Práctica 3

Box Filter

Julio Martín y Raquel Peces

ÍNDICE

DESCRIPCIÓN DEL CÓDIGO

La descripción de las principales funciones implementadas sería:

Box_filter

Este kernel es el encargado de aplicar el filtro a la imagen, tendrá en cuenta el radio sobre el que actúa el filtro. Tendremos especial cuidado con los pixeles que se encuentran en el borde, si uno de los valores sale de la imagen, le asignaremos el valor más cercano en dicho borde y que se encuentre dentro de la imagen. Además será dónde le indiquemos al programa el uso de la variable constante *filter* a través del uso de "__restrict__".

El filtro por defecto es el Laplace 5x5, puede compararse el resultado obtenido en la imagen resultado "resultado_laplace5.png".

SeparateChannels

Separa los valores RGBA, obteniendo variables para cada uno de los colores Red, Green, Blue. Esto supone una mejora porque hacemos que no haya dependencia de los datos, aspecto a evitar en un cauce masivamente paralelo como es el de la GPU y además favorece la coalescencia, al enviar un menor número de bytes a memoria de vídeo.

AllocateMemoryAndCopyToGPU

Realiza la reserva de memoria para las variables de color y del filtro respectivamente, a través de la función "cudaMalloc" y a su vez envía el filtro de memoria del host a memoria de video a través de la función "cudaMemcpy".

Create_filter

Es donde se definen cada uno de los filtros, hay que tener cuidado con el tamaño del filtro, ya que es una constante global y puede variar de un filtro a otro. Los que nosotros hemos definido pueden dividirse en 6 grandes familias, filtro gaussiano, Laplaciano 5x5, filtros de nitidez, filtros de gradiente, filtros de detección de línea y filtros de suavizado. Expondremos los filtros utilizados y los resultados obtenidos, en el siguiente apartado.

Convolution

Es la encargada de calcular el tamaño de grid y de bloque y de lanzar los distintos kernels, el tamaño de grid estará definido por el tamaño de la imagen y de bloque para aprovechar al máximo los recursos, mediante la expresión:

```
(numCols-1)/BLOCK_SIZE+1, (numRows-1)/BLOCK_SIZE+1
```

El tamaño de bloque de 32 x 32 viene definido por las especificaciones de nuestra tarjeta gráfica, una GeForce 840M, que puede lanzar hasta un máximo de 1024 threads por bloque y con un tamaño de Warp de 32, como podemos observar en la captura de pantalla que se muestra debajo de estas líneas en la *Figura 1*:

Figura 1: Especificaciones de GeForce 840M

Cleanup

En esta función ya se realizaba la liberación de memoria para las variables de color y únicamente hemos introducido la liberación de memoria para la variable del filtro.

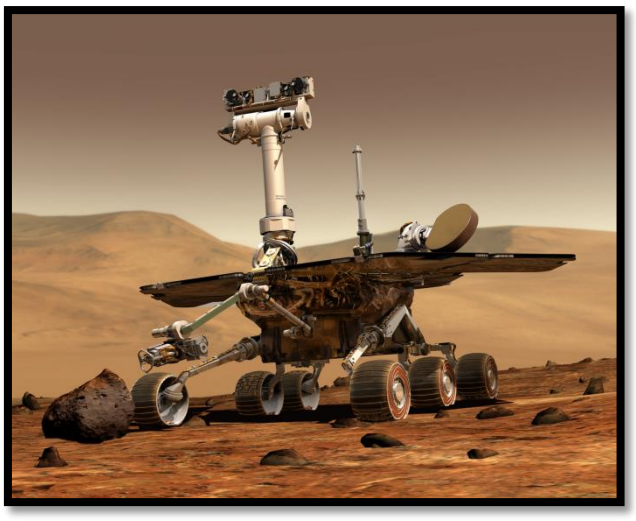
FILTROS IMPLEMENTADOS

En este apartado vamos a mostrar los resultados de aplicar cada uno de los filtros especificados, mostraremos una tabla dónde aparecerán los valores del filtro, la imagen resultante y el tiempo en calcularlos.

Los dividiremos en las 6 grandes familias ya mencionadas anteriormente, filtro gaussiano, Laplaciano 5x5, filtros de nitidez, filtros de gradiente, filtros de detección de línea y filtros de suavizado:

