Universidad de Castilla-La Mancha
Escuela Superior de Informática (CR)

Práctica Final

Implementación de Filtros

CUDA

Sergio Jiménez del Coso
Profesora: María José Santofimia Romero

Computadores Avanzados
18 de enero de 2021

Índice

Índice de figuras

1. Introducción

Este proyecto consistirá en el razonamiento, aprendizaje, realización y transformación de un programa secuencial a uno paralelo, cuya temática consiste en el filtrado de imágenes y frames de video (y también WebCam) a través del framework de CUDA. En él incluiré una serie de comparativas tanto a nivel de arquitectura como a nivel de rendimiento y realizaré diversas observaciones en cuanto al tiempo de ejecución y de memoria.

2. Filtros

En esta sección, explicaré cada uno de los filtros que he utilizado para la implementación del programa paralelo. Como ya sabemos, estos filtros tienen la funcionalidad de modificar el valor de cada uno de los píxeles a partir de un **kernel**. Este kernel consiste en una matriz de n-dimensiones que opera sobre un píxel en una determinada posición, cuyo resultado total es la suma de cada una de estas operaciones parciales. En este caso, realizaré la implementación de dos filtros:

1. **Filtro Sobel**. Se trata de un filtro de detección de bordes. Para la implementación de este filtro, se calcula los grandientes en cada píxel (componente x y la componente y). Estos detalles serán explicados en el siguiente apartado. Las matrices que emplearé para la implementación final de cada uno de los gradientes son las siguientes:

$$\mathbf{G_x} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{y} \quad \mathbf{G_y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

Figura 1: Matrices de gradientes

$$\mathbf{G} = \sqrt{\mathbf{G_x}^2 + \mathbf{G_y}^2}$$

Figura 2: Cálculo del resultado total

2. **Filtro Sharpening**. Se trata de un filtro de perfilamiento. Para la implementación de este, se calcula el valor total procedente del sumatorio de los cálculos independientes de cada uno de los píxeles que son determinados por la longitud del propio kernel.

0	-1	0
-1	5	-1
0	-1	0

Figura 3: Matriz del filtro Sharpen

3. Diseño del programa

Para realizar el diseño del programa, primero partimos de un programa secuencial que a partir de ciertos detalles, podemos convertirlo a un programa paralelo. En mi caso, para transformarlo a un programa paralelo, el punto de partida está en el cálculo de cada uno de los píxeles junto la posición específica del kernel.

Además, el programa que he diseñado acepta una lista de argumentos con el objetivo de aplicar los filtros de tres maneras diferentes: a una imagen, a un video y a través de los frames captados de la WebCam. Las instrucciones de ejecución vendrá registradas en el archivo *README.md* de la carpeta donde se encuentra el código. En el caso de la imagen, obtenemos dos argumentos: el **filtro** y la **imagen**. Tanto la lectura como la conversión a blanco y negro de la imagen, se realizan a través de la librería **OpenCV**. En mi caso, el acceso a los píxeles de la imagen se realiza a través de un indexado de un *array* de una dimensión. No obstante, a la hora de calcular el número de hebras y el número de bloques, he definido ambas variables de tipo **dim3** con el objetivo de asociar un conjunto de hebras de un determinado bloque a un píxel determinado, teniendo en cuenta los límites que presenta la GPU. Una vez que tengamos la imagen tratada y lista para el filtrado, se aplica el kernel correspondiente.

A la hora de aplicar el filtro seleccionado, primero debemos obtener cada uno de los índices de las hebras asociado al píxel correspondiente, obteniendo la componente **x** (fila) e **y** (columna). Una vez que tengamos ambos índices definidos, lo transformamos a un índice de una dimensión, como hemos definido anteriormente:

```
src_image[x*width+y];

# Donde width es el número de columnas que tiene la imagen
```

Un dato muy importante a la hora de la transformación de la imagen, el valor del píxel se verá modificado por un kernel determinado. Este valor debe ser positivo, de ahí por lo que utilizo el tipo de variable **unsigned char**. Este tipo de variable permite no perder información a la hora de la transformación del píxel ya que tenemos que incluir los valores que no tengan signo. Una vez realizado la transformación del píxel, se realiza la eliminación del ruido restante, es decir, establecer el valor de cada uno de los píxeles al rango [**0-255**] y así obtener una imagen nítida y correcta.

Para guardar la imagen modificada, he inicializado una variable con la función **cudaMemset**. Esta función permite inicializar la variable correspondiente a cero de un área determinada. Finalmente, aplicamos el kernel correspondiente obteniendo así el valor final del píxel modificado. El resultado será copiado a través de la función **cudaMemcpy** y la mostraremos por pantalla a través de la librería de **OpenCV**.

