la ejecución de sus funciones en paralelo mediante la librería Numba que cuenta con soporte para GPU CUDA, las modificaciones se muestran en el Anexo 2.

El entorno de trabajo se detalla en la sección [3.2] referente a Límites y Alcances.

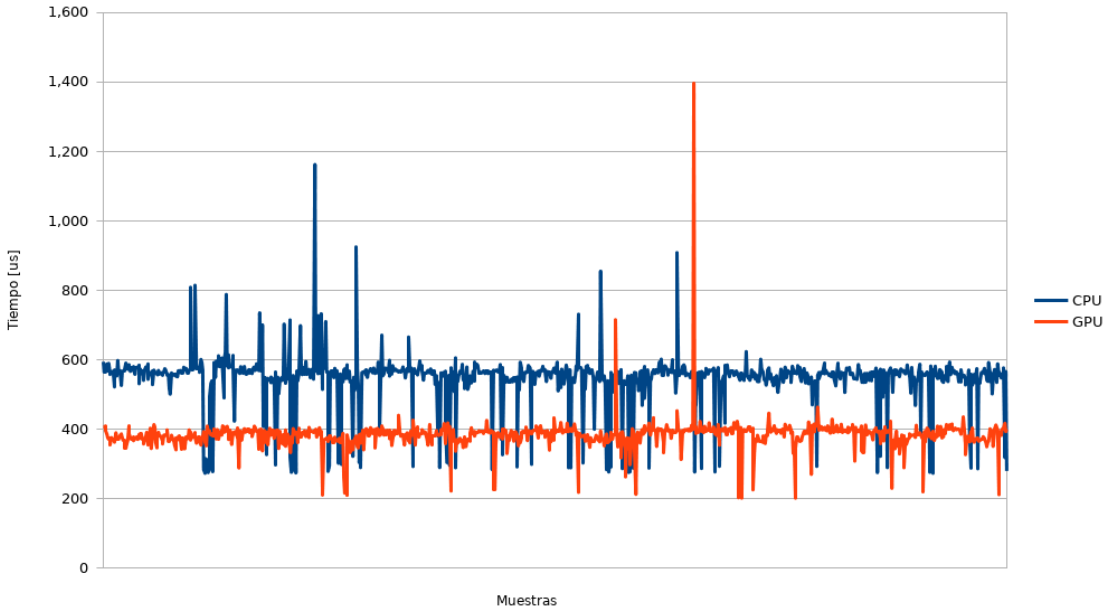
El tamaño de bloque a cifrar fue de 128 bits, lo que elimina el tiempo de ejecución de la operación de padding o el del recorte en sub-bloques.

Los tiempos de cifrado y descifrado fueron tomados de acuerdo a las longitudes de clave estándar que son: 128, 192 y 256 bits. El proceso se ejecutó un total 1000 veces por cada longitud de clave. Cada proceso generó una muestra cuyo resultado generó un promedio de la diferencia de tiempos de ejecución entre CPU y GPU.

5.3.1. Comparación de tiempos de ejecución

5.3.1.1. Cifrado con llave de 128 bits

Figura 11: Cifrado con llave de 128 bits para un total de 1000 muestras



Fuente: Elaboración propia

Tiempo promedio de ejecución del algoritmo en CPU:

$$\overline{t_{CPU}} = 545{,}33\mu s \tag{5.7}$$

Tiempo promedio de ejecución del algoritmo en CPU:

$$\overline{t_{GPU}} = 382{,}99\mu s \tag{5.8}$$

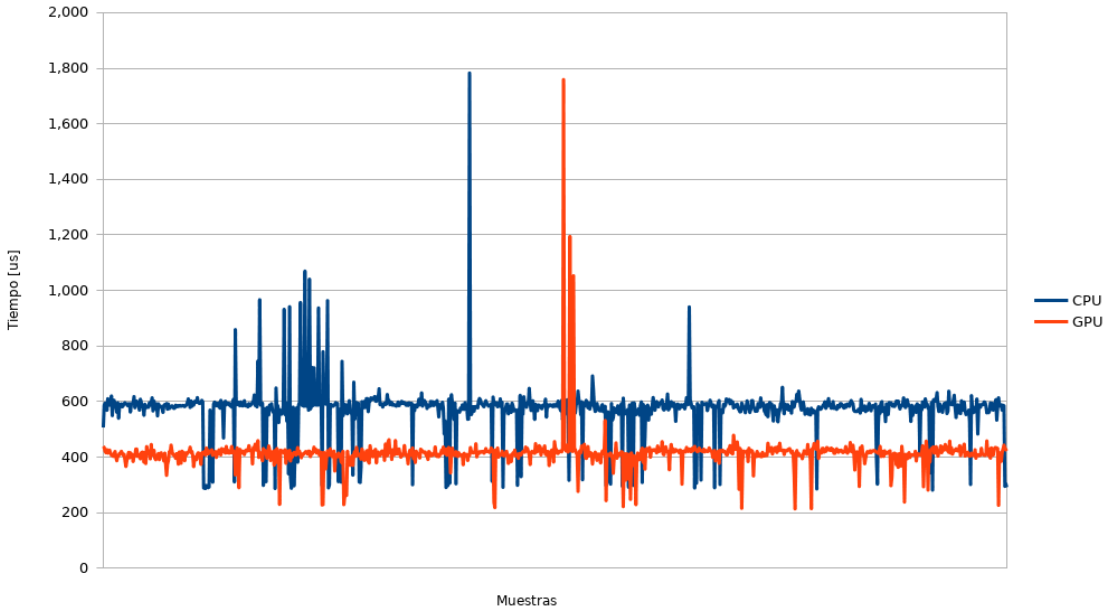
Aceleración del algoritmo ejecutado en GPU respecto a la ejecución en CPU:

$$\frac{\overline{t_{CPU}}}{\overline{t_{GPU}}} = \frac{545{,}33\mu s}{382{,}99\mu s} = 1{,}42x \tag{5.9}$$

Por tanto, la ejecución del algoritmo de cifrado en GPU es 1.42 veces más rápido que la ejecución en CPU, para un tamaño de llave de 128 bits.

5.3.1.2. Cifrado con llave de 192 bits

Figura 12: Cifrado con llave de 192 bits para un total de 1000 muestras



Fuente: Elaboración propia

Tiempo promedio de ejecución del algoritmo en CPU:

$$\overline{t_{CPU}} = 571{,}27\mu s \tag{5.10}$$

Tiempo promedio de ejecución del algoritmo en CPU:

$$\overline{t_{GPU}} = 412{,}92\mu s \tag{5.11}$$

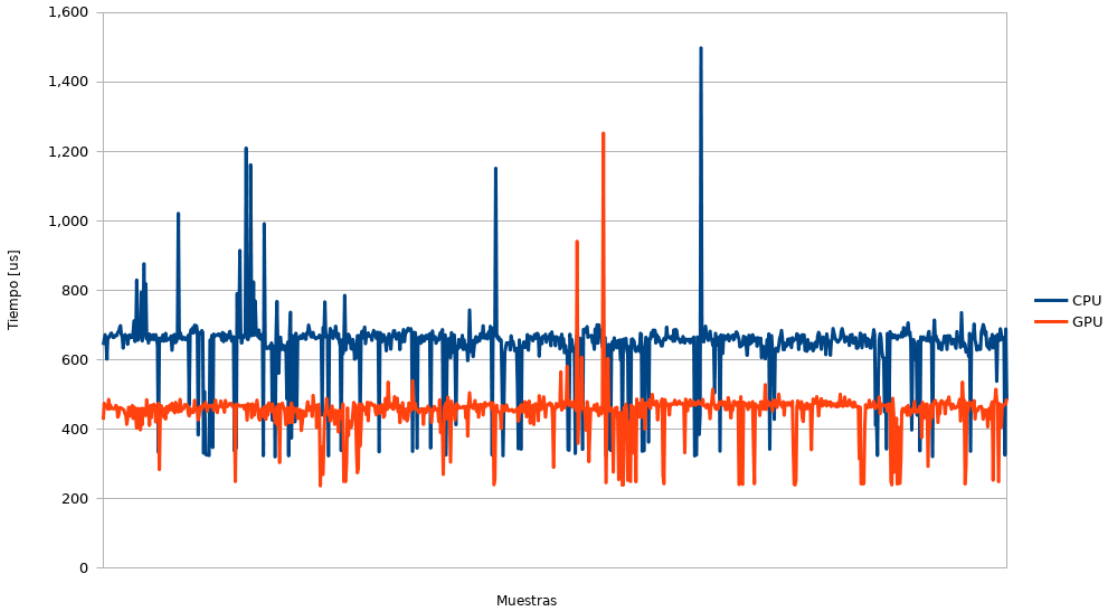
Aceleración del algoritmo ejecutado en GPU respecto a la ejecución en CPU:

$$\frac{\overline{t_{CPU}}}{\overline{t_{GPU}}} = \frac{571{,}27\mu s}{412{,}92\mu s} = 1{,}38x \tag{5.12}$$

Por tanto, la ejecución del algoritmo de cifrado en GPU es 1.38 veces más rápido que la ejecución en CPU, para un tamaño de llave de 192 bits.

5.3.1.3. Cifrado con llave de 256 bits

Figura 13: Cifrado con llave de 256 bits para un total de 1000 muestras



Fuente: Elaboración propia

Tiempo promedio de ejecución del algoritmo en CPU:

$$\overline{t_{CPU}} = 641{,}68\mu s \tag{5.13}$$

Tiempo promedio de ejecución del algoritmo en CPU:

$$\overline{t_{GPU}} = 451{,}16\mu s \tag{5.14}$$

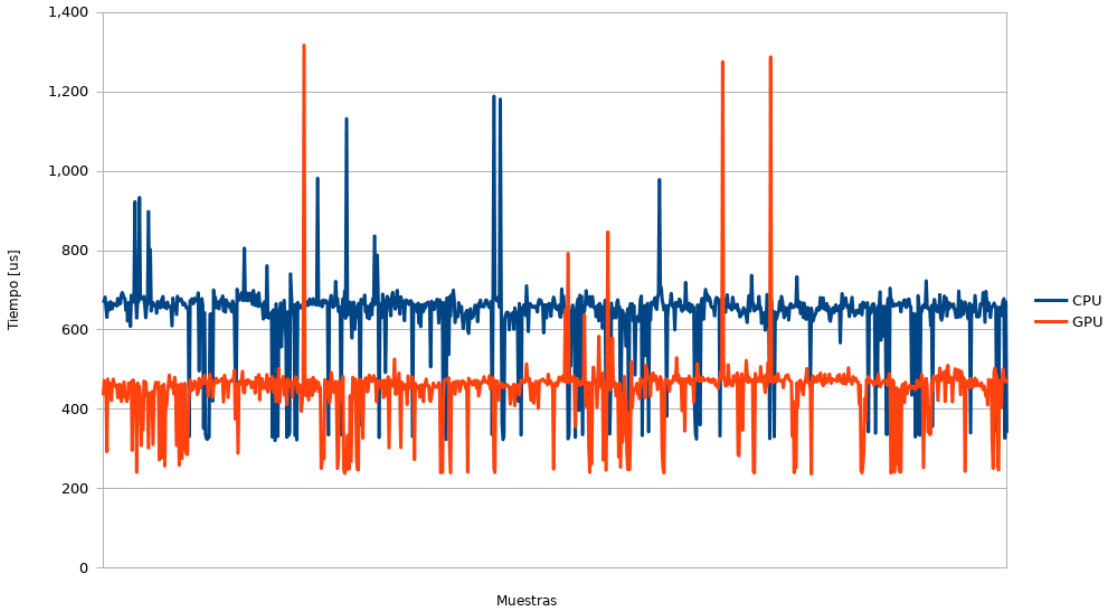
Aceleración del algoritmo ejecutado en GPU respecto a la ejecución en CPU:

$$\frac{\overline{t_{CPU}}}{\overline{t_{GPU}}} = \frac{641{,}68\mu s}{451{,}16\mu s} = 1{,}42x \tag{5.15}$$

Por tanto, la ejecución del algoritmo de cifrado en GPU es 1.42 veces más rápido que la ejecución en CPU, para un tamaño de llave de 256 bits.

5.3.1.4. Descifrado con llave de 128 bits

Figura 14: Descifrado con llave de 128 bits para un total de 1000 muestras



Fuente: Elaboración propia

Tiempo promedio de ejecución del algoritmo en CPU:

$$\overline{t_{CPU}} = 640{,}24\mu s \tag{5.16}$$

Tiempo promedio de ejecución del algoritmo en CPU:

$$\overline{t_{GPU}} = 449{,}68\mu s \tag{5.17}$$

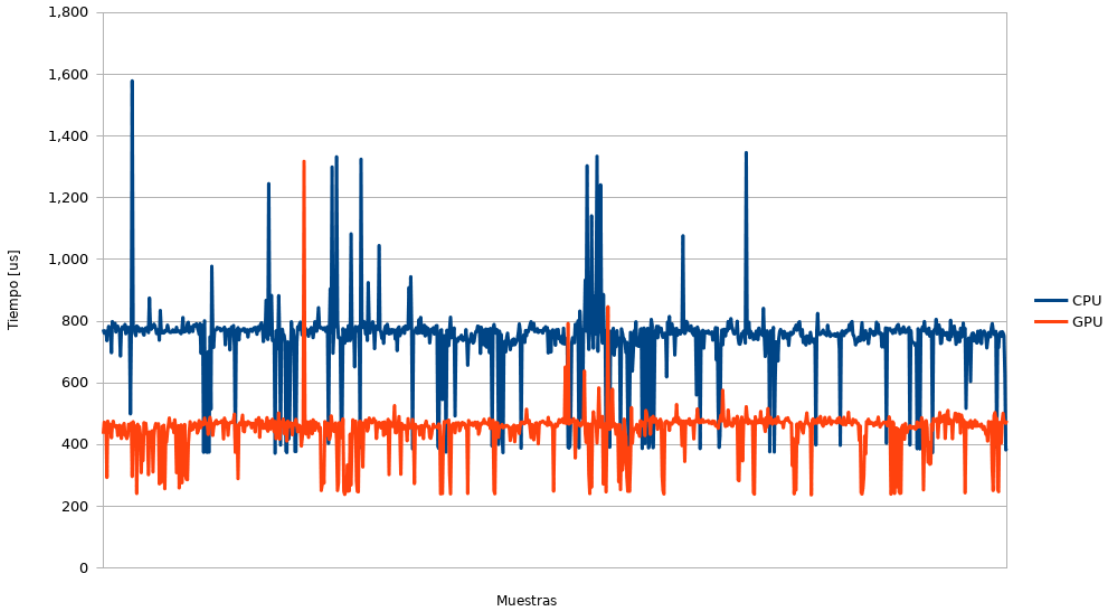
Aceleración del algoritmo ejecutado en GPU respecto a la ejecución en CPU:

$$\frac{\overline{t_{CPU}}}{\overline{t_{GPU}}} = \frac{640{,}24\mu s}{449{,}68\mu s} = 1{,}42x \tag{5.18}$$

Por tanto, la ejecución del algoritmo de descifrado en GPU es 1.42 veces más rápido que la ejecución en CPU, para un tamaño de llave de 128 bits.

