Práctico 3 del curso GPGPU

2023

Grupo 10

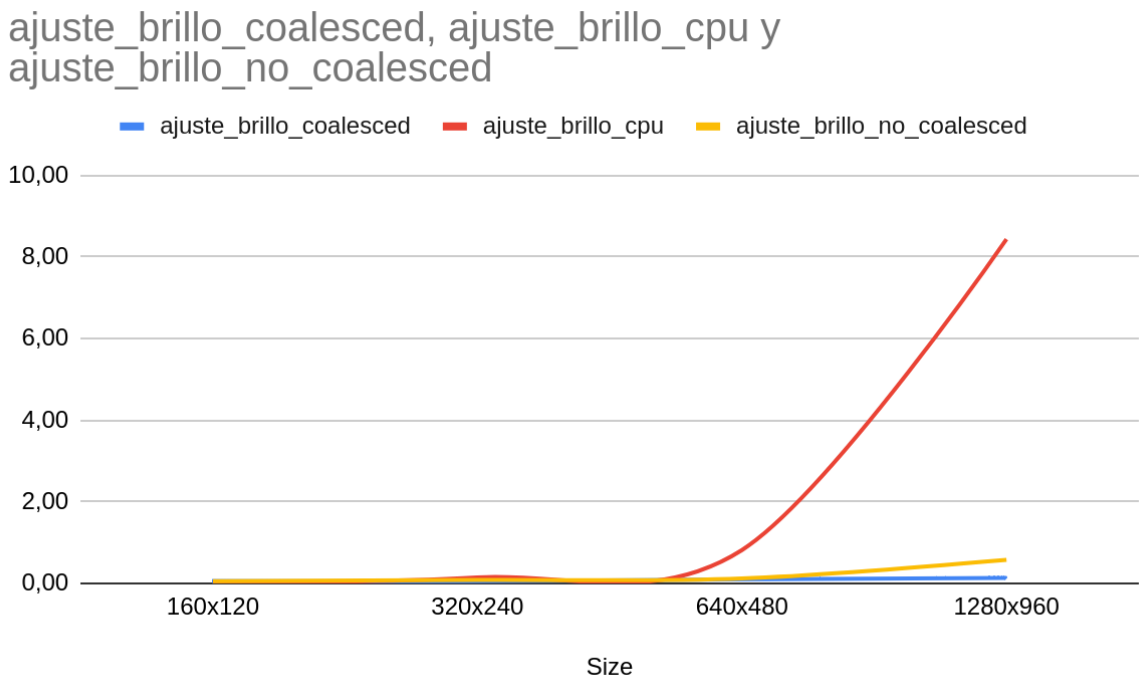
Daniel Padron Simon	5.147.163-4	daniel.padron@fing.edu.uy
Bruno Cabrera Martínez	5.397.585-8	bruno.cabrera@fing.edu.uy

