Computación de propósito general de unidades de procesamiento gráfico

Práctico 4

Accesos y tipos de memoria en GPU

Integrantes

Nombre	CI	Correo
Renzo Gambone	5.155.486-4	renzo.gambone@fing.edu.uy
Germán Ouviña	4.823.566-1	german.ouvina@fing.edu.uy

Índice

1. Introducción

En el presente informe se detallan los resultados obtenidos durante la evaluación experimental de los ejercicios correspondientes al **Práctico 4**, así como su pertinente análisis. A continuación, se destacan ciertos aspectos a tener en cuenta del análisis y de los archivos entregables:

1. La implementación de los algoritmos se realizó empleando el lenguaje **C** y la API de **CUDA**.
2. La evaluación experimental se realizó por medio de la plataforma ClusterUY. Durante la evaluación experimental los recursos que nos fueron asignados y con los que ejecutamos el algoritmo fueron:
   - **GPU:** Tesla P100-PCIE-12GB
   - **CPU:** Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz
3. Los datos manipulados fueron imágenes representadas por vectores de números decimales en punto flotante (*float* de **C**).
4. Los tiempos registrados al realizar distintas ejecuciones de un mismo script no fueron siempre constantes, sin embargo su variación osciló con un radio lo suficientemente bajo como para que las observaciones y conclusiones presentadas se preserven.

2. Desarrollo

2.1. Ejercicio 1 - Transposición de imagen

En esta sección se detallan los resultados obtenidos de la experimentación realizada con los algoritmos de transposición de imagen. Dicha experimentación consistió en registrar los tiempos de cada algoritmo, así como analizar distintas métricas en referencia al uso eficiente de la memoria. Para capturar estos datos, se utilizó la herramienta de profiling `nvprof`.

2.1.1. Construcción de Algoritmos

Por consigna, se trabajó con dos implementaciones de `transpose_gpu`, una función que, dada una imagen de entrada, intercambia cada píxel (x, y) por el píxel (y, x), generando la versión transpuesta de la imagen. Las implementaciones fueron las siguientes:

- `transponse_global_gpu`, versión paralela del algoritmo donde hay tantos threads como píxeles, y cada uno genera un píxel transpuesto leyendo los píxeles originales de la memoria global.
- `transponse_shared_gpu`, versión paralela del algoritmo donde hay tantos threads como píxeles, y cada uno carga un píxel en la memoria compartida de su bloque, para luego leer el píxel transpuesto dentro de dicho bloque y escribirlo en la posición correspondiente en memoria global. Se utiliza una estructura matricial en la memoria compartida, la cual cuenta con una celda por thread del bloque.

Cabe destacar que para el algoritmo en memoria compartida, se generó una versión con bloques de 16×16 y otra con bloques de 32×32, con el fin de comprobar si dicho cambio influenció de alguna forma en las métricas tomadas. A su vez, por sugerencia de la consigna, se probó agregar una columna extra a la estructura matricial de la memoria compartida y comparar las métricas una vez más.

2.1.2. Análisis de Métricas

A continuación se adjunta una tabla comparando tiempos y métricas de eficiencia de memoria según lo pedido en la consigna para cada versión del algoritmo. Se destaca que el algoritmo "Extra" refiere al algoritmo que utiliza memoria compartida con una columna extra en la matriz de la misma.

Algoritmo	***Tiempo*** (μs)	***gld_efficency*** (%)	***gst_efficency*** (%)	***shared_efficency*** (%)
Global (B 32×32)	109.28	100,00	12,50	0,00
Compartida (B 16×16)	29.312	100,00	100,00	22,50
Compartida (B 32×32)	43.007	100,00	100,00	6,06
Extra (B 16×16)	28.832	100,00	100,00	50,00
Extra (B 32×32)	32.159	100,00	100,00	100,00

Se pueden realizar las siguientes observaciones:

- Todos los algoritmos cuentan con *gld_efficency* del 100 %.
- Los algoritmos con memoria compartida rondan entre los $25 - 45\mu s$ de tiempo y cuentan con *gst_efficency* del 100 %, en contraposición con el algoritmo de memoria global que ronda los $110\mu s$ de tiempo y cuenta con una *gst_efficency* del 12.50 %.
- Los algoritmos con memoria compartida y bloques de 16×16 parecen contar con mejores tiempos en contraposición con los algoritmos de 32×32.
- Por otra parte, la *shared_efficency* mejoró en los algoritmos de memoria compartida con columna extra, siendo del 100 % al tener bloques de 32×32.

Se pueden extraer múltiples conclusiones de los datos anteriormente observados.

Respecto a la eficiencia de carga global *(gld_efficency)*, es coherente observar que resultó ser del 100 % en cada algoritmo, ya que cada thread trabajó con un sólo píxel de la imagen de entrada y se organizaron bloques y threads de forma tal que siguieran una estructura indizada de forma similar a la imagen en cuestión. De esta manera, todos los threads de un warp trabajan con píxeles consecutivos, y por ende cada lectura a memoria global usa el 100 % de los datos traídos.

