Organización del Computador II

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

Trabajo Práctico Final

Grupo *i*

Integrante	LU	Correo electrónico
Pablo Gauna	334/09	gaunapablo@gmail.com
Agustín Santiago Gutiérrez	513/08	elsantodel90@gmail.com

Índice

1. Introducción

Este trabajo practico se propone hacer una comparación de performance entre distintas lenguajes de programación, Implementando códigos con funcionalidades idénticas en distintos lenguajes que tienen sus fortalezas y debilidades.

Son los siguientes 4 lenguajes los que van a ser tenidos en cuenta para la comparación:

- C++ (fácil de usar)
- Assembler (difícil y rápido)
- Assembler + SSE (mas difícil y más rápido)
- C++ + CUDA (pasa el cálculo a la GPU y paraleliza el procesamiento)

Con el fin de ver la performances de cada uno de estos se implementaran 14 filtros distintos en cada uno de estos lenguajes y se aplicaran en tiempo real a un video *.avi, *.wmv, *.mp4 o a la captura de la webcam.

Los filtros antes mencionado son los siguientes:

- (1) pasar a escala de grises
- (2) invertir colores
- (3) Thresholding (filtro binario)
- (4) Zoom usando nearest neighbour
- (5) suavizado la imagen con un filtro gaussiano
- (6) Detección de bordes rápidamente
- (7) Detección de bordes mediante el algoritmo de Sobel
- (8) Detección de bordes mediante el algoritmo de Canny
- (9) Todos los anteriores juntos
- (Q) Zoom usando interpolación bilineal
- (W) Realce de contraste
- (E) Ajuste de color
- (R) Bokeh (Efecto desenfoque)
- (T) Rotación usando nearest neighbour

Para hacer la comparación se aplica al mismo video estos filtros y se chequea el rendimiento de las distintas implementaciones.

2. Instrucciones de Compilación y Uso

2.1. Codigo

El código y el informe pueden ser obtenidos mediante svn:

https://tpsh.dc.uba.ar/svn/gaunap-tpfinalorga2/

2.2. Compilación

El programa está desarrollado en linux.
Para la compilación es necesaria una máquina con OpenCV y CUDA instalados correctamente. En dicho caso, basta simplemente con ejecutar el comando make desde el directorio `src/linux-src`.
De no tener el OpenCV 2.2 instalado no se podrá correr el ejecutable.

2.3. Uso

Es un programa que puede reproducir videos *.avi o la captura de la Webcam. De no tener parámetros el programa intentara reproducir un video con el nombre "$default.avi$" en la carpeta donde se encuentre el binario.

A este programa se le puede pasar algunos parámetros, entre ellos el -h que muestra una ayuda de los parámetros que se le puede pasar al programa. Mostrando una ayuda similar a la siguiente:

Ayuda:

- -f seguido del nombre, sin espacios, de un archivo *.avi que se quiera reproducir
- en caso de no especificar nombre con -f, intentara abrir el archivo default.avi
- -c para abrir la cámara y usarla en vez de un video
- -h para mostrar ayuda
- -hc para mostrar los comandos que se pueden usar mientras se ejecuta el programa
- -rgb para elegir los valores a utilizar en el filtro de ajuste de color

Dentro del programa se pueden aplicar los filtros y ver los resultados en tiempo real. Para ver todos los comandos que se pueden efectuar se puede pasar como parámetro -hc al programa y mostrara una lista de los comandos similar a la siguiente:

Comandos:

- 1 al 9 y Q a T, para encender o apagar filtros:
  (1) pasar a escala de grises
  (2) invertir colores
  (3) Thresholding (filtro binario)
  (4) zoom
  (5) suavizado la imagen con un filtro gaussiano
  (6) Detección de bordes rápidamente
  (7) Detección de bordes mediante el algoritmo de Sobel
  (8) Detección de bordes mediante el algoritmo de Canny
  (9) Todos los filtros anteriores juntos

(Q) zoom (billinear interpolation)
(W) realce de contraste
(E) color balance
(R) bokeh
(T) rotacion (nearest neighbour)
(O) Disminuye el bokeh
(P) Aumenta el bokeh