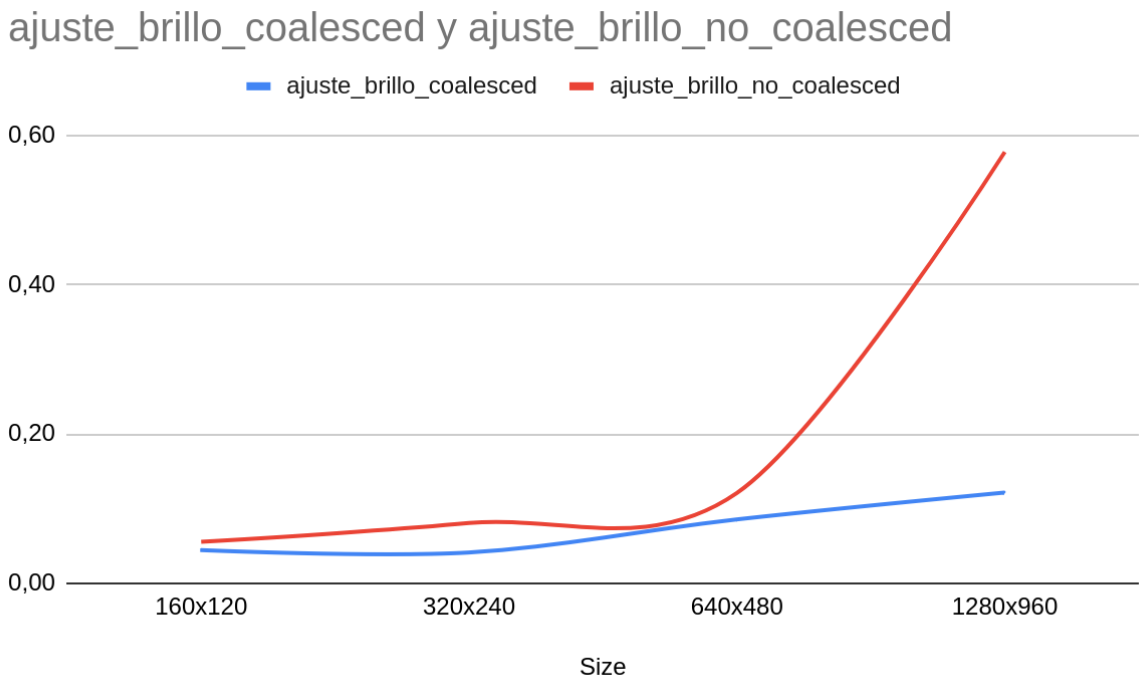
Ejercicio 1

Parte a)

Se realizan 2 versiones del algoritmo de suma, uno coalesced y otro no coalesced. Se ejecuta en la misma imagen con las siguientes dimensiones: 160x120, 320x240, 640x480, 1280x960.

En las siguientes gráficas se puede observar el tiempo de ejecución (en ms) de los algoritmos para las dimensiones mencionadas. En la primera también se incluyen los datos del algoritmo en CPU, pero es necesario aclarar que para GPU no se están tomando en cuenta los tiempos de copia de y a memoria de GPU, por lo que únicamente estamos comparando el poder de procesamiento para este caso.

