Instituto Tecnológico y de Estudios Superiores de Monterrey

"Campus Querétaro"

Implementación del algoritmo AES en paralelo

Proyecto final de Multiprocesadores

Autor:

A01328867

Daniel Jerónimo Gómez Antonio

Profesor:

Pedro Oscar Pérez Murueta

26 de noviembre de 2018

Índice

Índice de figuras

Índice de cuadros

Resumen

El propósito del presente documento es utilizar diferentes tecnologías para la implementación en paralelo del algoritmo de cifrado AES, considerado el estándar actualmente. La necesidad de utilizar estrategias de implementación concurrente y paralelo surge de la actual arquitectura de las computadoras, las cuales contienen múltiples núcleos y procesadores. De esta forma, se pueden conseguir mejores tiempos de ejecución y un desempeño más eficiente, como se demostrará en este documento, al comparar el speedup obtenido en cada prueba. Se utilizan diferentes tecnologías, como OpenMP, Java Threads, Java ForkJoin y CUDA, con el objetivo de comparar su eficiencia y contrastar sus ventajas y desventajas desde el punto de vista de un desarrollador de software.

1. Introducción

La demanda y necesidad de incrementar la velocidad de ejecución de las aplicaciones de software ha continuado creciendo, debido a que cada vez se desarrollan programas que requieren mayor capacidades de cómputo. Hasta el año 2004, la ley de Moore había permitido que la cantidad de transistores en un procesador incrementara de forma automática el desempeño del software [8], es decir, si el procesador ejecuta más instrucciones por segundo, entonces el software también se ejecutará más rápido. Sin embargo, debido a las limitaciones físicas de los procesadores, así como al calentamiento de los componentes, no es posible continuar confiando en la ley de Moore para conseguir algoritmos más rápidos.[8]

Una de las alternativas que inició su auge en el 2004 y 2005, es el cómputo en paralelo. El cual, toma ventaja de la arquitectura multi-núcleo de las computadoras. En este modelo, los multiprocesadores se comunican entre sí a través de caché compartido, contenido en el hardware [2]. No obstante, es necesario adaptar el software para que pueda utilizar múltiples procesadores para trabajar en la misma tarea, es decir, utilizar técnicas de paralelismo para cambiar la forma en que se escribe y ejecuta el código.

Durante el desarrollo de este artículo, se presentan 4 tecnologías distintas que implementan diferentes estrategias para conseguir la ejecución de los algoritmos en paralelo, las cuales son: *OpenMP*, *Java Threads*, *Java ForkJoin* y *CUDA*. El algoritmo implementado en paralelo, durante este documento, es el AES. El cual corresponde al algoritmo de cifrado estándar por bloques, que se utiliza en una gran variedad de aplicaciones de seguridad hoy en día. Así también, se enuncian las ventajas y desventajas de cada una de las tecnologías mencionadas, resaltando las aplicaciones en las que es conveniente utilizar cada una de las herramientas. De igual manera, se presenta un análisis del desempeño de cada una al ejecutar el algoritmo AES bajo circunstancias similares.

2. Desarrollo

El desarrollo del proyecto consiste en la implementación del algoritmo AES utilizando 4 de las tecnologías más usadas actualmente para obtener software corriendo en paralelo. A continuación, se presenta la definición del cómputo en paralelo, Posteriormente una sección para describir el funcionamiento del algoritmo estándar de cifrado avanzado (AES, por sus siglas en inglés), así como las propiedades que permiten que este algoritmo sea paralelizado. En secciones seguidas, se introduce cada una de las tecnologías utilizadas y sus principales características para conseguir código en paralelo.

2.1. Cómputo en paralelo - Multiprocesadores

El cómputo en paralelo surge de la necesidad de explotar los múltiples núcleos que actualmente contiene la arquitectura de los procesadores. En el pasado, las tareas solían ejecutarse una a la vez, o múltiples, pero nunca al mismo tiempo. Es decir, la multi-tarea solo se conseguía con la administración del procesador, de tal forma que se alternaba entre tareas para que siempre hubiera un proceso para ejecutarse y que ninguno se ejecutara por demasiado tiempo.

