Mineria Bitcoin:
Rendimiento CPU vs GPU

Alejandro Gómez Ochoa
Alvaro Moreno Ribot
cuda0025

Índice

1. Introducción

Nuestra idea principal era encarar el proyecto hacia el análisis de algún algoritmo de encriptado (AES o ChaCha) pero posteriormente pensamos que sería mejor el uso de algoritmos de hashing ya que se puede hacer un mejor uso de las GPUs ya que están pensadas para hacer cálculos y procesos matemáticos específicos de forma extremadamente rápida y paralela (para el renderizado de imágenes) en comparación con una CPU.

Pensando en cómo podíamos aplicar un algoritmo de hashing en algo del mundo actual, lo primero que pensamos fue en la minería de criptomonedas. La minería de criptomonedas ha empezado a dar un segundo (y cada vez más común) uso a las tarjetas gráficas y teníamos serios intereses en saber cómo podía rendir una GPU (y varias) en una minería de una criptomoneda como puede ser Bitcoin. Así que al final nuestro tema es este.

1.1 Qué es Bitcoin y el Proof of Work

Bitcoin es un protocolo distribuido, de par a par. Como tal, no hay un servidor "central" o punto de control. No hay una sola entidad que emita y controle las transacciones. Básicamente podríamos decir que Bitcoin es un libro de cuentas distribuido y ligado (blockchain) que aprovecha la potencia de procesamiento de la red para asegurar las transacciones (Proof of Work).

Los Bitcoin se crean a través de un proceso llamado **minería**, este proceso implica competir para encontrar soluciones a un problema matemático (un **hash** determinado a partir de un bloque nuevo) mientras se procesan las transacciones de bitcoin. Esto es lo que llamamos **Proof of Work** o Prueba de Trabajo. Cualquier participante en la red de bitcoin (es decir, cualquier persona que utilice un dispositivo que ejecute el protocolo bitcoin) puede operar como un minero, utilizando la potencia de procesamiento de su ordenador para verificar y registrar las transacciones. Cada 10 minutos, en promedio, un minero de bitcoin puede validar las transacciones de los últimos 10 minutos y es recompensado con bitcoin nuevo.

El problema a resolver es encontrar, a partir de los datos del bloque actual más el hash resultado del bloque anterior (así quedan ligados, por eso se llama **blockchain**) y un campo donde los mineros pueden introducir una secuencia aleatoria, un hash que tenga un número determinado de 0s; el número de 0s es lo que llamamos dificultad. Hemos creado una figura para representar de forma gráfica esto que hemos explicado:

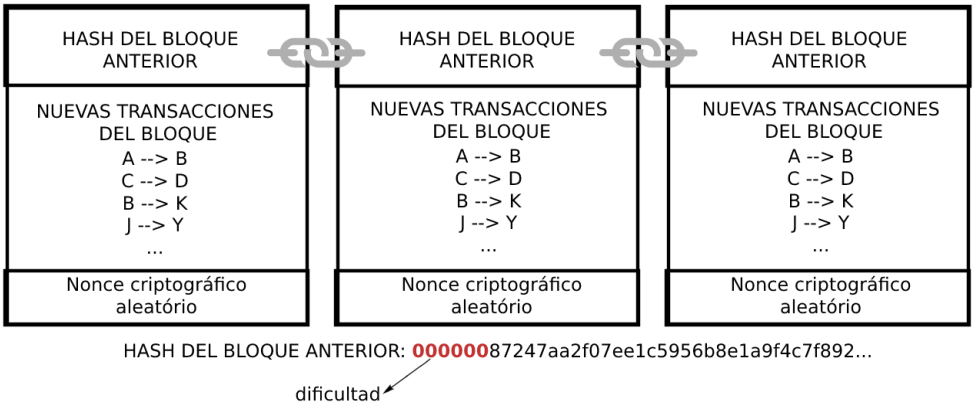

Figura 1

A pesar de ser computacionalmente difícil, las computaciones mineras pueden ser eficientes explotando el paralelismo masivo disponible en las GPUs. Utilizando la plataforma CUDA de Nvidia se puede lograr un speedup de más de 100 veces en comparación de una simple implementación de CPU secuencial.

