Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Organización del computador II

Trabajo Práctico Final - Procesamiento de imágenes

C++, Assembler y CUDA

Integrante	LU	Correo electrónico
Juan Manuel Martinez Caamaño	276/09	jmmartinez@dc.uba.ar
Leandro Lera Romero	187/09	lleraromero@dc.uba.ar
Ariel Eduardo Camblor	95/09	arielcamblor@gmail.com

Índice

1. Introducción

Este Trabajo Práctico consiste en la programación de algoritmos de procesamiento de imágenes utilizando distintos lenguajes y tecnologías, con el fin de conseguir una buena performance de estos algoritmos. En primer lugar, los algoritmos fueron programados en C++, para luego abordar Assembler y CUDA. En Assembler, utilizamos los registros XMM de los procesadores Intel y sus juegos de instrucciones SSE, del tipo SIMD, los cuales aprovechamos para procesar varios pixeles en paralelo. Por último, utilizamos CUDA, que procesa en paralelo sobre los cores de las placas de video NVIDIA, lo cual fue aprovechado para procesar múltiples pixeles a la vez.

2. Reseña histórica

En esta sección pretendemos hacer una breve reseña historia del surgimiento de las tecnologías utilizadas en este trabajo.

2.1. C++

C++ es un lenguaje de programación diseñado a mediados de los años 1980 por Bjarne Stroustrup. La intención de su creación fue el extender al exitoso lenguaje de programación C con mecanismos que permitan la manipulación de objetos. Estas nuevas características mantienen siempre la esencia del lenguaje C: otorgan el control absoluto de la aplicación al programador, consiguiendo una velocidad muy superior a la ofrecida por otros lenguajes. Un hecho fundamental en la evolución de C++ es sin duda la incorporación de la biblioteca STL años más tarde, obra de Alexander Stepanov y Adrew Koening. Esta biblioteca de clases con contenedores y algoritmos genéricos proporciona a C++ una potencia única entre los lenguajes de alto nivel.

Debido al éxito del lenguaje, en 1990 se reúnen las organizaciones ANSI e ISO para definir un estándar que formalice el lenguaje. Este proceso culmina en 1998 con la aprobación del ANSI C++.

2.2. OpenCV

OpenCV es un proyecto open source cuyo objetivo es proveer funcionalidad para el procesamiento de imágenes. Nació en los laboratorios de investigación de Intel con el objetivo de unificar distintos desarrollos en el área de procesamiento digital. Esta biblioteca está escrita en C y C++ y corre bajo Linux, Windows y Mac OS X.

Fue diseñada para que tenga un poder de cómputo eficiente, cuenta con optimizaciones realizadas en C y hace uso de los procesadores multicore. Fundamentalmente fue pensada con un fuerte foco en aplicaciones de tiempo real.

2.3. SSE

SSE (Streaming SIMD Extensions) es una extensión al grupo de instrucciones MMX para procesadores Pentium III, introducida por Intel en febrero de 1999. En febrero de 2001, AMD agregó esta tecnología en su procesador Athlon XP. Estas instrucciones operan con paquetes de operandos en coma flotante de precisión simple (FP).

Con la tecnología SSE, los microprocesadores x86 fueron dotados de setenta nuevas instrucciones y de ocho registros nuevos: del xmm0 al xmm7. Estos registros tienen una extensión de 128 bits (es decir que pueden almacenar hasta 16 bytes de información cada uno). A diferencia de su antecesor, MMX, la utilización de SSE no implicaba la inhabilitación de la unidad de punto flotante (FPU en inglés) por lo que no era necesario habilitarla nuevamente, lo que significaba para MMX una significativa pérdida de velocidad.

2.4. CUDA

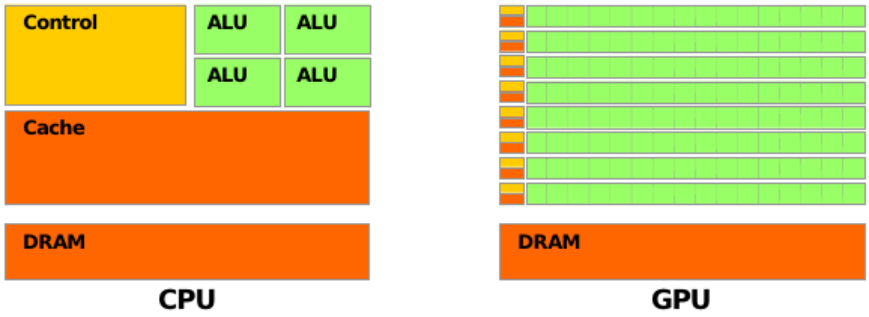
En noviembre del 2006, NVIDIA introduce CUDA, una arquitectura de procesamiento paralelo de propósito general, con un nuevo modelo de programación paralela y un nuevo set de instrucciones que utilizan la capacidad de procesamiento paralelo de los GPUs de las placas de video para resolver complejos cómputos de una manera más eficiente que con un CPU.

2.4.1. Descripción de la arquitectura

Una de las razones por las cuales las GPUs resultan óptimas para procesamiento de imágenes y grandes cálculos con matrices es debido a su arquitectura orientada a cómputo intensivo y paralelo. Están diseñadas con una mayor cantidad de transistores dedicados al cómputo, que al control de flujo (predicción de saltos y ejecución especulativa) y accesos a memoria.

Los cómputos que se ejecutan en las GPUs se caracterizan por realizar una gran cantidad de operaciones, sobre un

conjunto de datos grande, que se pueden procesar en paralelo de forma totalmente independiente.

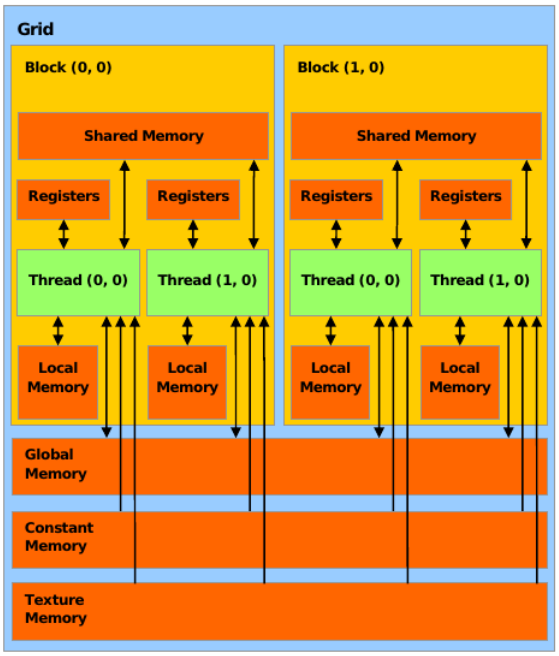


Representación comparando la arquitectura de un CPU con la de un GPU.

