

UNIVERSITAT DE BARCELONA

PROGRAMACIÓN PARALELA

Introducción a CUDA

Laboratorio 2

Trabajo realizado por Marcos Plaza

7 de Mayo del 2021

Introducción de la práctica

En este segundo laboratorio de la asignatura, utilizamos el mismo algoritmo de conversión de un espacio de color a otro, aunque esta vez existe un cambio de paradigma. Hasta el momento habíamos visto como exprimir los recursos de la *CPU* para aplicar paralelismo mediante la librería de *OpenMP*. En esta práctica, no tan solo usaremos la *CPU* sino que también aprenderemos el entorno de programación *CUDA* para aplicar el algoritmo mencionado mediante el uso de las *GPU* de *nVidia*. Veremos también como podemos mejorar progresivamente el rendimiento de la gráfica de un apartado a otro, teniendo en cuenta distintos factores; como los accesos a memoria, uso de la memoria compartida (*shared memory*) o el uso de *streams*.

De manera secundaria, también nos familiarizaremos con el entorno basado en celdas de *google collab*, el cual nos facilita el uso de *CUDA* y la implementación del algoritmo, sin necesidad de disponer de una *GPU* directamente.

En este informe iremos comentando las soluciones propuestas para cada apartado. El problema de la conversión de color se basa en una imagen con resolución *4K*, y en contraste con la anterior práctica (donde el espacio "origen" era *GRB*) se va de un espacio *BRG* a *RGBA*.

A continuación, apuntamos el modelo de gráfica *nVidia* (junto con algunas de sus especificaciones) con el que se han ejecutado todas las secciones:

Device name: Tesla T4
Memory Clock Rate (KHz): 5001000
Memory Bus Width (bits): 256
Peak Memory Bandwidth (GB/s): 320.064000

Realización de los ejercicios

Sección 1 En primer lugar se pide completar algunos métodos ya declarados; `allocGPUData`, `copyAndInitializeGPUData` y `freeCUDAPointers`. Hace falta recordar que si estamos realizando llamadas a la *CUDA Runtime API*, en este caso para las funciones de gestión de memoria. Conviene "envolver" la llamada con `CU_CHECK`. Dichas funciones siempre devuelven un código de error. De este modo, si no se devuelve `cudaSuccess`, podemos conocer que tipo de error se ha producido.

- `allocGPUData` para reservar memoria en el dispositivo¹. En nuestro caso conviene reservar los punteros para la imagen en ambas codificaciones.
- `copyAndInitializeGPUData` copiará la información de la imagen en *GBR* inicializada en el *host*, enviándola al *device*. Acto seguido inicializa a 0 los punteros para la imagen *RGBA*.
- `freeCUDAPointers` libera la memoria reservada en el dispositivo. Internamente se emplea el `cudaDeviceReset`, con lo que hacemos es destruir y liberar todos los recursos asociados con el dispositivo actual, en el proceso actual.

Una vez comprobada la correcta implementación de dichos métodos, entre otros, todos ellos incluidos en `experiment.h`, pasamos a la implementación de la función de conversión; `convertBRG2RGBA`. En *OpenMP* paralelizamos los bucles *for*, pero como ya he mencionado en la introducción existe un cambio en la visión de como programar en este entorno. Ahora lo que tenemos son los llamados *kernels*. Cada *kernel* será ejecutado por los hilos de cada `block`, de cada `grid` definidos antes del procesado. Para esta sección implementamos un *kernel* bidimensional, donde emplearemos las coordenadas *2D* de cada hilo para indexar cada píxel en la imagen y realizar la conversión. Para seleccionar dichas coordenadas, aprovechamos la fórmula vista en clase para identificar unívocamente cada *thread*:

```
int x = threadIdx.x + (blockIdx.x * blockDim.x);
int y = threadIdx.y + (blockIdx.y * blockDim.y);
```

De este modo pasamos de la posición relativa del hilo en cada `block`, a la posición "general" dentro de cada `grid`.

Con esta implementación, para 100 iteraciones, obtenemos un tiempo de ejecución de 48666 μ s y **Executed!! Results OK..**

¹Dispositivo es otra forma de referirnos a la tarjeta gráfica o *GPU*.