1. 2 Motivación y objetivos

Nuestra principal motivación en este trabajo es experimentar con un tema que ya conocemos y que consideramos, además, que puede dar grandes frutos al combinarlo con lo aprendido en esta asignatura. De esta manera podremos también conocer dicho tema en mayor profundidad a nivel de programación, ya que era algo que no habíamos hecho hasta ahora.

1.3 Fuente del código usado

A pesar de que nuestro objetivo principal era desarrollar nuestro propio código para poder realizar este proyecto únicamente por nosotros mismos, nos hemos visto obligados a inspirarnos en un código ya existente debido a diferentes problemas que nos estábamos encontrando.

1.3.1 Problemas con el código

Al intentar compilar el código, empezamos a recibir distintos errores que debimos solucionar. El primer error, al intentar compilar con make all en el **boada-5** fue el siguiente:

```
nvcc -O1 -v -lrt -lm -arch=sm_20 -o gpu_miner main.cu utils.o sha256.o
nvcc fatal    : Value 'sm_20' is not defined for option 'gpu-architecture'
makefile:9: recipe for target 'gpu_miner' failed
make: *** [gpu_miner] Error 1
```

Figura 2

Analizando el error vimos que hay un problema con la especificación de la arquitectura de la gpu especificada. Buscando por internet la versión 2.0 no está soportada por las GPUs de Boada-5 por lo que lo hemos tenido que cambiar a la versión 3.5 (sm_35) que sí es compatible. Este cambio se ha tenido que introducir dentro del archivo **Makefile** en dos líneas distintas.

Una vez solucionado esto, volvimos a ver de nuevo un error; esta vez distinto. El compilador se quejaba de una librería que se incluía en el archivo `cuPrintf.cu` llamada `sm_11_atomic_functions.h` que es la librería que permite hacer uso de funciones atómicas con cuda (evitando data races sobre variables compartidas). La versión 1.1 está deprecada y nuestra versión `sm_35` no la soporta. Cambiamos la línea a **`#include <sm_35_atomic_functions.h>`** y compilamos de nuevo. Esta vez todo funciona en **boada-5** y ya tenemos los dos archivos **`cpu_miner`** y **`gpu_miner`** listos.

1.3.2 Solapamiento con otras asignaturas

Además de ciertos problemas a nivel de programación nos hemos encontrado con falta de tiempo, ya que a ambos se nos han solapado una gran cantidad de exámenes y trabajos este último mes y esto, sumado al resto de problemas que se nos generaban, terminó en la decisión de inspirarnos en un código externo.

2. Minando con CPU

Para minar con CPU usaremos el ejecutable **cpu_miner** que se ha creado después de hacer el make.

El minado de Bitcoin sigue un patrón muy simple que podemos ver en el pseudocódigo de la derecha. Se busca un bloque válido para minar, entonces dentro de un bucle el minero debe calcular el doble hash SHA-256 por cada posible valor del nonce y comparar los resultados con el nivel de dificultad actual (los 0s que hemos mencionado en la introducción).

```
while true
        if not block.isValid()
                block = getNewBlock()
        else
                block.updateTimestamp()
        for nonce = 0 to 2^32
                block.nonce = nonce
                hash = sha256(block)
                hash = sha256(nonce)
                if(hash < difficulty)
                        submitBlock(block)
                        block.setValid(false)
                        break
        end
end
```

Figura 3

El tiempo de cálculo del algoritmo secuencial está dominado por las dos computaciones de hash SHA-256 que deben ocurrir para cada uno de los valores del nonce. No vamos a entrar en más detalle del código ya que no es lo que nos interesa para este trabajo. Nosotros hemos intentado comprender las distintas partes del código ya que nos ha parecido muy interesante.

