

TGA - Proyecto de laboratorio

Tratamiento de imágenes

Albert Bernal e Ismael Quiñones

Junio de 2021



UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

ÍNDICE

1.- Introducción. Problema a resolver.	2
1.1.- Blanco y negro secuencial	2
1.2.- Saturar secuencial	2
1.3.- Sepia secuencial	3
1.4.- Cambiar brillo secuencial	3
1.5.- Ecuallización con histograma secuencial	3
2.- Implementaciones CUDA e imágenes generadas	5
2.1.- Blanco y negro CUDA	5
2.2. - Saturar CUDA	6
2.3. - Sepia CUDA	6
2.4. - Brillo CUDA	7
2.5.- Ecuallización con histograma CUDA	8
3.- Rendimiento	10
3.1.- integer vs float?	10
3.2.- Blanco y negro	10
3.3.- Saturación	11
3.4.- Sepia	11
3.5.- Brillo	11
3.6.- Ecuallizador	12

1.- Introducción. Problema a resolver.

El tratamiento de imágenes es uno de los campos donde más se puede aprovechar el paralelismo en los datos, debido a que generalmente se quiere aplicar un mismo tratamiento sobre todos los píxeles de una imagen, sin haber ningún tipo de dependencia entre ellos.

Por este mismo motivo, nuestro trabajo de la asignatura se va a centrar en aplicar una serie de filtros sobre imágenes, en CUDA, ya que éste nos ayudará a explotar el paralelismo de este problema de forma inmensa en tarjetas Nvidia.

Los filtros que vamos a aplicar son los siguientes:

- Blanco y negro
- Saturar
- Sepia
- Cambiar brillo
- Ecualización con histograma

A continuación mostramos los códigos secuenciales utilizados:

1.1.- Blanco y negro secuencial

```
int C; int size = width*height*3;
for(int i=0; i<size; i=i+3){
    C = 299 * image[i];
    C += 587 * image[i+1];
    C += 114 * image[i+2];
    image[i] = image[i+1] = image[i+2] = C/1000;
}
```

Las proporciones (299, 587, 114) las extrajimos del último comentario a la última respuesta de la siguiente fuente:

<https://stackoverflow.com/questions/17615963/standard-rgb-to-grayscale-conversion>

Nota que estamos utilizando ints en lugar de floats (0.299, 0.597, 0.114). Cuando hagamos el código CUDA lo faremos con ambas versiones (ints y floats) y compararemos el rendimiento, a ver si hay una diferencia destacable.

1.2.- Saturar secuencial

```
int size = width*height*3; int R, G, B; float P;
for(int i=0; i<size; i=i+3){
    R = image[i]; G = image[i+1]; B = image[i+2];
    P = sqrt( R*R*Pr + G*G*Pg + B*B*Pb ) ;
```

```

R= MIN(P+(R-P)*saturacion , 255);
G= MIN(P+(G-P)*saturacion , 255);
B= MIN(P+(B-P)*saturacion , 255);
image[i] = R; image[i+1] = G; image[i+2] = B;
}

```

El código secuencial para saturar una imagen lo encontramos en la siguiente fuente:

<https://alienryderflex.com/saturation.html>

Y lo adaptamos ligeramente (por ejemplo añadimos el “MIN (X, 255)”).

1.3.- Sepia secuencial

```

int size = width*height*3; int R, G, B;
for(int i=0; i<size; i=i+3){
    R = MIN(255, (image[i]*0.393) + (image[i+1]*0.769) +
(image[i+2]*0.189));
    G = MIN(255, (image[i]*0.349) + (image[i+1]*0.686) +
(image[i+2]*0.168));
    B = MIN(255, (image[i]*0.272) + (image[i+1]*0.534) +
(image[i+2]*0.131));
    image[i] = R; image[i+1] = G; image[i+2] = B;
}

```

El código secuencial lo encontramos en la siguiente fuente:

<https://abhijitnathwani.github.io/blog/2018/01/08/colorosepia-Image-using-C>

También fue ligeramente adaptado de acuerdo a nuestras necesidades.

