DANIEL MONJAS MIGUÉLEZ
70274432-W
12/05/2022

ACAP: PRÁCTICA 3

Explicación del código original:

Si nos fijamos en el código original vemos que en la función *main()* lo que se hace es declarar una imagen, que es un puntero a *char* y que se inicializa con la función i*nicializarImagen()*. Esta función lo que hace es reservar en memoria dinámica un vector de *char* y se inicializa cada posición del vector a un número entero aleatorio entre 0 y *IMDEP-1*, que en este caso es 255. Tras esto se ejecuta la función *histogramaCPU()*, esta función tiene como parámetro un vector de *int* de tamaño el mismo que valores pueden tomar los píxeles de la imagen, es decir, en este caso 256 (del 0 al 255 ambos incluidos). Tras esto se recorre la imagen y en cada posición del vector se almacena el conteo de píxeles que almacenan el mismo valor que la posición del vector. Por ejemplo, si en un píxel se encuentra el valor 200, en la posición 200 del vector se aumenta en uno el valor almacenado. Todo este cálculo se ha hecho con la CPU.

Tras esto empezarán los cálculo y las operaciones con la GPU. En primer lugar se hacen dos llamadas a *cudaMalloc()*, la primera es para reservar en el dispositivo espacio para un vector lineal de *char* de tamaño *SIZE*, y se apunta *dev_imagen* a dicha memoria reservada. El segundo es para reservar espacio para el histograma. Se reserva espacio para un vector de *IMDEP* enteros sin signo y se apunta *dev_histo* a dicha memoria reservada. Tras el *cudaMalloc()* de la imagen se hace un *cudaMemcpy()* para copiar la imagen ya inicializada de la memoria del heap a la memoria del dispositivo que hemos reservado.

Una vez inicializado lo necesario se inicializan dos objetos *cudaEvent_t* que se usarán para posteriormente registrar el tiempo de cálculo empleado por la función *kernelHistograma*. Tras esto se hace un bucle for de 11 iteraciones, donde la primera no se tiene en cuenta, sino que es sólo para poner la gráfica en marcha. En cada iteración efectiva se hace un *cudaMemset()*, para poner todas las posiciones de *dev_histo* a 0. Después se hace un *cudaDeviceSynchronize()* , para esperar a que la gráfica haya terminado su actividad y esté lista para su uso. Tras esto se guarda el instante previo a la ejecución de la función de cálculo. Seguidamente se ejecuta la función de cálculo, y se le pasan el número de bloques y el número de hebras por bloque que se desean utilizar (siempre deben estar dentro de los límites permitidos por la gráfica, o sino dará error), donde *NBLOCKS*THREADS_PER_BLOCK* será el número de hebras que se van a utilizar para la ejecución de nuestro kernel. Además se pasan como parámetros, la imagen, su tamaño y el histograma.

KernelHistograma inicializa una variable en memoria compartida de cada bloque, que es un vector de enteros sin signo de tamaño *IMDEP*, y se inicializa a 0 cada posición de esta variable. El como se inicializa en el código original es un error, ya que aprovecha de que coincide de que tenemos tantas hebras por bloque como posiciones tiene el vector, de forma que lo que hace es que cada hebra inicialice la posición de su id dentro del bloque a 0. Esto no funcionaría se hubiera más hebras o menos hebras por bloque que posiciones tiene el vector. Para solucionarlo lo que se hace es un bucle for el cual empieza en el identificador de cada hebra dentro del bloque y cada hebra pone a 0 las posiciones *threadIdx.x+n*blockDim.x* donde *n* va desde 0 hasta el máximo número entero donde la fórmula anterior sea menor que IMDEP.
Tras esto se ejecuta *__synthread()* para sincronizar las hebras de un mismo bloque, se almacena en *i* el identificador de la hebra dentro del número total de hebras lanzadas y se almacena en *offset* el número total de hebras lanzadas.