Utilizando la arquitectura multi-núcleo, ahora es posible que la misma tarea se trabaje al mismo tiempo y de esta forma optimizar la ejecución de los programas. Esto se realiza dividiendo el algoritmo en secciones, las cuales son ejecutadas por diferentes hilos. En este sentido, un proceso se define como el código ejecutable de un programa, y un hilo es una rama en la que se puede dividir un proceso. De tal manera que un proceso siempre tiene al menos un hilo. La ventaja del cómputo en paralelo es que los procesos tienen múltiples hilos, cada uno ejecutandose en diferentes núcleos físicos del hardware. Por ejemplo, si se tiene un proceso que itera sobre un arreglo sumando sus elementos, este puede dividirse en N hilos, cada uno realizando la operación correspodiente sobre una región del arreglo. Al final, el resultado será el mismo, sin embargo el tiempo de ejecución disminuye al utilizar hilos.

Una de las consideraciones importantes al utilizar hilos, es lo que se conoce como

condición de carrera (*race condition*), la cual sucede cuando más de un hilo trata de escribir a una región de memoria al mismo tiempo. Es por esto que la sincronización es importante al usar hilos. Existen estrategias para evitar condiciones de carrera y otros problemas al trabajar con hilos, como distribución y manejo correcto de la memoria, operaciones atómicas, semáforos, etc., sin embargo, no forman parte del alcance del presente documento.

Es importante aclarar que no todos los algoritmos se pueden ejecutar en paralelo y no en todos se obtiene el mismo desempeño. Es decir, usar 2 hilos para ejecutar un proceso no garantiza la optimización de este al doble, ya que la creación de cada hilo consume tiempo. Algunos ejemplos de algoritmos que pueden implementarse en paralelo son multiplicación de matrices, suma de vectores, convolución, procesamiento de imágenes, etc.

Para calcular que tanto se ha optimizado el algoritmo, respecto a la versión secuencial del mismo se utiliza lo que se conoce como *speedup*. Se calcula como se muestra en la secuación 1

$$s = \frac{T_{seq}}{T_{par}} \tag{1}$$

2.2. El algoritmo AES - Advanced Encryption Standard

El AES se convirtió en el nuevo estándar de cifrado el 26 de noviembre de 2001. Se basa en el algoritmo de Rijndael creado por Vincent Rijmen y Joan Daemen. Se propuso como una versión para sustituir el DES, que fue quebrantado en 1998. Según lo establecido por Selent, el algoritmo de Rijndael " utiliza una combinación de operaciones OR exclusivas (XOR), sustitución de octetos por una matriz S-box rotaciones de fila y columna, y mezcla de columnas. Tuvo éxito porque es fácil de implementar y se puede ejecutar en un tiempo razonable en una computadora normal. " [7]

El algoritmo AES funciona en ambas direcciones, se utiliza para cifrar y descifrar cualquier tipo de archivo binario. Recibe como entrada el archivo dividido en bloques de 128 bits (16 bytes) y una clave de 128, 196 o 256 bits. Este proyecto utiliza una clave de

128 bits. Para cada bloque, la salida es otro bloque de 128 bits con los datos cifrados. Las operaciones se realizan sobre una matriz de 4x4 que contiene el bloque correspondiente, denominado *estado*. [1] La Fig. 1 muestra la representación gráfica del algoritmo, destacando las principales transformaciones que toman lugar sobre el estado. Es importante tener en cuenta que estas transformaciones se aplican varias veces utilizando el modo CBC, lo que significa que después de realizar la primera ronda de cifrado en el texto plano, las operaciones adicionales se aplicarán sobre un texto cifrado. Al hacer esto, la seguridad aumenta. Estas transformaciones se describen a continuación:

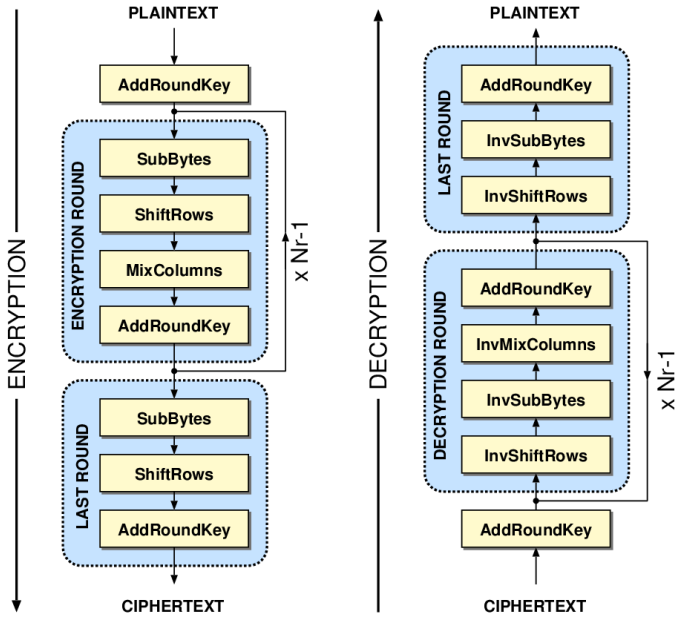

Fig. 1. Diagrama de flujo de cifrado y descifrado utilizando el algoritmo AES

- **Key expansion:** Es una rutina que se usa para expandir la clave de entrada y obtener $16 * (Nr + 1)$ bytes nuevos. Por lo tanto, se utiliza una clave diferente en cada ronda del algoritmo. El número de rondas depende de la longitud de la clave, ya que se utilizará una clave de 16 bytes, luego habrá 10 rondas y la clave expandida será de 176 bytes. Los métodos utilizados para expandir la clave involucran permutaciones cíclicas y operaciones aritméticas sobre los campos de Galois. Para más detalles consulte [1].

- **Add Round Key:** En esta parte del proceso, la clave se agrega al estado mediante una operación XOR. La parte de la clave expandida utilizada depende del número de

ronda.

- **Sub Bytes/Inverse Sub Bytes:** Esta transformación utiliza una tabla de búsqueda para realizar una sustitución de byte por byte, es decir, el valor de cada byte es cambiado por otro dentro de la matriz de S-box, o S-box inverso para descifrar. La matriz utilizada fue elegida cuidadosamente para proporcionar seguridad. Nuevamente, los valores de las matrices son el resultado de la aritmética modular sobre los campos de Galois. [1]

- **Shift rows/Inverse Shift Rows:** En este paso, las últimas tres filas del estado se desplazan hacia la izquierda para el cifrado o hacia la derecha para el descifrado. La fila 1 se desplaza con un desplazamiento de 1, la fila 2 con 2 y la fila 3 con 3.

- **Mix Columns/ Inverse Mix Columns:** Este proceso es realmente complejo de explicar, ya que utiliza las operaciones de los campos de Galois en cada columna de la matriz de estado. Afortunadamente, las operaciones aritméticas se pueden calcular previamente y almacenar en diferentes tablas de búsqueda. Por lo tanto, el proceso se reduce a realizar multiplicaciones y operaciones XOR con cada columna del estado y la tabla de búsqueda correspondiente.[7]

Todas las transformaciones anteriores se aplican a cada bloque en el que se divide el archivo de entrada 10 veces consecutivas. Cada bloque que se cifra es independiente de los demás, lo que brinda la posibilidad de hacer una implementación paralela del algoritmo.

2.3. Paralelización del algoritmo AES

El algoritmo AES implementado en modo CBC no se puede paralelizar por definición, porque cada ronda depende de los resultados de la ronda anterior de cifrado. Si bien, las transformaciones se pueden paralelizar, no tiene sentido usar diferentes hilos para calcular las operaciones en un estado de solo 16 bytes.

El enfoque utilizado en este proyecto se centra en ejecutar en paralelo el cifrado de los bloques en los que se divide el archivo. Es posible hacer esto porque el proceso de cifrado

para cada bloque es independiente de los otros. Por ejemplo, supongamos que un archivo de 1,048,576 bytes necesita cifrarse. Este archivo se dividirá en bloques de 16 bytes. Así, el algoritmo AES se aplicaría a 65,536 bloques. Aquí es donde se puede usar la paralelización, para distribuir el trabajo entre los hilos disponibles dependiendo de la tecnología usada.

2.4. AES paralelo usando OpenMP

De acuerdo a su sitio oficial, OpenMP es una API que permite definir un conjunto de directivas de compilación, rutinas de biblioteca y variables de entorno que pueden utilizarse para especificar paralelismo de alto nivel en programas de Fortran y C/C++.[5]

La principal característica de OpenMP es la fácil migración de un código secuencial a uno paralelo con tan solo utilizar directivas de preprocesamiento que le indican al compilador como se ejecutarán las secciones de código en paralelo. De esta forma, se vuelve fácil para el programador paralelizar un código. Es importante considerar, que los datos compartidos y privados son la clave de OpenMP y deben ser especificados en las directivas, de esta forma se establece la communicación y sincronización entre hilos.