1.4.- Cambiar brillo secuencial

```

int size = width*height*3; int R, G, B;
for(int i=0; i<size; i=i+3){
    R = MIN(255, (image[i]*brillo/100));
    G = MIN(255, (image[i+1]*brillo/100));
    B = MIN(255, (image[i+2]*brillo)/100);
    image[i] = R; image[i+1] = G; image[i+2] = B;
}

```

Para este filtro no hemos mirado en ninguna web ya que conocemos que subir o bajar el brillo es simplemente aumentar o disminuir los valores RGB.

1.5.- Ecualización con histograma secuencial

Este filtro tratará de ecualizar una imagen. Y hemos decidido hacerlo con código propio de forma que funcione para todas las imágenes, ya sean en color o monocromáticas.

Para generar el histograma ejecutamos:

```

for (int i=0; i<size; i=i+3) {
    redH[image[i]]++;
    greenH[image[i+1]]++;
    blueH[image[i+2]]++;
}

```

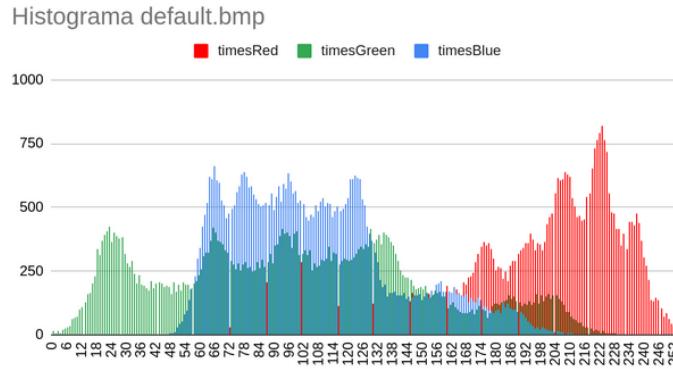


Figura 1. Histograma obtenido al procesar la imagen **default.bmp**

Para obtener los valores mínimos y máximos del histograma¹ (código para encontrar mínimo rojo):

```

for (int i=0; !found && i<256; ++i) {
    if (redH[i]>3) {
        minR=i;
        found = true;
    }
}

```

Para ecualizar tendremos que aplicar la transformación a cada canal: R, G y B. Separamos la imagen por canales:

```

for (int i=0; i<height*width; i++) {
    redChannel[i] = image[i*3];
    greenChannel[i] = image[i*3+1];
    blueChannel[i] = image[i*3+2];
}

```

Ahora usaremos los valores mínimos y máximos para expandir la parte del histograma con más información relevante y actualizaremos la imagen con los nuevos canales R, G y B:

```

double multR = (double)255/((double)maxR-(double)minR);
double multG = (double)255/((double)maxG-(double)minG);
double multB = (double)255/((double)maxB-(double)minB);

for (int i=0; i<height*width; ++i) {
    R = (redChannel[i]-minR)*multR;
    image[i*3] = MIN(255, R);
    G = (greenChannel[i]-minG)*multG;
    image[i*3+1] = MIN(255, G);
    B = (blueChannel[i]-minB)*multB;
}

```

¹ En este ejemplo usamos 3 como valor mínimo de apariciones, pero puede ser alterado para obtener un filtro más agresivo, es decir, que descarte más valores a izquierda y derecha de la sección con más información.

```

    image[i*3+2] = MIN(255, B);
}

```

2.- Implementaciones CUDA e imágenes generadas

En esta sección mostraremos las implementaciones CUDA de los filtros cuyos códigos secuenciales se han mostrado en la anterior sección. Así mismo, demostraremos que funcionan correctamente mostrando las imágenes generadas por cada uno de los filtros. El fichero que contiene estas implementaciones se llama *filtrar.cu*.

En la siguiente sección se hará un análisis de la performance de nuestros códigos.