Sección 2 En esta sección se pide adaptar de nuevo el código para una `grid` unidimensional. Tal y como se puede ver en la función `executeKernelconvertBRG2RGBA`, pasamos a tener un `block` y una `grid` cuyas dimensiones vienen definidas por la constante con valor 256 `BLOCK_SIZE`. También se pide probar con diferentes valores para esta constante.

En el *kernel* `convertBRG2RGBA`, modificaremos el código anterior tratando ambas imágenes como vectores unidimensionales. Pasaremos de tener coordenadas `int x`, `int y`, a un único valor `int x`. Por tanto únicamente emplearemos:

```
int x = threadIdx.x + (blockIdx.x * blockDim.x);
```

Si comparamos los resultados con el apartado anterior, para un número diferente de iteraciones (con un `BLOCK_SIZE = 256`), obtenemos los siguientes tiempos:

Iteraciones	Sección 1 (μs)	Sección 2 (μs)
100	48666	39876
1000	420091	368419
10000	3693547	3599177

Como se puede observar en los anteriores resultados, existe una mejora significativa por lo que respecta al primer apartado. Hemos de revisar un concepto que ya salió en la anterior práctica; *localidad de datos*. Las lecturas entre posiciones de memoria consecutivas se dan más fácilmente al tratar la imagen como array unidimensional, en contraste a utilizar x e y para indexar cada píxel.

Por otro lado si variamos el tamaño de cada bloque o `BLOCK_SIZE`:

BLOCK_SIZE	Sección 2 (μs)
256	39876
512	42209
1024 (max)	43650
128	39378
64	40031

Si tomamos como mejor en rendimiento, el modelo de la sección 2 y tan solo variamos el tamaño del bloque, obtenemos el mejor tiempo en dividir entre dos el valor inicial, es decir en `BLOCK_SIZE = 128`.

Sección 3 Para este apartado continuaremos con el `BLOCK_SIZE` que nos dió mejores resultados en el anterior. Trataremos de optimizar los accesos a memoria en contraste con las anteriores secciones, con la condición de no emplear aún la *memoria compartida*. La solución propuesta para este apartado pasa por, inicialmente, calcular la coordenada del píxel mediante la formula que hemos utilizado con anterioridad. En la sección dos, estamos realizando demasiados accesos a la *memoria global* del dispositivo. En concreto para recuperar cada uno de los valores de cada canal del píxel *BRG*, para acabar asignándolos al nuevo píxel en codificación *RGBA* (añadiendo 255 para el factor de opacidad), uno por uno. La solución que se propone es; traer de golpe el `uchar3 brg[x]` (donde *x* es el índice del píxel dentro de la imagen) a una variable estática temporal `tmp`, con lo que reducimos los accesos a *memoria global*. Para asignar el nuevo píxel convertido a *RGBA*, tan solo debemos utilizar `make_uchar4` además del índice *x*, accediendo a los valores del temporal (a nivel de memoria reduce el coste de acceso al estar en *memoria privada*) uno a uno. Para ver mejor esta conversión, se detallan las dos siguientes líneas:

```
if (x < width * height) {
    uchar3 tmp = brg[x];
    rgba[x] = make_uchar4(tmp.y, tmp.z, tmp.x, 255);
}
```

El tiempo de ejecución se reduce significativamente a un valor de $29125\mu s$.

Por otro lado, se comenta si existe algun problema con los accesos a memoria. A parte de que, al principio, se realizan demasiados accesos, si tomamos en cuenta que la memoria de la *GPU* esta estructurada en *palabras* de 4 bytes en 4 bytes, existirá un problema parecido al que ya vimos en el primer laboratorio con *OpenMP*. Anteriormente lo remediamos empleando el `__attribute__((aligned(4)))`, de manera que, los píxeles que ocupaban 3 bytes quedaban alineados y reducíamos los ciclos por acceso a un número justo y necesario. En esta ocasión ocurre lo mismo. La mayoría de ellos, realizará dos accesos por píxel dado que cada píxel en *BRG* ocupa 3 bytes, y como hemos dicho la estructura de la memoria es de 4 bytes en 4 bytes. Pero esta vez el compilador *nvcc* no nos ayuda de la misma manera, con lo que no podemos obtener lecturas coalescentes. Este es el principal problema que se solucionará con la implementación de las siguientes secciones.