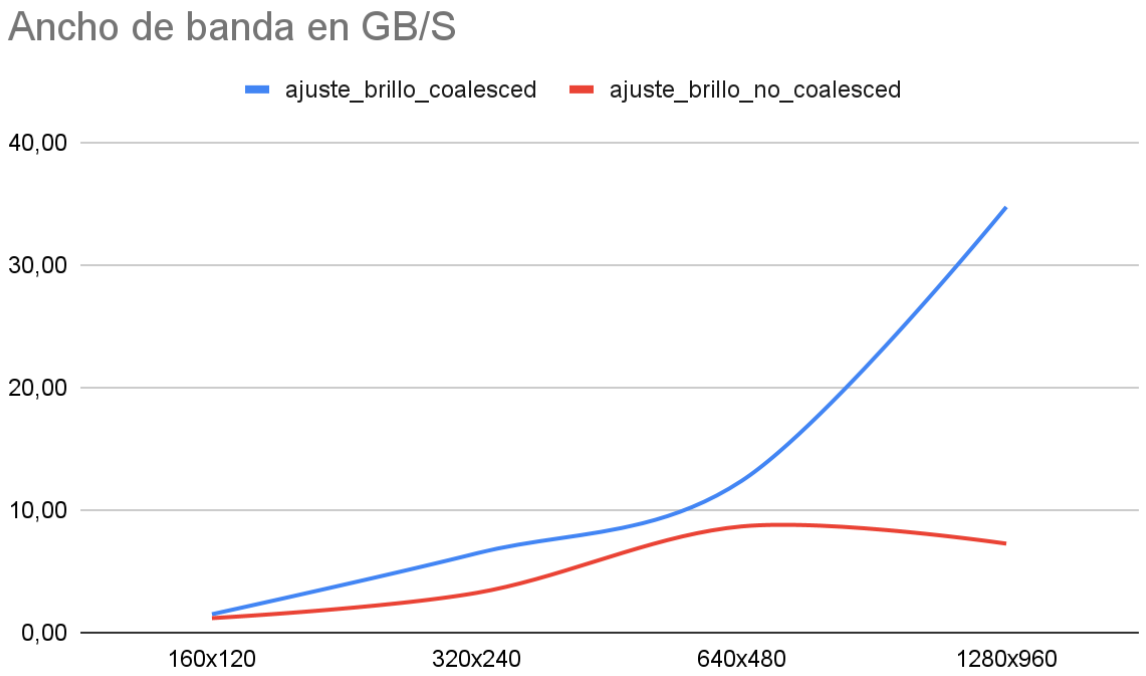




Al comparar entre el algoritmo coalesced y el no coalesced se puede notar una mayor eficiencia en el coalesced. El algoritmo coalesced es el que mejor realiza los accesos a memoria dentro del GPU. Este algoritmo utiliza bloques de 32x16, donde cada thread procesa 1 pixel correspondiente a su posición, entonces en cada warp los accesos se realizan de forma contigua y alineada. En este caso cada thread lee un pixel (flotante 4 bytes), podemos asumir que cada transacción de lectura a memoria global lee en palabras de 32 bytes, así que por cada warp se realizaran 4 * 32 -> 4 transacciones.

En el caso del algoritmo no coalesced, utilizamos bloques de tamaño 32x16, donde cada thread procesa el pixel con posición transpuesta (Ej: Si el thread dentro del bloque es él (x,y), entonces procesa el pixel (y,x)). De forma que, cada hilo en un warp lee en una fila distinta de la imagen. Si el ancho de la imagen es mayor a 32 bytes, podemos ver que cada hilo ejecuta una transacción distinta, 32 * 32 -> 32 transacciones.

Si bien en la gráfica se puede notar la diferencia en la performance entre ambos algoritmos, no es tan amplia como la diferencia entre cantidad de transacciones. Esto se puede deber a la existencia de caché en la GPU. Como se puede ver a medida que el tamaño de la imagen aumenta se comienza a notar la diferencia en la performance.



En esta gráfica se puede observar la cantidad de GB procesados por segundo, notamos la misma tendencia que en las gráficas anteriores. El algoritmo coalesced procesa una mayor cantidad de datos por segundos que el no coalesced, por lo que es más eficiente.

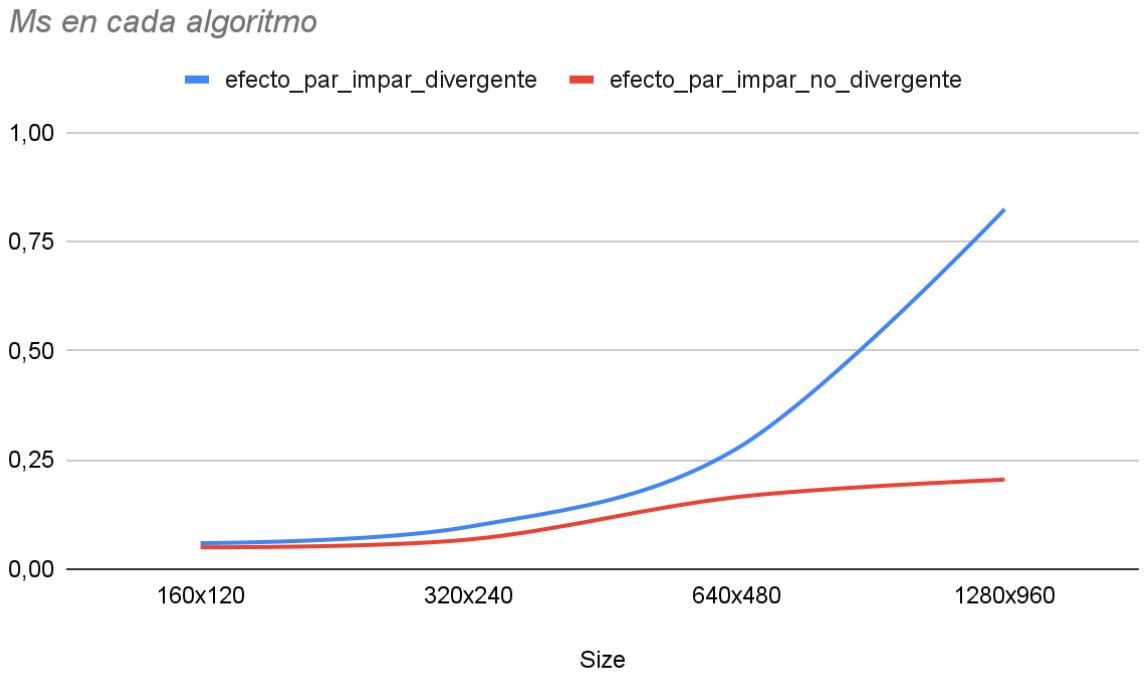
Parte b)

