Paralelización de una búsqueda de fuerza bruta sobre lista de hashes en MD5

Víctor Ramírez
Universidad Nacional de Colombia
vgramirezc@unal.edu.co

***Abstract*— El presente trabajo tiene dos objetivos principales. Primero ver que tan paralelizable es una búsqueda de fuerza bruta en el cual se encriptan todas las cadenas de texto que cumplan cierto formato usando el algoritmo MD5, y se comprueba si hay colisiones con una lista de hashes que se desean descifrar. Segundo se busca comparar los resultados de paralelizar el programa usando 4 diferentes herramientas: OpenMP, MPI, CUDA y OpenCL. Para las versiones de OpenMP y MPI se utiliza una CPU con dos nucleos fisicos y dos virtuales, mientras que para las versiones de CUDA y OpenCL se utiliza una GPU Tesla K80. Se obtiene al final una mayor reducción de los tiempos de ejecución usando CUDA, siendo 50 veces más rápido que la versión secuencial, seguido por OpenCL con una aceleración de 25. MPI y OpenMP solo alcanzan a duplicar la velocidad debido a las características de la CPU utilizada.**

I. INTRODUCCIÓN

Una función hash es una función que se encarga de tomar datos como entrada y retorna un valor dentro de un rango finito. Una función hash criptográfica es un tipo especial de función hash con ciertas propiedades que la hacen ideal para su uso en la seguridad de la información. Una de estas propiedades es que debe ser imposible usar el valor retornado para generar el dato original al cual se le aplicó dicha función. Por esta razón, una búsqueda de fuerza bruta resulta ser uno de los pocos métodos que se pueden emplear para obtener los datos originales. En esta búsqueda lo que se hace es tomar un conjunto de valores que se sospecha pueden ser los valores originales, se les aplica la función hash y se compara el valor final con el que intentamos descifrar. En el caso particular de este trabajo se supone que los datos originales son cadenas de texto y se reduce el espacio de búsqueda a aquellas cadenas que cumplan un formato dado, con el cual se fija su largo y se define que tipo de carácter se encuentra en cada posición, pudiendo ser estos un dígito, una letra minúscula o una letra mayúscula del alfabeto inglés. Esta búsqueda resulta ser paralelizable debido a que cada valor del dominio puede ser procesado de forma independiente. Se busca entonces evaluar la eficiencia de la paralelización usando distintas herramientas de programación y comparando los tiempos de ejecución al modificar la cantidad de unidades de procesamiento y divisiones de trabajo usadas. También se desarrolla una versión secuencia del programa, el cual sirve como base para medir la aceleración de las versiones paralelas al dividir el tiempo de ejecución del programa secuencial sobre el tiempo de ejecución del programa paralelo (1).

$$S = \frac{T_S}{T_P} \tag{1}$$

Todas las versiones del programa reciben como parámetros el formato de cadenas de texto sobre el cual se aplicará la búsqueda y el nombre de un archivo de texto donde se encontrará el listado de hashes que se desean descifrar. En este caso solo se trabajará con encriptado MD5, por lo cual cada hash se puede ver como una cadena compuesta por 32 símbolos hexadecimales. Se trabajará con las siguientes herramientas para las versiones paralelas:

A. OpenMP

API que permite escribir programas multiprocesos. Para esta versión se debe especificar la cantidad de hilos que se quieren crear para distribuir la carga de trabajo por igual entre ellos.

B. Message Passing Interface (MPI)

Permite desarrollar programas que corren en múltiples procesadores y se sincronizan mediante el paso de mensajes. Resulta ideal para su uso en sistemas distribuidos, sin embargo en este caso se limitó su uso al de una sola máquina, usando los núcleos de la CPU. Esta versión del programa recibe como atributo la cantidad de copias del programa a ejecutar.

C. CUDA

Es una plataforma de programación paralela que permite desarrollar programas que corren en paralelo mediante el uso de unidades de procesamiento gráfico (GPU) diseñadas por nvidia. Para esta versión se deben especificar la cantidad total de hilos y la cantidad de bloques a usar en la ejecución.