A continuación, se presenta la sección principal encargada de paralelizar el segmento de código que controla la división de bloques del AES, implementada usando C con directivas de OpenMP.

```
1 void cipher_control(byte *file_in, byte *file_out, long long file_size,
    unsigned long blocks, byte *expanded_key) {
2   unsigned long block;
3   int padding, res;
4
5   // Check if the size of the input file is multiple of 16
6   res = file_size %16;
7
8   #pragma omp parallel for shared(file_in, file_out, expanded_key)
9   for(block = 0; block < blocks; block++) {
10      // Check if it is necessary to add padding to the last block
11      if(block == blocks - 1 && res != 0) {
12          padding = 16 - res;
```

```
13            for(int i = res;i < res + padding;i++) {
14                file_in[block * 16 + i] = 0x00;
15            }
16        }
17        // Invoke the cipher process for the corresponding block
18        cipher(file_in + block * 16, expanded_key);
19        // Copy the encrypted block to the output file
20        memcpy(file_out + block * 16, file_in + block * 16, 16 * sizeof(byte));
21    }
22 }
```

Listing 1: Función principal de cifrado en el AES usando OpenMP

Como se observa, este fragmento de código corresponde a la división del trabajo en bloques. La parte más importante es la línea en que se invoca al método *cipher*. El cual, ejecuta el algoritmo AES para cifrar el bloque actual. No hay cambios significativos, respecto a la implementación secuencial. Excepto por la línea 8, en que se especifica que se utilizará un ciclo *for* en paralelo. El cual puede compartir en memoria el archivo de entrada, el de salida y la llave para cifrar. Es así, como de forma simple el algoritmo es paralelizado usando OpenMP.

2.5. AES paralelo usando CUDA

Según el sitio web de NVIDIA, " CUDA es una arquitectura revolucionaria de computación paralela creada por NVIDIA. CUDA hace posible utilizar los muchos núcleos de computación en un procesador gráfico para realizar cálculos matemáticos de propósito general, logrando aceleraciones dramáticas en el rendimiento de computación ". [4] Como se mencionó, CUDA necesita una unidad de procesamiento de gráficos (GPU) para ejecutarse. Debido a la arquitectura de los GPU, es posible administrar cientos de hilos y obtener grandes aceleraciones. En la Fig. 2 se pueden observar las diferencias generales entre la CPU y la arquitectura de la GPU. La GPU tiene un gran número de núcleos que se agrupan en bloques y se gestionan mediante su propia unidad de control. Además, cada bloque tiene acceso a una pequeña porción de caché, por lo tanto, varios hilos pueden ejecutarse simultáneamente

pero operan en datos diferentes.

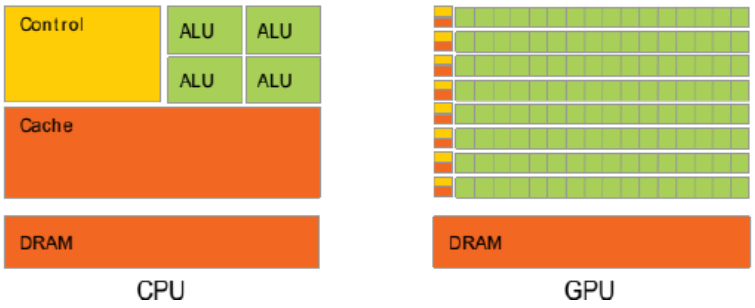

Fig. 2. Arquitectura de un CPU vs GPU

La principal ventaja de un GPU es que pueden ejecutar números masivos de hilos paralelos. Por otro lado, las desventajas son el costoso costo del hardware y el alto consumo de energía durante la ejecución.

Ahora es el momento de presentar el código escrito en CUDA C utilizado para administrar el proceso de cifrado. Este método se ejecuta en la GPU. Administra la parte de paralelización, luego invoca el `cipher`. Método, que realiza el AES sobre el estado recibido como parámetro.

```
__global__ void cipher_control(byte *file_in, byte *file_out, long long *
    file_size, unsigned long *blocks, byte *expanded_key, byte *d_sbox, byte *
    d_m2, byte *d_m3)
{
    byte state[16];
    int block;
    int padding, res;
    // Get the number of the block that the current thread is managing
    block = blockIdx.x * blockDim.x + threadIdx.x;
    // Check if the size of the input file is multiple of 16
    res = *file_size % 16;
    // Verify the current block is not out of boundaries
    while(block < *blocks) {
        //Copy the corresponding input data to the state matrix
        memcpy(state, file_in + block  * 16, 16 * sizeof(byte));

        // Check if it is necessary to add padding to the last block
```

```
16          if(block == ((*blocks) - 1) && res != 0) {
17              padding = 16 - res;
18              // Add padding only to the required spaces
19              for(int i = res;i < res + padding;i++) {
20                  state[i] = 0x00;
21              }
22          }
23          // Invoke the cipher process for the corresponding block
24          cipher(state, expanded_key, d_sbox, d_m2, d_m3);
25          // Copy the encrypted block to the output file
26          memcpy(file_out + block  * 16, state, 16 * sizeof(byte));
27          // Update the current block moving to the next section of memory
    allowed for this thread
28          block += gridDim.x * blockDim.x;
29      }
30  }
```