Figura 2: Imagen original

Blur

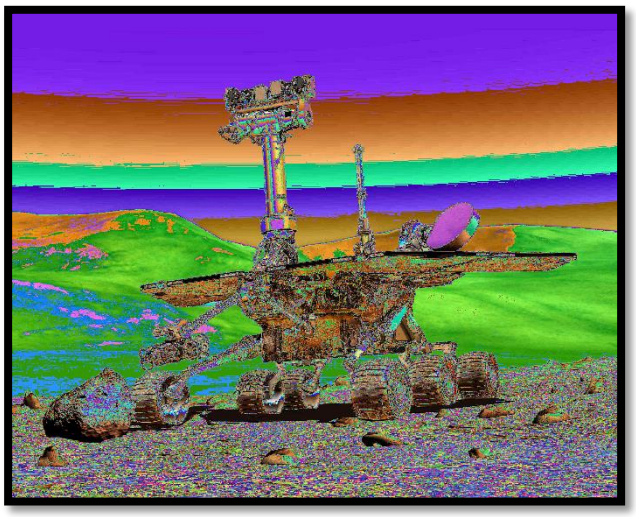


```
Your code ran in: 721.245361 msecs.
```

Figura 3: Filtro Blur ya implementado en la práctica

Laplace 5x5

0	0	-1	0	0
1	-1	-2	-1	0
-1	-2	17	-2	-1
1	-1	-2	-1	0
1	0	-1	0	0



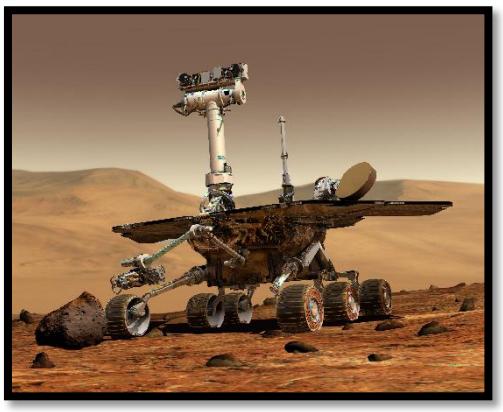
```
Your code ran in: 718.298340 msecs.
```

Figura 4: Filtro Laplace 5x5

Filtros de nitidez 3x3

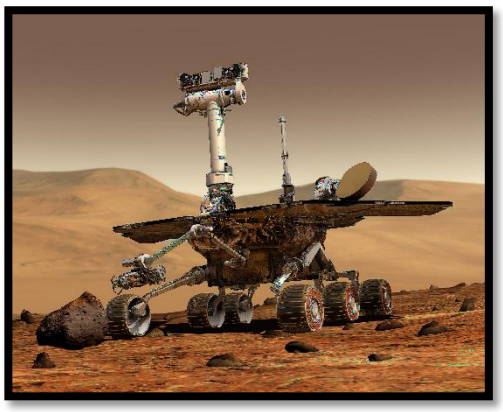
0	-0.25	0
-0.25	2	-0.25
0	-0.25	0

-0.25	-0.25	-0.25
-0.25	3	-0.25
-0.25	-0.25	-0.25



```
Your code ran in: 373.002777 msecs.
```

Figura 5: Filtro I Nitidez 3x3

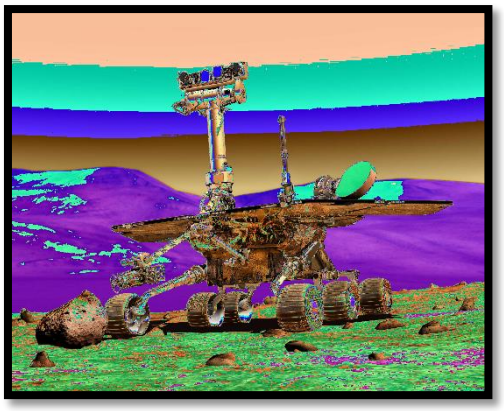


```
Your code ran in: 366.714386 msecs.
```

Figura 6: Filtro II Nitidez 3x3

Filtros de gradiente 3x3

```
1  0  1
2  0 -2
1  0 -1
```



```
Your code ran in: 372.526062 msecs.
```

Figura 7: Filtro gradiente Este

```
0 -1 -2
1  0 -1
2  1  0
```

```
Your code ran in: 374.576935 msecs.
```

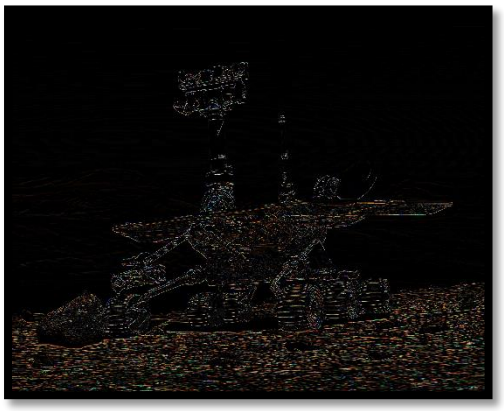
Figura 8: Filtro gradiente NordEste

```
-2 -1  0
-1  0  1
 0  1  2
```

```
Your code ran in: 375.106262 msecs.
```

