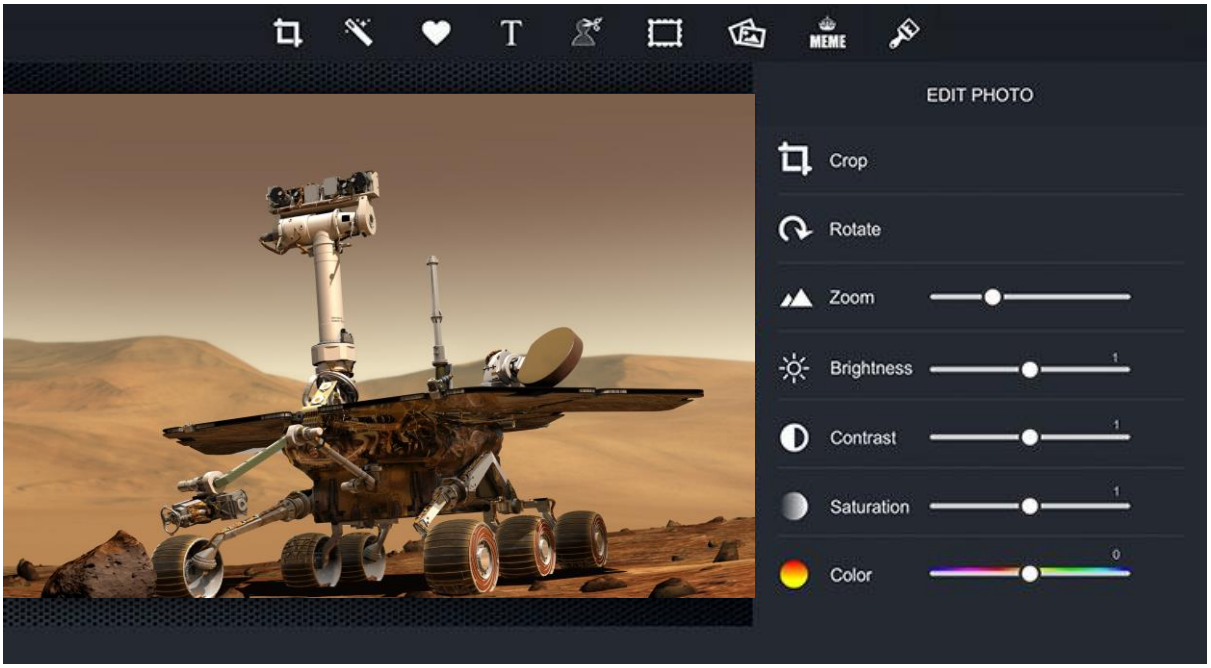
Trabajo de Investigación – Procesamiento de imágenes

Autor: Antonio Marco Rodrigo Jiménez

Fecha: 15/06/2019



Índice

Introducción

Mi trabajo de investigación para la asignatura consiste en un editor básico de imágenes programado en CUDA de forma paralela. Partiendo del código de la práctica 4 (box filter) y reutilizando sus métodos para cargar, guardar imágenes y descomponer en canales, he añadido varias funcionalidades al mismo, basadas en editores de imágenes básicos para mejorar o alterar el aspecto de las mismas, funcionalidades entre las cuales encontramos cambiar el brillo, contraste, saturación, apoyándonos en la documentación ofrecida por: *Harley R. Myler, Arthur R. Weeks. The Pocket Handbook of Image Processing Algorithms in C*

Entre las funcionalidades que ofrece el programa, encontramos:

1. Filtros de convolución (hecho de antemano en la p4)
2. Brillo
3. Contraste Legacy
4. Contraste
5. Saturación
6. Tinte
7. Inversión de color
8. Flip Horizontal
9. Flip Vertical
10. Ruido salt & pepper (efecto de granulado de foto antigua)

El programa usa como entrada una imagen y devuelve una imagen alterada como salida, con alguno de estos efectos. Como el formato de entrada y salida del programa es el mismo, se puede usar la salida del programa para usarla como entrada del mismo posteriormente, lo que permite combinar los efectos programados uno a uno para conseguir una imagen final personalizada. (Por ejemplo, ejecutar el programa para disminuir la saturación al mínimo (blanco y negro) y ejecutarlo una segunda vez con esa imagen como entrada aplicándole un ruido salt & pepper. De esta manera se consigue que la foto tenga un aspecto antiguo).

El programa permite la modificación completa de los valores que queremos utilizar para cada uno de los efectos, como la cantidad de brillo que queremos aumentar, la saturación, cuánto ruido salt & pepper queremos, el color de tinte para la imagen, etc.

Funcionamiento

Para utilizar una funcionalidad del programa, hay una sección en el código que permite elegir entre los diferentes efectos que hay programados, además del valor de los mismos. Esta sección se encuentra en el fichero "func.cu", en la función "operations" (Línea 512).