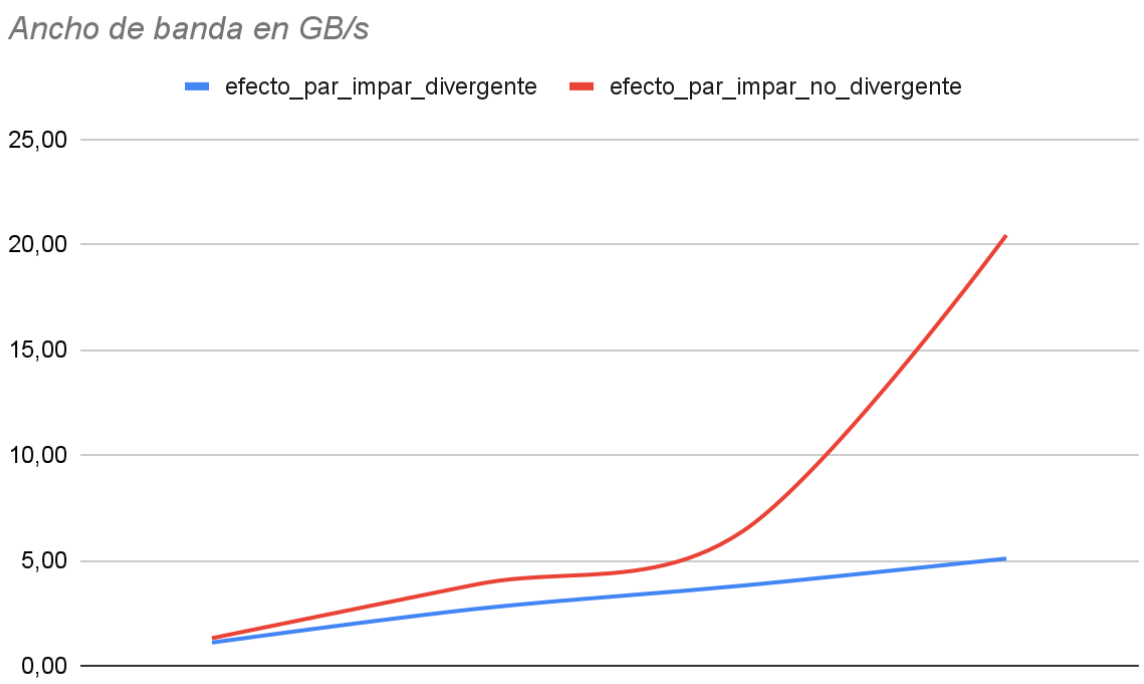
Se realizan 2 algoritmos, uno sin divergencias y otro con divergencias.

El algoritmo consiste en sumarle o restarle el coeficiente en 10 iteraciones al valor de cada píxel dependiendo de si el mismo es par o impar. (Esta suma se realiza dentro de un for para evitar optimizaciones del compilador.

La versión sin divergencias calcula un valor 1 o -1 dependiendo de si es par o impar (sin usar if - else), mientras que la que posee divergencias usa un if - else.

En la siguientes gráfica se muestra la performance (en ms y en GB/s) de ambos algoritmos en la misma imagen con las siguientes dimensiones: 160x120, 320x240, 640x480, 1280x960.