Finalmente en un bucle while lo que se hace es, que si tu identificador de hebra dentro del total de las lanzadas, es menor que el tamaño de la imagen, esta hebra suma 1 atómicamente a la posición del vector que coincida con el contenido del píxel, y actualiza su valor de *i* sumándole el número total de hebras lanzadas. Una vez terminado el bucle *while* se vuelve a ejecutar *__synthread()* para sincronizar las hebras y finalmente se pasan los resultados locales hebra a hebra que se tenían en la variable en memoria compartida *temp* al histograma con la función *atomicAdd()*. De nuevo esto es un error, pues si el número de hebras por bloque no coincide con el tamaño del histograma no funciona. Volvemos a hacer un bucle for con la misma estructura que el usado para inicializar, es decir cada hebra hace la suma atómica de las posiciones *threadIdx.x+n*blockDim.x*. Darse cuenta que esto funciona aunque se lancen más hebras que tamaño de la imagen, ya que las hebras que su identificador global quede fuera del tamaño de laimagen simplemente tienen a cero esto valores, luego solo suman 0 al histograma. Con esto termina el *kernel* y se vuelve a main.

En *main* se recoge el evento de finalización, se espera a que termine el evento *stop* y se guarda en *milliSeconds* los milisegundos que hay entre los eventos *start* y *stop*. Esto se suma a la variable *aveGPUSUMS*, que contendrá la suma de los milisegundos de cálculo empleados en las 10 iteraciones. Tras el bucle for se destruyen los objetos de tipo *cudaEvent*. Se copia desde el dispositivo a la memoria del host le histograma recién calculado (el histograma se reestablecía a 0 en cada iteración del for), y se compara si coincide con el calculado por la CPU, imprimiendo el Check-sum y si este es correcto en ambos casos. Por último se imprime el tiempo media que ha tardado la GPU, para ello se toma *aveGPUMS* y se divide entre el número de iteraciones del for para obtener la media y se divide entre 1000 para pasar a segundos. Finalmente liberamos los objetos imagen y dev_imagen y dev_histo, finalizando así el programa.

RESULTADOS GENMAGIC.UGR.ES

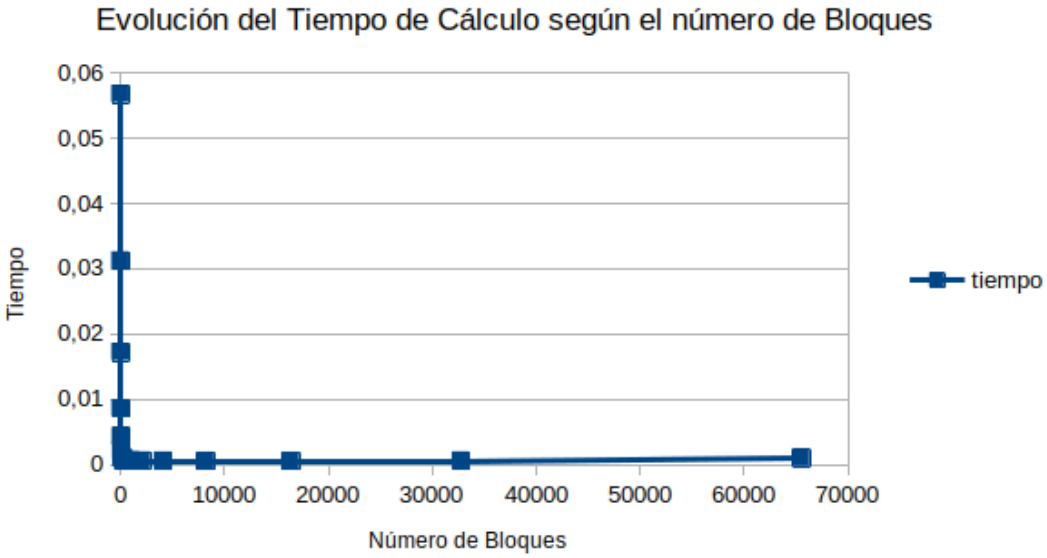
En primer lugar he consultado las características de la CPU de GENMAGIC, para ver cuantos CUDA cores posee y cuales son los tamaños de bloque y de número de hebras por bloque máximo que se pueden usar obteniendo la siguiente recopilación:

```
estudiante20@genmagic:~/PRACTICA_3$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce RTX 2080 Ti"
  CUDA Driver Version / Runtime Version          11.4 / 11.6
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 11018 MBytes (11553341440 bytes)
  (068) Multiprocessors, (064) CUDA Cores/MP:    4352 CUDA Cores
  GPU Max Clock rate:                            1545 MHz (1.54 GHz)
  Memory Clock rate:                             7000 Mhz
  Memory Bus Width:                              352-bit
  L2 Cache Size:                                 5767168 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.4, CUDA Runtime Version = 11.6, NumDevs = 1
Result = PASS
estudiante20@genmagic:~/PRACTICA_3$
```