Para elegir una funcionalidad, hay que cambiar el valor del "bool" correspondiente a "true", y cambiar el valor de los demás a "false":

```
// Box Filter
bool b_boxFilter = false;
// Brillo
bool b_brightness = false;
// Contraste legacy
bool b_contrastLegacy = false;
// Contraste
bool b_contrast = false;
// Saturacion
bool b_saturation = false;
// Tinte
bool b_tint = true;
// Voltear horizontalmente
bool b_horizontalFlip = false;
// Voltear verticalmente
bool b_verticalFlip = false;
// Invertir colores
bool b_invert = false;
// Aplicar ruido saltpepper (efecto granulado)
bool b_saltpepper = false;
```

Para elegir el valor del efecto (cantidad de brillo a aumentar, por ejemplo), cambiamos el valor de los "int" o "float" correspondientes:

```
// Cantidad de brillo a aumentar o disminuir en la im
int brightnessAmount = 100; //(-255 , 255)
// -255 = todo negro
// 0 = no hay cambios
// 255 = todo blanco

// Cantidad de contraste Legacy
int contrastLegacyAmount = 10; //(0, 127)
// 0 = no hay cambios
// 127 = contraste legacy maximo

// Cantidad de contraste a cambiar
float contrastAmount = 0.5; // (0, 255) (recomm <10)
// 0 = todo gris
// 0.5 = contraste reducido
// 1 = no hay cambio
// 10 = contraste muy alto

// Saturacion de cada canal de color (rojo, verde y a
// (0, +10000) (recom <5000)
```

*(Se adjunta como comentario del código una guía para elegir valores apropiados)

Funcionalidades

Las funcionalidades del programa se pueden dividir en 2 categorías:

-Aquellas que precisan descomponer la imagen en sus 3 canales de color RGB y trabajar individualmente sobre cada uno de ellos para recombinarlos al final

-Aquellas que realizan operaciones sobre la imagen entera, sin descomponer en los canales

Operaciones sobre canales

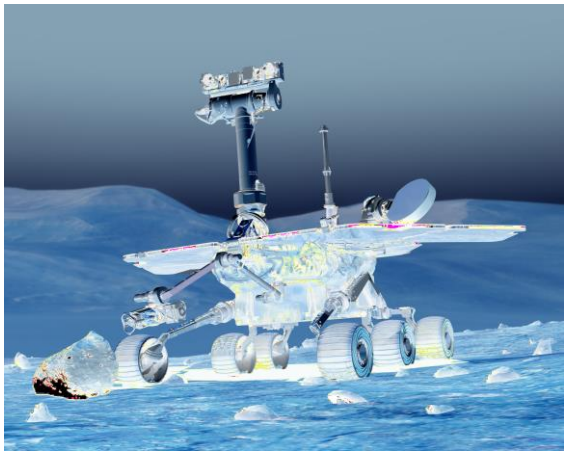
En este tipo de funcionalidades, se hace uso de 2 kernels: "separateChannels" y "recombineChannels". El primer kernel descompone la imagen de entrada en sus 3 canales de color RGB (Red, Green, Blue), y el segundo kernel une de nuevo los 3 canales por separado en la imagen final. Las operaciones que se realizan en los canales para cada funcionalidad se llevan a cabo después de llamar a "separateChannels" y antes de llamar a "recombineChannels". En esta categoría se encuentran las operaciones de invertir color, brillo, contraste Legacy, contraste nuevo, flip horizontal y box filter.

Invertir colores

Para esta tarea basta con sustituir cada valor del canal por su color inverso. El color inverso se define como: 255 – color. De manera que, si tenemos, en el canal rojo, por ejemplo, el valor 100, en la imagen final utilizamos el valor 255-100 = 155.

De esta manera, recombinando todos los canales de color, se obtendrá una imagen con los colores opuestos a los de la imagen de entrada.

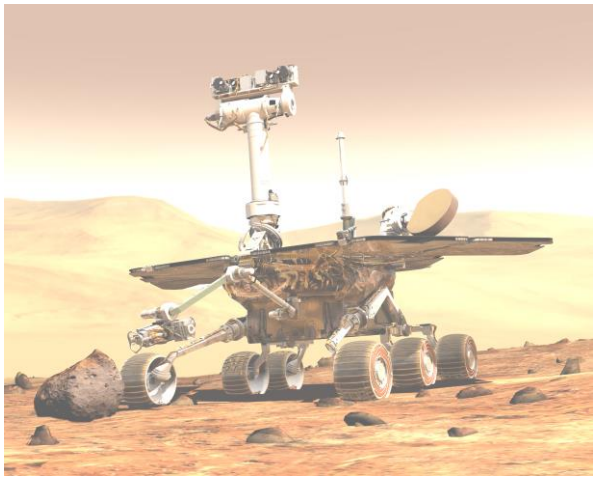
A continuación, se muestra un ejemplo del uso de esta funcionalidad:

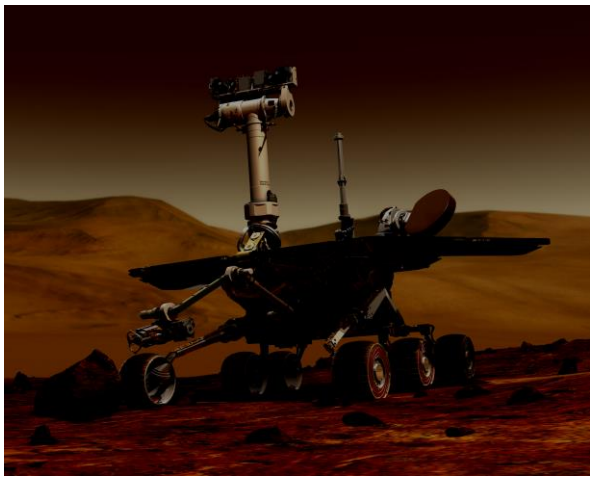


Brillo

Para realizar esta funcionalidad simplemente sumamos al valor del canal la cantidad de brillo a aumentar, o restamos la cantidad de brillo a disminuir. Antes de devolver el color final, nos aseguramos de que el valor se sigue encontrando entre 0 y 255. Si no es así, truncamos a dichos extremos.

En estas imágenes se muestra un aumento de brillo de 100, y una disminución de brillo de -100, respectivamente:



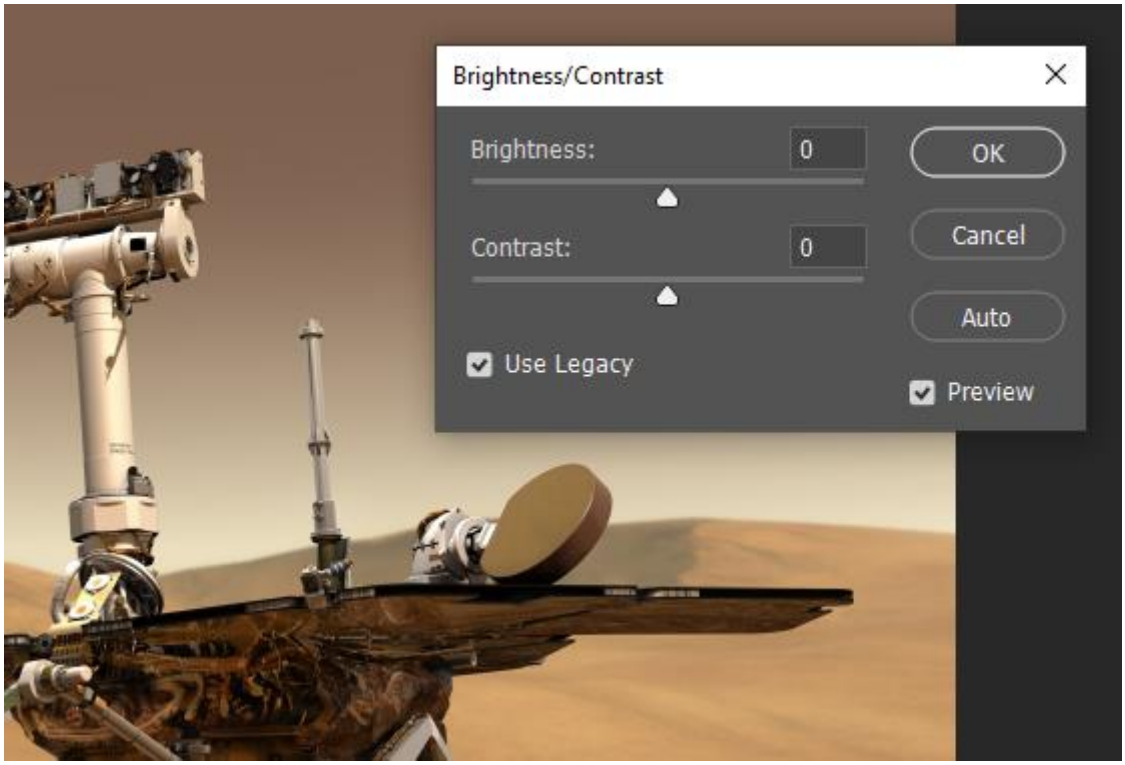


Contraste (Legacy y nuevo)

Existe una diferencia entre la funcionalidad de contraste Legacy y contraste nuevo (la funcionalidad de contraste normal actual que incorporan las herramientas modernas), ambas programadas en la aplicación. La funcionalidad de contraste Legacy aumenta o disminuye el contraste de la imagen en su totalidad, todas sus partes por igual, mientras que el contraste nuevo aumenta o disminuye el contraste de la imagen calculando el valor medio de la imagen, dando lugar a un resultado mucho más profesional.

