UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Laboratorio de TGA

Trabajo final

Filtrado de imágenes

Lluc Clavera
Pol Forner

2023/24 Q1

Contenidos

1 Problema

En este trabajo exploraremos distintos algoritmos de filtrado de imágenes y compararemos la mejora de rendimiento que puede proporcionarnos usar CUDA en lugar de una ejecución completamente secuencial.

1.1 Funcionamiento general de los filtros

Hemos escogido diversos filtros que nos permitan explorar distintos casos de ejecución. Por este mismo motivo, cada uno de los filtros escogidos tiene una particularidad diferente a los demás, y nos plantea unas dificultades distintas en cuanto a su paralelismo.

Todos los filtros que veremos en este proyecto consistirán en modificar el color de un píxel concreto. Dependiendo del filtrado que usemos cambiará el cálculo del nuevo píxel, pero la esencia del cálculo será la misma.

Para la mayoría de filtrados usaremos un concepto llamado convolución. Una convolución consiste en aplicar una suma ponderada con distintos pesos de los píxeles de alrededor del píxel que estamos tratando. En función de los pesos, que se expresan en una matriz llamada *kernel*, se aplicará un filtro u otro.

1.2 Blanco y negro

El filtro blanco y negro és el filtro más sencillo que implementaremos. Su implementación consistirá únicamente en obtener la media de las intensidades de los componentes **rojo**, **verde** y **azul** y substituir todos los componentes por esa intensidad media.

Puesto en forma matemática:

$$m = \frac{R_{i,j} + G_{i,j} + B_{i,j}}{3}$$

$$R'_{i,j} = G'_{i,j} = B'_{i,j} = m$$

Donde $R_{i,j}, G_{i,j}, B_{i,j}$ representan las intensidades del píxel (i,j) de **rojo**, **verde** y **azul** respectivamente.

El resultado de aplicar el filtro de blanco y negro debería ser el siguiente (aplicado en la imagen IMG00.png):

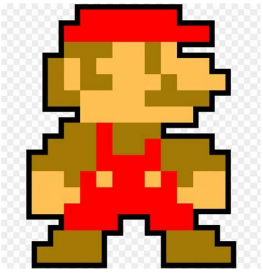

Figure 1: Imagen 0 sin filtrar

Figure 2: Imagen 0 con filtro blanco y negro

1.3 Rotar imagen

Para este filtro simplemente rotaremos una imagen 180^{o}, eso simplemente significa que la primera fila se moverá a la última fila, pero "girada", es decir, el primer elemento se moverá al último elemento de la última fila

De forma general, i definiendo $P[i, j]$ como el valor del píxel y n_i y n_j como el número de filas y columnas:

$$P[i, j]' = P[n_i - i - 1, n_j - j - 1]$$

Este filtro, sobre la imagen 0, se muestra en la figura 4:

1.4 Filtro de media

Para este filtro, cada píxel recibirá los valores de los píxeles de alrededor y el nuevo pixel consistirá en la media de estos valores. Dicho de otra forma tendremos que aplicar una convolución con una matriz con todos los valores a 1.

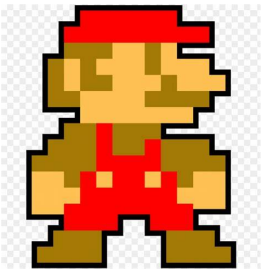

Figure 3: Imagen 0 sin filtrar

Figure 4: Imagen 0 con rotación

El kernel de la convolución será el siguiente:

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

Como veis, esto simplemente consistirá en sumar todos los píxeles en un rango de 3x3 alrededor del pixel que queramos tratar y luego dividir el resultado entre 9. Por lo tanto, el nuevo valor del píxel será el siguiente:

$$P[i,j]' = \frac{1}{9}\sum_{k=i-1}^{i+1}\sum_{l=j-1}^{j+1} P[k,l]$$

Como vemos en la siguiente figura, el filtro de media no se nota demasiado (de notaria más si ampliásemos el tamaño del kernel de convolución). Igualmente, podemos notar como el color es ligeramente distinto tras aplicar el filtro.