- GENMAGIC dispone de 4352 CUDA cores, respartidos en 68 SM de 64 CUDA cores cada SM
- El tamaño máximo de hilos por bloque es (1024, 1024, 64).
- El tamaño máximo de grid, es decir, de número de bloques es (2147483647, 65535, 65535)

Teniendo esto en cuenta, para el primer experimento lo que he hecho es ir aumentando el tamaño de bloque en forma de potencia de 2, manteniendo fijo el número de hebras por bloque a 256 (el que venía en el código original), de forma que he obtenido los siguientes datos.

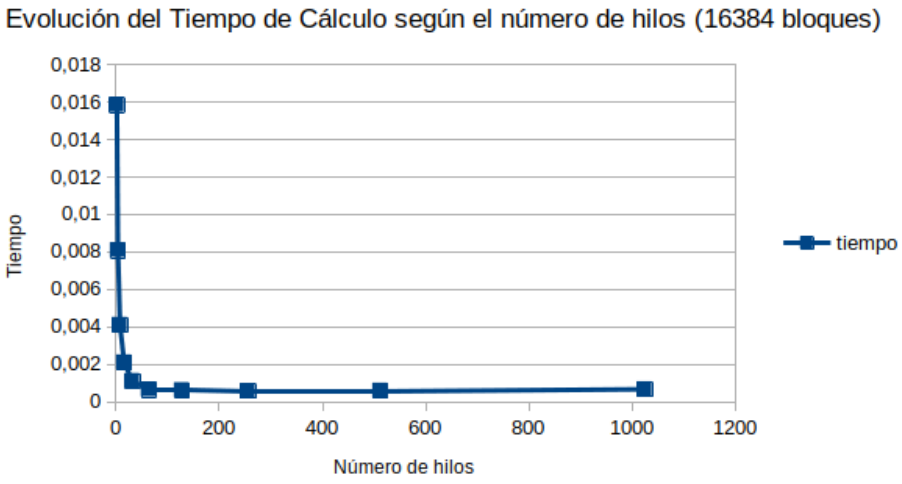
N_BLOCK	TIEMPO
2	0,056721411
4	0,031231686
8	0,017218781
16	0,008626823
32	0,004332608
64	0,002183065
128	0,001146886
256	0,000676083
512	0,000659453
1024	0,00065344
2048	0,000627206
4096	0,0006052
8192	0,000593357
16384	0,000587315
32768	0,00062223
65536	0,001037603



Fijándonos, vemos que el mejor tiempo se obtiene para 16384 bloques, y viendo que luego comienza a empeorar, aunque en la gráfica este empeoramiento es casi despreciable a la vista.