Aumentar el contraste de una imagen implica oscurecer las partes más oscuras (valor del color menor de la mitad 255/2=127), y aclarar las partes más claras (valor del color menor mayor que 127), siendo 0 negro y 255 blanco. Esto significa que, mientras menos contraste tenga una imagen, más gris (127) se verá todo, y mientras más contraste tenga, más se diferenciarán los colores puros entre ellos y menos gris se verá (valores de 0 o 255 para cada color).

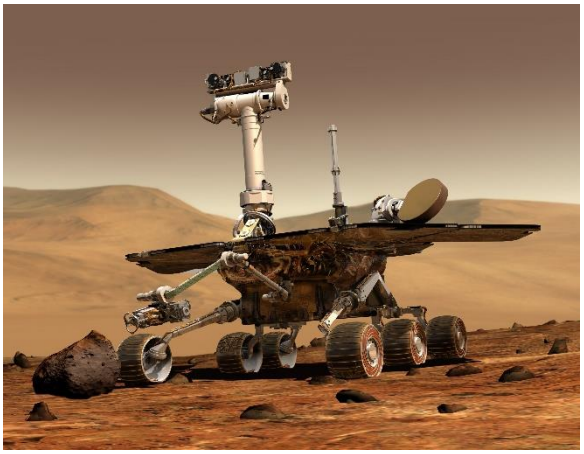
Se denomina contraste Legacy porque es la primera funcionalidad de contraste que introdujo Adobe en 2003 con la primera versión “Creative Suite (CS)” de Photoshop. En versiones posteriores, hasta el día de hoy, mantuvieron esta funcionalidad por si se quería utilizar para lograr algún efecto visual, añadiendo un menú que permite cambiar entre el contraste actual y el Legacy:



Para programar el contraste Legacy, comprobamos el valor del canal de color. Si es mayor que 127, aclaramos, sumando el valor de contraste al canal. Si es menor que 127, oscurecemos, restando el valor de contraste del canal.

Para programar el contraste nuevo, calculamos primero el valor de cada canal por separado, y hacemos la media para calcular el valor medio de la imagen. Una vez obtenido, lo usamos para calcular el cambio de contraste de la imagen ponderándolo con el valor medio calculado.

A continuación, se observa la imagen sin alterar, una imagen en la que se ha aumentado el contraste Legacy y otra en la que se ha aumentado el contraste nuevo, respectivamente:



Operaciones sobre imagen

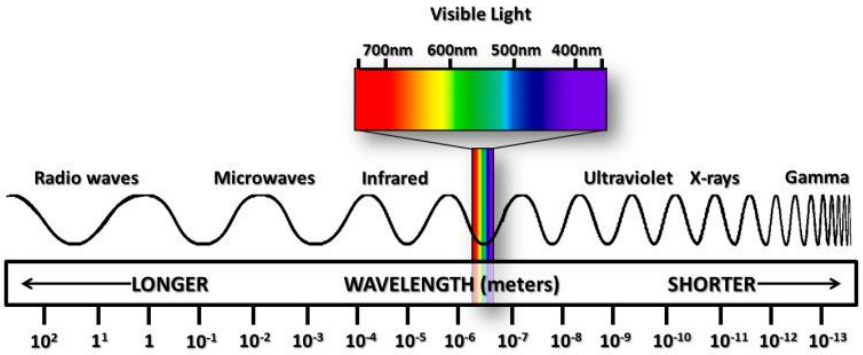
En estas operaciones, los kernel reciben como entrada la imagen entera, sin descomponer en canales, se trabaja sobre su totalidad, y se devuelve la imagen entera con los cambios realizados. A esta categoría pertenecen las funcionalidades de: saturación, tinte, flip vertical y ruido salt&pepper.

Saturación

Saturación y tinte necesitan un concepto llamado: nivel equivalente de escala de grises. Cuando convertimos una imagen a color en una imagen a blanco y negro, se puede hacer de dos maneras: realizando la media de los 3 canales de color, o realizando una media ponderada (método de la luminosidad).

-Método de la media: Se divide la imagen en sus 3 canales de color, se suman los valores de los 3 y se divide, es decir, se hace la media exacta de los 3 colores.

-Método de la media ponderada: Se tiene en cuenta que los 3 colores tienen diferente longitud de onda y diferentes efectos visuales para la vista humana. Por ejemplo, el rojo tiene una longitud de onda mucho mayor que el verde y el azul, que ganan en frecuencia, y el verde tiene un efecto de suavidad y calma más acentuado para la vista humana que los otros dos.