- Cambiar el tipo de código:

(N) Ejecución normal (CPU, código en C++)
(A) Ejecución con Assembler (CPU, código hecho en Assembler)
(S) Ejecución con SSE (CPU, código Optimizado con SSE)
(C) Ejecución con CUDA (GPU, código en C++/CUDA)

- Aumentar o disminuir la velocidad del video:

(+) Acelera la aparición de cada frame del video 1ms
(−) Retrasa la aparición de cada frame del video 1ms

- Otros:

(Esc) para salir de la reproducción
(Enter) guarda el frame original y el frame que luego que es pasado por los filtro en un *.jpg

3. Desarrollo

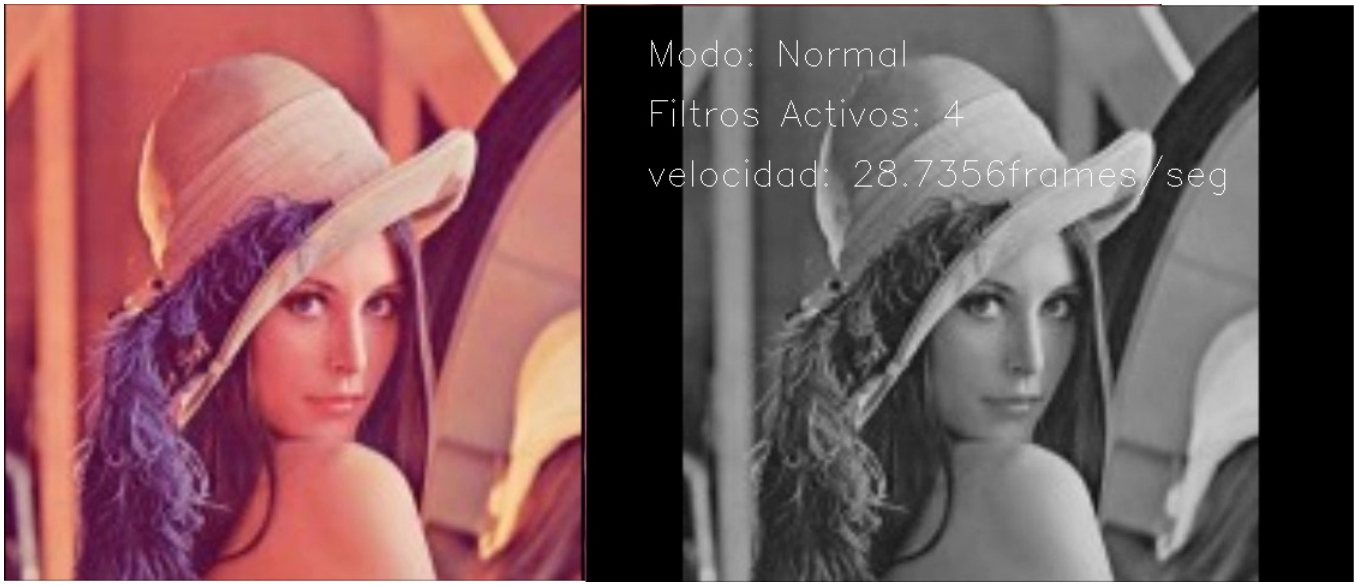
3.1. General

3.1.1. Filtro (1) Pasar la imagen a escala de grises

Para pasar la imagen a escala de grises lo que hago en cada pixel p es lo siguiente:

$$R'(p) = G'(p) = B'(p) = R(p) * 0{,}30 + G(p) * 0{,}59 + B(p) * 0{,}11$$

Los números 0.3, 0.59 y 0.11 son los recomendados para la transformaciones, no encontré nada más fiable que wikipedia[1] para sustentar esto, pero en práctica parece funcionar muy bien.



3.1.2. Filtro (2) Invertir de colores

Es un filtro muy simple que invierte los colores de cada pixel de la imagen.