Respecto a la eficiencia de almacenamiento global *(gst_efficency)*, el algoritmo que sólo utiliza memoria global cuenta con una performance subóptima a la hora de escribir. Esto tiene sentido, ya que las escrituras a memoria global siguen el mismo principio que las lecturas: se hacen por filas. Para tener una eficiencia del 100 % en esta métrica, todos los threads de un warp tendrían que escribir píxeles de forma consecutiva **por filas**. Sin embargo, por la naturaleza del algoritmo, lo hacen **por columnas**. El algoritmo de memoria compartida corrige esto, ya que realiza la lectura inicial de memoria global por filas y carga los tiles de memoria compartida como paso intermedio. Una vez hecho esto, se lee de memoria compartida por columna, escribiendo en memoria global por fila (la noción inversa al algoritmo que utiliza sólo memoria global). De esta manera, se alcanza una eficiencia del 100 %.

Respecto a la eficiencia del uso de memoria compartida *(shared_efficency)*, es importante entender como funcionan los accesos a la misma. Tomando como ejemplo el caso donde se generan bloques de 32×32, el tile cargado en memoria compartida es una matriz de 32×32 celdas, cada una siendo una palabra de 32 bits. De esta manera, la celda [0][0] se almacena entera en el banco 0, la celda [0][1] en el banco 1, y así sucesivamente hasta la celda [0][31] en el banco 31. Lo mismo sucede para el resto de filas, quedando cargadas las celdas $[x][y]$ en el banco y, con $x, y \in [0.,31]$. Otra forma de verlo, es que el banco y tiene la columna y del tile en cuestión. Como se estableció en el párrafo anterior, la lectura de memoria compartida en este algoritmo se hace por columnas, por lo que los 32 threads de un warp van a leer las 32 celdas de alguna columna. Los conflictos de banco se producen cuando varios threads de un warp quieren acceder a distintas palabras de un mismo banco, y en este caso hay 32 threads intentando acceder a las 32 palabras de un banco en concreto. Se puede comprobar esto observando la baja eficiencia para este algoritmo. No obstante, al agregar una columna extra, la eficiencia pasa a ser del 100 %, ya que cada columna del tile pasa a estar almacenada en los 32 bancos a la vez, una celda por banco. Esto puede contrastarse en las siguientes figuras. A la izquierda, el tile de 32×32, a la derecha, el tile de 32×33. En cada celda, el número de banco al que pertence:

$$\begin{pmatrix} 0 & 1 & ... & 30 & 31 \\ 0 & 1 & ... & 30 & 31 \\ ... & ... & ... & ... & ... \\ 0 & 1 & ... & 30 & 31 \\ 0 & 1 & ... & 30 & 31 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & ... & 30 & 31 & 0 \\ 1 & 2 & ... & 31 & 0 & 1 \\ ... & ... & ... & ... & ... & ... \\ 30 & 31 & ... & 28 & 29 & 30 \\ 31 & 0 & ... & 29 & 30 & 31 \end{pmatrix}$$

Si se comprueba la eficiencia para un tamaño de bloque de 16×16, la misma es reducida a la mitad, lo cual condice con lo observado en la anterior figura. Si bien el tamaño del tile pasa a ser de 16×17, un warp sigue teniendo 32 threads, por lo que siguen existiendo conflictos (la mitad de las veces).

Como conclusión final, es coherente comprobar que el uso de memoria compartida disminuye los tiempos de ejecución notoriamente. Por otra parte, la eficiencia en el uso de la memoria compartida no pareció influir mucho en los tiempos medidos, aunque es probable que al aumentar el tamaño de imagen, se perciba una mejora sustancial.

2.2. Ejercicio 2 - Filtro Gaussiano

En esta sección se detallan los resultados obtenidos de la experimentación realizada con los algoritmos de filtro gaussiano. Dicha experimentación consistió en registrar los tiempos de ejecución de cada algoritmo y realizar comparaciones. Para capturar estos datos, se utilizó la herramienta de profiling `nvprof`.

2.2.1. Construcción de Algoritmos

Por consigna, se trabajó con distintas versiones del algoritmo de filtro gaussiano, partiendo de la implementación del práctico anterior. Cabe destacar que mientras se implementaban las optimizaciones pedidas, también se desarrollaron otras mejoras a nivel de código a modo de reducir la cantidad de iteraciones y operaciones que ejecuta cada kernel. Dichas mejoras también se incluyeron en la implementación del anterior práctico, con el fin de hacer más justa la comparación de performance.