Es por esto que en lugar de cada color aportar el mismo peso, se distribuyen mediante la siguiente fórmula:

$$E'_Y = 0.299\,E'_R + 0.587\,E'_G + 0.114\,E'_B$$

Siendo ER el color rojo, EG el verde y EB el azul, según la
"ITU-R BT.601-7, Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios".

En este trabajo de investigación, se ha optado por utilizar el método de la media ponderada para convertir la escala de grises, y, para programar el ajuste de saturación y tinte de la imagen original.

***[Nota**: En el código redondeamos a 0.3*R + 0.59*G + 0.11*B**]**

Aquí se muestra una imagen convertida a escala de grises utilizando el método de la media, y el de la media ponderada, respectivamente:

Para conseguir la funcionalidad de la saturación, nos apoyamos en el método de la media ponderada. En primer lugar, definimos los componentes de color en términos de RGB de la siguiente manera:

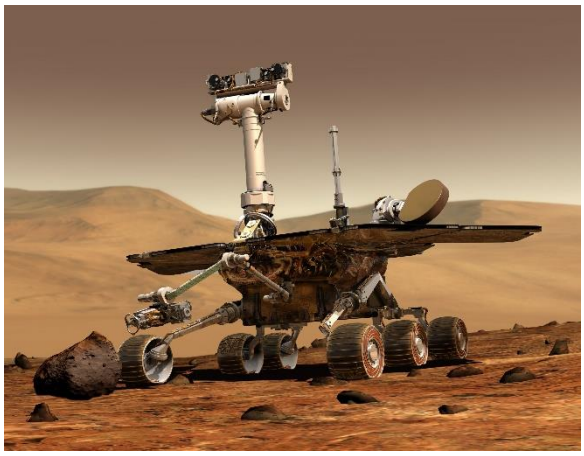
$$R - Y = 0.70R - 0.59G - 0.11B$$
$$B - Y = -0.30R - 0.59G + 0.89B$$
$$G - Y = -0.30R + 0.41G - 0.11B,$$

donde la Y se define como: $Y = 0.30R + 0.59G + 0.11B$

Pasamos al kernel una variable saturación, que representa el porcentaje de cantidad de color a ser añadido a la imagen en blanco y negro. De manera que, si la variable es 0, la imagen será en escala de grises, mientras que si mayor, “menos en blanco y negro” será la imagen, es decir, con los colores más vivos.

El programa primero calcula los componentes R-Y, G-Y, B-Y antes definidos. Luego, escala estos valores utilizando la variable saturación, y, por último, los añade de nuevo al componente de iluminación para generar los nuevos valores RGB.

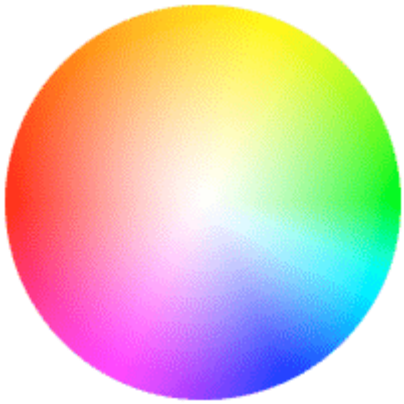
El programa permite alterar la saturación de cada uno de los 3 canales de color por separado, pudiendo mantener la saturación del rojo y verde, pero aumentando la de los azules, por ejemplo.
A continuación, se muestran imágenes con la saturación a 0% y a 200% respectivamente:



Tinte

La modificación del tinte de la imagen es muy similar a la de la saturación explicada anteriormente. Cambiar el tinte de una imagen consiste en visualizar la misma a través de un color determinado, es decir, tintar la imagen con rojo hará que todo en la misma se vea con tonos rojizos.

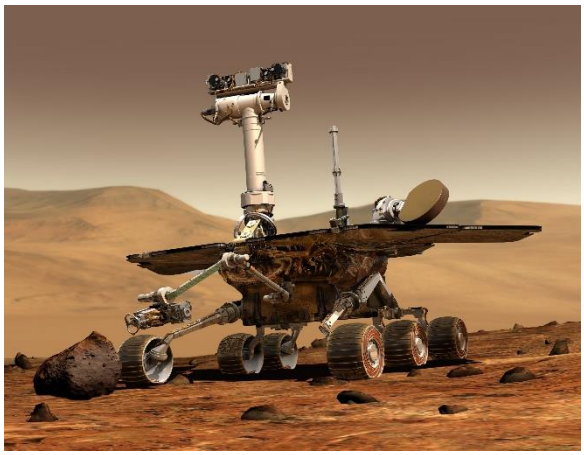
Para cambiar el tinte de la imagen utilizaremos una variable que se moverá en términos de grados (entre -180º y 180º), ya que los colores con los que tintar la imagen se pueden definir como un círculo:



Utilizando los mismos componentes R-Y, G-Y, B-Y, Y definidos en el apartado anterior, pasamos al programa la variable tinte. Los ángulos positivos acercan el tono de la imagen al verde, mientras que los negativos la acercan al rojo.