Para invertir un pixel se realiza la siguiente cuenta[2]:

$$\begin{aligned} \text{Inv(R)} &= (255 - \text{R}) \\ \text{Inv(G)} &= (255 - \text{G}) \\ \text{Inv(B)} &= (255 - \text{B}) \end{aligned}$$

[1] http://en.wikipedia.org/wiki/Grayscale
[2] http://es.wikipedia.org/wiki/Inversion_(imagen)

3.1.3. Filtro (3) Thresholding

Transformar la imagen en otra que solo utilice dos colores, para esto se escoge un valor U y todo pixel que cumpla que R(pixel)+G(pixel)+B(pixel) < U se transforma en un pixel blanco y de no cumplir en uno negro.

Para decidir que U usar se usa el siguiente algoritmo[3]:

1 se comienza con un valor U random

2 la imagen I es segmentada en dos conjuntos de pixeles, los que se encuentres por encima del umbral U y los que no:
G1 = $I_{m,n} : I_{m,n}$ >U (pixeles que se encuentra por encima del umbral)
G2 = $I_{m,n} : I_{m,n}$ <=U (el resto)

3 se obtiene el promedio de cada conjunto:
m1 = valor promedio G1
m2 = valor promedio G2

4 un nuevo umbral es creado usando m1 y m2
U' = (m1+m2)/2

5 se vuelve al paso 2 usando el nuevo valor de U, repitiendo hasta que U sea similar a U'

[3] http://en.wikipedia.org/wiki/Thresholding_(image_processing)

3.1.4. Filtro (4) ZOOM

El Zoom se encarga de agrandar el centro de la imagen haciéndola 4 veces más grande. Para esto lo que hace es hacer 4 copias de cada pixel del centro en la nueva imagen, es decir, utiliza la técnica de Nearest Neighbor[4].

3.1.5. Filtro (5) suavizado de la imagen con un filtro gaussiano

Para sacar ruido de las imágenes y tener mejores resultados en la detección de bordes es bueno suavizar la imagen usando este filtro.[5]. Este filtro también da la sensación de desenfocar la imagen.

[4]http://en.wikipedia.org/wiki/Nearest-neighbor_interpolation
[5]http://www.seccperu.org/files/DeteccióndeBordes-Canny.pdf

3.1.6. Filtro (6) Detección rápida de bordes

Para este filtro lo que se hace es implementar un algoritmo muy básico de detección de bordes.

Defino las siguiente funciones:

$$\begin{aligned} R(pixel) &= Intensidad\ del\ rojo\ en\ el\ pixel \\ G(pixel) &= Intensidad\ del\ verd\ en\ el\ pixel \\ B(pixel) &= Intensidad\ del\ azul\ en\ el\ pixel \end{aligned}$$

Dada A imaginen y $a_{i,j}$ un pixel en la posición i,j defino las siguientes variables:

$$\begin{gathered} DifferenciaConElPixelHoriztal_{i,j} = \\ abs(R(a_{i,j}) - R(a_{i,j+1})) + abs(G(a_{i,j}) - G(a_{i,j+1})) + abs(B(a_{i,j}) - B(a_{i,j+1})) \\ DifferenciaConElPixelVertical_{i,j} = \\ abs(R(a_{i,j}) - R(a_{i+1,j})) + abs(G(a_{i,j}) - G(a_{i+1,j})) + abs(B(a_{i,j}) - B(a_{i+1,j})) \end{gathered}$$

El pixel $a_{i,j}$ es considerado borde si y solo si se cumple que:

$$DifferenciaConElPixelHorizontal_{i,j} + DifferenciaConElPixelVertical_{i,j} > 75$$

Donde 75 es un número que empíricamente funciono bien.

Todo pixel considerado borde en la imagen se le asigna el color blanco. De no ser así se le asigna el color negro.

3.1.7. Filtro (7) Detección de bordes mediante el algoritmo de Sobel

Para encontrar bordes con mayor precisión implemente el método de Sobel [6].