Listing 2: Función para controlar el AES ejecutándose en el GPU

Esta función incluye la sintaxis C normal y algunas palabras especiales de CUDA. El `__global__` indica que el método se ejecutará en la GPU, lo que significa que el mismo segmento de código se copiará y ejecutará en varios hilos al mismo tiempo. Las instrucciones más importantes son las que se encuentran en las líneas 7, 24 y 28. En la línea 7, la instrucción`block = blockIdx.x * blockDim.x + threadIdx.x;` obtiene el número de bloque que el hilo actual necesita para cifrar. Dado que el programa se está ejecutando con varios bloques de hilos, es importante obtener el índice correcto para evitar las condiciones de carrera. En la línea 24, el método `cipher` es invocado, aquí es donde el algoritmo AES en realidad tiene lugar y realiza operaciones sobre el estado actual. Finalmente, en la línea 27, el bloque para cifrar actual se actualiza sumando `gridDim.x * blockDim.x`.Esta operación garantiza que el siguiente bloque que será cifrado por el hilo actual no sea cifrado por otro.

La función `decipher_control`es similar al previo `decipher_control`, la principal diferencia es la parte de relleno. Dado que el tamaño de los archivos de entrada podría no ser un múltiplo de 16, se agrega un relleno para completar el último bloque durante el proceso de cifrado. El número de bytes que se agregaron se almacena en el primer byte del archivo

de salida. Luego, en el proceso de descifrado, se lee la información sobre el relleno y esos últimos bytes se ignoran al guardar el archivo descifrado.

El método `cipher_control`se invoca usando la siguiente instrucción dentro del `main` del programa: `cipher_control <<< 128, 128>>> (/*params*/)`. Esta instrucción le dice a la GPU que use 128 * 128 = 16,384 hilos para ejecutar la función. Este número debe ser elegido cuidadosamente dependiendo de la aplicación. En este caso fue seleccionado como una recomendación encontrada en [6], donde se indica que, debido a limitaciones de hardware, no es posible usar más de 65,535 hilos ni más de 512 hilos por bloque. En este proyecto, la cantidad óptima de hilos debe ser igual al tamaño del archivo de entrada, sin embargo, este tamaño es variable. Cuando la cantidad de hilos es mayor que la requerida, habrá hilos que no hagan más que consumir recursos, por otra parte, si el número es menor, entonces los hilos necesarios tendrán que realizar más trabajo. Una cuadrícula de 128 * 128 es un número razonable que puede tratar con archivos grandes y pequeños.

Todos los métodos para leer/escribir los archivos de entrada/salida son administrados por el CPU. Todos los otros métodos relacionados con el algoritmo AES, como AddRoundKey, MixColumns, ShiftRows, etc. son administrados por el GPU.

2.6. AES paralelo usando Java Threads

Java, integra dentro de su API soporte para operaciones en paralelo utilizando diferentes estrategias, una de las cuales es utilizar objetos de tipo `Thread`. Estos objetos están diseñados para ejecutarse de forma paralela. La creación y administración de los `Threads` es responsabilidad del programador.

En el caso del uso de Java Threads, la implementación del método de descifrado se realizó al crear un objeto `AES_decipher`, el cual se instancia dentro de cada uno de los Threads creados. En el caso del código de Java, la cantidad de hilos que se utilizan depende del hardware disponible. Se recomienda utilizar solo 2 por núcleo del procesador.[3]

Una vez que los Threads son creados, se asigna el trabajo a cada uno de ellos, dividiendo equitativamente la cantidad de bloques a descifrar. Por ejemplo, si se tuvieran

800 bloques y 8 hilos disponibles, cada uno de los Threads debería realizar el trabajo de descifrado de 100 bloques. Es importante cuidar la distribución correcta de la memoria, ya que si se asigna el mismo bloque a más de un hilo, se pueden dar condiciones de carrera.