En primer lugar, el programa calcula los componentes R-Y, & B-Y únicamente, y luego, rota ambos vectores según el ángulo de la variable tinte. Finalmente, el programa recalcula los nuevos valores RGB.

A continuación, se muestran imágenes tintadas de rojo (-30º) y verde (90º) respectivamente:



Flip

El flip (o volteo) de una imagen es una operación diferente a la rotación, aunque parecida: la imagen se refleja sobre sí misma como un espejo.

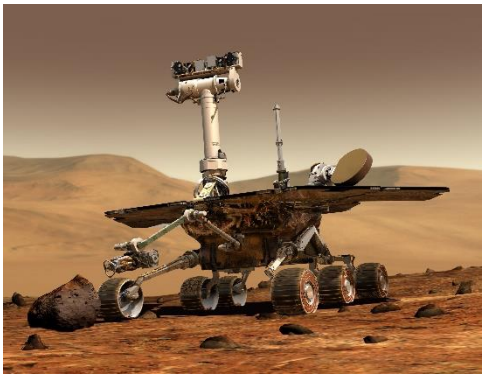
Flip Horizontal (Operación de canal):

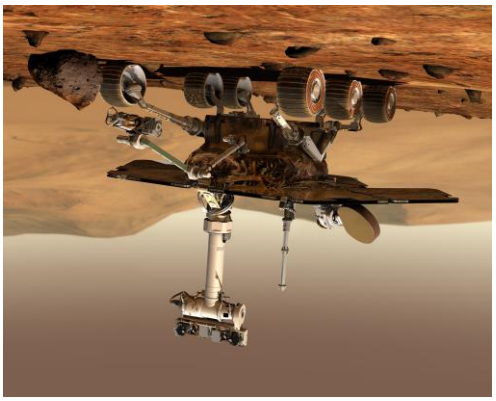
La forma más sencilla para conseguir el flip horizontal es jugar con el id de los threads que se ejecutan en el kernel. Si en lugar de utilizar
`"thread_2D_pos.y * numCols + thread_2D_pos.x"`
para pintar el píxel que se corresponde con dicho thread, utilizamos
`"thread_2D_pos.y * numCols - thread_2D_pos.x"`
obtenemos como resultado todos los píxeles volteados horizontalmente.

Flip Vertical (Operación de imagen):

Obtener el flip vertical resulta algo más complicado. Necesitamos crear un array copia de la imagen de entrada, que represente una imagen auxiliar con el mismo número de píxeles. En ella, en lugar de la imagen original, la copiaremos con las columnas cambiadas de sitio, de manera que en donde estaba la primera columna copiamos la última, en la segunda la penúltima, y así sucesivamente, manteniendo las filas igual. Una vez conseguido, copiamos la imagen auxiliar a la imagen de salida con la ayuda de un kernel, consiguiendo así el flip vertical.

A continuación, se muestran las imágenes en las que se ha aplicado flip horizontal y vertical, respectivamente:





Salt & Pepper Noise

El filtro de ruido “salt & pepper” emula el efecto de las fotografías realizadas con algunas cámaras antiguas, las cuales contenían píxeles erróneos. El ruido recibe su nombre debido al hecho de que los píxeles erróneos son blancos o negros (como la sal y la pimienta).

Esta tarea contiene un componente estocástico, ya que la generación de los píxeles erróneos en las cámaras antiguas es impredecible y aleatoria. Necesitaremos generar números aleatorios en los kernel, y la mejor manera para conseguirlo es con la ayuda de la librería NVIDIA cuRAND. Esta librería resulta apropiada para generar grandes cantidades de números aleatorios (uno por píxel), con precisión simple y doble, distribuciones uniforme y normal, y generar números aleatorios dentro de los kernel (paralelizando y acelerando así la tarea).

El procedimiento para generar el ruido salt & pepper es el siguiente: En primer lugar, hay que definir una probabilidad P (entre 0 y 1) de que haya ruido salt & pepper. En segundo lugar, generamos un número aleatorio en cada píxel, y lo comparamos con la probabilidad P definida anteriormente. Si el número aleatorio es mayor o igual que la probabilidad P, pintamos ese píxel de blanco o negro (salt & pepper) a igual proporción. Si es menor que P, dejamos el píxel de la imagen de entrada tal y como está. De esta manera, habrá píxeles blancos y negros de manera aleatoria con probabilidad P, provocando efecto de ruido y de fotografía antigua.

A nivel de código, utilizamos un kernel para generar el número aleatorio en cada píxel llamado “generate”, y, una vez generados, llamamos al kernel “saltpepper” que es el que comprara dicho número aleatorio con la probabilidad P y pinta los píxeles en consecuencia.