La característica importante a señalar es que este problema es altamente paralelizable ya que la computación SHA-256 que se debe calcular para cada valor de nonce es completamente independiente entre otros valores. También podemos remarcar que se puede encajar el problema cómodamente en la caché L1 debido a las pequeñas longitudes de los mensajes. Estas características hacen de la GPU un candidato ideal para la minería.

Así pues, hemos ejecutado el **cpu_miner** en el **boada-1**, como output nos da en número de hashes calculado y el tiempo de hash en microsegundos. A partir de estos datos hemos podido obtener la siguiente tabla de rendimiento:

# de hashes	Tiempo de hash	Hashes/s
1	0,003646	274,273
2	0,007088	282,167
4	0,01656	241,546
8	0,02833	282,386
128	0,464959	275,293
256	0,9117	280,794
512	1,882171	272,026
1024	3,63659	281,582
2048	6,299997	325,08

Figura 4

A partir de los datos de la tabla hemos podido obtener el siguiente gráfico:

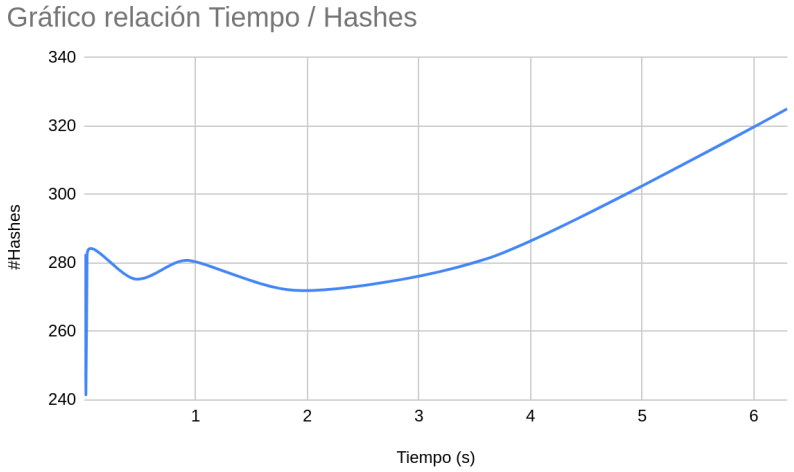

Figura 5

Vamos a ver ahora el código C de la implementación del pseudo código mencionado anteriormente. El código secuencial se encuentra en el archivo **serial_baseline.c**

```
for(i=0; i<32; i++) {
    tick();
    for(j=0; j<hashes; j++) {
        //Hash the block header
        sha256_init(&ctx);
        sha256_update(&ctx, data, 80);
        sha256_final(&ctx, hash);
        //Hash
        sha256_init(&ctx);
```

```
        sha256_update(&ctx, hash, 32);
        sha256_final(&ctx, hash);
```

Figura 6

Como vemos y como ya habíamos mencionado en el pseudocódigo, hasheamos el header del bloque dos veces.

```
//Check the difficulty
k=0;
while(hash[k] == difficulty[k]) k++;
if(hash[k] < difficulty[k]) {
    nr.nonce_found = true;
    nr.nonce = j;
    #ifdef MINING_MODE
    break;
    #endif
}
```

Figura 7

3. Minando con GPU vs diversas GPUs

A continuación utilizaremos el ejecutable **gpu_miner**. Comenzaremos haciendo uso de una única unidad de GPU para posteriormente ir incrementando el número, observando y comparando los distintos comportamientos y resultados que nos ofrecen todas las versiones.