A continuación, se presenta el segmento de código utilizado para manejar la creación, inicialización y finalización de cada uno de los hilos. La instrucción `.start()` indica que el hilo debe comenzar a ejecutarse en paralelo. Por otro lado, la sentencia `.join()` se utiliza para indicar que se debe esperar hasta que todos los hilos hayan terminado su ejecución antes de continuar con el flujo del programa.

```
1     // Instantiate the object to cipher
2     ac = new AES_cipher(file_in, key);
3           the runtime
4           // Assign the work to the threads
5           for(int thr = 0; thr < MAXTHREADS; thr++) {
6              if(thr < MAXTHREADS - 1) {
7                 threads[thr] = new Thread(new AES_cipher(thr * thread_blocks, (
    thr + 1) * thread_blocks));
8              }
9              else {
10                threads[thr] = new Thread(new AES_cipher(thr * thread_blocks,
    aes_blocks));
11             }
12          }
13          // Start the execution of the threads
14          for(int thr = 0; thr < MAXTHREADS; thr++) {
15             threads[thr].start();
16          }
17          // Wait for every thread to finish its work
18          for(int thr = 0; thr < MAXTHREADS; thr++) {
19             try {
20                threads[thr].join();
21             } catch (InterruptedException e) {
22                e.printStackTrace();
23             }
```

```
24          }
```

Listing 3: Segmento que distribuye y administra el trabajo del AES ejecutándose con Java Threads

2.7. AES usando Java ForkJoin

Java también provee un framework para manipular los hilos, de esta forma, el programador no tiene que lidiar con la administración y sincronización de hilos, pues estos son creados dentro de un *pool*. Este framework se llama Java ForkJoin. La forma en que funciona es a través de la división recursiva del bloque inicial de trabajo en más pequeños, hasta alcanzar un tamaño aceptable que es asignado a alguno de los hilos que se encuentre libre. Funciona a través del algoritmo de work-stealing, por el cual, si un hilo termina su trabajo y esta en espera, robará el de otro hilo que esté ocupado. Esto con el objetivo de hacer más eficiente la ejecución y mantener a todos los hilos siempre trabajando.[3]

Al igual que con Java Threads, se implementaron 2 clases, una para el cifrado y la otra para el descifrado. La función más importante se encuentra dentro de cada una de las clases y es muy similar, a continuación se presenta el fragmento que divide la tarea e invoca la creación de nuevas tareas para los hilos. Como puede observarse, si la tarea es lo suficientemente pequeña, esta es procesada invocando el método de control de descifrado, el cual es similar al presentado en la sección de OpenMP. En caso contrario, se divide el bloque a la mitad y se crean 2 nuevos objetos `AES_decipher`, cada uno con la mitad del trabajo inicial. Los objetos nuevos se asignan al *pool* utilizando el método *invokeAll*. En el `main` lo único que se hace es crear el *pool* y asignarle el objeto inicial, conteniendo todo el trabajo.

```
1
2       public void compute() {
3         if(this.end - this.start <= THRESHOLD) {
4             this.decipher_control();
5         }
6         else {
7             long mid = (this.start + this.end) / 2;
8             invokeAll(new AES_decipher(this.start, mid),
```

```
9                       new AES_decipher(mid, this.end));
10          }
11      }
```

Listing 4: Segmento que divide y crea nuevas tareas del AES ejecutándose con Java ForkJoin

3. Resultados

Para evaluar el desempeño de las implementaciones realizadas, se procedió a utilizar los programas para cifrar un archivo, y posteriormente descifrarlo. De esta forma se comprobó la correcta ejecución, pues el archivo descifrado correspondía al mismo archivo original. De igual manera, se midió el tiempo de ejecución para hacer las comparaciones pertinentes. Es necesario resaltar, que se ejecutó el algoritmo 10 veces consecutivas para reportar el tiempo promedio de ejecución, así se elimina la influencia de factores externos en el tiempo reportado. El archivo de prueba para cifrar y descifrar corresponde a un libro en formato PDF de 17.9 MB. La llave utilizada se lee de un archivo TXT y fue la misma para todos los casos.

En todos los casos, la computadora utilizada para realizar las pruebas tiene las características técnicas que se enuncian en la Tabla 1.