El cómputo realizado por un programa de CUDA, es ejecutado por un thread (hilo de ejecución), estos threads se agrupan en bloques de threads, y todos los bloques forman partes de un 'grid' (una malla) de bloques.
La memoria en CUDA, está organizada de forma jerárquica, de forma visible al programador. Se puede direccionar en los siguientes espacios de memoria:

- Conjunto de registros privado para cada thread.
- Memoria local privada para cada thread.
- Memoria compartida por un bloque de threads.
- Memoria global visible por todo el Grid.
- Memoria de texturas visible por todo el Grid.
- Memoria constante visible por todo el Grid.

La memoria constante, de texturas y global no solo son visibles por todo el Grid, sino que además el 'host' (el código que se ejecuta en la CPU y se encarga de llamar a las distintas funciones que se ejecutaran en la GPU) también puede escribir y leer en estos espacios de memoria. No es un dato menor la existencia de distintos tipos de memoria, debido a que es tarea del programador utilizarlas de forma adecuada para mejorar la performance del programa.
Un ejemplo de esto es que cada bloque de threads carga en memoria compartida los datos necesarios para que el bloque realice el cómputo, a fin de reducir los accesos a memoria global ya que esta es de acceso lento.



Esquema de la jerarquia de memoria.

Las placas de video están compuestas por varios *multiprocessors*, cada uno de los cuales sigue el modelo de ejecución *Single Instruction Multiple Data*.

Cada uno de estos *multiprocessors* posee un conjunto de registros por procesador, una cache para la memoria constante, una cache para la memoria de textura y una memoria compartida.

Un kernel se ejecuta de la siguiente manera, cada **thread block** del **grid** es asignado a un *multiprocessor*, estos se ejecutan de esta forma para poder utilizar la memoria local de cada *multiprocessor* para reducir los accesos a memoria global.
La cantidad de **thread blocks** que ejecuta un *multiprocessor* depende de la cantidad de registros disponibles en el *multiprocessor* y de la cantidad de memoria compartida que utiliza el **thread block**.

Cada **thread block** se divide en unidades llamadas **wraps**, estos son un conjunto de threads. Un scheduler va alternando entre distintos **wraps** a fin de maximizar la utilización del *multiprocessor*.

3. Desarrollo

Tal como fue presentado en la introducción nuestro objetivo es realizar los diferentes filtros de imágenes con las tres tecnologías mencionadas. A continuación detallaremos los diferentes algoritmos que utilizamos para implementar dichos efectos.

3.1. Corrección de brillo

Este filtro, en primer lugar, calcula el brillo actual de cada pixel ($\frac{r+g+b}{3}$) y guarda los valores de brillo máximo y mínimo. Luego se calculan dos valores a y b, cuyo cálculo dependerá del valor del brillo mínimo: si es cero, $b = 0$ y $a = \frac{255}{max}$; si no es cero, $b = \frac{255}{\frac{-max}{min}+1}$ y $a = \frac{-b}{min}$. Luego volverá a visitar cada pixel, y definirá un nuevo valor de brillo de la siguiente forma: $brilloNuevo = a * brilloViejo + b$. Acto seguido, reemplazará el valor actual de cada uno de los canales de color: si $brilloViejo = 0$, los tres valdrán cero; sino, el valor de cada canal se definirá de la siguiente manera: $colorNuevo = \frac{brilloNuevo * colorActual}{brilloViejo}$. Si el resultado es mayor a 255, al canal se le asignará como valor 255.

3.2. Eliminación de canales

Este filtro, como su nombre lo indica, nos permite eliminar un canal de la imagen. La eliminación consiste en darle valor cero a ese color en todos los pixeles de la imagen. Mediante una máscara de 3 digitos (uno por canal, en el orden rgb), que el usuario pasa por parámetro, el filtro decide si dejar el canal como está o eliminarlo.

3.3. Intercambiar canales

Filtro similar al anterior, pero este brinda la posibilidad de intercambiar los valores de color de cada pixel. La máscara que el usuario pasará por parámetro (también de 3 dígitos, en el orden bgr) indicará para cada uno de los canales el número de canal ($0 = r$, $1 = g$, $2 = b$) que desea colocarse en ese lugar. Con ello, el filtro visita cada pixel y coloca en cada canal el valor en ese pixel del canal elegido.

3.4. Gauss

El filtro de Gauss trabaja, para cada pixel, con sus vecinos dentro de un cuadrado que, en este caso, tiene una dimensión de 3x3 pixeles, pero bien podría ser otras dimensiones, como 5x5 o 7x7. Lo que hace es, para cada color, obtener el valor nuevo a partir de un promedio entre todos los valores del mismo color perteneciente a los pixeles del vecindario antes descripto. Estos valores pueden utilizarse tal cual están o multiplicados por distintos coeficientes, para variar el resultado final del filtro.

3.5. Interpolación

La idea de este filtro es hacer una ampliación 2x de la imagen de forma digital. El procedimiento utilizado para el desarrollo del mismo se puede describir mediante tres etapas.
La primera consiste en copiar los pixeles de la imagen original a una nueva imagen, del doble de tamaño, espaciados por una columna y una fila. Por ejemplo, el pixel en la posición $(0, 0)$ se copia en la $(0, 0)$, el de la $(0, 1)$ a $(0, 2)$ y el de la $(1, 0)$ en $(2, 0)$. De esta manera la imagen de salida nos queda con la imagen original y con espacios "vacios"para calcular.
La segunda etapa consiste en rellenar los pixeles cuyas cuatro esquinas, si consideramos que el pixel a calcular es el centro de una matriz de 3x3, son pixeles de la imagen original. La idea en este caso es realizar un promedio de los

valores de las cuatro esquinas y completar el pixel con este valor. Como resultado de este proceso, nos queda que la imagen de salida queda conformada como un "tablero de ajedrez", quedan alternados pixeles con la información de salida y pixeles sin completar.
Por último, realizamos la misma operación con los pixeles que aún no están completos pero esta vez considerando los pixeles de los laterales, el superior y el inferior.

3.6. Laplace

El filtro de Laplace consiste en a un pixel central, multiplicar cada uno de sus canales, y restarle los canales correspondientes de los pixels que lo rodean. En nuestra implementación, al pixel central lo multiplicamos por 8, y le restamos los 8 pixels inmediatos al central.
De esta forma, si todos los pixels de la región tomada, son similares al central, podemos asegurar que nos encontramos en una sección homogénea, y nos dará un resultado cercano al 0. En cambio, si alguno de estos pixels fuera distinto al central, la diferencia seria mayor a 0.
Entonces, las regiones de una imagen cercanas a un borde, donde habría una variación ya sea de brillo o de color, la resta de los canales del pixel central, por los de los vecinos, dará distinto de 0, dejando al pixel resultado 'brillante'.
Este filtro se lo puede utilizar para resaltar los bordes de las imágenes.