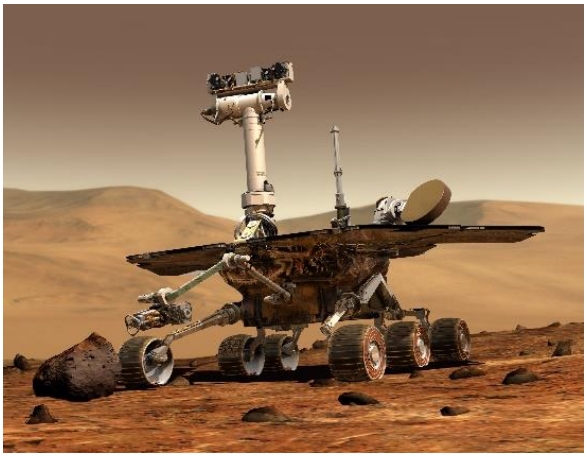
El funcionamiento del kernel “generate” es el siguiente: Primero se le pasa una semilla y crea un estado de cuRAND (curandState). Mediante el método “curand_init”, inicializa el estado a través de la semilla. Acto seguido, mediante el método “curand_uniform”, se consigue un número aleatorio entre 0 y 1 a partir del estado de cuRAND inicializado con “curand_init”.

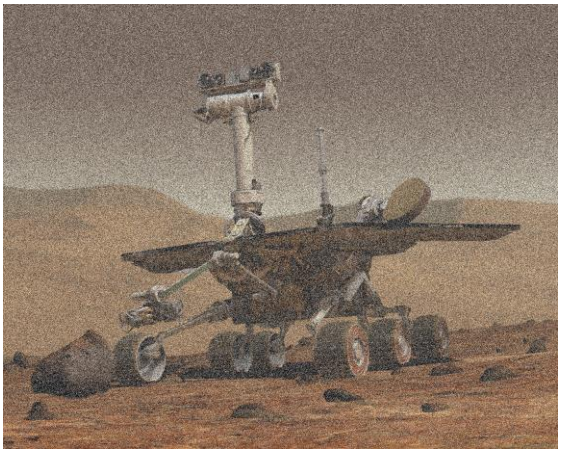
Este kernel tiene una particularidad. Si se llama con un número de threads igual al de todos los demás kernels: un thread por píxel de la imagen, el tiempo de ejecución se hace demasiado alto, al ser "curand_init" una función que consume mucho tiempo de ejecución. Para que funcione, es necesario acceder a las opciones de administrador de Nsight, y desactivar el TDR (timeout detection & recovery). Es una funcionalidad que detecta cuando un programa sobrepasa un tiempo de ejecución "recomendado" (y configurable) de, por defecto, 2 segundos, y, cuando eso ocurre, resetea la tarjeta gráfica e impide la consecución de la ejecución. En este caso, el programa consigue aplicar el ruido saltpepper en un tiempo de 13 minutos, lo cual no parece muy óptimo en términos de procesamiento de imágenes y paralelismo.

En lugar de eso, y sin ser necesario desactivar el TDR, llamamos al kernel sólo con un número de threads igual al número de filas de la imagen. Inicializamos con "curand_init" el curandState de cada fila, reduciendo muchísimo el número de veces que se llama a esta función (en lugar de una vez por píxel, se realiza una vez por fila). Y, una vez inicializado el curandState para cada fila, se generan números aleatorios con "curand_uniform" para cada una de las columnas en cada fila (para todos los píxeles). Al ser "curand_uniform" una función mucho más rápida, este método agiliza mucho los cálculos, permitiendo aplicar el ruido saltpepper en un tiempo medio de 4 segundos.

Para salt, se define un int=255 (color blanco), y para pepper, se define un int=0 (color negro). Convertimos en char estos valores a la hora de pintar los píxeles de la imagen que cumplan la condición de la probabilidad P, y así conseguimos nuestro filtro de ruido saltpepper.

A continuación, se muestra un ejemplo de imagen con ruido saltpepper:





Análisis de las mejoras y tiempos

A continuación, exponemos el resultado de la ejecución del programa probando las diferentes funcionalidades, midiendo el tiempo en milisegundos. Las pruebas se han realizado con la imagen de entrada “Mars_Rover.jpg”, usando un tamaño de bloque de 16x16 threads, en modo Release, y en un ordenador con las siguientes características:

-CPU: Intel Core i7-6500U

-GPU: NVIDIA GeForce 940M

-Mars_Rover.jpg: 3000 x 2400 píxeles

Test	Funcionalidad		Tiempo
1	Inversión de color		12.40ms
2	Brillo		12.40ms
3	Contraste Legacy		12.40ms
4	Contraste nuevo		23.50ms
5	Saturación		13.30ms
6	Tinte		14.10ms
7	Flip Horizontal		12.20ms
8	Flip Vertical		54.00ms
9	Ruido Salt & Pepper	1 Thread por pixel	780000.00ms (TDR desactivado)
		1 Thread por fila	4220.00ms
10	Box Filter		39.60ms

Comprobamos que en general, las operaciones por canal, en la que es necesario dividir en los 3 canales RGB, tienen un tiempo de ejecución mayor que las operaciones de imagen, excepto en el ruido saltpepper, que, por usar funciones de cuRAND en sus kernels, los tiempos destacan sobre los demás. El resto de funcionalidades tienen un tiempo de ejecución directamente proporcional a la complejidad de las operaciones que en ellos se realizan, como cabría esperar.