CPU	Intel Core i5 (8th Gen) 8250U / 1.6 GHz
Number of cores	8
Memory	8 GB DDR4 SDRAM
Storage	256 GB SSD
GPU	NVIDIA GeForce MX150 - 2 GB GDDR5 SDRAM
OS	Linux Mint 18.3 Sylvia 64-bits

Cuadro 1: Características técnicas de la computadora utilizada en las pruebas

Los resultados del speedup obtenido, se calcularon al implementar el algoritmo AES de forma secuencial usando C y Java. En el caso de OpenMP y CUDA, el speedup se compara con la implementación en C; mientras que para Java Threads y ForkJoin, se utiliza la versión secuencial hecha en Java. Los resultados se resumen en la Tabla 2 y en la gráfica de la Fig. 3

PRUEBA	Tiempo en milisegundos											
	C Secuencial		OpenMP		CUDA		Java secuencial		Java Threads		Java ForkJoin	
	Cipher	Decipher	Cipher	Decipher	Cipher	Decipher	Cipher	Decipher	Cipher	Decipher	CIpher	Decipher
Book.pdf de 17.9 MB	*2500.9*	*3089.4721*	*826.2884*	*1004.1843*	*0.0089*	*0.0099*	*504.6*	*704*	*210.6*	*212.6*	*203.8*	*219.3*

Cuadro 2: Resumen del tiempo de ejecución para la implementación de AES en paralelo

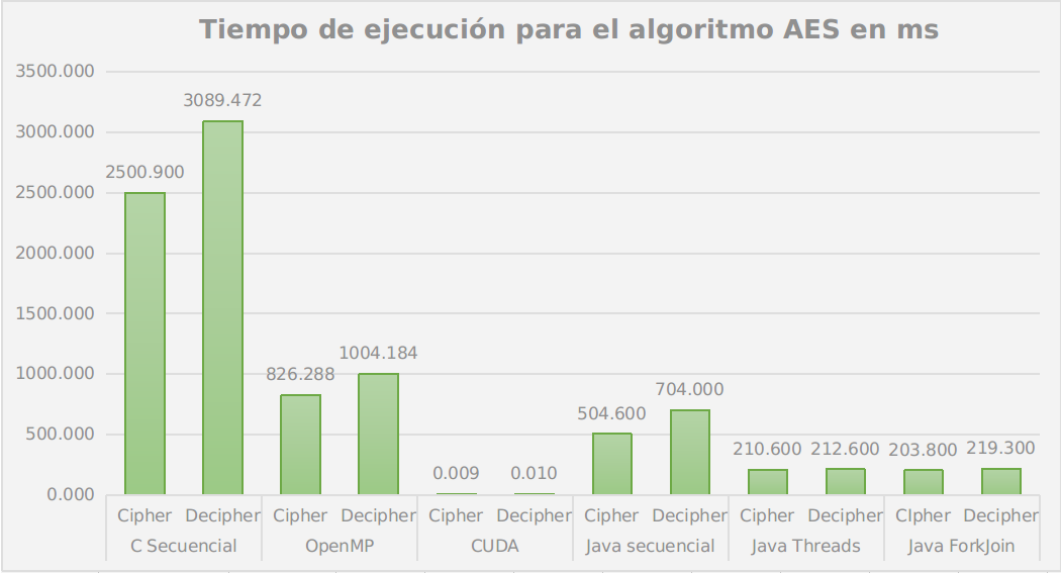

Fig. 3. Gráfica comparativa del tiempo de ejecución para el algoritmo AES paralelo

En la Tabla 3 se muestra el speedup obtenido para cada una de las tecnologías utilizadas.

Tecnología	**Speedup**	
	Cipher	**Decipher**
OpenMP	*3.027*	*3.077*
Java Threads	*2.396*	*3.311*
Java ForkJoin	*2.476*	*3.210*
CUDA	*281000.000*	*312067.889*

Cuadro 3: Speedup obtenido con las 4 tecnologías para el AES en paralelo

Como se puede ver, el mayor speedup se obtiene utilizando CUDA para paralelizar el algoritmo AES de más de 300,000 lo cual es una mejora muy significativa. Seguido de eso, se obtiene una mejora de 3 para OpenMP y de 2.8 en promedio para el cifrado y descifrado usando ambas tecnologías de Java. En este sentido, se puede concluir que el speedup que se obtiene al optimizar el AES se encuentra cerca de 3 al usar cómputo en un CPU de 8 núcleos. Sin embargo, al utilizar el GPU para el algoritmo AES se obtiene un desempeño

considerablemente grande. Se puede concluir que la mejor implementación del algoritmo paralelo AES en un CPU se obtiene usando OpenMP. No obstante, el tiempo de ejecución de Java es menor respecto a C, aún cuando el speedup no lo supera. Esto tiene que ver con la manera en que Java trabaja y administra la memoria. Finalmente, en el caso de tener un GPU disponible, la mejor optimización sin duda se da al usar CUDA.