5.3.1.5. Descifrado con llave de 192 bits

Figura 15: Descifrado con llave de 192 bits para un total de 1000 muestras



Fuente: Elaboración propia

Tiempo promedio de ejecución del algoritmo en CPU:

$$\overline{t_{CPU}} = 745{,}25\mu s \tag{5.19}$$

Tiempo promedio de ejecución del algoritmo en CPU:

$$\overline{t_{GPU}} = 448{,}18\mu s \tag{5.20}$$

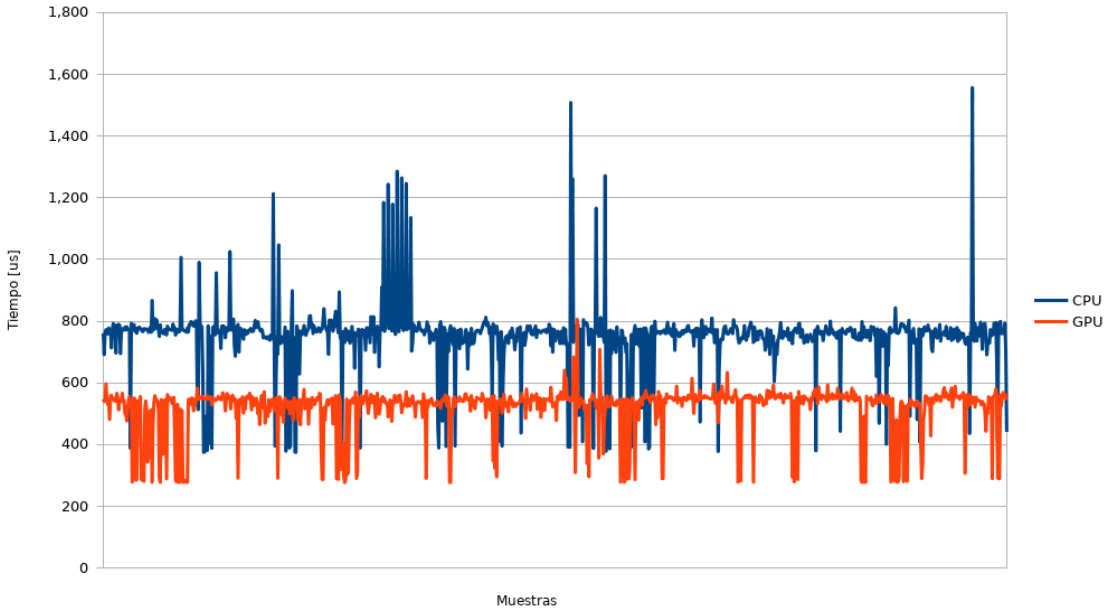
Aceleración del algoritmo ejecutado en GPU respecto a la ejecución en CPU:

$$\frac{\overline{t_{CPU}}}{\overline{t_{GPU}}} = \frac{745{,}25\mu s}{448{,}18\mu s} = 1{,}66x \tag{5.21}$$

Por tanto, la ejecución del algoritmo de descifrado en GPU es 1.66 veces más rápido que la ejecución en CPU, para un tamaño de llave de 192 bits.

5.3.1.6. Descifrado con llave de 256 bits

Figura 16: Descifrado con llave de 256 bits para un total de 1000 muestras



Fuente: Elaboración propia

Tiempo promedio de ejecución del algoritmo en CPU:

$$\overline{t_{CPU}} = 748{,}93\mu s \tag{5.22}$$

Tiempo promedio de ejecución del algoritmo en CPU:

$$\overline{t_{GPU}} = 520{,}02\mu s \tag{5.23}$$

Aceleración del algoritmo ejecutado en GPU respecto a la ejecución en CPU:

$$\frac{\overline{t_{CPU}}}{\overline{t_{GPU}}} = \frac{748{,}93\mu s}{520{,}02\mu s} = 1{,}44x \tag{5.24}$$

Por tanto, la ejecución del algoritmo de descifrado en GPU es 1.44 veces más rápido que la ejecución en CPU, para un tamaño de llave de 256 bits.

5.3.2. Cálculo de la aceleración del algoritmo ejecutado en GPU con respecto a la ejecución en CPU

De acuerdo a los resultados mostrados anteriormente, se puede calcular el promedio de aceleración del algoritmo de cifrado ejecutado en GPU con respecto a la ejecución en CPU.

$$\frac{1{,}42 + 1{,}38 + 1{,}42}{3} = 1{,}41x \tag{5.25}$$

De igual manera se puede calcular el promedio de aceleración del algoritmo de descifrado ejecutado en GPU con respecto a la ejecución en CPU.

$$\frac{1{,}42 + 1{,}66 + 1{,}44}{3} = 1{,}51x \tag{5.26}$$

Capítulo 6
Conclusiones

En este capítulo se muestran las conclusiones de acuerdo a los resultados obtenidos.