4. Rendimiento y Comparativas

A la hora de definir el rendimiento, he establecido varias pautas: a nivel de código, a nivel de CPU y a nivel de frames.

4.1. Nivel de Código

En cuanto a nivel de código, podemos observar que se trata de un código que no tiene mucha variación ya que el procedimiento es el mismo: lectura de la imagen o frame, transformación a través del kernel y muestra de la imagen o frame. A la hora de realizar un código mucho más eficiente he supuesto varias ideas como la utilización de memoria *pinned*. No obstante, no está implementado debido a que la latencia se incrementa exponencialmente dependiendo del tamaño de la imagen o frame. A su vez, la idea de implementar un mecanismo de memoria compartida tampoco se pudo realizar. Esto es debido a que mi tarjeta gráfica permite establecer unos límites computacionales y al tratarse de datos con mucha información, saltaría una serie de excepciones indicando que el core no es lo suficientemente capaz como para manipular un número muy grande de datos.

4.2. Nivel de CPU

En cuanto a nivel de CPU, he realizado una serie de comparativas entre la ejecución utilizando la CPU y la utilización de la GPU. Para establecer dicha comparativa he creado un script en C++, *cpu_kernels.cpp* mostrando el tiempo de ejecución para un determinado filtro de una imagen. Al realizar la ejecución de ambos programas a la vez con una determinada imagen, obtenemos esta comparativa:

```
gsergey@ubuntu:~/Desktop/CUDA_Filter$ make gpu-sobel
./cuda_kernels 0 "sobel" img/tesla.jpg & ./cpu_kernel img/tesla.jpg "sobel"
[CPU MANAGER] Using Image img/tesla.jpg | ROWS = 2160 COLS = 3840
[CUDA MANAGER] Using Image img/tesla.jpg | ROWS = 2160 COLS = 3840
[CUDA MANAGER] Time GPU 8.11 microseconds
[CUDA MANAGER] Memory occupied by the picture is 14745600 Bytes
[CPU MANAGER] Time CPU 439675 microseconds
[CPU MANAGER] Memory occupied by the picture is 14745600 Bytes
```

Figura 4: Comparativa entre CUDA y CPU

Como podemos observar el rendimiento de ejecución con CUDA frente a la ejecución de la CPU es muy alto, debido a que la GPU representa una arquitecura SIMD (Single Instruction, Multiple Data), frente a una de SISD (Single Instruction, Single Data).

4.3. Nivel de Frames

En esta sección de rendimiento, como he definido anteriormente, mi programa es capaz de implementar una serie de filtros a un video o incluso al conjunto de frames captados de la WebCam. He establecido una comparativa entre dos videos, cuya calidad es diferente utilizando el filtro *Sharpen*.

```
gsergey@ubuntu:~/Desktop/CUDA_Filter$ make video-sharpen
./cuda_kernels 1 "sharpen" video/video.mp4
[CUDA MANAGER] Frames per second: 1
[CUDA MANAGER] Frames per second: 11
[CUDA MANAGER] Frames per second: 23
[CUDA MANAGER] Frames per second: 23
[CUDA MANAGER] Frames per second: 24
[CUDA MANAGER] Frames per second: 24
[CUDA MANAGER] Frames per second: 24
[CUDA MANAGER] Frames per second: 25
```

Estos son los datos del video de baja calidad.

```
gsergey@ubuntu:~/Desktop/CUDA_Filter$ make video-sharpen
./cuda_kernels 1 "sharpen" video/4k.mp4
[CUDA MANAGER] Frames per second: 7
[CUDA MANAGER] Frames per second: 19
[CUDA MANAGER] Frames per second: 20
[CUDA MANAGER] Frames per second: 19
[CUDA MANAGER] Frames per second: 21
[CUDA MANAGER] Frames per second: 19
```

Y estos son los datos asociados al vídeo con una resolución 4K. Como podemos observar, la única diferencia es el número de frames por segundo. Esto es debido a la calidad de cada vídeo, ya que el primero presenta una calidad inferior al segundo. Esto nos permite demostar que el filtro realiza de forma satisfactoria su trabajo ya que en todo momento los frames tienen su filtro asociado gracias a la ejecución paralela de la GPU.