Vamos a mostrar únicamente los kernels, ya que es lo realmente relevante de la aplicación, porque el resto de código de la función main es realmente parecido al resto de sesiones de laboratorio y puede ser visto en los ficheros adjuntos a esta entrega.

Para todas las versiones se ha hecho una distribución de 1024 threads por bloque, en la que cada thread se encargará de 1 cierto píxel (es decir, de 3 elementos de la imagen: las componentes RGB de 1 píxel). En todos los casos se ha comprobado si ha habido error en las llamadas CUDA.

2.1.- Blanco y negro CUDA

```

__global__ void KernelBlancoNegro(unsigned char *image, int N) {
    int elem = (blockIdx.x * blockDim.x + threadIdx.x)*3;
    if(elem < N){
        int C;
        C = 299 * image[elem];
        C += 587 * image[elem+1];
        C += 114 * image[elem+2];
        image[elem] = image[elem+1] = image[elem+2] = C/1000;
    }
}

```

Como la imagen no tiene por qué ser múltiplo de nThreads (1024), hacemos la comprobación de “if(elem < N)” para asegurarnos que los últimos threads de todos no actúen sobre unas posiciones inexistentes de la imagen.

La figura 2 muestra, en la parte izquierda, la imagen de la que partimos y, en la parte derecha, la imagen resultante de aplicarle el filtro con el código CUDA:



Figura 2. Izquierda: imagen original. Derecha: filtro blanco y negro aplicado.

2.2. - Saturar CUDA

```

int elem = (blockIdx.x * blockDim.x + threadIdx.x)*3;
if(elem < N){
    int R, G, B; float P;
    R = image[elem]; G = image[elem+1]; B = image[elem+2];
    P = sqrt( R*R*0.299 + G*G*0.587 + B*B*0.114 );
    R= MIN(P+(R-P)*saturacion , 255);
    G= MIN(P+(G-P)*saturacion , 255);
    B= MIN(P+(B-P)*saturacion , 255);
    image[elem] = R; image[elem+1] = G; image[elem+2] = B;
}

```

Ese código mostrado es el kernel de la saturación, el motivo del if ya ha sido explicado anteriormente. La figura 3 muestra la aplicación de este filtro con un **factor de saturación de 1.5**.



Figura 3. Izquierda: imagen original. Derecha: filtro saturación con factor 1.5 .

Este filtro, en caso de que se quiera bajar la saturación, se puede probar con cualquier imagen de la que se nos proporcionó en la práctica; pero si se quiere utilizar para aumentar la saturación recomendamos únicamente utilizar **IMG01.jpg** e **IMG02.jpg**. Esto se debe a que las demás imágenes nos dan la sensación de que de por sí ya están algo saturadas y, por ello, al subirla aún más la saturación se queda una imagen que no es nada vistosa, con algunos colores sobresaturados.

2.3. - Sepia CUDA

```
int elem = (blockIdx.x * blockDim.x + threadIdx.x)*3;
if(elem < N){
    int R, G, B;
    R = MIN(255, (image[elem]*393 + image[elem+1]*769 + image[elem+2]*189)/1000
);
    G = MIN(255, (image[elem]*349 + image[elem+1]*686 + image[elem+2]*168)/1000
);
    B = MIN(255, (image[elem]*272 + image[elem+1]*534 + image[elem+2]*131)/1000
);
    image[elem] = R; image[elem+1] = G; image[elem+2] = B;
}
```

La figura 4 muestra la aplicación de este filtro:



Figura 4. Izquierda: imagen original. Derecha: filtro sepia.

2.4. - Brillo CUDA

```
int elem = (blockIdx.x * blockDim.x + threadIdx.x)*3;
if(elem < N){
    int R, G, B;
    R = MIN(255, (image[elem]*brillo/100));
    G = MIN(255, (image[elem+1]*brillo/100));
    B = MIN(255, (image[elem+2]*brillo/100));
    image[elem] = R; image[elem+1] = G; image[elem+2] = B;
}
```

La figura 5 muestra la aplicación de este filtro con un **brillo del 150%**.