Se puede observar que el algoritmo no divergente es más eficiente que el divergente. Cada warp de un bloque en GPU es de 32 hilos y se puede asumir que ejecutan la misma instrucción del kernel a la misma vez. Cuando el kernel posee puntos donde la siguiente instrucción varía dependiendo el hilo en un mismo warp ocurre una divergencia. El warp se parte en 2, y cada divergencia se ejecuta por separado hasta que el código se vuelva a unir. Este tipo de ejecución genera que queden cuda cores inactivos, por lo que no son eficientes. Este es el caso del algoritmo “efecto_par_impar_divergente”. Podemos ver

como su ancho de banda es menor y su tiempo de ejecución es mayor al del algoritmo no divergente.

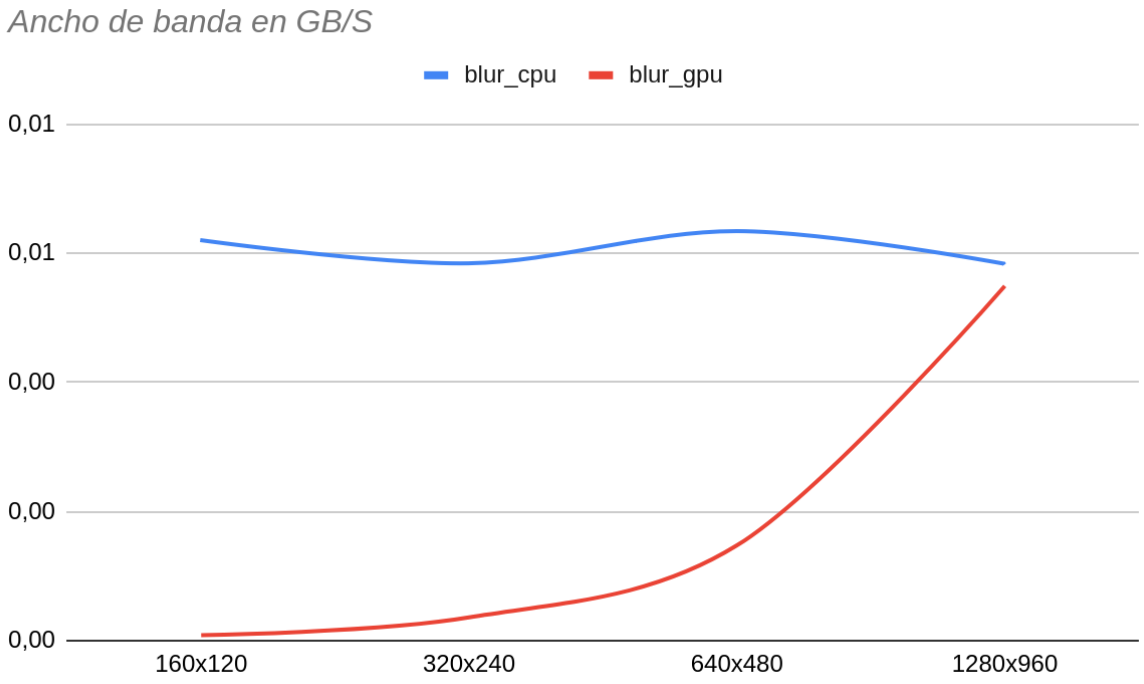
Ejercicio 2

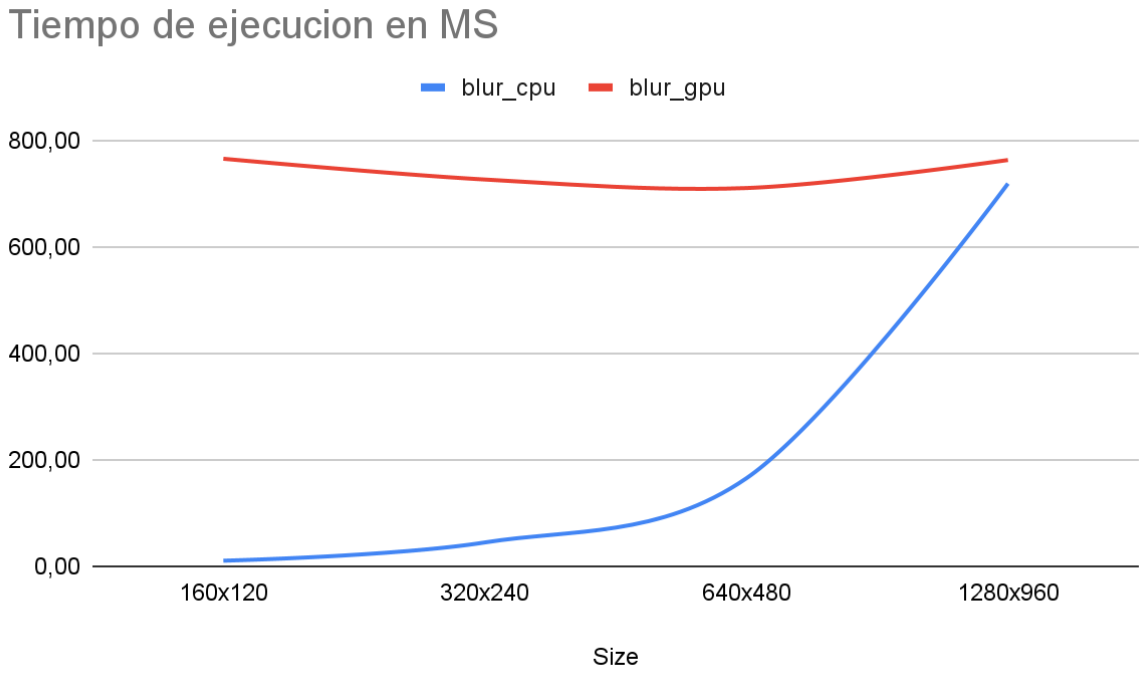
Parte a)

Se construye el kernel (GPU) solicitado en blur_gpu dentro de blur.cu. El resto de las operaciones (por ejemplo transferencia) se realizan en main_blur_gpu (nuevamente en blur.cu). El kernel no se programó de forma más simple posible, donde cada thread (dentro de un bloque de 32x16) se encarga de leer todos los píxeles que requiere para llevar a cabo su cálculo. Creemos que el uso de la memoria compartida para este algoritmo sería muy provechoso debido al número de veces que los datos de un mismo pixel es consultado.

Parte b)

La construcción del análogo para CPU de la parte anterior fue construido.