Se recomienda no desactivar nunca el TDR, para detectar posibles fallos en las aplicaciones gráficas. En el caso de obtener timeouts de los kernel, se recomienda aumentar el límite de TDR a no más de 10 segundos (por defecto, 2). Si se pasa de 10 segundos, seguramente haya una manera de optimizar el trabajo, reordenar el código, o paralelizar más partes del mismo.

Comentarios personales

Realizar este trabajo de investigación me ha resultado muy didáctico. Como en todos los ámbitos, cuando uno se encuentra solo ante el peligro, es cuando se pone las pilas para aprender y mejorar. Es cierto que he echado de menos más ejemplos de trabajos de investigación, alguna guía de cómo enfocarlo, de qué nivel de requerimiento se exige, etc. En general he estado un poco perdido en ese sentido, y no sabía si lo que hacía era demasiado o se iba a quedar corto. Pero gracias a mis conocimientos avanzados en Photoshop y a la práctica 4, me ha motivado el tema de procesamiento de imágenes y no me ha costado mucho decidirme por el tema.

Por otro lado, está bien que la temática sea libre, ya que nos empuja a usar nuestra creatividad, y a pegarnos con los elementos de programación que nos vayan haciendo falta mientras avanzamos. En mi caso, he aprendido mucho acerca del uso de número de threads para pasar a los kernels, cómo dividirlos, y qué threads se utilizan en cada momento para cada cosa. Además, he aprendido mucho de la librería cuRAND, la cual es muy útil (y casi la única manera efectiva) para generar números aleatorios en CUDA para la tarea del ruido salt&pepper. Dicha tarea ha sido la principal en mi trabajo de investigación, la que más tiempo me ha consumido (2 o 3 días), la más compleja, y con la que más he investigado y aprendido. Incluso he participado activamente en foros de NVIDIA en los cuales me han ayudado moderadores de NVIDIA a resolver mis problemas.

Como comentario final, me gustaría decir que al principio planteé convertir esta aplicación de CUDA en un programa utilizable, con interfaz gráfica, sliders y botones, que permitiera utilizar todas las funcionalidades a la vez de manera cómoda (como un Photoshop en miniatura). Sería además muy positivo para añadir a mi portfolio. Pero consideraba entrar en terreno desconocido, que no tiene que ver con la asignatura, y sin tiempo para hacerlo correctamente. Aun así, considero profundizar en el tema de crear una aplicación gráfica para sacar partido útil a este trabajo de investigación en el futuro.

Antonio Marco Rodrigo Jiménez

Referencias

1. Diapositivas de la asignatura, Sección CUDA, Utilización de CUDA
2. Daniel Shiffman. Images and Pixels. Disponible en: https://processing.org/tutorials/pixels/
3. **Harley R. Myler & Arthur R. Weeks. The Pocket Handbook of Image Processing Algorithms in C:** http://adaptiveart.eecs.umich.edu/2011/wp-content/uploads/2011/09/The-pocket-handbook-of-image-processing-algorithms-in-C.pdf
4. Tolga Soyata. GPU Parallel Program Development Using CUDA
5. NVIDIA. CUDA Toolkit Documentation. cuRAND. Disponible en: https://docs.nvidia.com/cuda/curand/index.html
6. CUDA, random numbers inside kernels: http://aresio.blogspot.com/2011/05/cuda-random-numbers-inside-kernels.html
7. NVIDIA. CUDA Toolkit Documentation. Timeout Detection & Recovery (TDR). Disponible en: https://docs.nvidia.com/gameworks/content/developertools/desktop/timeout_detection_recovery.htm
8. Steve Patterson. Better Brightness and Contrast in Photoshop. Disponible en: https://www.photoshopessentials.com/photo-editing/brightness-contrast/
9. Digital Image Processing, Grayscale to RGB Conversion: https://www.tutorialspoint.com/dip/grayscale_to_rgb_conversion.htm
10. Recommendation ITU-R BT.601-7 (03/2011) Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios: https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.601-7-201103-I!!PDF-E.pdf
11. **NVIDIA -> CUDA ZONE -> Forums. Robert Crovella.** Disponible en: https://devtalk.nvidia.com/default/topic/1036414/gpu-accelerated-libraries/why-i-need-setup_kernel-for-curand-states-/2