Informe práctico 3

Maximiliano Bove - Mauricio Vanzulli
Curso: Computación de Propósito General en Unidades de Procesamiento Gráfico.
9 de mayo de 2021

Plataforma computacional:

Este trabajo se ejecutó en un sistema operativo Linux con arquitectura de 64 bits en un procesador i5-8250U a una frecuencia del reloj de 1.6 GHz y una memoria RAM de 8 Gb. Se utilizó una tarjeta gráfica NVIDIA GM108M [GeForce MX130] basada en la arquitectura Maxwell con compute capability 5.3.

Ejercicio 1

Para este ejercicio se implementó la función *ajustar_brillo_gpu* con dos posibles *kernels*. Esta función consiste en sumar a cada píxel de la imagen original un valor fijo modificando su valor de forma tal, que si este valor fijo es positivo, lo aclara y viceversa lo oscurece. El primer kernel titulado *ajustar_brillo_coalesced_kernel* se programó de forma tal que para *threads* con índices consecutivos en la dirección x, se acceden píxeles contiguos en cada fila de la imagen. A diferencia de esta lógica, en el segundo kernel titulado *ajustar_brillo_no_coalesced_kernel*, los *threads* con índices consecutivos en la dirección x acceden al píxel inferior de una misma columna de la imagen. Para ambos *kernels* se midieron los tiempos en milisegundos utilizando *gettimeofday*, *cudaEvents* y *nvprof*. En todos los casos se impusieron 32 *threads* por bloque en la dirección x e y. Por otra parte la cantidad de bloques en cada dirección se dejó como variable a ajustar en función del tamaño total de la imagen y la cantidad de *threads* por bloque, de forma tal que se apliquen los filtros a todos los píxeles que integran la imagen original.

Resultados experimentales

En la Figura a continuación se presentan los resultados obtenidos para el Ejercicio 1:

ajustar_brillo_gpu								
blockDim.x = 32	coalesced				no coalesced			
blockDim.y = 32	cudaEvents (ms)	gettimeofday (ms)	nvprof (ms)	gld_efficiency	cudaEvents (ms)	gettimeofday (ms)	nvprof (ms)	gld_efficiency
reserva de memoria	0.118	0.132	0.113	100.00%	0.117	0.132	0.111	12.50%
transferencia host-device	1.948	1.955	1.769		1.966	1.974	1.774	
ejecución kernel	0.326	0.335	0.323		1.151	1.161	1.129	
transferencia device-host	1.651	1.659	1.534		1.653	1.661	1.524	
liberar memoria device	0.391	0.398	0.079		0.392	0.399	0.080	
total	**4.552**	**4.611**	**3.818**		**5.396**	**5.459**	**4.619**	

Figura 1: Resultados para la función *ajustar_brillo_gpu*.

En la Figura 1 se observa que los tiempos medidos con la función *nvprof* son ligeramente menores que los arrojados por la función *cuda events* y menores aun que los obtenidos por *gettimeofday*. Esto se debe a la pequeña diferencia temporal entre la terminación de un evento de CUDA y la ejecución de la función gettimeofday.

Con respecto al desempeño de acceso a memoria *coalesced* y no *no coalesced* se observan apreciables diferencias en la Figura 1. En primer lugar la función *coalesced* presenta tiempos de ejecución del orden de 3.5 veces respecto a la *no coalesced*. Este resultado es independiente de la métrica de tiempo que se utilizó y se observa también a la hora de cuantificar la eficiencia del acceso a memoria global. Este indicador se calculó utilizando el comando *nvprof –metrics gld_efficiency* y se obtuvo un valor de acceso a memoria global de 8.3 veces mayor para el caso *no coalesced*.

Este comportamiento del código se explica por la forma en que *threads* consecutivos de un mismo *warp* acceden a la memoria global. En el caso *coalesced* los accesos memoria desde el kernel son adyacentes mientras que para la lógica no *coalesced* se realizan de forma salteada. Los pedidos a la memoria global se realizan por *warp* (grupos de 32 *threads*) y se hacen en bancos de 32 bytes (para

compute capabilities mayores a 6.0), por lo que en los accesos *no coalesced* se necesitará (en el peor de los casos) transferir 32 bancos de 32 bytes, uno por thread, ya que como los *threads* consecutivos no están asociados a pixels consecutivos, los espacios en memoria adyacentes a cada píxel requerido no son de utilidad. Por el contrario, en un acceso *coalesced*, como a *threads* consecutivos le corresponden píxeles consecutivos, solamente se precisarán 8 bancos para realizar el total de las transferencias solicitadas por cada *warp*.

Ejercicio 2

Para este ejercicio se implementó la función de filtro *blur* que consiste en aplicar una máscara a cada píxel interpolando el valor del mismo a partir de los aledaños, con diferentes pesos asignados en la matriz de la mascara. Se implementó el código en GPU y se comparó con la versión de CPU incluida en el *template*. En la Figura 1 se observan los tiempos medidos con la función *cuda events*, *gettimeofday* y *nvprof* para un tamaño de bloque de 32x32 *threads* por bloque.

blur					
blockDim.x = 32	**GPU**				**CPU**
blockDim.y = 32	**cudaEvents (ms)**	**gettimeofday (ms)**	**nvprof (ms)**	**gld_efficiency**	**gettimeofday (ms)**
reserva de memoria	0.324	0.319	0.317	71.12%	
transferencia host-device	3.898	3.904	3.563		
ejecución kernel	2.235	2.245	2.232		
transferencia device-host	1.648	1.655	1.533		
liberar memoria device	0.863	0.870	0.802		
total	**9.291**	**9.312**	**8.447**		**44.212**

Figura 2: Tiempos de desempeño de la función *blur* en milisegundos.

En la Figura 2 se observa nuevamente que los tiempos de ejecución capturados por las funciones *cudaEvents*, *gettimeofday* y *nvprof* replican el mismo patrón de comportamiento explicados anteriormente. Además, el tiempo total de ejecución del código en GPU es 4.7 menor respecto del tiempo de ejecución en CPU. Aquí se evidencia la potencia de programar en tarjetas gráficas para algoritmos que se ajusten al paradigma *SMPT*, como es el caso de la función *blur*, donde a cada píxel se le ejecuta la misma operación en términos algebraicos. Otro resultado interesante se observa al comparar el tiempo consumado en la transferencia de memoria de la CPU a la GPU y en sentido opuesto. Tanto en la Figura 1 con en 2 transferir los miSmos datos de la CPU a la GPU es más costosO que devolverlos de GPU a la CPU.

Por último se analiza el valor de eficiencia del acceso a memoria global del software *blur* para GPU. Para este código se alcanza el valor de 71 %, ya que si bien el acceso de cada thread a cada píxel de la imagen se realiza de forma *coalesced*, el valor de salida aplicado por la máscara en ese píxel depende de los aledaños en una celda de 5 x 5. Debido a esto para cada thread debe realizarse un loop que recorra cada uno de los píxeles de la imagen en la celda de la máscara para luego aplicarle el peso correspondiente. Este acceso no es perfectamente coalesced y también es desalineado. Por ende no es eficiente ya que para los píxeles en las mismas columnas se copian datos de la memoria global a los registros que no son utilizados.