Resultados obtenidos con una única GPU

Comenzaremos con una ejecución usando 32 threads / bloque. Para poder obtener unos resultados concretos y fiables, repetiremos esta ejecución hasta 5 veces, variando el número de bloques existentes. Comenzaremos con 1024 bloques e iremos incrementándolos en un 100%, de forma que después utilizaremos 2048, seguido de 4096, 8192 y 16984. Obtenemos los siguientes resultados:

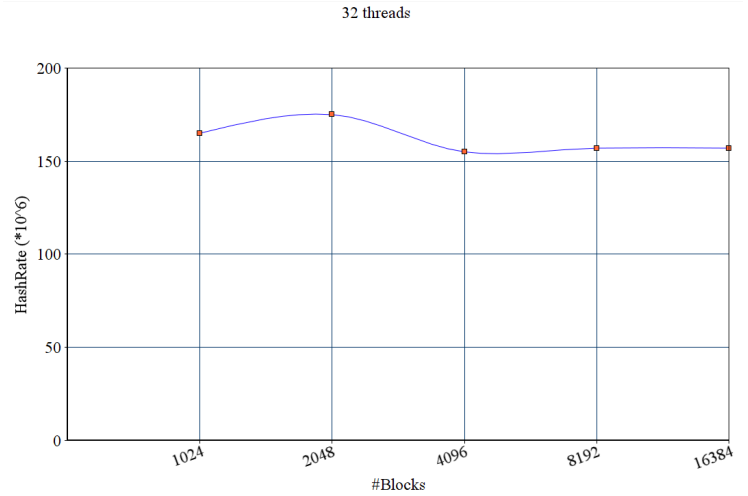

Figura 8

Posteriormente repetiremos el ejercicio incrementando el número de threads / bloque, también en un 100%. Ahora utilizaremos 64.

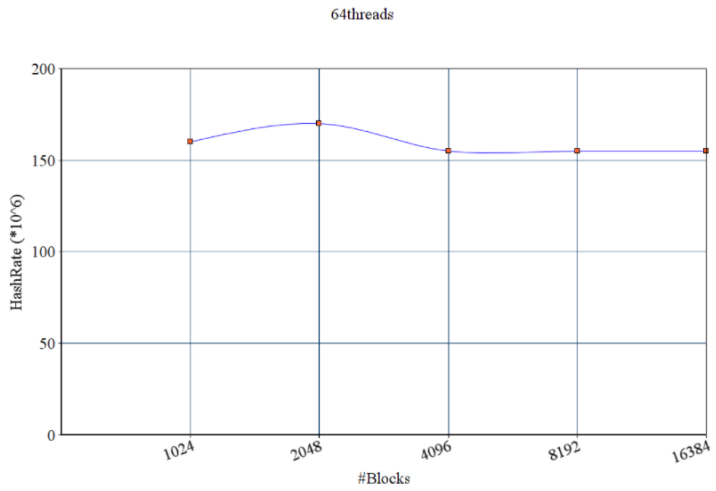

Figura 9

Y de nuevo repetiremos el procedimiento hasta en 2 ocasiones más, esta vez con 128 y 256 threads, obteniendo así los siguientes resultados:

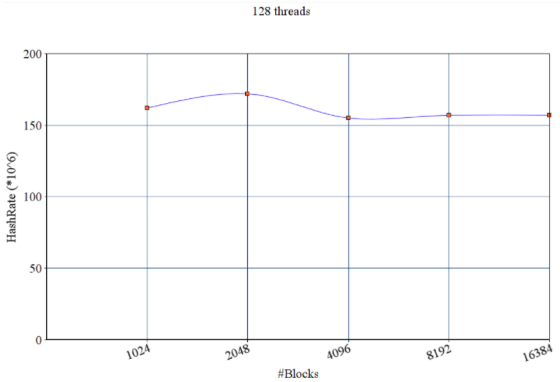

Figura 10

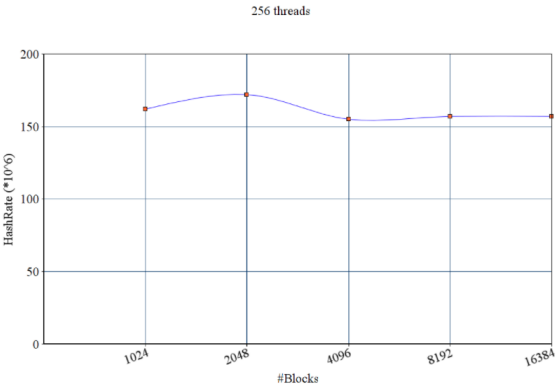

Figura 11

Podemos observar como el *Hashrate* no presenta demasiada variedad, por no decir que son prácticamente idénticos en cada ejecución. Podemos sacar la conclusión de que, si bien el cambio de datos iniciales provoca un cambio en la efectividad del miner, no lo hace con el rendimiento.

Mientras realizamos este proceso, nos percatamos de que en ciertas ocasiones, dependiendo del número de threads / bloque y del número de bloques, se podía encontrar o no el hash deseado, debido seguramente a que el código no está lo suficientemente bien adaptado a la posibilidad de realizar estos cambios con esta facilidad.

#Blocks / #Threads	32	64	128	256
1024	No encontrado	No encontrado	No encontrado	Encontrado
2048	No encontrado	No encontrado	No encontrado	Encontrado
4096	No encontrado	No encontrado	Encontrado	Encontrado
8192	No encontrado	Encontrado	Encontrado	Encontrado
16284	No encontrado	Encontrado	Encontrado	Encontrado

Figura 12

Código:

```
CUDA_SAFE_CALL(cudaMemcpy(d_ctx, (void *) &ctx, sizeof(SHA256_CTX), cudaMemcpyHostToDevice));
CUDA_SAFE_CALL(cudaMemcpy(d_nr, (void *) &h_nr, sizeof(Nonce_result), cudaMemcpyHostToDevice));

float elapsed_gpu;
long long int num_hashes;
#ifdef ITERATE_BLOCKS
//Try different block sizes
for(i=1; i <= 512; i++) {
    dim3 DimBlock(i,1);
#endif
    //Start timers
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    //Launch Kernel
    kernel_sha256d<<<DimGrid, DimBlock>>>(d_ctx, d_nr, (void *) d_debug);

    //Stop timers
    cudaEventRecord(stop,0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&elapsed_gpu, start, stop);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

#ifdef ITERATE_BLOCKS
    //Calculate results
    num_hashes = GDIMX*i;
    //block size, hashrate, hashes, execution time
    printf("%d, %.2f, %.0f, %.2f\n", i, num_hashes/(elapsed_gpu*1e-3), num_hashes, elapsed_gpu);
}
#endif
//Copy nonce result back to host
CUDA_SAFE_CALL(cudaMemcpy((void *) &h_nr, d_nr, sizeof(Nonce_result), cudaMemcpyDeviceToHost));
```

Figura 13

Es gracias a los timers inicializados (“//Start timers”) que podemos controlar los tiempos de ejecución y evaluar el rendimiento de nuestro ejecutable. Más adelante llamamos al kernel y le pasamos las medidas del Grid y del bloque ya definidas, además del *d_ctx* que contiene el header actual que se desea minar, i el *d_dbug* que contiene la información de control.
Es entonces cuando detenemos el timer y se calculan y guardan las diferencias de tiempo transcurrido para finalmente hacer un *flush* de las variables de inicio y fin.

Para terminar, se copia el *nonce* encontrado en el host para mostrarlo en los resultados, siempre y cuando se encuentre.