6.1. Conclusiones

- Se lograron paralelizar las funciones que contienen bucles que realizan operaciones repetitivas e independientes de cada vuelta del algoritmo AES Rijndael para las operaciones de cifrado y descifrado, con la ejecución del algoritmo en entorno Python, utilizando la malla de procesadores CUDA de que dispone la GPU Nvidia 650Ti, mediante la librería Numba y los decoradores de funciones @jit. El programa modificado se muestra en el Anexo 2.
- Con la modificación a este programa se pudieron obtener los tiempos de ejecución de las operaciones de cifrado y descifrado del algoritmo AES Rijndael para las longitudes de llave estándar de 128, 192 y 256 bits. De acuerdo a los resultados obtenidos se puede observar que la ejecución del algoritmo distribuyendo la carga de trabajo a la GPU es acelerada en 1.41 veces para la operación de cifrado y 1.51 veces para la operación de descifrado.
- Por tanto, se demostró que se puede liberar a la CPU de la carga de trabajo en la ejecución del algoritmo AES Rijndael realizando los cálculos necesarios en la GPU, y de esta forma hacer que la CPU puedad atender otros procesos hasta la obtención del resultado para cada ejecución de cifrado o descifrado. La razón por la cual actualmente este proceso todavía no es utilizado es debido a las limitaciones físicas del bus de conexión

entre la CPU y la GPU, por lo cual el tiempo es incrementado en sobremanera durante el traslado de datos de la CPU a la GPU, para implementar este proceso es necesaria otra tecnología de conexión en el bus de intercambio de datos de la CPU hacia y desde la GPU.

Bibliografía

Adve, S. V., Adve, V. S., Agha, G., Frank, M. I., Garzarán, M. J., Hart, J. C., ... Zilles, C. (2008). Parallel Computing Research at Illinois, The UPCRC Agenda. University of Illinois at Urbana-Champaign. Recuperado desde http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.471.2755&rep=rep1&type=pdf

Caro, P. (2013). Python-AES. 21 de Diciembre de 2018. Recuperado desde https://github.com/pcaro90/Python-AES

Daemen Joan, R. V. (2015). AES implementations. 10 de Diciembre de 2018. Recuperado desde https://en.wikipedia.org/wiki/AES_implementations

Daemen, J. & Rijmen, V. (2002). *The Design of RijndaeL: AES - The Advanced Encryption Standard (Information Security and Cryptography)*. Springer. Recuperado desde https://www.amazon.com/Design-RijndaeL-Encryption-Information - Cryptography / dp / 3540425802 ? SubscriptionId = AKIAIOBINVZYXZQZ2U3A & tag = chimbori05 - 20 & linkCode = xm2 & camp = 2025&creative=165953&creativeASIN=3540425802

DataQUBO. (2013). ¿Qué tan seguro es AES? 17 de Diciembre de 2018. Recuperado desde https://www.top500.org/system/179397

ExtremeTech. (2018). PCIe 5.0 Arriving in 2019 With 4x More Bandwidth Than PCIe 3.0. 16 de Diciembre de 2018. Recuperado desde https://www.extremetech.com/computing/250640-pci-sig-announces-plans-launch-pcie-5-0-2019-4x-bandwidth-pcie-3-0

IBM. (2018). Summit Super-Computer. 17 de Diciembre de 2018. Recuperado desde http://www.dataqubo.com/encriptacion-que-tan-seguro-es-aes/

Information, F. (2001). *Advanced Encryption Standard (AES)*.

ISO/IEC JTC 1, Information technology, Subcommittee SC 27, Security techniques. (2015). ISO/IEC 18033-1:2015. Recuperado desde https://www.iso.org/obp/ui/#iso:std:iso-iec:18033:-1:ed-2:v1:en

Jimenez, D. & Medina, A. (2014). Cluster de Alto Rendimiento. *Instituto de Electrónica Aplicada, Boletín Anual 2014, Universidad Mayor de San Andrés*. https://www.scribd.com/document/376915872/Cluster-Alto-Rendimiento.

Kaliski, B. (1998). *PKCS #7: Cryptographic Message Syntax Version 1.5*. RFC. http://www.rfc-editor.org/rfc/rfc2315.txt. doi:10.17487/rfc2315

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, *38*(8), 5.