Sección 4 La explicación de la solución a el problema anterior se hace explícita en el *pdf* del enunciado de la práctica. Como vemos en la solución, se propone traer la información de la imagen *BRG* (donde cada píxel supone un espacio de 3 bytes) en grupos de 4 bytes. De esta manera podemos tener un único acceso por *thread*, para leer la totalidad de los datos de la imagen en *BRG*. Para realizar este "truco", lo que haremos será calcular la posición de cada hilo cuando leemos *BRG* como si tuviera elementos de 4 bytes, mediante la constante `N_ELEMS_3_4_TBLOCK`. Con este cálculo, únicamente utilizamos el 75% de los hilos de cada bloque, con lo cual, se requiere de un número mayor de bloques para traer toda la imagen. Calcularemos el índice de cada lectura, por lo tanto, de la siguiente forma:

```
int position = threadIdx.x + (blockIdx.x * N_ELEMS_3_4_TBLOCK);
```

En esta sección, transferimos los datos desde *device memory* a *shared memory*, una memoria de menos latencia y de menos tamaño, pero que a la vez es compartida por todos los bloques de un *Streaming Multiprocessor*. Es decir, cada *thread* de un bloque particular posee una memoria compartida en común con los otros *threads* del mismo bloque. Debemos tener cuidado y saber buscar el balance en la *memoria compartida* usada para cada *threadblock*, ya que como he dicho esta memoria esta físicamente compartida por los mismos.

Guardaremos cada *palabra*, al array de *memoria compartida* con nombre `bgrShared`. Después de esta transferencia de datos de *memoria global* a *compartida*, debemos explicitar una barrera para sincronizar todos los hilos de ejecución, con tal de asegurarnos que los datos son correctos antes de proceder. Una vez sincronizamos, continuamos recalculando la posición de cada píxel para asignar el nuevo bloque de *RGBA*. Para ello, utilizamos de nuevo la función `make_uchar4`.

Como hemos podido ver, hemos solucionado el problema de la alineación de memoria para la lectura coalescente de cada *thread*. Por lo que los accesos para traer cada píxel del vector `bgr`, disminuyen a 1 por *thread* en lugar de (en la mayoría de casos) 2.

Por otro lado;

- utilizamos más accesos a memoria (en concreto dos accesos a *memoria compartida* adicionales, aunque si es verdad que son más "baratos"),
- utilizamos la sincronización de hilos mediante el `syncthreads`,
- y usamos un mayor número de bloques (los cuales deben sincronizarse y a la vez compartir memoria).

Parece que la solución al anterior problema sale mas cara que usar la *memoria global* del dispositivo. Como consecuencia, este nuevo *kernel* disminuye el rendimiento en contraste al apartado anterior, con un tiempo de $32300\mu s$. En el próximo apartado, se implementa un algoritmo en el que, cada tres bloques de 4 bytes, se genera la información necesaria para la conversión de 4 píxeles por un mismo *thread*. Utilizaremos también *memoria compartida*.

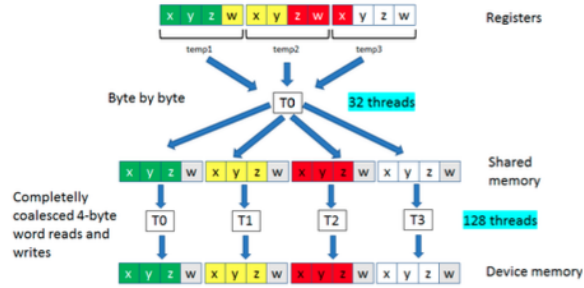


Figure 2: Segunda parte del algoritmo. Conversión y escritura en memoria

De esta manera, estamos conservando siempre la coalescencia de las lecturas y las escrituras. Cabe decir que para realizar la escritura de *memoria compartida* a *memoria global*, escribiendo desde `pix_write` al vector `rgba`, hace falta de nuevo, sincronizar hilos y recalcular las posiciones de cada hilo que vaya a realizar la transferencia. Con esto, junto a los accesos a memoria añadidos en el ejercicio, el tiempo sube un poco más que en el anterior apartado ($36050\mu s$), aunque como siempre se obtiene **Executed!! Results OK..**

Sección 6 Por último, en este ejercicio se pide implementar el uso de *streams*, mediante un ejemplo dado. Cabe observar que el *kernel* será el mismo que en el ejercicio 3, que es el ejercicio con el menor tiempo de ejecución en la conversión. Pero como he dicho, esta vez contaremos con *streams*.