Una vez obtenido el mejor número de bloque lo fijamos y repetimos el experimento, esta vez cambiando el número de hebras por bloque. Lo hacemos de la misma forma que en el experimento anterior, variando en potencias de dos, llegando a un tope de 1024, donde ya nos da error la gráfica por los límites vistos. Estos son los resultados:

N_THREAD	TIEMPO
2	0,015839813
4	0,008088611
8	0,004106525
16	0,002104883
32	0,001111344
64	0,000660074
128	0,000625146
256	0,000585882
512	0,00060031
1024	0,000679434



De nuevo si nos fijamos vemos que el mejor tiempo se obtiene para 256 hebras por bloque y que tras este empiezan a empeorar los tiempos, aunque vuelve a ser despreciable a la vista en la gráfica.

Con estos dos experimentos he obtenido la mejor configuración, cuyo tiempo se usará para calcular la ganancia respecto al tiempo obtenido por la CPU. Los tiempos obtenidos por la CPU, y por la versión general (no la mejor configuración con los siguientes).

TiempoCPU	0,160549879
TiempoGen	0,004348934
TiempoTrans	0,013097937

- **TiempoCPU**: tiempo de cálculo de la CPU
- **TiempoGen**: tiempo de cálculo de la GPU para la versión general del programa sin la mejor configuración
- **TiempoTrans**: tiempo de cálculo teniendo en cuenta las transferencias de la GPU

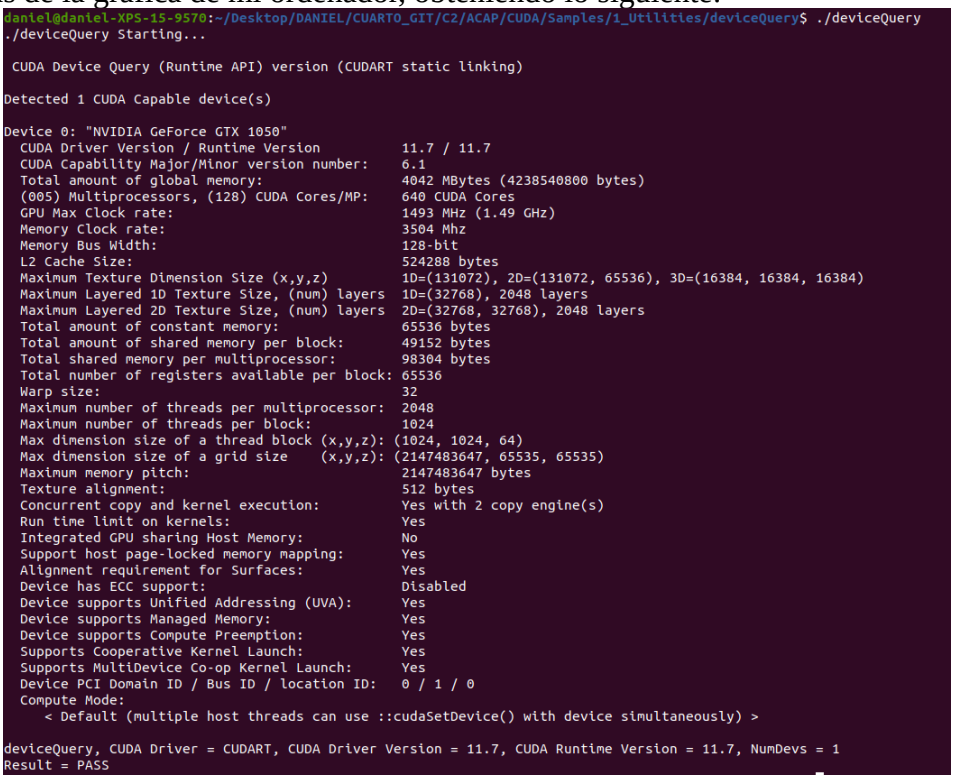
De esta forma obtenemos que:

- La ganancia obtenida por la GPU para la versión general es de 36,91706496
- La ganancia obtenida por la GPU para la mejor configuración es de 274,0310831
- La ganancia obtenida para la mejor configuración teniendo en cuenta las transferencias es de 12,2576463