3.7. Pixelizar

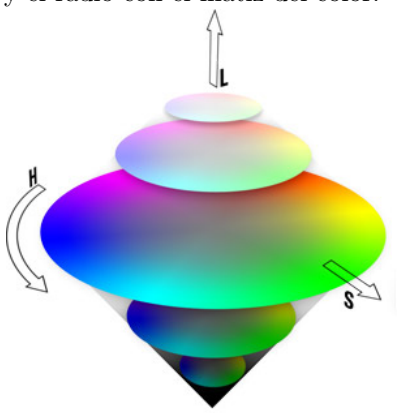
Básicamente una imagen se "pixela" cuando se agranda demasiado y no se cuenta con información para poder completar los nuevos pixeles que aparecen producto de la ampliación. Esto provoca que cada pixel sea copiado tantas veces como espacio a rellenar tanto a lo ancho como a lo alto. Es por esto que este filtro funciona bajo este concepto, pero de manera inversa. En vez de agrandar la imagen lo que hacemos es copiar cada una cierta cantidad de pixeles (en el trabajo ocho) tanto a lo ancho como a lo alto formando un cuadrado de 8x8 con el mismo pixel, esto crea la misma sensación que al estirar una imagen.

3.8. Range Operator

El filtro 'Range Operator' sirve para detectar bordes. Consiste en tomar los valores máximos y mínimos para cada color de cada vecindario de 3x3 pixeles (pueden tomarse también vecindarios de mayores dimensiones), y asignarle al pixel central como valor para cada color la diferencia entre esos valores calculados. De esta forma, quedarán expuestas las zonas en donde hay una pronunciada diferencia de valores, ya que serán las únicas en donde en la imagen resultante haya valores distintos o lejanos al cero, y por lo tanto resaltarán sobre las zonas uniformes, que quedarán en negro. Como estas diferencias pronunciadas suceden en los bordes, entonces logramos una imagen resultante en el cual quedarán resaltados.

3.9. Conversión a HSL

El modelo de color HSL, define al espacio de color como un cono doble. Los dos vertices se corresponden con el color blanco y negro. La distancia al blanco y al negro se corresponde con la luminocidad del color. El radio al eje del doble cono se corresponde con la saturación y el radio con el matiz del color.



Llamemos m al menor compontente del formato **rgb** y a M al mayor.
La luminosidad se calcula como $l = \frac{m+M}{2}$, esto es el promedio entre el mayor y menor componente del formato **rgb**.
Para calcular la saturación se utiliza la siguiente formula:

$$s = \begin{cases} si\ m = M, s = 0 \\ si\ l < 0{,}5, \frac{M-m}{M+m} \\ si\ l \geq 0{,}5, \frac{M-m}{2{,}0-(M+m)} \end{cases}$$

El valor del *hue* se calcula de la siguiente forma:

$$h = \begin{cases} si\ m = M, h = 0 \\ si\ M = r, \frac{\frac{g-b}{M-m}}{6} \\ si\ M = g, \frac{2+\frac{b-r}{M-m}}{6} \\ si\ M = b, \frac{4+\frac{r-g}{M-m}}{6} \end{cases}$$

En nuestro caso, utilizamos este formato para realizar una modificación a la saturación (simplemente sumando o restando un valor) y regresamos al formato **rgb**.
Este formato de representación del espacio de color es el elegido por numerosas aplicaciones dedicadas al diseño.

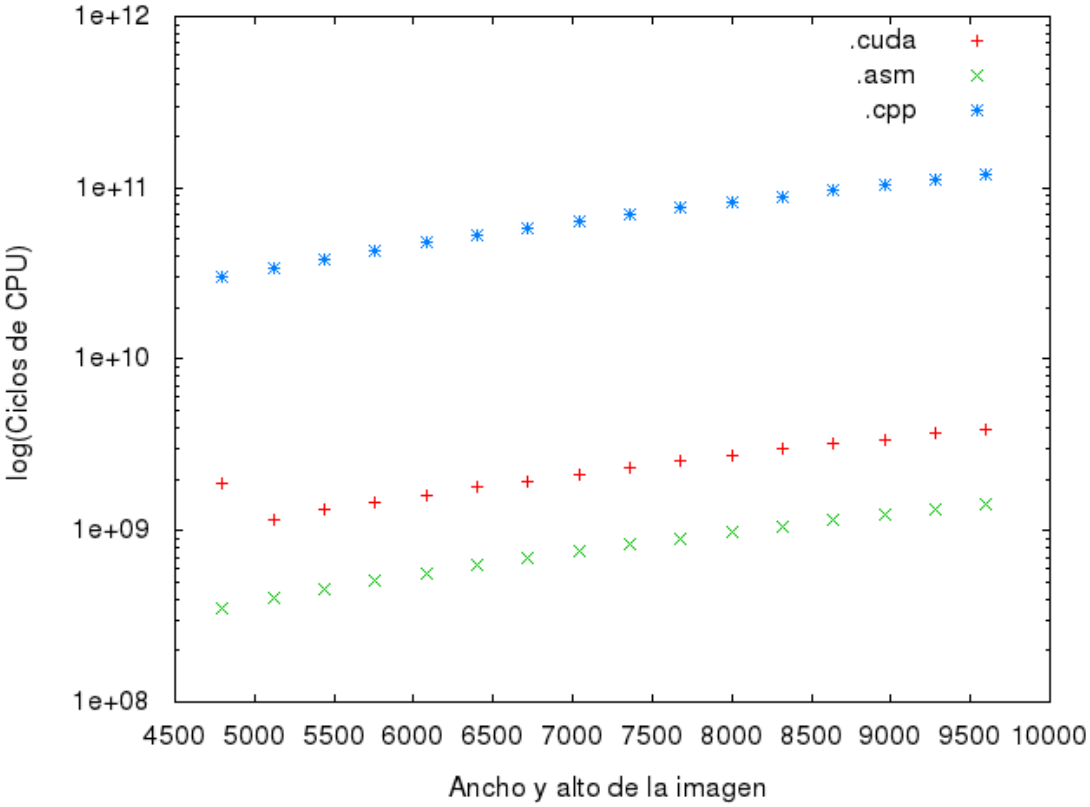
3.10. Sobel

El filtro sobel es otro de los filtros que implementamos para realizar una detección de bordes. Este filtro toma cada pixel de la imagen, con excepción de los de los bordes, y realiza dos cálculos diferentes con sus ocho pixeles vecinos. Dicho proceso está especialmente diseñado para rescatar las diferencias bruscas de color que se producen en pixeles contiguos. Una vez realizadas dichas operaciones, que vale aclarar que para realizarlas se utiliza la saturación para calcular el valor final, se realiza una última operación, se aplica la raíz cuadrada a la suma de los cuadrados de los valores resultantes de las primeras operaciones con los vecinos. Como resultado, se obtiene el valor del pixel que estábamos considerando.