Figura 5. Izquierda: imagen original. Derecha: filtro brillo con un brillo del 150%

Es muy interesante ver cómo queda la imagen si, por ejemplo, se le baja el brillo al 50% (es decir, se reduce a la mitad la iluminación); no lo mostramos por no alargar más la práctica con imágenes (igual que no mostramos tampoco el caso de saturación 0.5) pero es realmente interesante de probar.

Nota que al ejecutar los archivos de la práctica el brillo se le indica en tanto por ciento (a diferencia de la saturación que se indicaba en tanto por uno). Eso es porque nos parecía más lógico hablar de brillos en porcentajes y de saturaciones en factores de tanto por uno.

2.5.- Ecualización con histograma CUDA

En esta ocasión hay bastante más código, sobre todo para generar el histograma y buscar los mínimos y máximos. Esa parte del código es secuencial y la ejecutamos antes de llegar al kernel, cuyo código es el siguiente:

```
int elem = (blockIdx.x * blockDim.x + threadIdx.x);
if (elem < N) {
    int R, G, B;
    R = (redChannel[elem]-minR)*multR;
    image[elem*3] = MIN(255, R);
    G = (greenChannel[elem]-minG)*multG;
    image[elem*3+1] = MIN(255, G);
    B = (blueChannel[elem]-minB)*multB;
    image[elem*3+2] = MIN(255, B);
}
```

Como se puede apreciar, en este kernel procesamos la imagen de forma distinta ya que en realidad recorremos los 3 canales, y eso hace que *elem* sea 3 veces menor que en los demás filtros, donde recorremos la imagen en sí.

Para la ejecución de la ecualización usamos el archivo **default.bmp**, claramente saturada. Para obtener la siguiente imagen, se aplicó la transformación con 51-255, 0-228 y 48-208 como mínimos y máximos para los canales rojo, verde y azul, respectivamente:



Figura 6. Izquierda: imagen original. Derecha: imagen ecualizada con filtro *hardcoded*

Este ha sido el mejor escenario que hemos encontrado tras probar distintos valores de multiplicador para cada canal. Pese a que ha sido más trabajo, después de crear un filtro con valores *hardcoded* que funcionase sobre una imagen en color como es **default.bmp**, **hemos optado por hacer un tratamiento del histograma que funciona para cualquier imagen, ya sea en color o monocromo.** Al ser más generalista y por tanto sin valores *hardcoded*, pensábamos que la transformación no obtendría un resultado tan vistoso, pero, tal como podemos ver en la Figura 7, hemos conseguido un resultado prácticamente idéntico:

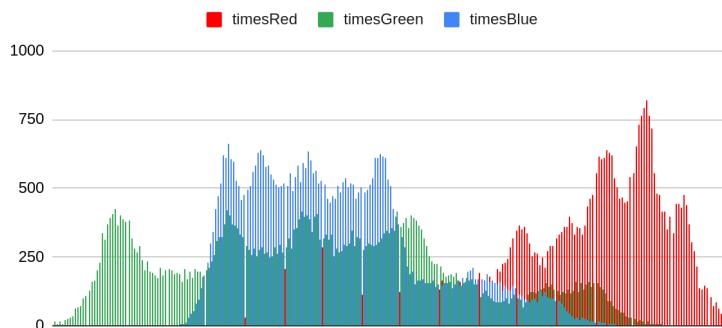


Figura 7. imagen ecualizada con filtro generalista

Y ahora sí podemos afirmar que es capaz de mejorar la ecualización de cualquier imagen y no solamente de **default.bmp**, gracias a haber cambiado el filtro de forma que sea generalista.