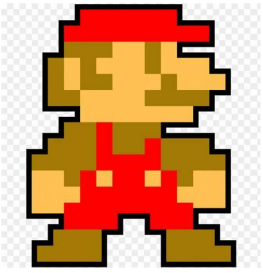

Figure 5: Imagen 0 sin filtrar

Figure 6: Imagen 0 con filtro de media

1.5 Ruido gaussiano

Para este filtro aplicaremos una matriz de convolución un poco más compleja que en el caso anterior. La matriz de convolución será una matriz de 5x5 con pesos basados en una distribución normal.

El kernel de convolución de este filtro será el siguiente:

$$G = 1/159 \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

De este modo, el nuevo valor del píxel se calculará de la siguiente manera:

$$P[i,j]' = \sum_{k=i-2}^{i+2} \sum_{l=j-2}^{j+2} G[k,l] * P[k,l]$$

El resultado de aplicar este filtro es una difuminación de la imagen, aunque como vemos en la imagen de ejemplo, el resultado se parece bastante a la aplicación del filtro de media:

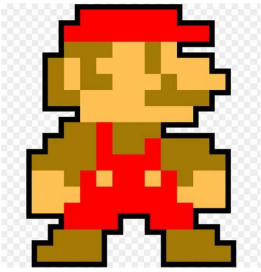

Figure 7: Imagen 0 sin filtrar

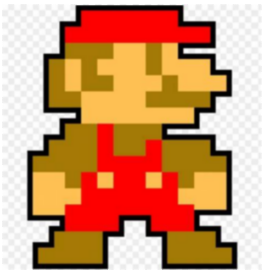

Figure 8: Imagen 0 con ruido Gaussiano

1.6 Filtro de Sobel

El filtro de Sobel, es un filtro dedicado a la detección de bordes. La esencia del filtro consiste en aproximar las derivadas para detectar los cambios de

color, cambios grandes de color suelen implicar la presencia de bordes.

Para este filtro aplicar una convolución no nos bastará. Tendremos que aplicar 2 convoluciones (una para detectar los cambios en dirección horizontal y otra para detectar cambios en dirección vertical) y luego combinarlas.

Los 2 Kernels de conolución que necesitaremos son los siguientes:

$$Y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}, \; X = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Para calcular el nuevo valor de cada píxel primero tendremos que aplicar las 2 convoluciones por separado y luego combinarlas. El cálculo que tendremos que hacer será el siguiente:

$$P_y[i,j] = \sum_{k=i-1}^{i+1} \sum_{l=j-1}^{j+1} Y[k,l] * P[k,l]$$

$$P_x[i,j] = \sum_{k=i-1}^{i+1} \sum_{l=j-1}^{j+1} X[k,l] * P[k,l]$$

$$P[i,j]' = \sqrt{P_y[i,j]^2 + P_x[i,j]^2}$$

Como se puede ver, necesitaremos calcular por separado 2 valores intermedios (P_x y P_y) para luego combinarlos. Esta característica del filtro dará pie a varias descomposiciones factibles en CUDA, que explicaremos con más detalle durante la discusión de la implementación.

En la siguiente figura se puede ver el resultado de aplicar el filtro a la IMG00 a demás del resultado con los colores invertidos (la inversión se hace usando $P[i,j]' = 255 - P[i,j]$). Como este filtro se usa para detectar bordes, realmente da igual si usamos la fórmula original o invertimos los colores. De hecho, a veces se ve de forma más clara con los colores invertidos.

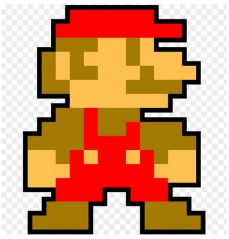

Figure 9: Imagen sin filtrar

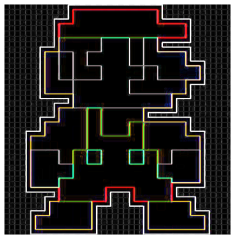

Figure 10: Imagen con filtro de Sobel

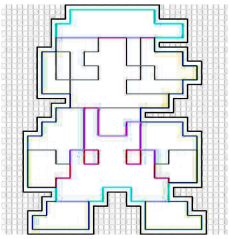

Figure 11: Imagen con filtro de Sobel con colores invertidos

2 Implementación CUDA

2.1 Esquema general

Para ejecutar los 5 filtros usaremos un único programa (filtrar.cu) que importará los 5 filtros que vamos a usar y, dependiendo de un parámetro de entrada, decidirá que filtro ejecutará.