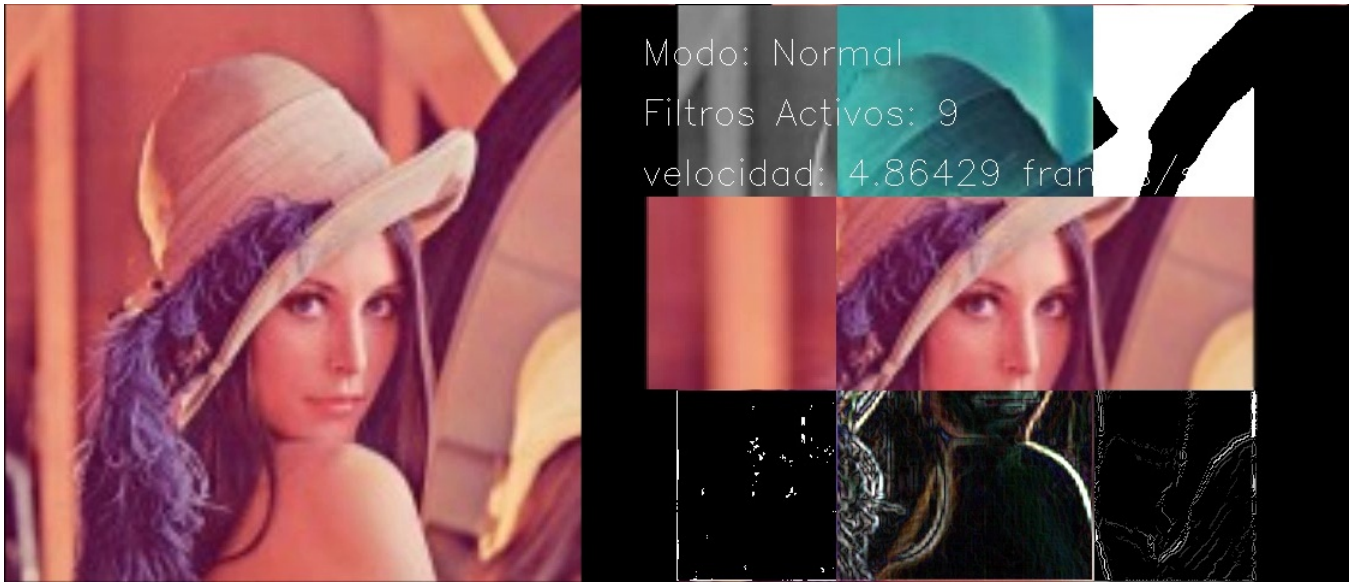
3.1.8. Filtro (8) Detección de bordes mediante el algoritmo de Canny

Detección de ejes con Canny[7] es un método eficaz de encontrar bordes. Para la implementación me base en un trabajo que encontré en internet [8].

[6] http://en.wikipedia.org/wiki/Sobel_operator
[7] http://en.wikipedia.org/wiki/Canny_edge_detector
[8] http://www.seccperu.org/files/DeteccióndeBordes-Canny.pdf

3.1.9. Filtro (9) Todos juntos

Simplemente divide la pantalla en 9 cuadrados y le aplica un filtro distinto a cada una menos al cuadrado central que queda sin modificaciones.



3.1.10. Filtro (Q) Zoom con interpolación bilineal

Se hace un zoom similar el filtro (4), pero utilizando interpolación bilineal.[9]