Muñoz, A. M. (2004). *Seguridad Europea para EEUU. Algoritmo Criptográfico Rijndael*. Recuperado desde http://www.tierradelazaro.com/wp-content/uploads/2016/04/AES.pdf

Smith, R. (2018). Micron begins mass production of GDDR6. 16 de Diciembre de 2018. Recuperado desde https://www.anandtech.com/show/13012/micron-begins-mass-production-of-gddr6

SuperComputing Applications and Innovation. (2012). GPGPU (General Purpose Graphics Processing Unit). 13 de Diciembre de 2018. Recuperado desde http : / / www . hpc . cineca . it / content / gpgpu - general - purpose - graphics - processing-unit

UserBenchmark. (2018). i9-9900k vs i7-3770. 10 de Diciembre de 2018. Recuperado desde https://cpu.userbenchmark.com/Compare/Intel-Core-i9-9900K-vs-Intel-Core-i7-3770/4028vs1979

Anexos

Anexo 1. Script AES Rijndael escrito en entorno Python

```
import sys
import os.path
from ProgressBar import ProgressBar
from AES_base import sbox, isbox, gfp2, gfp3, gfp9, gfp11, gfp13, gfp14, Rcon

if sys.version_info[0] == 3:
  raw_input = input

def RotWord(word):
  return word[1:] + word[0:1]

def SubWord(word):
  return [sbox[byte] for byte in word]

def SubBytes(state):
  return [[sbox[byte] for byte in word] for word in state]

def InvSubBytes(state):
  return [[isbox[byte] for byte in word] for word in state]

def ShiftRows(state):
  Nb = len(state)
  n = [word[:] for word in state]
  for i in range(Nb):
    for j in range(4):
      n[i][j] = state[(i+j) % Nb][j]
  return n

def InvShiftRows(state):
  Nb = len(state)
  n = [word[:] for word in state]
  for i in range(Nb):
    for j in range(4):
      n[i][j] = state[(i-j) % Nb][j]
  return n

def MixColumns(state):
  Nb = len(state)
  n = [word[:] for word in state]
  for i in range(Nb):
    n[i][0] = (gfp2[state[i][0]] ^ gfp3[state[i][1]] ^ state[i][2] ^ state[i][3])
    n[i][1] = (state[i][0] ^ gfp2[state[i][1]] ^ gfp3[state[i][2]] ^ state[i][3])
    n[i][2] = (state[i][0] ^ state[i][1] ^ gfp2[state[i][2]] ^ gfp3[state[i][3]])
    n[i][3] = (gfp3[state[i][0]] ^ state[i][1] ^ state[i][2] ^ gfp2[state[i][3]])
  return n

def InvMixColumns(state):
  Nb = len(state)
  n = [word[:] for word in state]
  for i in range(Nb):
    n[i][0] = (gfp14[state[i][0]] ^ gfp11[state[i][1]] ^ gfp13[state[i][2]] ^ gfp9[state[i][3]])
    n[i][1] = (gfp9[state[i][0]] ^ gfp14[state[i][1]] ^ gfp11[state[i][2]] ^ gfp13[state[i][3]])
    n[i][2] = (gfp13[state[i][0]] ^ gfp9[state[i][1]] ^ gfp14[state[i][2]] ^ gfp11[state[i][3]])
    n[i][3] = (gfp11[state[i][0]] ^ gfp13[state[i][1]] ^ gfp9[state[i][2]] ^ gfp14[state[i][3]])
  return n

def AddRoundKey(state, key):
  Nb = len(state)
  new_state = [[None for j in range(4)] for i in range(Nb)]
  for i, word in enumerate(state):
    for j, byte in enumerate(word):
      new_state[i][j] = byte ^ key[i][j]
  return new_state

def Cipher(block, w, Nb=4, Nk=4, Nr=10):
  state = AddRoundKey(block, w[:Nb])
  for r in range(1, Nr):
    state = SubBytes(state)
    state = ShiftRows(state)
    state = MixColumns(state)
    state = AddRoundKey(state, w[r*Nb:(r+1)*Nb])
```

```
  state = SubBytes(state)
  state = ShiftRows(state)
  state = AddRoundKey(state, w[Nr*Nb:(Nr+1)*Nb])
  return state

def InvCipher(block, w, Nb=4, Nk=4, Nr=10):
  state = AddRoundKey(block, w[Nr*Nb:(Nr+1)*Nb])
  for r in range(Nr-1, 0, -1):
    state = InvShiftRows(state)
    state = InvSubBytes(state)
    state = AddRoundKey(state, w[r*Nb:(r+1)*Nb])
    state = InvMixColumns(state)
  state = InvShiftRows(state)
  state = InvSubBytes(state)
  state = AddRoundKey(state, w[:Nb])
  return state

def KeyExpansion(key, Nb=4, Nk=4, Nr=10):
  w = []
  for word in key:
    w.append(word[:])
  i = Nk
  while i < Nb * (Nr + 1):
    temp = w[i-1][:]
    if i % Nk == 0:
      temp = SubWord(RotWord(temp))
      temp[0] ^= Rcon[(i//Nk)]
    elif Nk > 6 and i % Nk == 4:
      temp = SubWord(temp)
    for j in range(len(temp)):
      temp[j] ^= w[i-Nk][j]
    w.append(temp[:])
    i += 1
  return w

def prepare_block(block):
  c = []
  for word in block:
    for byte in word:
      c.append(byte)
  s = None
  for byte in c:
    if sys.version_info[0] == 3:
      if not s:
        s = bytes([byte])
      else:
        s += bytes([byte])
    elif sys.version_info[0] == 2:
      if not s:
        s = chr(byte)
      else:
        s += chr(byte)
  return s

def get_block(inf, Nb=4):
  return process_block(inf[:Nb*4], Nb), inf[Nb*4:]

def padding(inf, Nb=4):
  ''' PKCS#7 padding '''
  padding_length = (Nb*4) - (len(inf) % (Nb*4))
  if padding_length:
    if isinstance(inf, str):  # Python 2
      inf += chr(padding_length) * padding_length
    elif isinstance(inf, bytes):  # Python 3
      inf += bytes([padding_length] * padding_length)
  return inf

def unpadding(inf, Nb=4):
  ''' PKCS#7 padding '''
  padding_length = ord(inf[-1])
  if padding_length < (Nb*4):
    if len(set(inf[-padding_length:])) == 1:
      inf = inf[:-padding_length]
  return inf

def process_block(block, Nb=4):
  if sys.version_info[0] == 3:  # Python 3
    if type(block) == str:
      block = bytes(block, 'utf8')
    pass
  elif sys.version_info[0] == 2:  # Python 2
    block = map(ord, block)
  return [[block[i*4+j] for j in range(4)] for i in range(Nb)]

def process_key(key, Nk=4):
  try:
    key = key.replace(" ", "")
    return [[int(key[i*8+j*2:i*8+j*2+2], 16) for j in range(4)] for i in range(Nk)]
```

```
160     except:
161       print ("Password must be hexadecimal.")
162       sys.exit()
163
164   def print_block(block):
165     s = ''
166     for i in range(len(block[0])):
167       for j in range(len(block)):
168         h = hex(block[j][i])[2:]
169         if len(h) == 1:
170           h = '0'+h
171         s += h + ' '