D. OpenCL

Framework que permite escribir programas con paralelismo que funcionan en diferentes tipos de plataformas, permitiendo utilizar procesamiento tanto en CPU como en GPU. Esta versión del programa recibe la cantidad de work items totales y la cantidad de work groups a ser usados.

Después de correr todas las versiones sobre un mismo escenario se pasa a realizar una comparación de resultados mediante gráficas de tiempo de ejecución y de aceleración. Por último se llega a una serie de conclusiones sobre qué versiones resultaron más eficientes y bajo qué configuraciones o cantidad de recursos usados.

II. MATERIALES Y MÉTODOS

Para evaluar todas las versiones sobre un mismo escenario se decidió fijar el formato de cadenas de texto a uno de tamaño seis compuesto solamente por letras minúsculas, lo cual da un total de 308.915.776 cadenas de texto. Para poder realizar de forma sencilla la división de carga de trabajo se decidió representar cada cadena de texto como un número entre cero (inclusivo) y el total de cadenas (exclusivo), donde el número i representa a la i-ésima cadena lexicográficamente ordenada que cumple con el formato dado. Para almacenar los hashes que se quieren descifrar se utilizó una estructura de datos llamada Trie, ya que permite comprobar si un hash dado está almacenado en la estructura en una complejidad lineal con respecto al largo del hash (32 en el caso de MD5), y se puede implementar fácilmente con el uso de una matriz de enteros, lo cual resulta más sencillo de manejar en OpenCL que las clases. En total se pasaron 100.000 hashes en todas las ejecuciones, los cuales se obtuvieron de una base de datos con hashes de contraseñas de páginas web que se han filtrado en internet. Las versiones de OpenMP y MPI se ejecutaron en CPU con las carácteristicas que se encuentran en Fig. 1. Por otro lado, las versiones de CUDA y OpenCL corrieron en una GPU con las carácteristicas que se encuentran en Fig. 2.

```
CPU(s):                  4
On-line CPU(s) list:     0-3
Thread(s) per core:      2
Core(s) per socket:      2
Socket(s):               1
NUMA node(s):            1
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   58
Model name:              Intel(R) Core
(TM) i5-3337U CPU @ 1.80GHz
```

Fig. 1. Especificaciones CPU

```
Device 0: "Tesla K80"
  CUDA Driver Version / Runtime Version          10.0 / 8.0
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes (11996954624 bytes
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536),
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
```

Fig. 2. Especificaciones GPU

A continuación se presentan detalles de cada una de las versiones.

A. OpenMP

Para realizar la paralelización simplemente se distribuyó la cantidad de cadenas de texto a evaluar en cantidades iguales para cada hilo lanzado. Debido a que cada cadena se está representando como un número, el problema se reduce simplemente a paralelizar un ciclo for, donde cada hilo utilizará su id para saber el rango que le corresponde. Se probó con cantidades de hilos potencia de dos entre 1 y 16.

B. MPI

Ya que esta versión solo corre sobre una misma máquina y no sobre un sistema distribuido su implementación es en esencia la misma que en OpenMP pero dividiendo el trabajo en número total de procesadores en lugar de hilos.

C. CUDA

En esta versión se incrementa considerablemente el uso de memoria ya que por cada hilo se debe apartar la cantidad de memoria utilizada para almacenar la cadena de texto que está procesando, el hash de dicha cadena y la clase MD5 que se utiliza para encriptarla. La cantidad de bloques se iteró en múltiplos de 13 debido a que es el número de multiprocesadores. Como cantidad total de hilos se utilizaron las potencias de dos entre 1 y 8192.

D. OpenCL

Tiene las mismas consideraciones de uso de memoria que CUDA. Se utilizaron cantidad de work groups múltiplos de 32 como es la recomendación para el dispositivo usado. La cantidad total de work items se iteró en potencias de dos entre 1 y 8192.

III. RESULTADOS

El tiempo de ejecución del programa secuencial fue de 34.2 segundos. Este es el tiempo base que se usó para el cálculo de la aceleración de los versiones paralelas. A continuación los resultados separados por herramientas.