De esta manera, se generaron las siguientes versiones:

- `blur_kernel_global`, (Memoria Global): Kernel implementado en el práctico anterior, modificado acorde a las optimizaciones mencionadas anteriormente.
- `blur_kernel_a`, (Memoria Compartida): Kernel que itera sobre la estructura de `blur_kernel_global`, optimizando su eficiencia a modo que el filtro gaussiano se ejecute accediendo a los datos por medio de memoria compartida.
  La carga de la misma implica un overhead en el kernel y fue implementada siguiendo el algoritmo sugerido en el práctico, en donde cada hilo carga un píxel, luego el píxel correspondiente a shiftear la posición del píxel `blockDim.x`, `blockDim.y`, y luego en ambas direcciones (siempre que corresponda).
  Dado que para la aplicación de la máscara es necesario contar con un apron de todos los píxeles que bordean el bloque a escribir, los bloques de dimensión 32×32 trabajan con con una memoria compartida de: $(32 + 2 * radio) \times (32 + 2 * radio)$.
  A modo de poder aplicar el algoritmo sugerido, durante la carga de la memoria compartida, cada hilo cargó en memoria el píxel de la posición que le corresponde escribir desplazada $(-radio)$ posiciones vertical y horizontalmente en la imagen de entrada.
- `blur_kernel_b`, (Memoria Constante - `const __restrict__`): Kernel que itera sobre la estructura de `blur_kernel_a`, optimizando el acceso a la máscara en el kernel indicando `const __restrict__` en el parámetro de la misma.
  Esto indica al compilador que dicho parámetro es de solo lectura (`const`), también indicando que no otro puntero apunta a su dirección (`__restrict__`). Lo cual permite al compilador optimizar los accesos a memoria a través de dicho puntero (no volviendo a buscar datos del mismo en memoria tras cada acceso).
- `blur_kernel_c`, (Memoria Constante - `cudaMemcpyToSymbol`): Kernel que itera sobre la estructura de `blur_kernel_a`, optimizando el acceso a la máscara del kernel alojando y accediendo a la misma a través de la memoria constante de la GPU. Esto se logra declarando la máscara como `__constant__` (afuera del kernel), y de la misma forma que el método que invoca al kernel cargaba la máscara en la memoria global de la GPU la carga en la memoria constante a través del método `cudaMemcpyToSymbol`.
  Tras alojar la máscara en memoria constante, el acceso a la misma dentro del kernel es tan simple como acceder a una constante.
  La memoria constante de CUDA ofrece un acceso con velocidad de registro y optimizado para cuando todo el warp accede al mismo elemento.

Es importante destacar que, tras capturar los resultados con una máscara de $radio = 2$ (5×5) (la utilizada en el práctico anterior), se decidió probar cada versión del algoritmo con una máscara de $radio = 5$ (11×11). Esta decisión surgió por no haber observado una mejora sustancial a la hora de experimentar con las optimizaciones respecto al almacenamiento de la máscara de menor tamaño.

2.2.2. Análisis de Métricas

A continuación se adjunta una tabla comparando tiempos para cada versión del algoritmo y distinto radio.

Algoritmo	***Radio***	***Tiempo*** (μs)	***Radio***	***Tiempo*** (μs)
Global	2	153.09	5	606.6
Compartida	2	106.85	5	335.74
Constante (`const __restrict__`)	2	106.24	5	329.44
Constante (`cudaMemcpyToSymbol`)	2	108.10	5	227.13

Se pueden realizar las siguientes observaciones:

- El algoritmo con memoria compartida obtuvo una mejora en velocidad de 43,28 % para $radio = 2$ y de 80,70 % para $radio = 5$ respecto al algoritmo con memoria global.
- El algoritmo con memoria compartida y la directiva `const __restrict__` obtuvo una mejora en velocidad de 0,57 % para $radio = 2$ y de 1,91 % para $radio = 5$ respecto al algoritmo con memoria compartida.
- El algoritmo con memoria compartida y la máscara alojada en `__constant__` obtuvo una **disminución** en velocidad de 1,16 % para $radio = 2$ y una **mejora** de 47,82 % para $radio = 5$ respecto al algoritmo con memoria compartida.
- Las diferencias entre algoritmos son más pronunciadas entre sí a mayor tamaño de máscara.

Se pueden extraer múltiples conclusiones e interpretaciones a partir de los datos anteriormente observados.

Respecto a las implementaciones con memoria compartida, es fácil observar que pese a que los kernels tienen un overhead asociado a la hora de cargar datos en la memoria compartida, esta optimización causó una mejora significativa en el tiempo de ejecución del kernel.

A la hora de evaluar las dos optimizaciones realizadas sobre el acceso a la máscara se concluye que la optimización con la directiva `const __restrict__` si bien ofrece una mejora, no es significativa, siendo prácticamente despreciable para una máscara pequeña y bastante menor para una máscara de mayor tamaño.

Por el otro lado, la optimización que almacena la máscara en memoria `constant` y la transfiere mediante `cudaMemcpyToSymbol` parece tener una especie de overhead para la arquitectura de la tarjeta de video empleada al trabajar con máscaras pequeñas, el cual termina causando un aumento en el tiempo total de kernel. Por otra parte, para máscaras más grandes, este algoritmo logra obtener una mejora significativa en el tiempo de ejecución.