172       s += '\n'
173     print (s)
174
175   def str_block_line(block):
176     s = ''
177     for i in range(len(block)):
178       for j in range(len(block[0])):
179         h = hex(block[i][j])[2:]
180         if len(h) == 1:
181           h = '0'+h
182         s += h
183     return (s)
184
185   def help():
186     print ("Help:")
187     print("python AES.py -demo")
188     print("python AES.py (-e | -d) <file> [-c (128|192|256)]")
189     print("  -e: Encrypt")
190     print("  -d: Decrypt")
191     print("  -c <n>: <n> bits key (default 128)")
192     print("Note: a function mode (-e/-d) has to be specified.")
193     sys.exit()
194
195   def demo():
196     plaintext = "00112233445566778899aabbccddeeff"
197     Nb = 4
198
199     # AES-128
200     print("\n")
201     print("*"*40)
202     print("*" + "AES-128 (Nk=4, Nr=10)".center(38) + "*")
203     print("*"*40)
204     Nk = 4
205     Nr = 10
206
207     key = "000102030405060708090a0b0c0d0e0f"
208     print("KEY:\t\t{0}".format(key))
209     key = process_key(key, Nk)
210     expanded_key = KeyExpansion(key, Nb, Nk, Nr)
211
212     print("PLAINTEXT:\t{0}".format(plaintext))
213
214     block = process_key(plaintext)
215     block = Cipher(block, expanded_key, Nb, Nk, Nr)
216     print("ENCRYPT:\t{0}".format(str_block_line(block)))
217
218     block = InvCipher(block, expanded_key, Nb, Nk, Nr)
219     print("DECRYPT:\t{0}".format(str_block_line(block)))
220     print("\n")
221
222     # AES-192
223     print("*"*40)
224     print("*" + "AES-192 (Nk=6, Nr=12)".center(38) + "*")
225     print("*"*40)
226     Nk = 6
227     Nr = 12
228
229     key = "000102030405060708090a0b0c0d0e0f1011121314151617"
230     print("KEY:\t\t{0}".format(key))
231     key = process_key(key, Nk)
232     expanded_key = KeyExpansion(key, Nb, Nk, Nr)
233
234     print("PLAINTEXT:\t{0}".format(plaintext))
235
236     block = process_key(plaintext)
237     block = Cipher(block, expanded_key, Nb, Nk, Nr)
238     print("ENCRYPT:\t{0}".format(str_block_line(block)))
239
240     block = InvCipher(block, expanded_key, Nb, Nk, Nr)
241     print("DECRYPT:\t{0}".format(str_block_line(block)))
242     print("\n")
243
244     # AES-256
245     print("*"*40)
246     print("*" + "AES-256 (Nk=8, Nr=14)".center(38) + "*")
247     print("*"*40)
```

```
248     Nk = 8
249     Nr = 14
250
251     key = "000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f"
252     print("KEY:\t\t{0}".format(key))
253     key = process_key(key, Nk)
254     expanded_key = KeyExpansion(key, Nb, Nk, Nr)
255
256     print("PLAINTEXT:\t{0}".format(plaintext))
257
258     block = process_key(plaintext)
259     block = Cipher(block, expanded_key, Nb, Nk, Nr)
260     print("ENCRYPT:\t{0}".format(str_block_line(block)))
261
262     block = InvCipher(block, expanded_key, Nb, Nk, Nr)
263     print("DECRYPT:\t{0}".format(str_block_line(block)))
264     print("\n")
265
266   def main():
267     if len(sys.argv) > 1 and sys.argv[1] == '-demo':
268       demo()
269     if len(sys.argv) < 3:
270       help()
271     mode = sys.argv[1]
272     ifile = sys.argv[2]
273     if mode not in ['-e', '-d'] or not os.path.exists(ifile):
274       help()
275     try:
276       with open(ifile, 'rb') as f:
277         inf = f.read()
278     except:
279       print ("Error while trying to read input file.")
280       sys.exit()
281     Nb = 4
282     Nk = 4
283     Nr = 10
284     eggs = ''.join(sys.argv[3:])
285     spam = eggs.find('-c')
286     if spam > -1 and eggs[spam+2:spam+5] in ['128', '192', '256']:
287       Nk = int(eggs[spam+2:spam+5])//32
288     Nr = Nk + 6
289     key = raw_input(
290       "Enter a key, formed by {0} hexadecimal digits: ".format(Nk * 8))
291     key = key.replace(' ', '')
292     if len(key) < Nk * 8:
293       print ("Key too short. Filling with \'0\',"
294         "so the length is exactly {0} digits.".format(Nk * 8))
295       key += "0" * (Nk * 8 - len(key))
296     elif len(key) > Nk * 8:
297       print (
298         "Key too long. Keeping only the first {0} digits.".format(Nk * 8))
299       key = key[:Nk * 8]
300     key = process_key(key, Nk)
301     expanded_key = KeyExpansion(key, Nb, Nk, Nr)
302     if mode == '-e':
303       ofile = ifile + '.aes'
304     elif mode == '-d' and (ifile.endswith('.aes') or ifile.endswith('.cif')):
305       ofile = ifile[:-4]
306     else:
307       ofile = raw_input('Enter the output filename: ')
308       path_end = ifile.rfind('/')
309       if path_end == -1:
310         path_end = ifile.rfind('\\')
311       if path_end != -1:
312         ofile = ifile[:path_end+1] + ofile
313     if os.path.exists(ofile):
314       spam = raw_input(
315         'The file "{0}" already exists. Overwrite? [y/N] '.format(ofile))
316       if spam.upper() != 'Y':
317         ofile = raw_input('Enter new filename: ')
318     pb = ProgressBar(len(inf), 0)
319     output = None
320     if mode == '-e':  # Encrypt
321       inf = padding(inf, Nb)
322     print('')
323     while inf:
324       block, inf = get_block(inf, Nb)
325       c = pb.update(len(inf))
326       if c:
327         pb.show()
328       if mode == '-e':  # Encrypt
329         block = Cipher(block, expanded_key, Nb, Nk, Nr)
330       elif mode == '-d':  # Decript
331         block = InvCipher(block, expanded_key, Nb, Nk, Nr)
332       block = prepare_block(block)
333       if output:
334         output += block
335       else:
```

```
336         output = block
337     if mode == '-d':  # Decript
338       output = unpadding(output, Nb)
339     with open(ofile, 'wb') as f:
340       f.write(output)
341     print('')
342     sys.exit()
343 if __name__ == '__main__':
344   main()
```