A. OpenMP

En Fig. 3 y Fig. 4 se muestran los resultados de tiempo y aceleración. El tiempo se redujo casi a la mitad al usar dos hilos, después hay una reducción pequeña de tiempo al usar 4 hilos y por último se vuelven a incrementar los tiempos usando 8 y 16 hilos. Esto se debe a que la CPU tiene 2 núcleos físicos, los cuales permiten la primer reducción considerable de tiempo, y dos núcleos lógicos, que permiten la segunda reducción más pequeña. El incrementar a más de 4 hilos lo que va a hacer es gastar mayor cantidad de tiempo en cambios de contexto, por lo que se puede ver que el tiempo vuelve a aumentar.

B. MPI

En Fig. 5 y Fig. 6 se ve que el comportamiento en fue similar a OpenMP. Cuando se lanzan 1, 2 y 4 procesos los tiempos son muy similares a los de OpenMP y se debe a la misma explicación de la cantidad de núcleos. Sin embargo en este caso los llamados con 8 y 16 procesos no aumentaron el tiempo de forma tan rápida como en OpenMP, aunque se ve la tendencia a seguir aumentando.

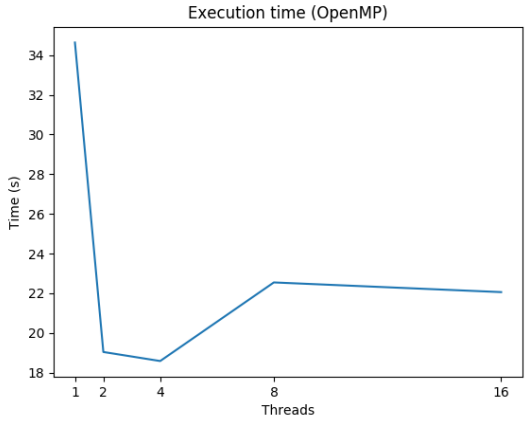

Fig. 3. Tiempo de ejecución en OpenMP

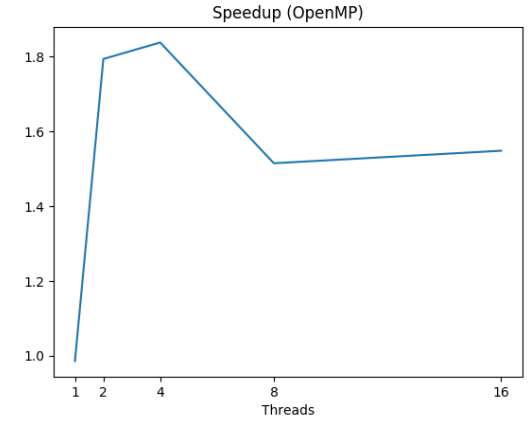

Fig. 4. Aceleración en OpenMP

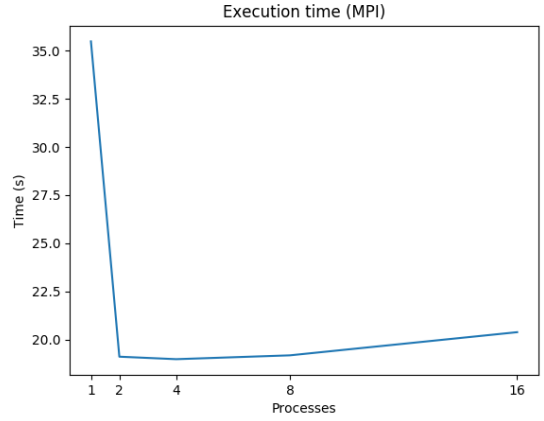

Fig. 5. Tiempo de ejecución en MPI

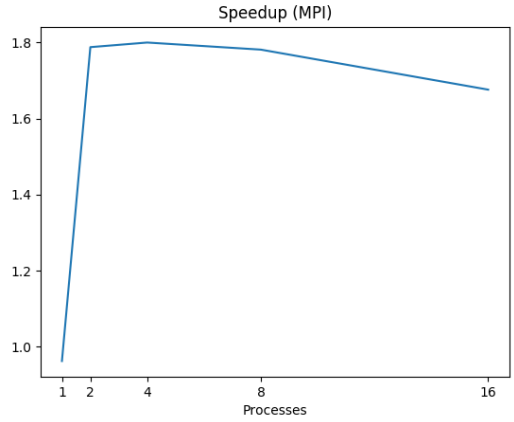

Fig. 6. Aceleración en MPI

C. CUDA

Entre Fig. 7 y Fig. 12 se muestra el comportamiento del tiempo y la aceleración para las diferentes configuraciones de bloques y total de hilos. Al utilizar pocos hilos los tiempos son mucho más grandes que la versión secuencial llegando a tardar casi 500 segundos. Esto se debe al costo que tiene todo el manejo de asignación, inicialización y copia de memoria que en las