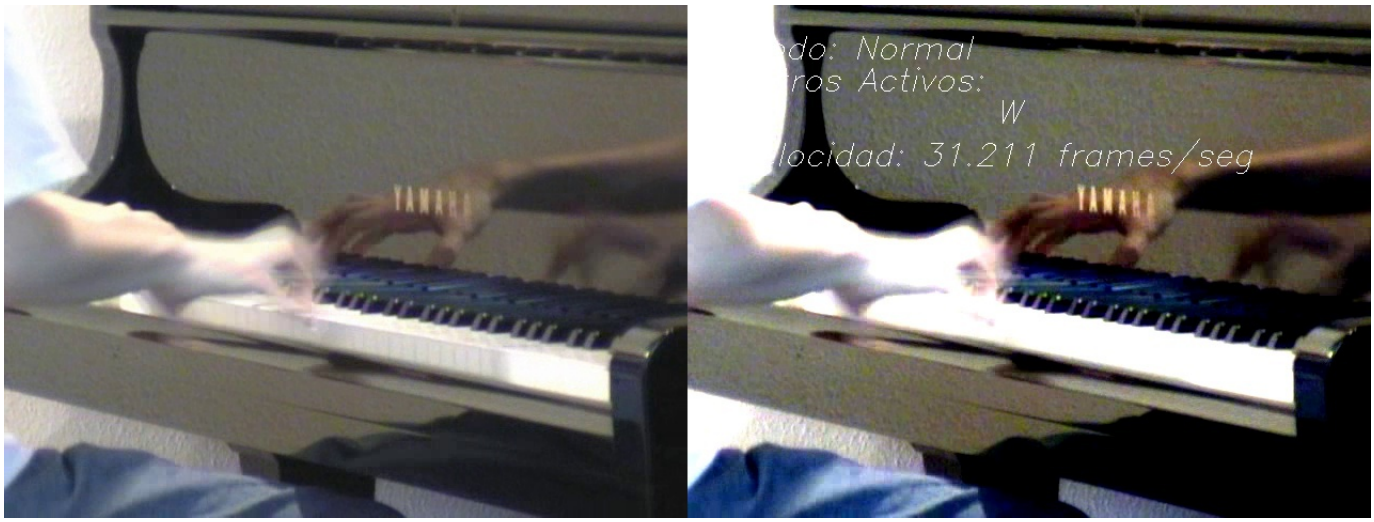
En nuestro caso, como la distribucion de los puntos de evaluación es siempre la misma (son grillas cuadradas uniformes), los coeficientes a tomar para la interpolación están fijos, y son $\frac{1}{16}$, $\frac{3}{16}$ o $\frac{9}{16}$.

[9] http://en.wikipedia.org/wiki/Bilinear_interpolation

3.1.11. Filtro (W) Realce de contraste.

Utilizamos una noción de constraste definida en [10], bajo el nombre de RMS contrast. Según esta definición, el contraste es el desvío estándar de las intensidades de los píxeles. Lo que proponemos con este filtro entonces es duplicar el contraste, sin alterar la media de los píxeles.

Para esto, si llamamos x_m a la media de intensidad de los píxeles, tenemos que aplicando la transformación $x \mapsto 2x - x_m$ se duplica el contraste sin modificarse la media. Esta transformación se aplica de forma independiente a cada color de la imagen.