Tras compilar el código, usaremos *filtrar.exe* ***fileIN OUTSequential OUTKernel filter***. Donde **fileIN** es la imagen de entrada, **OUTSequential** y **OUTKernel** son las salidas de la ejecución secuencial y la ejecución del kernel respectivamente y **filter** es una letra indicando que filtro se desea utilizar. Las opciones de filtrado son: B (Black and white), R (Rotate), M (Mean), G (Gaussian) y S (Sobel).

Todas las funciones de filtrado que implementemos van a recibir los mismos parámetros: imagen a tratar, dimensiones de la imagen (width y height) y tamaño del pixel (cuantos chars representan un píxel). A demás, algunos filtros necesitan una variable auxiliar adicional en la que se guarda la imagen original simplemente para consultarla sin riesgo a que se modifique.

A demás, hemos usado 32x32 como tamaño de bloque, con lo que se consigue que los bloques se llenen al máximo (el máximo son 1024 threads por bloque). Se podría investigar con distintos tamaños de bloque, incluso se podría buscar un tamaño de bloque múltiplo de la imagen para ver si el resultado mejora.

A continuación vamos a profundizar en la implementación concreta de cada filtro en CUDA.

2.2 Blanco y negro

Vamos a repasar el cálculo de un píxel para este filtro:

$$m = \frac{R_{i,j} + G_{i,j} + B_{i,j}}{3}$$

$$R'_{i,j} = G'_{i,j} = B'_{i,j} = m$$

Como se puede ver en la formulación, para calcular el valor de un píxel solo se necesita el valor previo del mismo píxel. Gracias a esto, este filtro es completamente paralelizable y cada thread se puede encargar, sin problema, de su propio píxel.

2.3 Rotar imagen

Vamos a repasar el cálculo de un solo píxel:

$$P[i,j]' = P[n_i - i - 1, n_j - j - 1]$$

En este caso nos encontramos que cada píxel necesita el valor de otro píxel distinto, esto nos puede llevar a pensar que quizás necesitemos guardarnos la imagen original en una variable original, ya que si los threads editan directamente la imagen esto dará pie a que un thread use el valor ya modificado de otro píxel cuando debería haber usado el original.

Igualmente, al ver la siguiente relación nos damos cuenta de que no necesitamos variables auxiliares:

$$P[i,j]' = P[n_i - i - 1, n_j - j - 1] \iff$$
$$P[n_i - i - 1, n_j - j - 1]' = P[i,j]$$

Como vemos en la anterior fórmula, el filtro consiste simplemente en intercambiar píxeles. Si cada thread realiza un solo intercambio (por lo tanto, modifica 2 píxeles) conseguimos que ningún thread modifique un valor que

otro thread necesite usar.

Con esto llegamos a una versión completamente paralelizable en la que cada thread se encargará de 2 píxeles

2.4 Filtro de media

Para el filtro de media tendremos que implementar un kernel que realice la siguiente operación por píxel:

$$P[i,j]' = \frac{1}{9} \sum_{k=i-1}^{i+1} \sum_{l=j-1}^{j+1} P[k,l]$$

El problema que nos surge al realizar la operación es que esta vez sí necesitaremos una variable auxiliar, pues la operación se tiene que hacer sobre los píxeles originales, y no queremos que un thread modifique el resultado de otro thread.

2.5 Ruido gaussiano

Para el kernel del ruido gaussiano tan solo necesitaremos aplicar la siguiente transformación a cada píxel:

$$P[i,j]' = \sum_{k=i-2}^{i+2} \sum_{l=j-2}^{j+2} G[k,l] * P[k,l]$$

Este kernel es extremadamente similar al anterior, aunque en este caso guardaremos la matriz G en una variable global del dispositivo para poder aplicar la operación de forma más cómoda.