versiones anteriores no era necesario. A partir de entre 16 y 32 hilos ya se comienza a igualar el tiempo de ejecución secuencial. En general el tiempo comienza a reducirse muy rápido, llegando a alcanzar tiempos hasta 50 veces menores que la versión secuencial. Cuando la cantidad de hilos es muy grande el incremento de aceleración comienza a ser más bajo que al principio, mostrando una tendencia a estabilizarse y llegar a un punto máximo en el que no se acelera más. Esto se debe principalmente a que una mayor cantidad de hilos totales implica mayor uso y administración de memoria, llegando incluso a un punto en el que los recursos no son suficientes. En cuanto a la configuración de bloques el resultado muestra que para las cantidades de hilos más pequeñas resulta más rápida la ejecución si se utilizan más bloques. Sin embargo a medida que se usan más hilos (entre 4096 y 8192) la tendencia cambia y resultan más rápidas las configuraciones con menor cantidad de bloques.

D. OpenCL

Entre Fig. 13 y Fig. 15 se muestran los resultados de tiempo y aceleración utilizando diferentes configuraciones de work groups y work items totales. Inicialmente el programa es mucho más lento que la versión secuencial, alcanzando tiempos de ejecución mayores a los 300 segundos. Entre 8 y 16 work items ya se iguala el tiempo de ejecución de la versión secuencial, y se alcanza una aceleración máxima de aproximadamente 25. Inicialmente un uso de mayor cantidad de work groups significa una mayor aceleración. Sin embargo para una cantidad grande de work items hay un cambio en la tendencia, mostrando que resulta mejor utilizar dos veces el múltiplo de work groups recomendados para el dispositivo.