Se han ejecutado ambas versiones con los tamaños de imagen 60x120, 320x240, 640x480, 1280x960, los resultados en cuanto a tiempo de ejecución (ms) y ancho de banda (GB/s) se visualizan en las gráficas a continuación.





Lo primero que podemos observar es algo que en un primer momento podría parecer desconcentrarte, la CPU en algunos casos es más veloz que la GPU!.

La realidad es que esta gráfica tiene en cuenta los tiempos de transferencia que se deben hacer previo y posterior a ejecutar el kernel en la GPU, es más, este tiempo es uno de los más significativos a considerar.
Cuando el tamaño de la imagen es muy pequeña, el tiempo que “ganamos” al utilizar la potencia de la GPU, no comparable con el tiempo perdido en la tranferencia. A medida que el tamaño de la imagen aumenta, el esfuerzo que la CPU requiere es cada vez mayor, pero con la GPU se mantiene considerablemente estable, tanto es así que para el tamaño de 1280x960 no vemos una diferencia sustancial entre ambas, y para 160x120 la CPU es 76.5 veces más rápida que la GPU (765ms para la GPU, 10ms para la CPU).
Esta misma tendencia se observa para la tasa de transferencia.
Por lo tanto, siguiendo con esta tendencia, no seria de extrañar que para tamaños mayores, la GPU consiga mejores velocidades en comparación con la GPU.

Es una clara reflexión que hacemos es que: aunque la GPU sea mejor para determinados tipos de algoritmos, los tiempos de transferencia son determinantes, por lo que hay que evaluar si para un volumen de datos dado, realmente conviene ejecutarlo en la GPU o en la CPU.