2.6 Filtro de Sobel

Primero vamos a repasar la fórmula para calcular el valor de un nuevo píxel:

$$P_y[i,j] = \sum_{k=i-1}^{i+1} \sum_{l=j-1}^{j+1} Y[k,l] * P[k,l]$$

$$P_x[i,j] = \sum_{k=i-1}^{i+1} \sum_{l=j-1}^{j+1} X[k,l] * P[k,l]$$

$$P[i,j]' = \sqrt{P_y[i,j]^2 + P_x[i,j]^2}$$

Nosotros hemos decidido invertir los colores al final para que los bordes destaquen más, de este modo cambiamos el tercer cálculo por:

$$P[i,j]' = 255 - \sqrt{P_y[i,j]^2 + P_x[i,j]^2}$$

En esencia, tenemos que hacer tres cálculos separados, los dos primeros son independientes y el tercero depende de los anteriores.

Como son tres cálculos separados podemos dividirlo fácilmente en tres kernels distintos. Los dos primeros se encargarán de calcular P_x y P_y mientras que el tercero se llamará después de los otros dos y realizará el cómputo final. Otra forma de dividir el trabajo es usar un solo kernel para que haga los tres cálculos, uno detrás del otro.

En este trabajo implementaremos la opción de que un solo kernel realice los tres cálculos, pues no tiene dependencias y nos evita incluir instrucciones de sincronización.

Finalmente, observamos que, como en los anteriores filtros, necesitamos una variable auxiliar para guardar el valor original de los píxeles de la imagen.

3 Análisis de rendimiento

3.1 Esquema de medición de rendimiento

Para cada experimento hemos realizado 3 mediciones: Tiempo Secuencial, que mide el tiempo en una ejecución secuencial; Tiempo Global, que mide el tiempo total de la ejecución en la GPU (transferencias de memória + kernel); i el Tiempo Kernel, que solo mide el tiempo de ejecución del kernel.

Para medir los speed-ups, hemos comparado el tiempo de ejecución global y de kernel con el tiempo secuencial para obtener 2 speed-ups distintos.

A demás, para medir los tiempos hemos decidido repetir cada experimento 10 veces, eliminar los dos datos más extremos (menor y mayor tiempo) y luego hacer la media de todas las mediciones para obtener los tiempos medios (i calcular los speed-ups con los tiempos medios).

3.2 Resultados generales

Primero de nada vamos a ver comparaciones globales y luego hablaremos específicamente de cada uno de los filtros en concreto.

Las siguientes 2 gráficas muestran la evolución del speedup con distintos filtros:

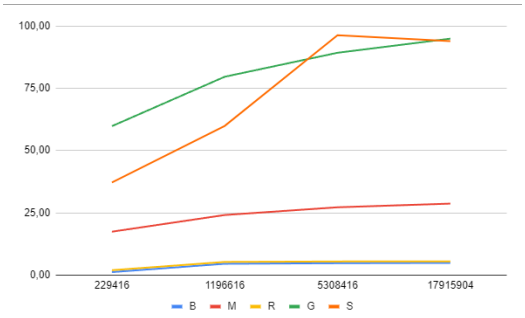

Figure 12: Speedup global en función del filtro

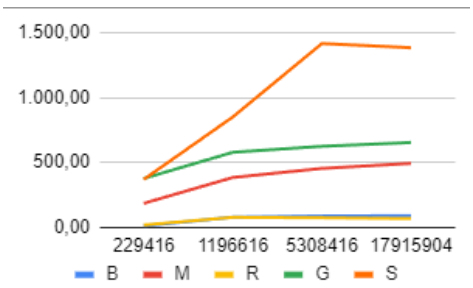

Figure 13: Speedup del kernel en función del filtro

Los filtros de *blanco y negro* y *rotación* son prácticamente idénticos en términos de speed-up. Este resultado no es una sorpresa, pues son los únicos kernels que no usan variable auxiliar, y ambos kernels realizan una operación muy básica (calcular una simple media o intercambiar 2 píxeles).