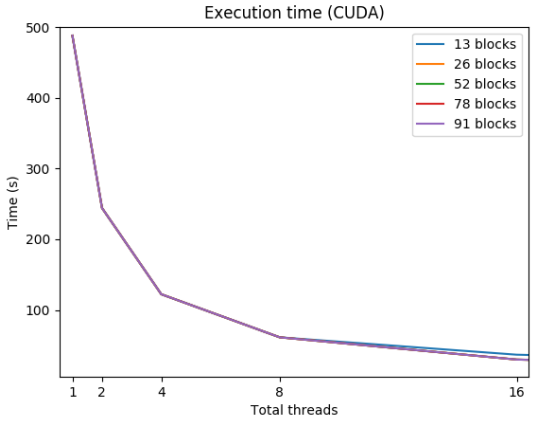

Fig. 7. Tiempo de ejecución en CUDA entre 1 y 16 hilos

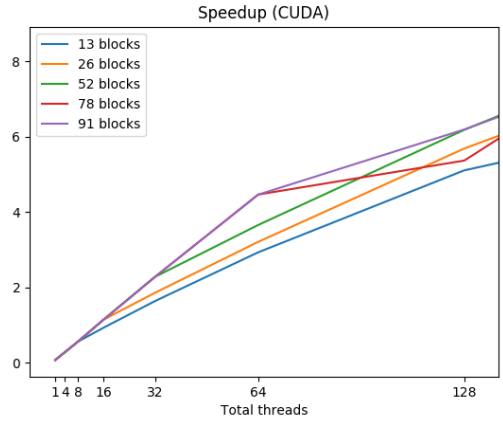

Fig. 10. Aceleración en CUDA entre 1 y 128 hilos

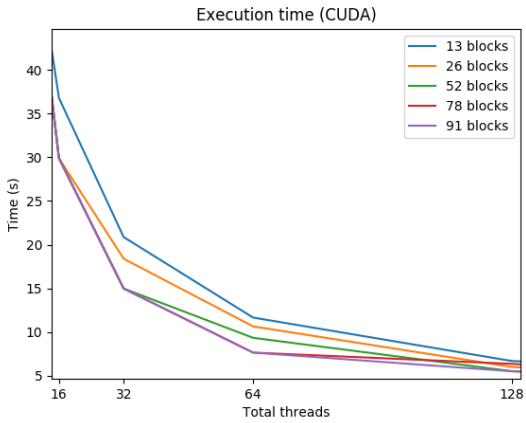

Fig. 8. Tiempo de ejecución en CUDA entre 16 y 128 hilos

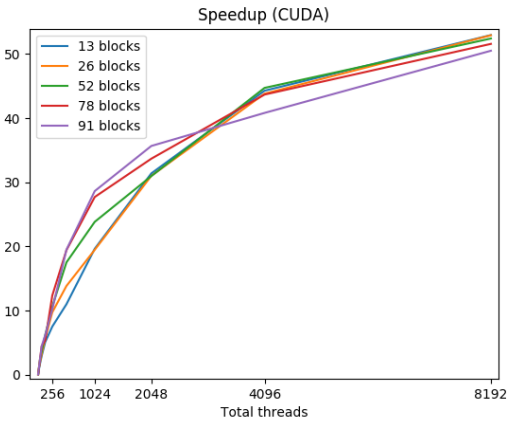

Fig. 11. Aceleración en CUDA entre 256 y 8192 hilos

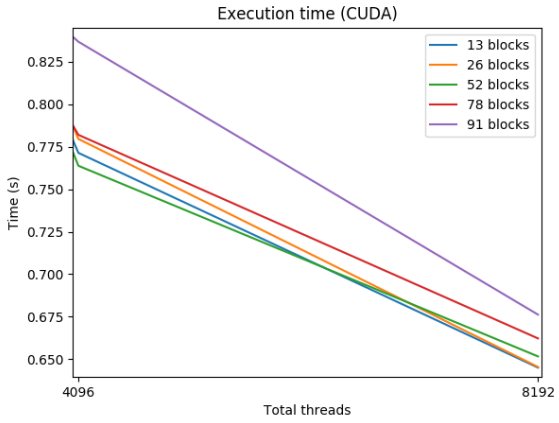

Fig. 9. Tiempo de ejecución en CUDA entre 4096 y 8192 hilos

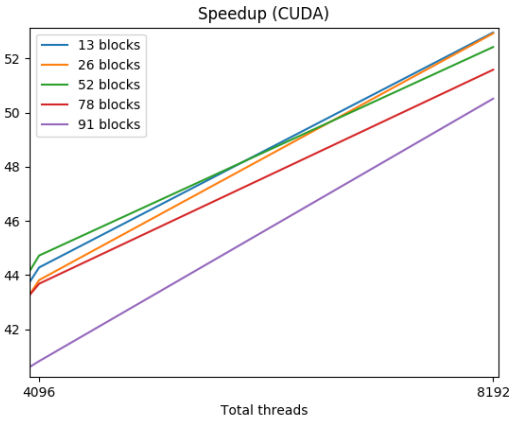

Fig. 12. Aceleración en CUDA entre 4096 y 8192 hilos

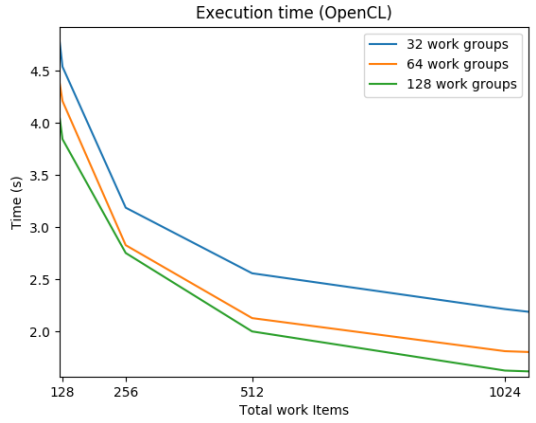

Fig. 13. Tiempo de ejecución en OpenCL entre 128 y 1024 work items

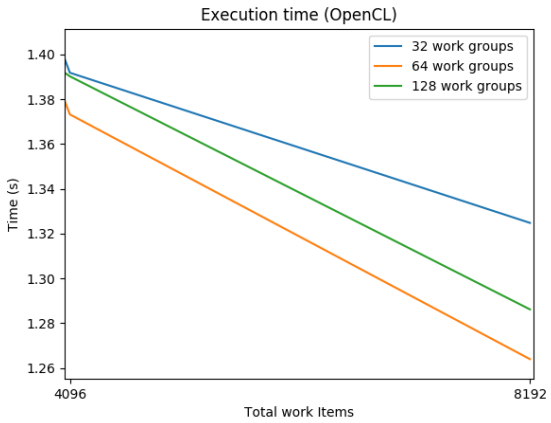

Fig. 14. Tiempo de ejecución en OpenCL entre 4096 y 8192 work items

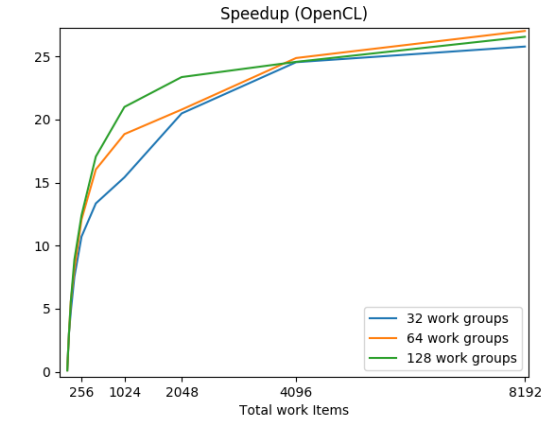

Fig. 15. Aceleración en OpenCL

IV. CONCLUSIONES

La búsqueda por fuerza bruta resultó ser una tarea notablemente paralelizable debido a la independencia de los cálculos necesarios para buscar colisiones de hashes y a la facilidad de distribuir las cargas de trabajo entre diferentes unidades de procesamiento. Los resultados iniciales de aceleración usando CPU tanto en OpenMP como en MPI fueron de 1.8 cuando su valor podía ser de máximo 2. Esta cercanía al valor ideal es muestra de que resulta eficiente aplicar computación