En este trabajo también he incluido la entrada de frames a través de la WebCam permitiendo aplicar un kernel a cada uno de ellos. En este caso, he realizado una demostración de la entrada de frames aplicando el filtro *Sharpen*:

```
gsergey@ubuntu:~/Desktop/CUDA_Filter$ make live-sharpen
./cuda_kernels 2 "sharpen" ""
[CUDA MANAGER] Frames per second: 4
[CUDA MANAGER] Frames per second: 26
[CUDA MANAGER] Frames per second: 22
[CUDA MANAGER] Frames per second: 22
[CUDA MANAGER] Frames per second: 21
[CUDA MANAGER] Frames per second: 21
[CUDA MANAGER] Frames per second: 22
[CUDA MANAGER] Frames per second: 22
[CUDA MANAGER] Frames per second: 22
[CUDA MANAGER] Frames per second: 24
```

Como podemos observar, la cantidad de frames por segundo es muy cercana al valor 30 ya que es la medida que la WebCam capta los frames sin ningún filtro aplicado. También debemos tener en cuenta que influye una serie de factores en la captación de frames, como por ejemplo la intensidad lumínica o la cercanía frente a la WebCam. Esto también nos quiere demostrar que la transformación de los frames aplicando un kernel específico es rápida, llegando a ser inmediata, realizando una operación transparente al usuario.

4.4. Otras comparativas

En esta sección, he realizado una serie de comparativas entre dos procesamientos diferentes de dos imágenes para llegar a una serie de conclusiones dependiendo de los valores que obtengamos. En este caso, vamos a comparar dos imágenes distintas con el filtro *Sobel*:

Figura 5: Visual Profiler de madrid.jpg

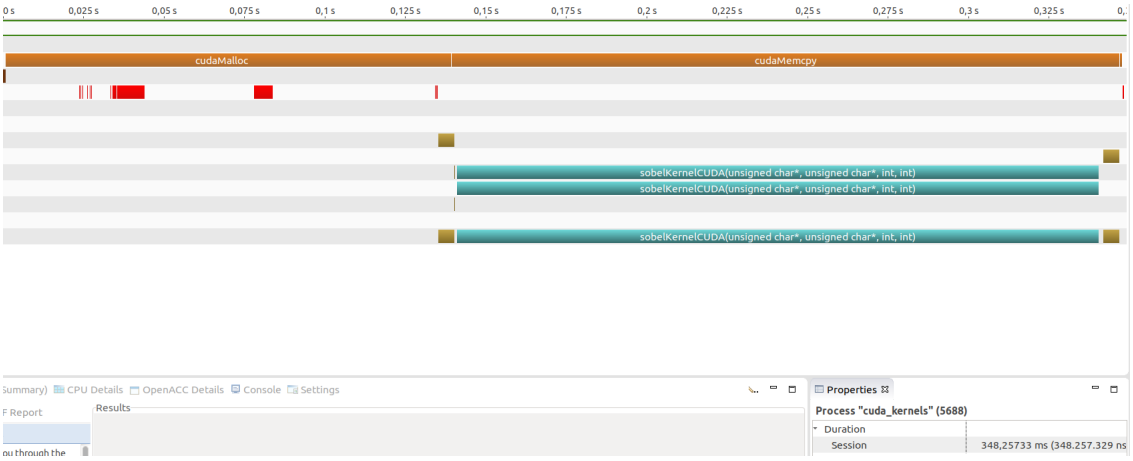

Figura 6: Visual Profiler de tesla.jpg

Como podemos observar, hay diferentes aspectos que tenemos que tener en cuenta. Si observamos detenidamente, en la imagen de *madrid.jpg* tiene un tiempo de ejecución muy bajo mientras que la imagen *tesla.jpg* el tiempo de ejecución es muy alto. Esto es debido a que el tiempo de ejecución de la aplicación del kernel es muy alto con respecto a la de *madrid.jpg*. Además, el tamaño de la imagen *madrid.jpg* es de 659x1172 píxeles mientras que *tesla.jpg* contiene 2160x3840 píxeles en total. En cuanto a la cantidad de memoria utilizada, observamos como la latencia de la instrucción **cudaMemcpy** en la imagen de *madrid.jpg* es mucho más pequeña que *tesla.jpg*, todo ello relacionado con el tamaño de la imagen. Este mismo razonamiento se puede asociar al tiempo de ejecución del kernel, conforme aumente el tamaño de la imagen, las hebras de cada bloque tardarán mucho más en aplicar el kernel.

5. Conclusión

Para concluir con este trabajo, pienso que se trata de un trabajo que engloba los contenidos que he aprendido a lo largo de esta asignatura, ya sea tanto a nivel de programación como a nivel estructural. Además, esta práctica permite entender muy bien cómo funciona la arquitectura *SIMD* haciendo posible realizar un programa paralelo que sea mucho más eficiente tanto en tiempo como en rendimiento que uno secuencial. Este trabajo está realizado con una tarjeta gráfica: NVIDIA-GEFORCE 920MX.

6. Referencias

1. *Repositorio de GitHub CUDA-OpenCV*, Autor: *Levid Rodriguez*
2. *Intalación y configuración de la WebCam con OpenCV*
3. *Cálculo de frames/segundo*