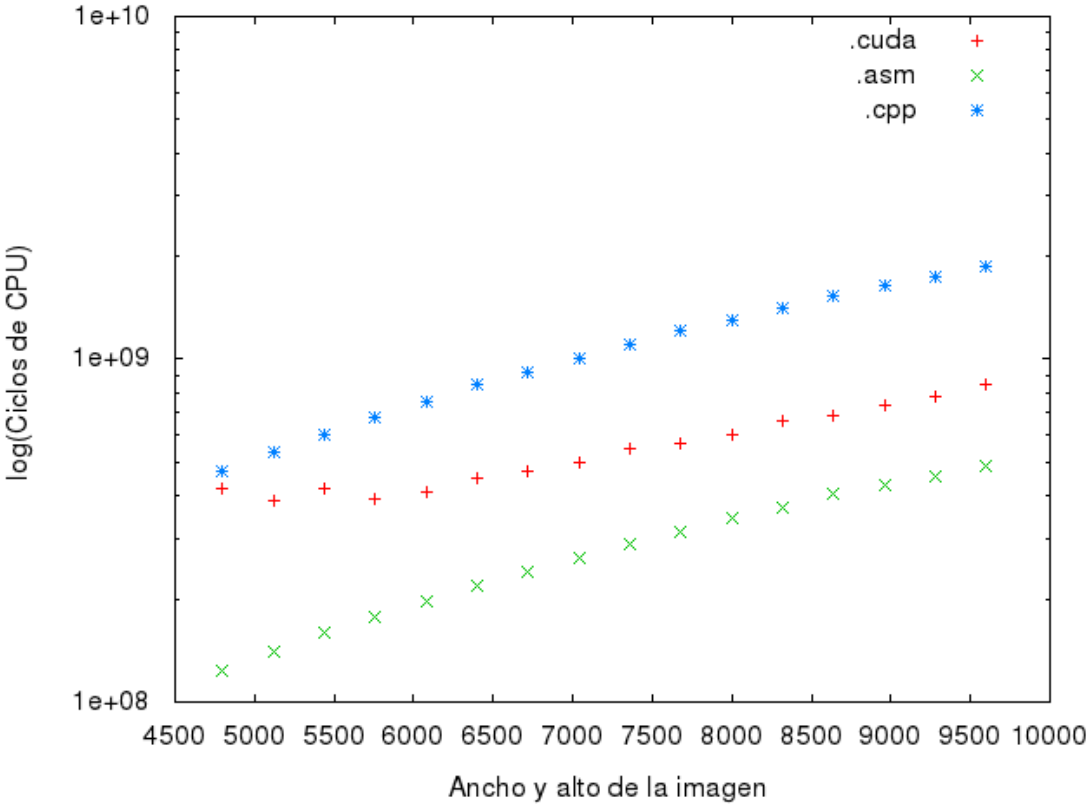
4. Resultados

A continuación presentamos los gráficos que muestran los resultados que obtuvimos al realizar algunas pruebas de performance de los diferentes filtros y sus respectivas implementaciones. La prueba consistió en la medición de tiempo de los algoritmos en C++, ASM y CUDA, utilizando imágenes cuadradas de tamaño creciente, desde 4600x4600 pixeles hasta 8000x8000 pixeles. Para esto utilizamos la siguiente configuracion de hardware: Core i5-2310 bajo Linux Ubuntu 11, con una placa de video GeForce 460 GTX. Para medir el tiempo de cada ejecución, se recurrió a la utilización del Time Stamp Counter(*rdtsc*), con el cual tomamos el instante en el que llamamos al filtro y el instante en el cual retorna, siendo la diferencia entre ambos momentos el tiempo de ejecución del algoritmo. Para facilitar la visualización de los datos, los gráficos se realizaron utilizando escala logarítmica.