paralela a esta tarea. En este caso las limitaciones de hardware no permitieron ver la tendencia que seguirán los tiempos de ejecución al aumentar la cantidad de núcleos físicos, pero se esperaría que los resultados sigan siendo buenos mientras que la carga de trabajo por cada unidad de procesamiento siga siendo considerablemente alta, lo cual depende completamente del formato de cadenas de texto sobre el cual se esté aplicando la búsqueda. El uso de GPU resultó ser igualmente muy bueno, llegando a valores máximos de aceleración de 50 y 25, usando CUDA y OpenCL respectivamente. En el caso de CUDA se obtuvieron los menores tiempos de ejecución utilizando una cantidad de bloques igual a la de multiprocesadores y con una gran cantidad de hilos totales (8192). Para OpenCL los mejores resultados se obtuvieron utilizando dos veces la cantidad de work groups recomendados para el dispositivo (32 y 64 en este caso). Ambas herramientas mostraron una tendencia a llegar a un punto máximo en el cual los tiempos ya no logran reducirse más debido a los costos adicionales en administración de memoria que se deben realizar en estas dos versiones.

REFERENCES

[1] Jun Zhang, Parallel Computing, Chapter 7, Performance and Scalability.
[2] Hashes leaks `https://hashes.org/leaks.php`
[3] Repositorio del proyecto `https://github.com/vgramirezc/brute-force-MD5-hash-searcher`