Pese a ser claramente visible la mejora en la imagen, es interesante ver el nuevo histograma para poder apreciar mejor el cambio:

Histograma default.bmp



Histograma default.bmp ecualizado

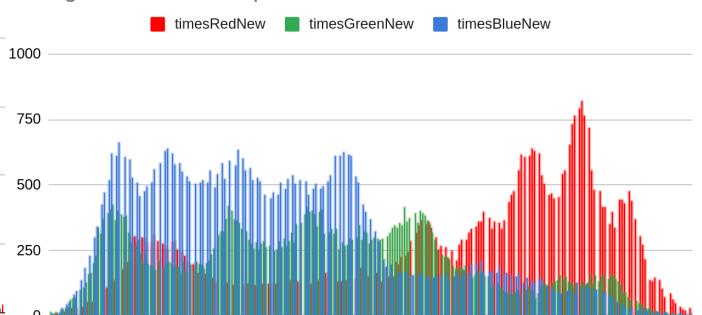


Figura 7. Izquierda: histograma imagen original. Derecha: histograma imagen ecualizada

Como se puede observar, sobre todo en la franja azul que tenía la información más centrada, hemos expandido los 3 sectores para obtener un resultado más uniforme.

3.- Rendimiento

En esta sección analizaremos el rendimiento de las implementaciones CUDA y lo compararemos con el rendimiento secuencial. Todas los datos sobre tiempos, GFLOPS, etc. serán la media de los obtenidos en múltiples ejecuciones aplicadas sobre la imagen **IMG01.jpg** y sin utilizar PINNED MEMORY.

3.1.- integer vs float?

En algunos de los kernel se podía utilizar indistintamente integers y floats. La versión con integers tiene la ventaja de que evita todas las operaciones de conversión entre tipos (las cuales son caras) y la versión con floats tiene la ventaja de que evita operaciones de división (entre 1000 en estos casos), las cuales son muy caras. Entonces, ¿hay alguna opción mejor que la otra? La respuesta es que sí, veámoslo con el filtro blanco y negro:

Media tiempos ejecución Kernel utilizando floats: 1.431 ms

Media tiempos ejecución Kernel utilizando ints: 1.240 ms

Por ende, el hecho de utilizar enteros nos ha proporcionado un **SpeedUp** del **15.40%**.

Debido a esta diferencia, en todos los kernels que se ha podido, se han utilizado enteros. La excepción ha estado en el filtro de saturación, ya que para el correcto funcionamiento ha sido necesario que utilice coma flotante; igual que para la ecualización.

3.2.- Blanco y negro

Media tiempo transferencia H->D: 20,16 ms

Media tiempo ejecución Kernel: 1,24 ms

Media tiempo transferencia D->H: 19,06 ms

La cantidad de píxeles de IMG01.jpg es de $5484 \times 3456 = 17.915.604$ (cada píxel representado por 3 Bytes, es decir, 53,75 MB). Por lo tanto:

Ancho de banda H->D: 2,67 GB/s

Ancho de banda D->H: 2,82 GB/s

La cantidad de operaciones aritméticas hechas por píxel es de 6. Por lo tanto:

GOP/s Kernel: $17.915.604 * 6 \text{ operaciones} / 0,00124 \text{ segundos} / 10^9 = 86,69 \text{ GOP/s}$

Nota que se han utilizado GOPs en lugar de GFLOPS porque este kernel no utiliza operaciones aritméticas de punto flotante.

El tiempo de la **ejecución secuencial** del filtro (únicamente contando el bucle que recorre la imagen y aplica el filtro) ha sido de 104,25 ms. Eso, al convertirlo en código para GPU se ha convertido en una serie de llamadas (cudaMalloc, cudaMemcpy, ejecución kernel, gestión de errores etc.) cuyo tiempo en total ha estado de 44,54 ms. Por tanto, el **SpeedUp** obtenido es de 2,34x. Y el SpeedUp que se obtendría si solamente tuviéramos en cuenta el Kernel sería de 87,07x.

3.3.- Saturación

Los **tiempos de transferencia, ancho de banda** obtenido y cantidad de datos transferidos, debido a que se trata de la misma imagen, son idénticos a los mostrados en la sección 3.1.