Resultados obtenidos con diversas GPUs

Ahora repetiremos todo el proceso anterior, aumentando el número de GPUs a utilizar.
Comenzaremos utilizando 32 threads / bloque y el mismo número distinto de bloques que antes; 1024, 2048, 4096, 8192 y 16384, obteniendo los siguientes resultados:

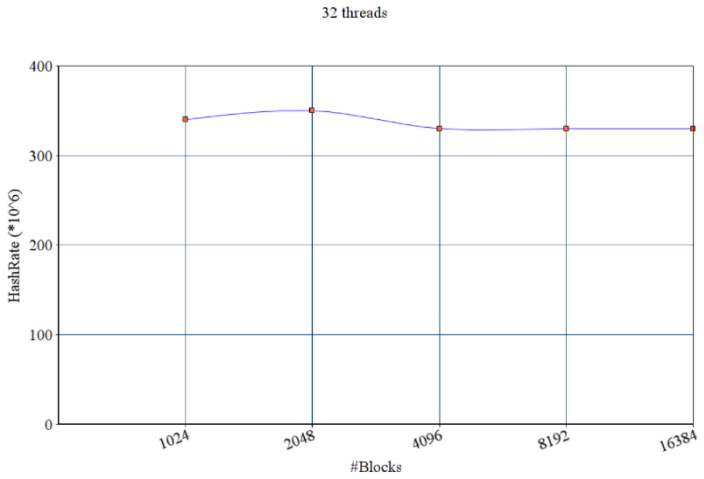

Figura 14

Aumentaremos de nuevo el número de threads / bloque a 64 gerando lo siguiente:

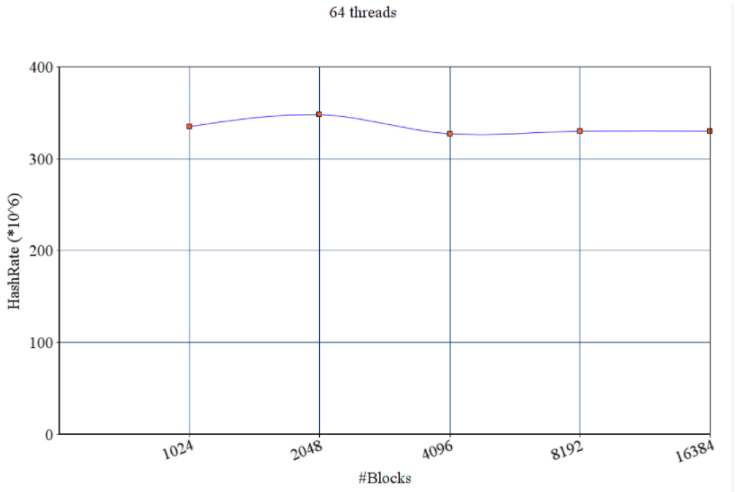

Figura 15

Y lo repetiremos una última vez, con 128 y 256 threads / bloque:

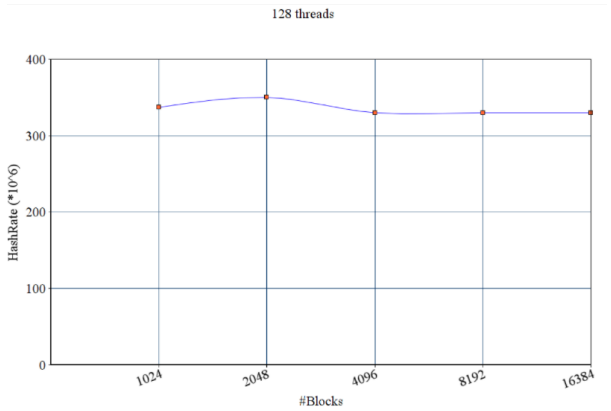

Figura 16

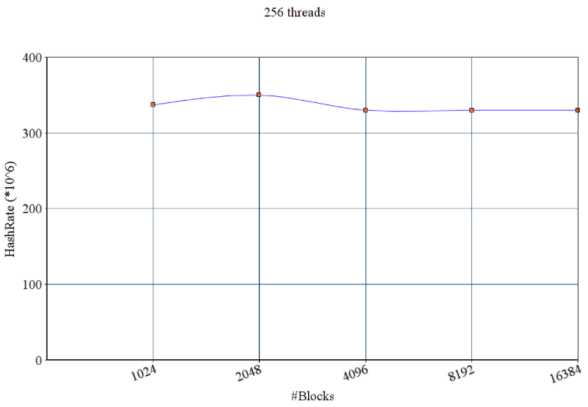

Figura 17

Vemos otra vez como los resultados son increíblemente parecidos, manteniéndose aproximadamente todos los casos en los mismos valores. Es decir, vemos otra vez como el cambio de variables no afecta al rendimiento del programa.