La selección de la tecnología para implementar un algoritmo en paralelo depende del uso que se requiera dar a la aplicación final. Por ejemplo, en el caso de AES se puede utilizar en distintos escenarios. En una aplicación de alto nivel que tiene contacto con el usuario, una opción viable sería Javva Threads o ForkJoin. No obstante, en una aplicación de bajo nivel, por ejemplo para cifrar mensajes que se envían a los registros de hardware conviene utilizar C y OpenMP. Finalmente, si se cuenta con el hardware dedicado, en este caso un GPU, la opción indiscutible sería CUDA.

La ventaja principal de OpenMP se relaciona a la facilidad con que se modifica un algoritmo en secuencial para hacerlo paralelo, pues solo requiere especificar las directivas de preprocesamiento. La desventaja, es que no se tiene la versatilidad y flexibilidad para modificar el comportamiento que se desea en paralelo. En el caso de Java, la desventaja se encuentra en que se requiere hacer la administración de los hilos de forma manual y lidiar con problemas de sincronización. Una de las ventajas sería la posibilidad de trabajar con aplicaciones de alto nivel de forma sencilla. Por ejemplo, procesar imágenes y archivos no requiere un gran nivel de complejidad. Finalmente, CUDA representa la ventaja de speedup muy grandes (y de verdad, muy grandes) gracias a la cantidad de hilos que brinda, sin embargo, la desventaja es que requiere de un hardware especial (GPU), que es costoso y además requiere mayores recursos, como más energía y se calienta más que un procesador normal.

4. Conclusiones

Al finalizar el proyecto de investigación, se comparó objetivamente el desempeño de las diferentes tecnologías actuales para paralelizar los algoritmos. Los resultados obtenidos

muestran que el tiempo de ejecución promedio del AES puede ser incrementado 3 veces usando un CPU, y miles de veces con un GPU. Al analizar las ventajas y desventajas de cada una de las tecnologías, es evidente que cada una de ellas puede ser adecuada bajo diferentes escenarios. Esta es la razón por la cual es importante, para un programador, conocer OpenMP, Java Threads, Java ForkJoin, CUDA y las demás tecnologías que existen para crear código en paralelo. Ya que de lo contrario, se podría hacer la elección incorrecta al desarrollar una nueva aplicación.

Es evidente, que el cómputo secuencial ya no es suficiente para resolver las necesidades que actualmente demandan los usuarios. Es necesario integrar el paralelismo en el desarrollo de software para obtener mejores resultados, en cuanto a eficiencia. Además, de esta forma se aprovecha al máximo la arquitectura de los sistemas multi-núcleo y multiprocesadores. Finalmente, es necesario resaltar que antes de usar las tecnologías disponibles para el multiprocesamiento, se debe realizar un análisis cuidadoso del algoritmo a implementar, con la finalidad de evitar problemas en tiempo de ejecución, como condiciones de carrera o incluso tratar de paralelizar un algoritmo que, por su naturaleza, no es posible. El paralelismo en el software es una herramienta muy poderosa que requiere un uso adecuado para conseguir resultados óptimos y contribuir a crear software más eficiente.

5. Anexos

El repositorio con el proyecto completo puede ser accedido a través del siguiente enlace público de Github: `https://github.com/dannyel2511/multiprocessors-final-project`

Referencias

[1] FIPS. «Advanced Encryption Standard (AES)». En: (2001). URL: `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[2] Maurice Herlihy y Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[3] Java. *The Java Tutorials*. URL: https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html (visitado 24-11-2018).

[4] NVIDIA. *What is CUDA?* URL: https://www.nvidia.com/object/io_69526.html (visitado 19-11-2018).

[5] OpenMP. *OpenMP FAQ*. URL: https://www.openmp.org/ (visitado 24-11-2018).

[6] Jason Sanders y Kandrot Edward. *CUDA by example. An introduction to general purpose GPU programming*. Addison-Wesley, 2010.

[7] Douglas Selent. *Advanced Encryption Standard*. 2010. URL: https://www2.rivier.edu/journal/ROAJ-Fall-2010/J455-Selent-AES.pdf.

[8] Herb Sutter. «The free lunch is over: A Fundamental Turn Toward Concurrency in Software». En: *Dr. Dobb's* (2005). URL: http://www.gotw.ca/publications/concurrency-ddj.htm.