Luego tenemos el filtro de media, que muestra un mayor rendimiento que los dos filtros anteriores. Comparado con el filtro simple de blanco y negro, este filtro realiza una media con 9 píxeles, en vez de realizar una media con los componentes de un solo píxel. Por este motivo, cada kernel realiza más operaciones y se extrae mejor el potencial de la tarjeta gráfica.

El caso del filtro de ruido gaussiano es muy similar, en vez de usar 9 píxeles usa 25, por lo que el trabajo realizado por kernel es aún mayor, y el rendimiento que obtenemos es incluso superior.

El caso más curioso es el del filtro de Sobel. En primer lugar, vemos que el speed-up global se cruza con el speed-up global del filtro de media, empezando por debajo, superándolo eventualmente y volviendo a un speed-up similar al final.

Este cruce de speed-ups no es difícil de explicar cuando tenemos en cuenta que el filtro de ruido gaussiano tarda más en ejecutar. Una parte importante del tiempo se gasta en la transferencia de datos (que tarda el mismo tiempo para todos los filtros), como el filtro de ruido gaussiano tarda más que el de sobel, el porcentaje de tiempo gastado en transferencias es inferior y, por lo tanto, muestra más rendimiento global.

Como el speed-up del filtro de Sobel aumenta más rápido, eventualmente termina alcanzando (y superando) al filtro de ruido gaussiano en rendimiento global, pues el incremento de speed-up acaba compensando la pérdida de tiempo en transferencias.

Lo que realmente nos ha sorprendido es que el filtro de Sobel tenga tan buen rendimiento cuando se mira solo el tiempo de kernel, especialmente porque el número de operaciones por kernel tampoco es muy distinta a la del filtro de gauss (2 filtros de 9 píxeles más una combinación de los 2 filtros). Sospechamos que puede tener algo que ver con la implementación de la función *sqrt* (o *sqrtf* en CUDA) de CUDA. Creemos que la GPU es más

eficiente que la CPU al realizar raíces cuadradas y por ese motivo vemos un speed-up mayor en este filtro que en todos los demás.

4 Conclusiones

En conclusión, hemos aprendido sobre diversas técnicas de filtrado y hemos visto como para algunos casos donde el cálculo es más grande, los speedup son significativos. Durante el proceso pensamos varias implementaciones de los kernels intentando buscar un mejor rendimiento, como el uso de memoria compartida. Debido a la complejidad decidimos centrarnos en kernels más sencillos pero hacerlos bien y hacer un mejor testeo.

También nos encontramos problemas donde observábamos que se realizaban los filtros y al ir a medir el tiempo de los kernels salía 0. Hicimos una búsqueda del problema hasta determinar que había un problema donde cuando se elegía una gráfica diferente a la 0 había un error. Debido a que no encontramos como solucionarlo, decidimos que siempre se usase la cpu = 0.

5 Referencias

[1] *Creating global variables in CUDA*. stackoverflow. 2022. URL: https://stackoverflow.com/questions/6255215/creating-global-variables-in-cuda.

[2] *cuda invalid resource handle*. stackoverflow. 2021. URL: https://stackoverflow.com/questions/6021449/cuda-invalid-resource-handle.

[3] *Cuda programming*. cuda-programming. 2012. URL: https://cuda-programming.blogspot.com/2013/01/performance-of-sqrt-in-cuda.html13.

[4] *Event Management*. nvidia developer doc. 2023. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html#group__CUDART__EVENT_1gf4fcb74343aa689f4159791967868446.

[5] *How to Implement Performance Metrics in CUDA C/C++*. nvidia developer. 2021. URL: https://developer.nvidia.com/blog/how-implement-performance-metrics-cuda-cc/.

[6] A. Walker R. Fisher S. Perkins and E. Wolfart. *Sobel Edge Detector*. 2003. URL: https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm.

[7] *Técnicas de filtrado*. Universidad de Murcia. 2015. URL: https://www.um.es/geograf/sigmur/teledet/tema06.pdf.