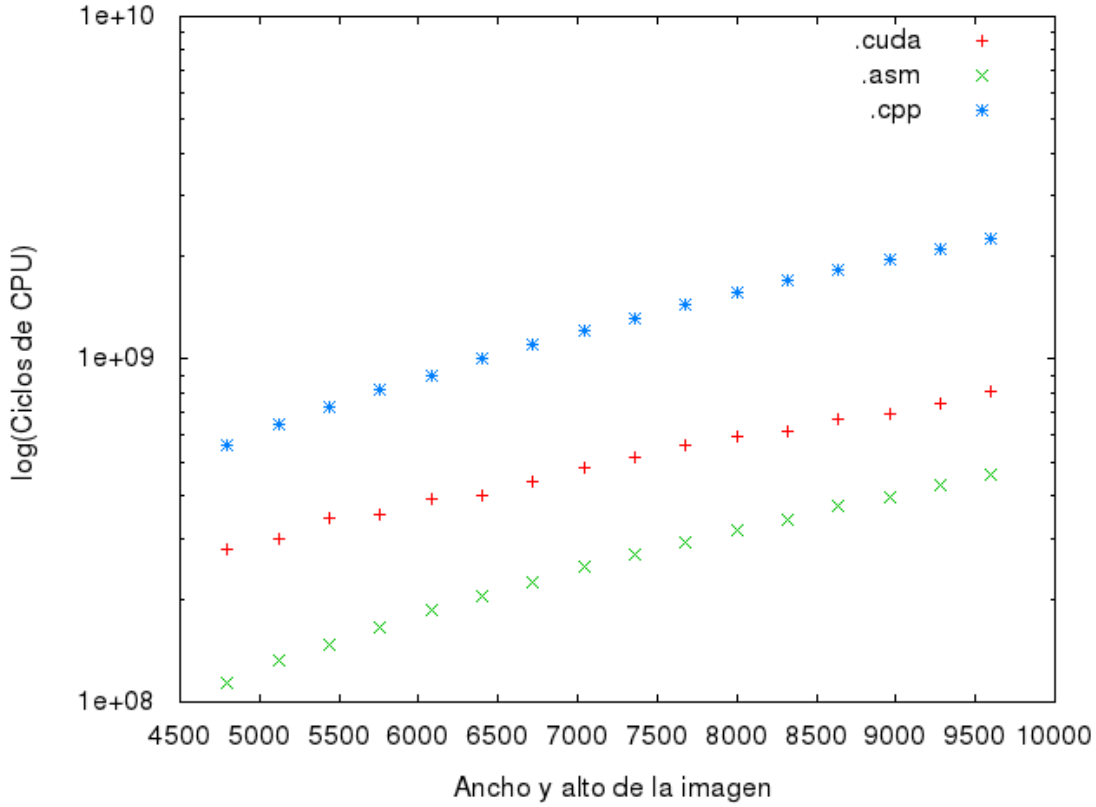
4.1. Corrección de brillo



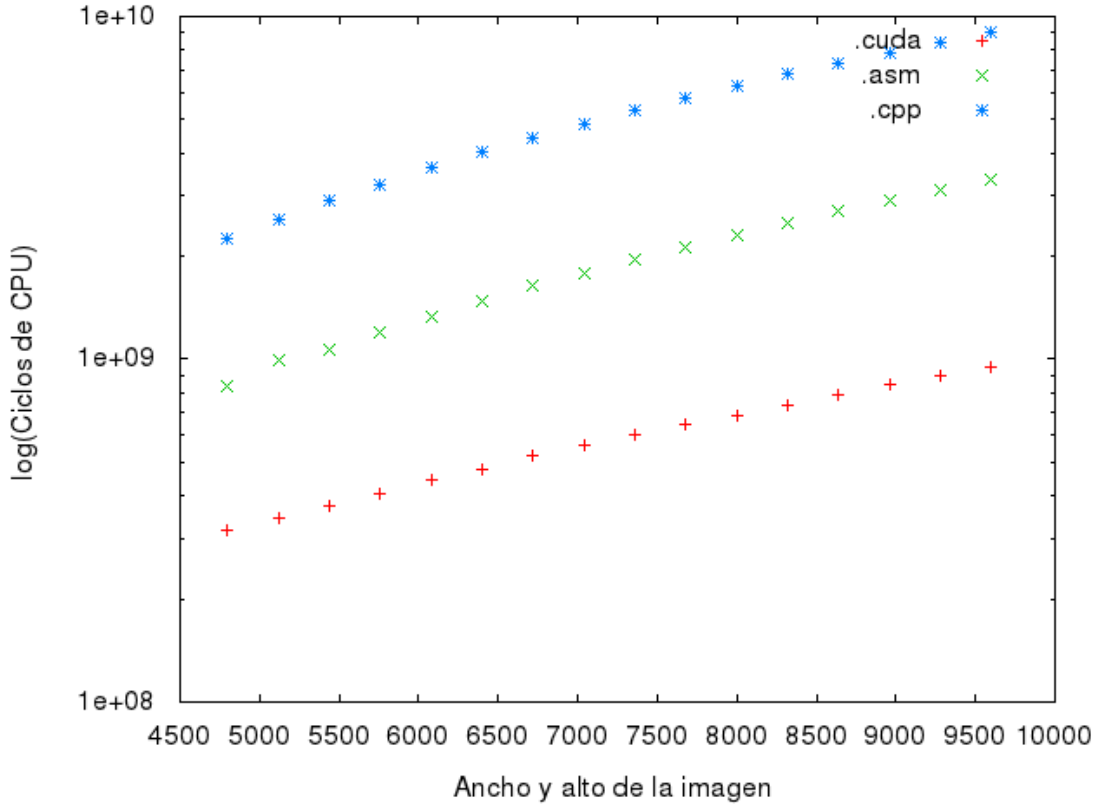
4.2. Eliminación de canales



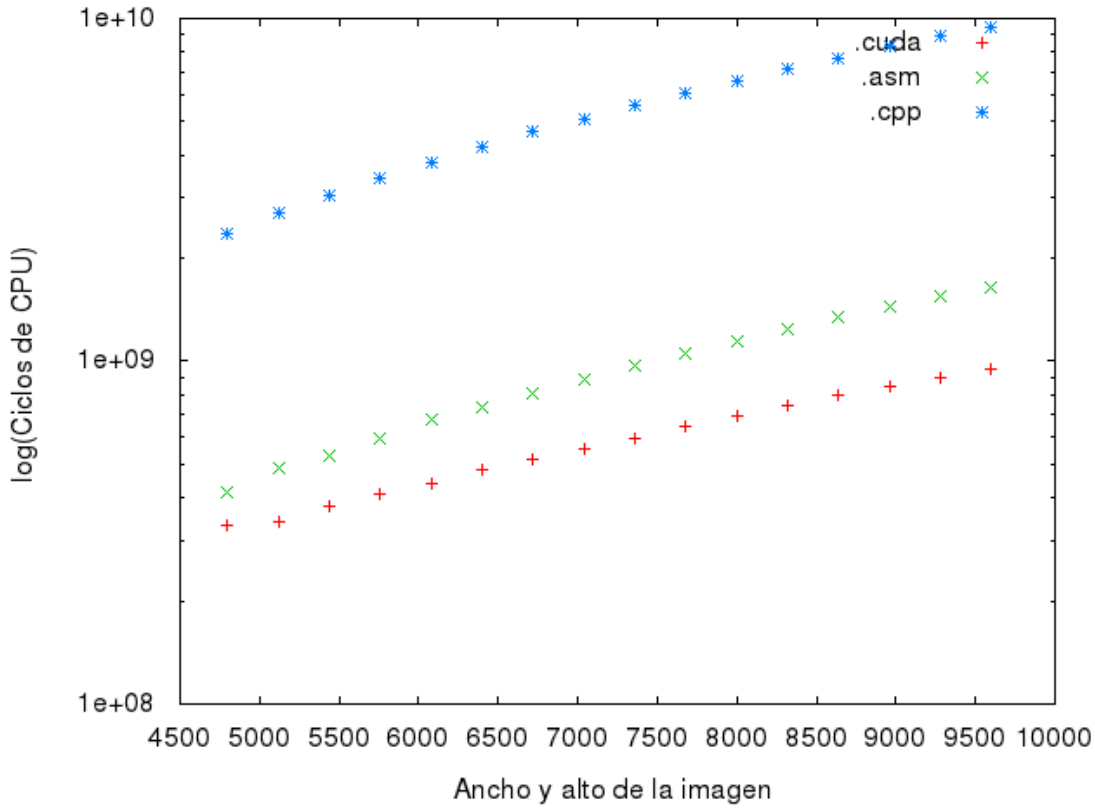
4.3. Intercambiar canales



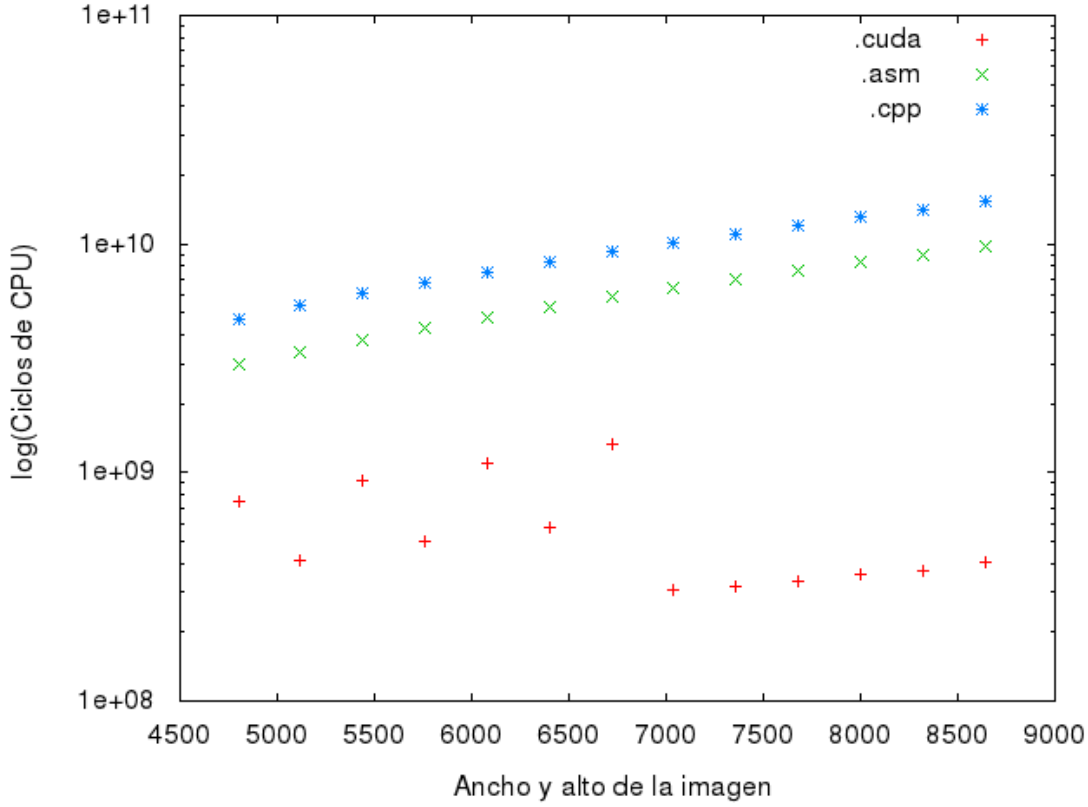
4.4. Gauss Escalada



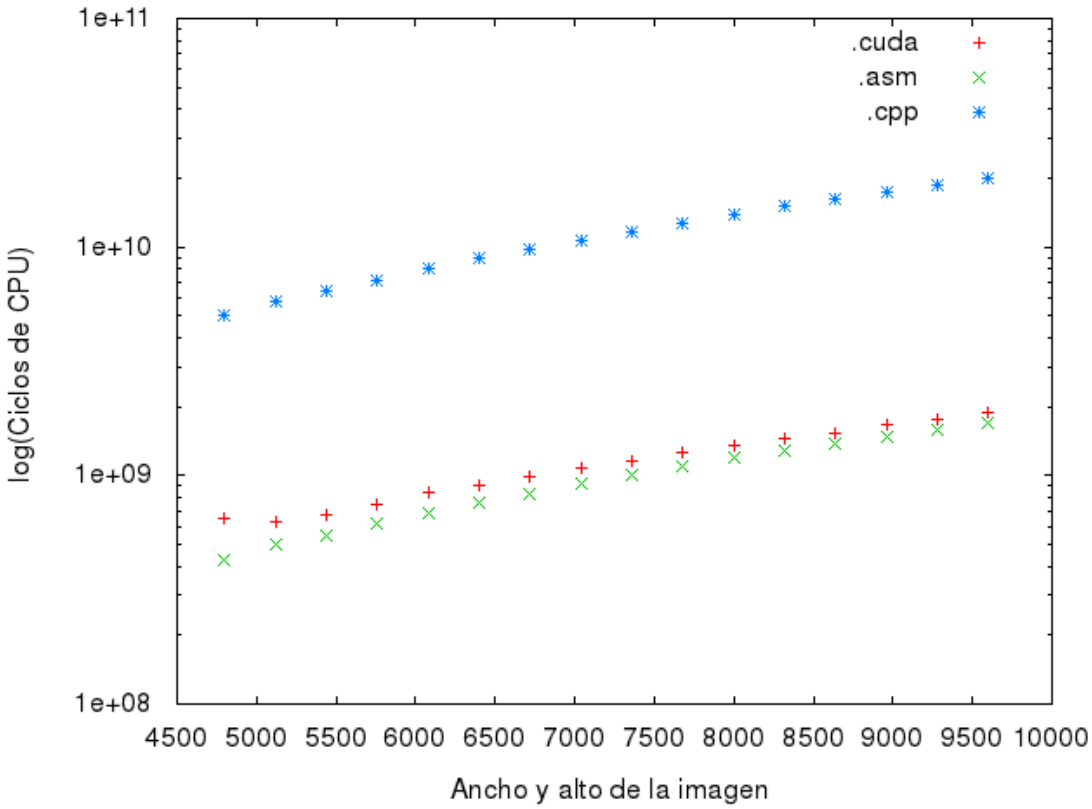
4.5. Gauss Simple



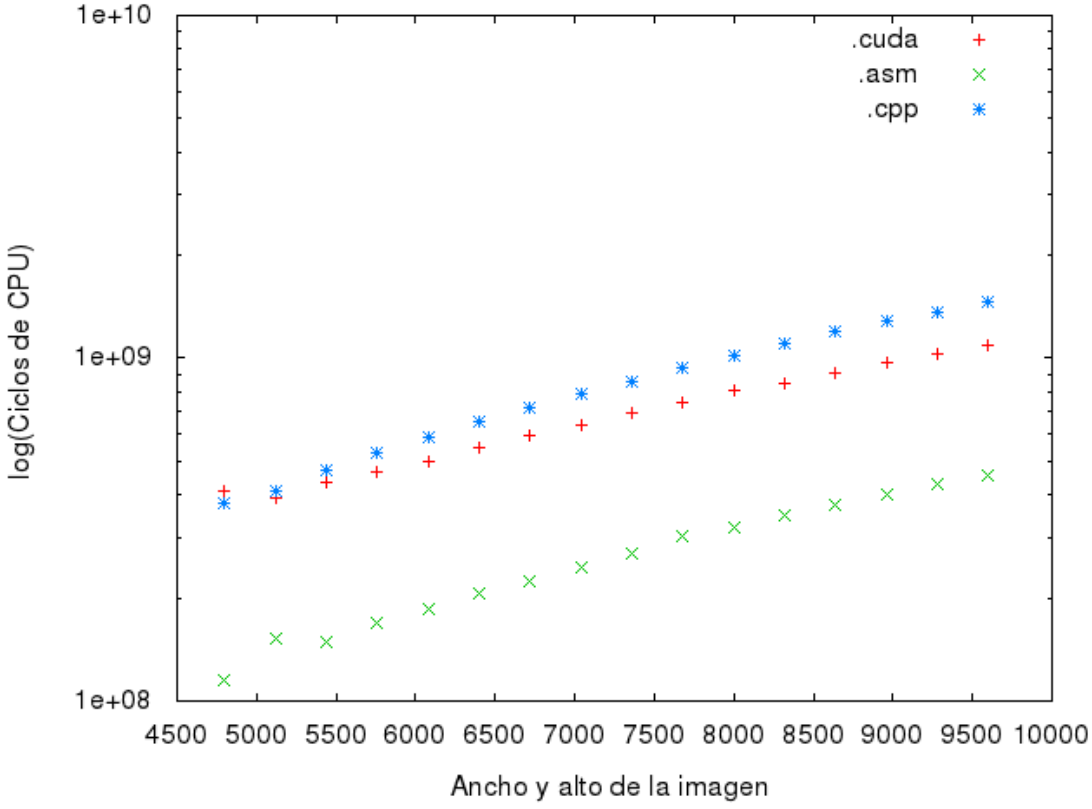
4.6. Interpolación



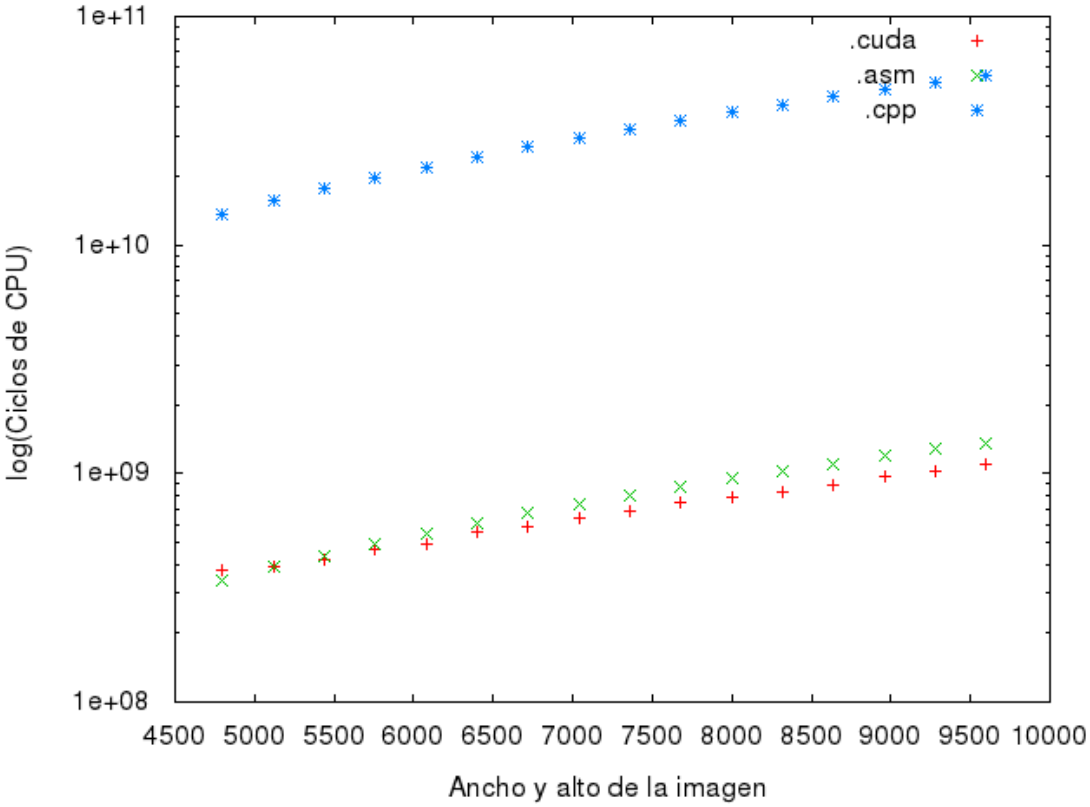
4.7. Laplace



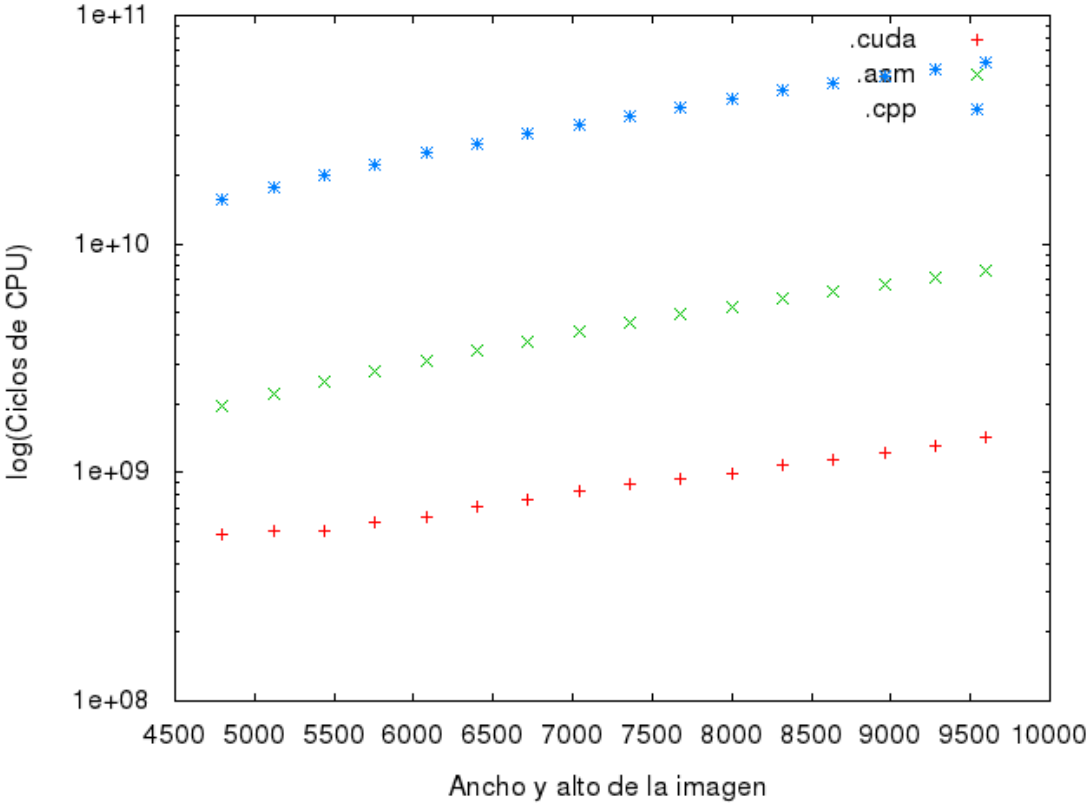
4.8. Pixelizar



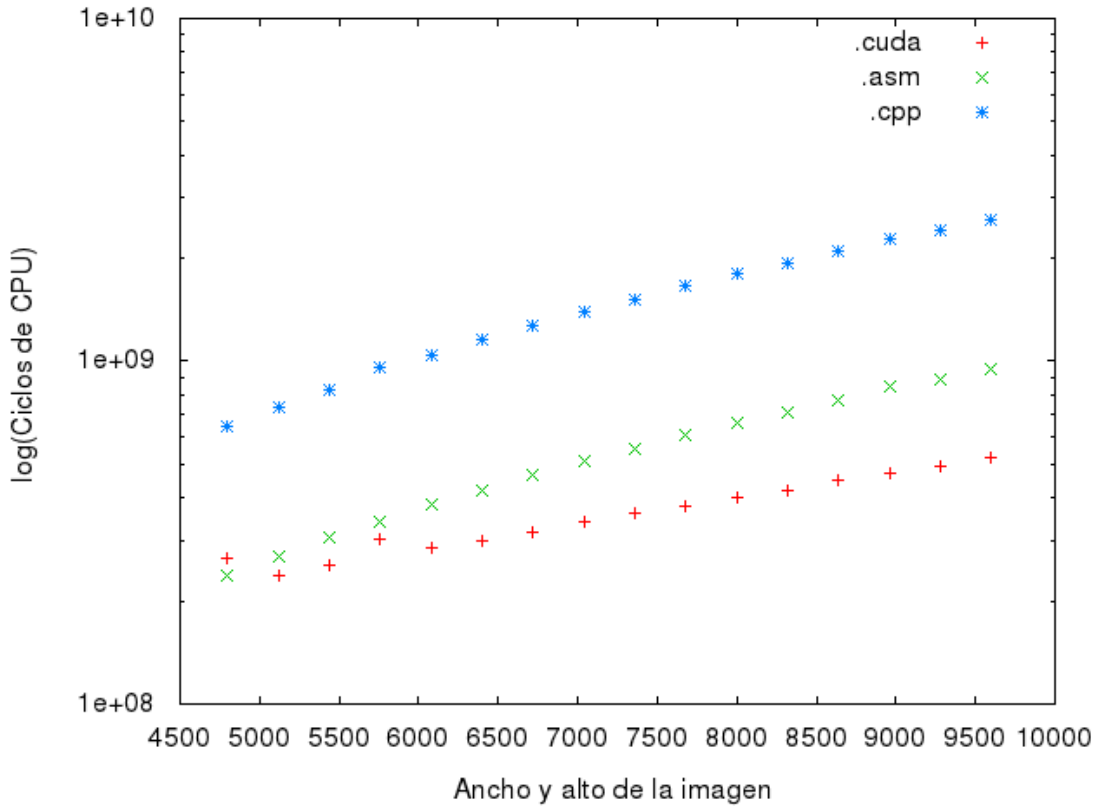
4.9. Range Operator



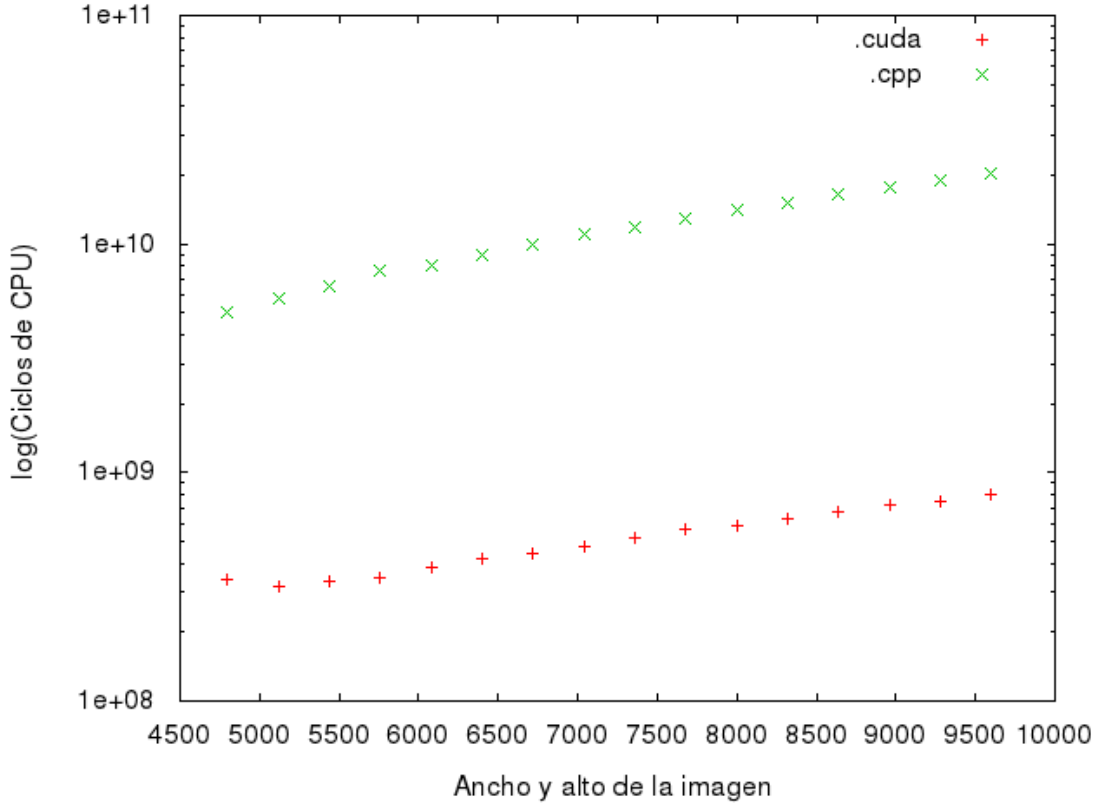
4.10. Sobel



4.11. GreyScale



4.12. RgbtoHsl



4.13. FPS conseguido en la reproducción de video