Figura 9: Filtro gradiente NordOeste

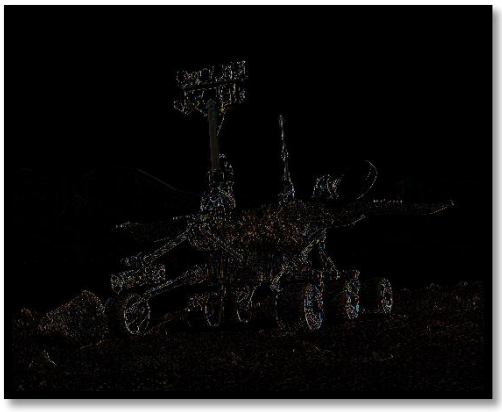
```
-1 -2 -1
 0  0  0
 1  2  1
```



```
Your code ran in: 375.450195 msecs.
```

Figura 10: Filtro gradiente Norte

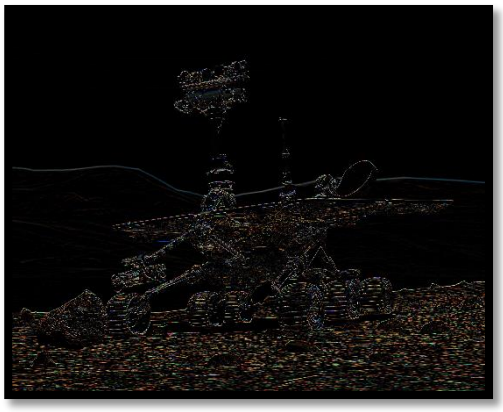
-1	0	1
-2	0	2
-1	0	1



```
Your code ran in: 374.265106 msecs.
```

Figura 11: Filtro gradiente Oeste

1	2	1
0	0	0
-1	-2	-1



```
Your code ran in: 369.944000 msecs.
```

Figura 12: Filtro gradiente Sur

Filtros de detección de línea 3x3

-1	-1	2
-1	2	-1
2	-1	-1

```
Your code ran in: 366.737427 msecs.
```

Figura 13: Filtro línea diagonal derecha

2	-1	-1
-1	2	-1
-1	-1	2

```
Your code ran in: 363.251190 msecs.
```

Figura 14: Filtro línea diagonal izquierda

-1	-1	-1
2	2	2
-1	-1	-1

```
Your code ran in: 367.805176 msecs.
```

Figura 15: Filtro línea Horizontal

-1	2	-1
-1	2	-1
-1	2	-1

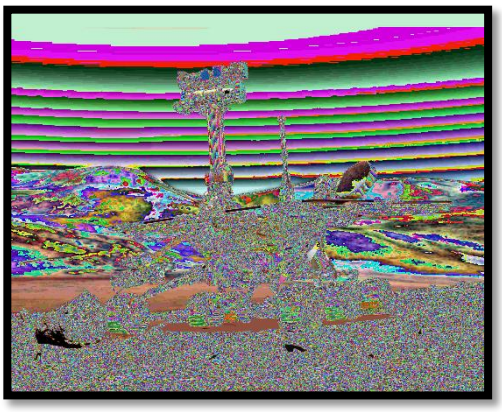
```
Your code ran in: 372.723175 msecs.
```

Figura 16: Filtro línea vertical

* El filtro vertical de la web marca el elemento de la fila 1 columna 2 como un 0, lo que creemos que es un error, lo hemos corregido y puesto un dos, obteniendo el resultado que se obtiene en la *"figura 16".*

Filtros de suavizado 3x3, 5x5, NxN

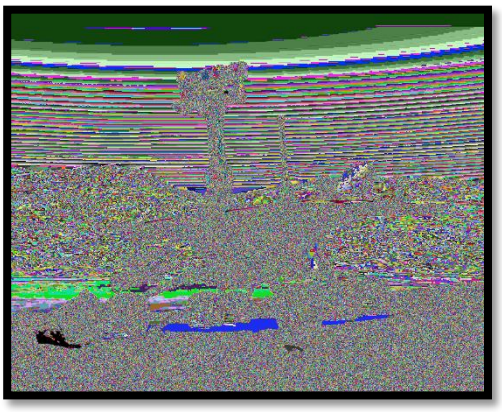
1	2	1
2	4	2
1	2	1



```
Your code ran in: 368.718933 msecs.
```

Figura 17: Filtro suavizado 3x3

1	**1**	**1**	**1**	**1**
1	**4**	**4**	**4**	**1**
1	**4**	**12**	**4**	**1**
1	**4**	**4**	**4**	**1**
1	**1**	**1**	**1**	**1**



```
Your code ran in: 730.784302 msecs.
```

Figura 18: Filtro suavizado 5x5

Filtros media aritmética NxN

Hemos realizado un algoritmo genérico, para poder calcular filtros de media aritmética, para ello nos hemos basado en el filtro 3x3, como se compone de nueve valores, dividimos 1/9 = 0.111, para 10x10 sería 1/100=0.01, para 16x16 sería 1/256= 0.0039:

0.111	0.111	0.111
0.111	0.111	0.111
0.111	0.111	0.111

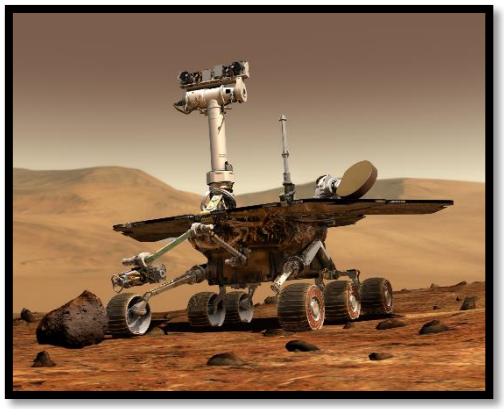


Figura 19: Media aritmética 3x3

Figura 20: Media aritmética 10x10

Figura 21: Media aritmética 16x16

***** El tiempo de cálculo se va incrementando de forma directamente proporcional al tamaño del filtro.

MEJORAS INTRODUCIDAS

Tamaño de bloque

Como se ha especificado en el primer apartado, la tarjeta gráfica sobre la que lanzamos el código es una nVidia GeForce 840M, que acepta 1024 threads por bloque como se muestra en la *"figura 22"*, por lo tanto para mejorar el rendimiento lo único que debemos hacer es especificar el tamaño de bloque a 32x32, que mejora con respecto a la especificación de 16x16 u 8x8.

```
Maximum number of threads per multiprocessor:  2048
Maximum number of threads per block:           1024
```

Figura 22: Resultado de deviceQuery sobre nVidia GeForce 840M

Memoria de constantes

En el kernel box_filter, marcaremos la variable filter como constante, para ello en la entrada al método pondremos:

```
const float* __restrict__ filter
```

Si tomamos tiempos los resultados obtenidos son:

```
Your code ran in: 614.408691 msecs.
```

Figura 23 : Tiempo SIN memoria de constantes

```
Your code ran in: 597.994446 msecs.
```

Figura 24: Tiempo CON memoria de constantes

Memoria compartida

En este caso vamos a realzar dos mejoras, por un lado el uso de memoria compartida para agilizar los accesos a memoria y por otro lado evitar los *"ifs"* para no tener divergencia en los threads. Traeremos a memoria compartida el tamaño de bloque, más un margen que vendrá definido por el radio del filtro, para evitar ir a memoria global, obteniendo la siguiente mejora de tiempos:

```
Your code ran in: 597.994446 msecs.
```

Figura 25: Tiempo SIN memoria compartida

```
Your code ran in: 485.096680 msecs.
```

Figura 26: Tiempo CON memoria compartida

Para evitar poner *"ifs"* vamos a utilizar la fórmula de operar con booleanas, de forma que utilizamos alguna operación más, pero evitamos la divergencia de las instrucciones condicionales:

```
        baux = !((fx < 0) || (fy < 0));
        ds_inputChannel[x0][y0] = inputChannel[fy*numCols + fx] * baux;
```

COMENTARIOS PERSONALES

Problemas encontrados

La dificultad a la hora de implementar memoria compartida, llegamos a implementar 3 métodos distintos, sin que en un primer momento nos dieran los resultados esperados, siempre nos tardaba más tiempo que el algoritmo que no hacía uso de esta memoria. Al final tuvimos que desarrollar distintos métodos al respecto para que nos funcionara todo bien.

También nos encontramos que el uso de memoria de constantes, en la documentación encontrada al respecto, realizada por otros autores, se implementaba de otra forma distinta a la que se pedía en la práctica, lo que nos retrasó mucho a la hora de realizar ese pequeño apartado.

Crítica constructiva

Sobre todo esta primera práctica de CUDA, debería haberse propuesto antes de acabar el periodo lectivo, ya que facilita la resolución de dudas con los profesores y con el resto de compañeros.

Propuesta de mejora

Creemos que el tema de la práctica está bien y que realmente se usan mucho este tipo de filtros, el orientarla a temas gráficos es algo que le aporta calidad a la asignatura de cara al master que realizamos.

Quizás como pega, no llegamos a conocer más casos reales de uso, lo que realmente se utiliza en la industria.

Evaluar tiempo dedicado

El tiempo de la práctica es adecuado, lo que más nos ha retrasado ha sido el uso de memoria compartida, pero el resto de la práctica se ha realizado en un tiempo prudencial, la mayor pega ha sido tener que realizarla durante el periodo de exámenes. Realmente la última semana lectiva no teníamos ninguna otra entrega y hubiese estado bien poderla adelantar.

ÍNDICE DE FIGURAS