Dichos *streams* sirven para que la ejecución de la *CPU* al mandar las instrucciones desde el *host code* no se detenga y así optimizar los tiempos. Para ello debemos sobrescribir las operaciones implementadas en el primer ejercicio; `allocGPUData`, `copyAndInitializeGPUData`, y `freeCUDAPointers`, mediante el uso de operaciones no bloqueantes como:

- `cudaMemcpyAsync`
- `cudaMemsetAsync`

Mediante las pertinentes modificaciones del código con el *stream* no bloqueante y diferente a 0, para una sola iteración y para el mismo `BLOCK_SIZE = 128`, obtenemos un tiempo de conversión igual a $500\mu s$. Por otro lado, el tiempo empleado por la *CPU* para la comunicación con la *GPU* es de $27170\mu s$.

Si establecemos el *stream* por defecto obtenemos los siguientes tiempos; para el tiempo de conversión $510\mu s$ y para el chrono establecido para el ejercicio $26856\mu s$.

Los tiempos de ejecución en la implementación con *streams* no es la esperada. Así como en el ejemplo propuesto, los tiempos con *streams* son bastante menores, en la implementación del algoritmo de conversión de color, no mejora. Todas las copias de datos de *CPU* a *GPU* deberían realizarse superpuestas con otras operaciones como el lanzamiento de *kernels* de otros *streams*. Esto puede deberse a que únicamente empleamos un único *stream*. Como no hay otro proceso en *CPU* que utilice otro *stream*, las llamadas a dichas operaciones se realizarán de manera secuencial de la misma manera que se haría en el *stream* por defecto.

Conclusiones

Finalmente comparamos todas las implementaciones para 100 iteraciones y con `BLOCK_SIZE` variables.

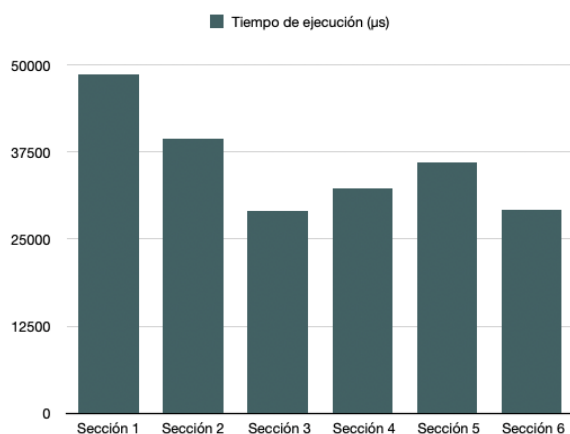


Figure 3: Comparación del rendimiento de los diferentes apartados

Del gráfico observamos que la solución más eficiente es la implementada en el tercer apartado, donde únicamente se accede a la *memoria global* del dispositivo para lectura y escritura una sola vez. Se carga el píxel en *memoria privada* y como cada píxel es independiente, cada *thread* puede realizar esta tarea en paralelo, utilizando a su vez `make_uchar4` para incluir el valor en `rgba`. Es una solución no demasiado compleja, en comparación al algoritmo propuesto en la sección 5.

Por otro lado, hemos podido familiarizarnos con el entorno de programación *CUDA*, así como las posibilidades que nos brinda. Accesos a memoria, estructura de memoria, lecturas coalescentes, *threads*, *bloques*, *grids*, jerarquía de memoria. Todos estos conceptos y más, se han revisado y estudiado pertinentemente mediante este laboratorio.

Como síntesis final, el objetivo de la computación paralela utilizando este tipo de dispositivos y entornos, consiste en mantener dichos recursos ocupados al máximo, de manera eficiente. A su misma vez la gestión de la memoria y la comprensión del problema a tratar, son siempre asuntos muy a tener en cuenta.

Algunos links de referencia

- Diapositivas de la teoría del campus virtual.
- Repaso CUDA.
- Mejores prácticas.
- Streams.