Media tiempo ejecución Kernel: 1,81 ms

En cuanto a la cantidad de operaciones, aquí sí que vamos a poder hablar de GFLOPs, ya que este kernel sí que trabaja con coma flotante. La cantidad de operaciones de coma flotante por píxel es 18, contando *sqrт* como 1 única operación en coma flotante. Por lo tanto:

GFLOP/s Kernel: $17.915.604 * 18 \text{ flop} / 0,00181 \text{ segundos} / 10^9 = 178,17 \text{ GFLOP/s}$

El tiempo de la **ejecución secuencial** del filtro (únicamente contando el bucle que recorre la imagen y aplica el filtro) ha sido de 322,67 ms. Eso, al convertirlo en código para GPU se ha convertido en una serie de llamadas (cudaMalloc, cudaMemcpy, ejecución kernel, gestión de errores etc.) cuyo tiempo en total ha estado de 49,10 ms. Por tanto, el **SpeedUp** obtenido es de 6,57x. Y el SpeedUp que se obtendría si solamente tuviéramos en cuenta el Kernel sería de 178,27x.

3.4.- Sepia

Los **tiempos de transferencia, ancho de banda** obtenido y cantidad de datos transferidos, debido a que se trata de la misma imagen, son idénticos a los mostrados en la sección 3.1.

Media tiempo ejecución Kernel: 1,51 ms

En cuanto a la cantidad de operaciones, vamos a volver a hablar de GOP/s, ya que este filtro vuelve a NO tener operaciones de coma flotante. La cantidad de operaciones por píxel son 18, y recordamos que había 17.915.604 píxeles en **IMG01.jpg**. Por tanto:

GOP/s Kernel: $17.915.604 * 18 \text{ op} / 0,00151 \text{ segundos} / 10^9 = 213,56 \text{ GOP/s}$

En este filtro y en el anterior se ha obtenido un mejor rendimiento que en blanco y negro. Eso probablemente sea debido a que cada thread se “exprime” más. Es decir, en el filtro blanco y negro el trabajo que hacía un thread era tan poco que probablemente ese tiempo representaba un porcentaje pequeño respecto al tiempo de lógica de “poner ese thread en funcionamiento”.

El tiempo de la **ejecución secuencial** del filtro (únicamente contando el bucle que recorre la imagen y aplica el filtro) ha sido de 194,57 ms. Eso, al convertirlo en código para GPU se ha convertido en una serie de llamadas (cudaMalloc, cudaMemcpy, ejecución kernel, gestión de errores etc.) cuyo tiempo en total ha estado de 42,46 ms. Por tanto, el **SpeedUp** obtenido es de 4,58x. Y el SpeedUp que se obtendría si solamente tuviéramos en cuenta el Kernel sería de 128,85x.

3.5.- Brillo

Los **tiempos de transferencia, ancho de banda** obtenido y cantidad de datos transferidos, debido a que se trata de la misma imagen, son idénticos a los mostrados en la sección 3.1.

Media tiempo ejecución Kernel: 2,05 ms

De nuevo aquí no hay operaciones de punto flotante así que contaremos GOP/s a secas. Por cada pixel se hacen 3 multiplicaciones y 3 divisiones, por tanto:

$$\text{GOP/s Kernel: } 17.915.604 * 6 \text{ op} / 0,00205 \text{ segundos} / 10^9 = 52,44 \text{ GOP/s}$$

El tiempo, si lo comparamos con el filtro de blanco y negro que tenía la misma cantidad de operaciones, vemos que ha sido bastante peor (2,05 ms vs 1,24 ms). Eso probablemente se deba a que este filtro hace operaciones mas costosas (ya que el blanco y negro tenía operaciones de suma) y a que en este filtro se hace la operación *mínimo*.