	Corrección de brillo	Eliminación de canales	Intercambiar canales	Gauss Escalada	Gauss Simple
C++	15	19	19	16	15
ASM	18	19	19	17	18
CUDA	18	19	20	20	20

	Interpolación	Laplace	Pixelizar	Range Operator	Sobel	GreyScale	Rgbtohsl
C++	12	13	20	6	6	18	11
ASM	13	19	19	18	15	19	-
CUDA	17	20	18	20	18	20	20

5. Conclusiones

Al observar los gráficos podemos ver que, para todos los filtros, el algoritmo hecho en Assembler con SSE tiene un mejor rendimiento que el de C++, disminuyendo considerablemente el tiempo de ejecución.

Por lo tanto, podemos decir que el aprovechamiento de la tecnología SSE que proveen los procesadores Intel nos permite obtener una eficiencia mucho mayor en el algoritmo, lo cual resulta muy importante, por ejemplo, al momento de procesar videos, en donde con esta mejora podemos tener un mayor número de frames por segundo y así conseguir que el video procesado pueda verse con fluidez.

Por otra parte, se puede observar que el rendimiento logrado, en muchos casos, utilizando CUDA es a su vez mejor que el obtenido con Assembler y con C++, de lo cual se desprenden las siguientes conclusiones. En primer lugar y el más evidente de los resultados, el tiempo que se tarda en procesar la imagen es claramente menor pero por otro lado al utilizar la tecnología provista por NVIDIA obtenemos una ventaja adicional que es la utilización del procesador principal de la computadora para otras tareas mientras la placa de video se encarga del procesamiento de la imagen, mejorando no solamente el tiempo de aplicación del filtro sino también el rendimiento general del equipo.

Además, una vez incorporado el paradigma de cómputo que ofrece CUDA, la interfaz de uso es mucho más amigable para el programador ya que esta biblioteca está basada en C++ lo cual facilita considerablemente tanto su programación como mantenimiento.

6. Referencias

[1] Wikipedia - http://es.wikipedia.org/wiki/C

[2] El mundo informático - http://jorgesaavedra.wordpress.com/2006/12/09/breve-historia-de-c-c-c/

[3] Learning OpenCV Computer Vision with the OpenCv Library - O'Reilly

[4] Wikipedia - http://es.wikipedia.org/wiki/SSE

[5] NVIDIA CUDA C Programming Guide