De nuevo, hemos visto cómo en ciertas ocasiones no se podía encontrar el hash deseado dependiendo del número de threads / bloque:

#Blocks / #Threads	32	64	128	256
1024	No encontrado	No encontrado	No encontrado	Encontrado
2048	No encontrado	No encontrado	No encontrado	Encontrado
4096	No encontrado	No encontrado	Encontrado	Encontrado
8192	No encontrado	No encontrado	No encontrado	Encontrado
16284	No encontrado	No encontrado	No encontrado	Encontrado

Figura 18

En este caso se puede deber tambien a que el programa conste de un timer que proque el aborto de la ejecución. Eso, sumado a que el tiempo que tarda en encontrar un bloque sigue una distribución exponencial y que el programa mantiene un comportamiento imprevisible en ciertas configuraciones, hace que no sea un resultado demasiado extraño.

Código:

```
float elapsed_gpu;
long long int num_hashes;
#ifdef ITERATE_BLOCKS
//Try different block sizes
for(i=1; i <= 512; i++) {
    dim3 DimBlock(i,1);
#endif
    //Start timers
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    //Launch Kernel
    cudaSetDevice(0);
    kernel_sha256d<<<DimGrid, DimBlock>>>(d_ctx_0, d_nr_0, (void *) d_debug);

    cudaSetDevice(1);
    kernel_sha256d<<<DimGrid, DimBlock>>>(d_ctx_1, d_nr_1, (void *) d_debug);

    cudaSetDevice(2);
    kernel_sha256d<<<DimGrid, DimBlock>>>(d_ctx_2, d_nr_2, (void *) d_debug);

    cudaSetDevice(3);
    kernel_sha256d<<<DimGrid, DimBlock>>>(d_ctx_3, d_nr_3, (void *) d_debug);

    //Stop timers
    cudaEventRecord(stop,0);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&elapsed_gpu, start, stop);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

#ifdef ITERATE_BLOCKS
    //Calculate results
    num_hashes = GDIMX*i;
    //block size, hashrate, hashes, execution time
    printf("%d, %.2f, %.0f, %.2f\n", i, num_hashes/(elapsed_gpu*1e-3), num_hashes, elapsed_gpu);
}
```

Figura 19

Nos encontramos con que este código es muy similar al anterior. Para distribuir correctamente el trabajo entre 4 aceleradores, hacemos uso de *cudaSetDevice* antes de llamar al kernel, además de crear sus propias variables de contexto *d_ctx_i* y *dn_nr_i* para cada GPU.

Además también hacemos uso de *cudaSetDevice* cuando hacemos el *allocate* del espacio a memoria global, cuando se establecen los tiempos de inicio y fin en cada acelerador y cuando se copian los resultado de cada GPU. Encontramos una única diferencia, entonces, con respecto al código anterior; estas gestiones se hacen 4 veces en vez de una, una vez para cada acelerador.

4. Conclusiones

Podemos concluir con el estudio que hemos hecho que el aumento de GPUs beneficia positivamente al rendimiento en cuando a Hashes por segundo, pues el hashrate sube casi multiplicándose por 2. Y si comparamos con los resultados obtenidos por CPU la diferencia es abismal. Realmente estos son los resultados que esperábamos y los que habíamos podido ver en algunas páginas de cálculo de Hash que hemos encontrado por internet.

Así que el uso de GPUs y la paralelización nos brindan una gran mejora en el rendimiento en cuanto a la computación rápida de hashes. Por este mismo motivo, se pueden usar las GPUs para crackear contraseñas a partir de hashes filtrados con diccionarios de palabras.