El tiempo de la **ejecución secuencial** del filtro (únicamente contando el bucle que recorre la imagen y aplica el filtro) ha sido de 212,82 ms. Eso, al convertirlo en código para GPU se ha convertido en una serie de llamadas (cudaMalloc, cudaMemcpy, ejecución kernel, gestión de errores etc.) cuyo tiempo en total ha estado de 44,01 ms. Por tanto, el **SpeedUp** obtenido es de 4,84 x. Y el SpeedUp que se obtendría si solamente tuviéramos en cuenta el Kernel sería de 103,81x.

3.6.- Ecualizador

Pese a que **IMG01.jpg** ya está bastante bien ecualizada y aplicarle el filtro no supone cambios muy apreciables, hemos decidido usarla para las pruebas de rendimiento ya que su gran tamaño nos permitirá ser más precisos en la toma de datos que si usáramos **default.bmp**, cuya resolución es únicamente 220x220. Cabe mencionar que en este filtro buscábamos algo funcional para cualquier imagen y por eso no es demasiado eficiente.

$$\text{Media tiempo transferencia H->D}^2: 20,16 + 3*6,64 \approx 40,88 \text{ ms}$$

$$\text{Media tiempo ejecución Kernel: } 1,66 \text{ ms}$$

$$\text{Media tiempo transferencia D->H: } 19,06 \text{ ms}$$

La cantidad de píxeles de **IMG01.jpg** es de $5484 \times 3456 = 17.915.604$ (cada píxel representado por 3 Bytes, es decir, 53,75 MB). Por lo tanto:

$$\text{Ancho de banda H->D: } 2,63 \text{ GB/s}$$

$$\text{Ancho de banda D->H: } 2,82 \text{ GB/s}$$

En cuanto a la cantidad de operaciones, aquí también vamos a poder hablar de GFLOPs, ya que este kernel también trabaja con coma flotante. La cantidad de operaciones de coma flotante por píxel es 3. Por lo tanto:

$$\text{GFLOP/s Kernel: } 17.915.604 * 3 \text{ flop} / 0,00166 \text{ segundos} / 10^9 = 32,38 \text{ GFLOP/s}$$

Nota que solo se están teniendo en cuenta las operaciones de punto flotante. Aparte de esas operaciones, hay otras 3 operaciones de sustracción de enteros por cada thread.

El tiempo de la **ejecución secuencial** del filtro (generando el histograma, encontrando mínimos y máximos y aplicando la transformación) ha sido de 201,95 ms. En el caso del filtro en sí (únicamente contando el bucle que recorre la imagen y aplica el filtro) ha sido de 119,76 ms. Eso, al pasarlo a código

² En este caso no solo transferimos la imagen, también los 3 canales que usaremos para aplicar la transformación. De los 4 tiempos mostrados, el primero se refiere a d_image y los siguientes a los 3 vectores de los 3 canales.

para GPU se ha convertido en una serie de llamadas (cudaMalloc, cudaMemcpy, ejecución kernel, gestión de errores etc.) cuyo tiempo en total ha estado de 224,33 ms y tan solo 1,66ms para el kernel. Por tanto, el **SpeedUp** obtenido en el caso de la aplicación en sí del filtro es de 72,14x. Pero, pese a este gran SpeedUp por la aplicación del filtro, el hecho de que se tenga que hacer el histograma en secuencial, encontrar mínimos y máximos, transferencias de datos desde y hacia la GPU y demás, nos hace tener en global un *downgrade* del rendimiento del $\frac{224,33 - 201,95}{201,95} * 100\% = 11,08\%$.

La causa de haber obtenido este *downgrade* en el global probablemente se deba a que, en primer lugar, el código no ha sido eficientado al máximo debido a que buscábamos un ecualizador que sirviese para cualquier foto incluso estando en color. En segundo lugar, a que no se ha utilizado *pinned memory* para mejorar tiempos de transferencia. Y, en tercer lugar, a que algunas cosas relativas al histograma podrían haberse paralelizado (como por ejemplo el hecho de dar valores a los vectores que representan cada uno de los canales, llamados redChannel, greenChannel y blueChannel).