Para tener en cuenta las transferencias lo que se ha hecho es extender el bucle for que se usaba para promediar el tiempo de cálculo de forma que también incluya las operaciones de transferencia. Antes de entrar al bucle se crean dos objetos de tipo cudaEvent_t y en cada iteración del bucle se toma una medida nada más empezar y una medida al final, tras liberar con cudaFree. Tras esto se hace la media de dicho bucle y obtenemos el tiempo medio de ejecución con transferencias. Además se han quitado aquellas líneas de código que conllevaban tiempo, pero no era trabajo realizado por la GPU, como pueden ser los printf innecesarios o el cálculo y comparación del checksum.

RESULTADOS ORDENADOR

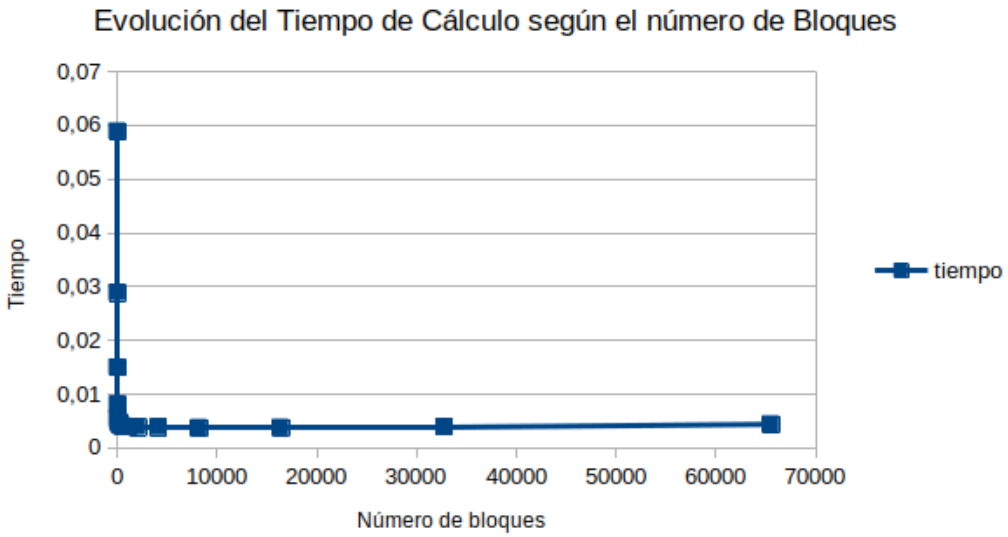
Repitiendo el proceso seguido para GENMAGIC.UGR.ES, lo primero que he hecho es consultar las características de la gráfica de mi ordenador, obteniendo lo siguiente:



- Mi gráficaes una GTX 1050 con 640 CUDA cores repartidos en 5 multiprocesadores con 128 CUDA cores cada multiprocesador.
- El número de hilos por bloque máximo es (1024, 1024, 64)
- El número de bloques máximo es (2147483647, 65535, 65535)
- Destacar que la frecuencia de reloj de memoria es más rápida en GENMAGIC y el bus de memoria también es más ancho, lo que puede justificar los resultados que vamos a ver, junto con la gran diferencia de CUDA cores disponibles.

He repetido el proceso seguido para GENMAGIC y he ido tomando los tiempos que se obtienen al ir aumentando el número de bloques en potencia de dos, llegando a los siguientes resultados:

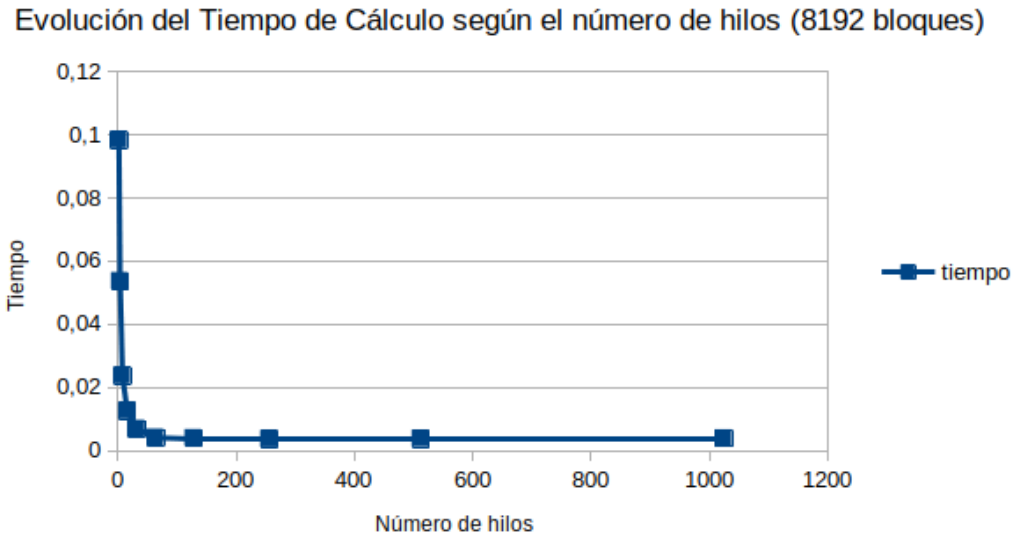
N_BLOCK	TIEMPO
2	0,058860803
4	0,028827316
8	0,014957974
16	0,008017338
32	0,00542809
64	0,004964378
128	0,004751735
256	0,00430688
512	0,004066345
1024	0,003910854
2048	0,003845274
4096	0,003810032
8192	0,003698186
16384	0,003740352
32768	0,003900563
65536	0,004371783



Se puede apreciar que el mejor tiempo se ha obtenido para 8192 bloques, y que tras este el tiempo comienza a aumentar. De nuevo, como ocurría con GENMAGIC, el aumento de tiempo es tan poco significativo que no se aprecia en la gráfica.

Una vez establecido cual es el mejor número de bloques, repetimos el proceso para el número de hilos por bloque, de nuevo aumentando este en potencia de dos, y llegando a los siguientes resultados:

N_THREAD	TIEMPO
2	0,098296387
4	0,05364563
8	0,023725807
16	0,012604922
32	0,006936509
64	0,004148211
128	0,00402352
256	0,003686698
512	0,003766096
1024	0,003950979



De aquí se obtiene que al igual que ocurría en GENMAGIC el mejor número de hilos por bloque es 256. Con esto he obtenido cual es la mejor configuración número de bloques, número de hilos por bloque en mi ordenador. Con eso y los tiempos obtenidos vamos a ver cual es la aceleración respecto al tiempo obtenido por el procesador:

TiempoCPU	0,160480976
TiempoGen	0,004616624
TiempoTrans	0,020527306

Habiendo usado los mismo nombres que se han usado para GENMAGIC las aceleraciones obtenidas son:

- La aceleración de la versión general respecto al tiempo de CPU es 34,7615435
- La aceleración de la mejor configuración respecto al tiempo de CPU es 43,52973203
- La aceleración de la versión con transferencias respecto al tiempo de CPU es 7,81792681416646

Vemos que las aceleraciones obtenidas son mucho peores que las obtenidas por GENMAGIC, menos en el caso general (esto se puede deber a que la configuración general no utiliza todo el potencial de ninguna de las dos gráficas, luego no se da una gran mejora). Esto se puede deber a lo ya comentado previamente. La aceleración en las versiones que tienen solo en cuenta el tiempo de cálculo puede ser peor debido a que la gráfica de GENMAGIC dispone de casi 7 veces más CUDA cores que la gráfica de mi ordenador, lo que le puede permitir realizar más trabajo en paralelo. Por otro lado la versión con transferencias puede ser peor porque la frecuencia del reloj de memoria y el ancho del bus de memoria son en ambos casos mucho mayores en la gráfica de GENMAGIC.