- Python-AES [Caro, 2013]:

  github.compcaro90Python-AES

Anexo 2. Script AES Rijndael escrito en entorno Python paralelizado con GPU CUDA

```
1  from numba import cuda, jit
2  import sys
3  import os.path
4  from ProgressBar import ProgressBar
5  from AES_base import sbox, isbox, gfp2, gfp3, gfp9, gfp11, gfp13, gfp14, Rcon
6
7  if sys.version_info[0] == 3:
8    raw_input = input
9
10 @cuda.jit('uint8(uint8)')
11 def RotWord(word):
12   return word[1:] + word[0:1]
13
14 @cuda.jit('uint8(uint8)')
15 def SubWord(word):
16   return [sbox[byte] for byte in word]
17
18 @cuda.jit('uint8[:](uint8[:])')
19 def SubBytes(state):
20   return [[sbox[byte] for byte in word] for word in state]
21
22 @cuda.jit('uint8[:](uint8[:])')
23 def InvSubBytes(state):
24   return [[isbox[byte] for byte in word] for word in state]
25
26 @cuda.jit('uint8[:](uint8[:])')
27 def ShiftRows(state):
28   ...
29
30 @cuda.jit('uint8[:](uint8[:])')
31 def InvShiftRows(state):
32   ...
33
34 @cuda.jit('uint8[:](uint8[:])')
35 def MixColumns(state):
36   ...
37
38 @cuda.jit('uint8[:](uint8[:])')
39 def InvMixColumns(state):
40   ...
41
42 @cuda.jit('uint8[:](uint8[:], uint8[:])')
43 def AddRoundKey(state, key):
44   ...
45
46 @cuda.jit('uint8[:](uint8[:], uint8, uint8, uint8)')
47 def KeyExpansion(key, Nb=4, Nk=4, Nr=10):
48   ...
49
50 ...
51
52 if __name__ == '__main__':
53   main()
```

- Python-AES-CUDA