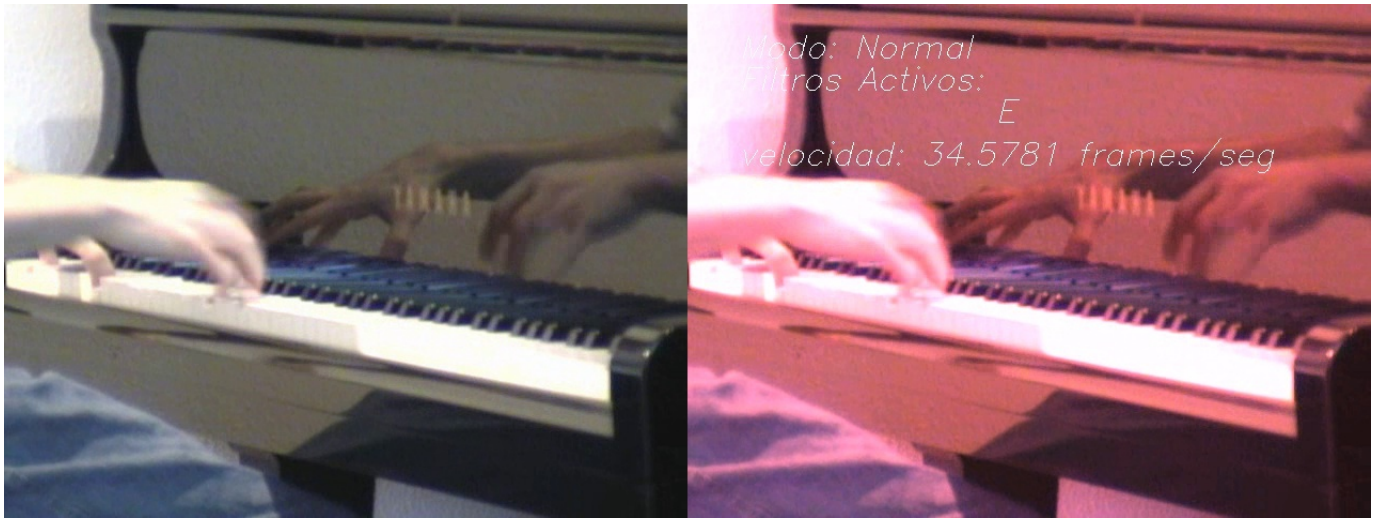
3.1.12. Filtro (E) Ajuste de color

Simplemente se ajusta la intensidad de cada color multiplicándola por un coeficiente de transformación k (tres coeficientes, uno por color). Los coeficientes se especifican al programa mediante la opción `-rgb`. Si no se utiliza se toman valores por defecto que tiñen la imagen de un color rojizo.

Para evitar salirse de los enteros innecesariamente, se utiliza punto fijo, de forma que si se especifica un valor de v para un coeficiente, el valor real del coeficiente es $k = \frac{v}{128}$. En particular, un valor de 128 no modifica un color, un valor de 0 lo anula y un valor de 256 duplica la intensidad de ese color.

En caso de que el valor de la intensidad de un píxel supere el máximo permitido de 255, se satura, quedando 255 como valor final.

[10] http://en.wikipedia.org/wiki/Contrast_(vision)

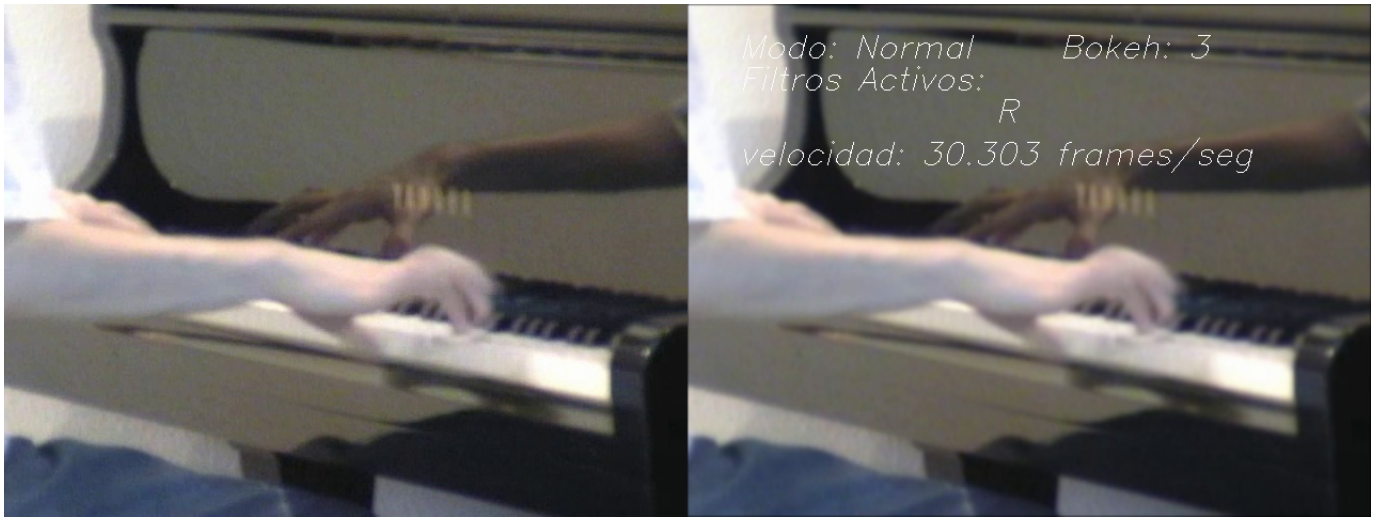


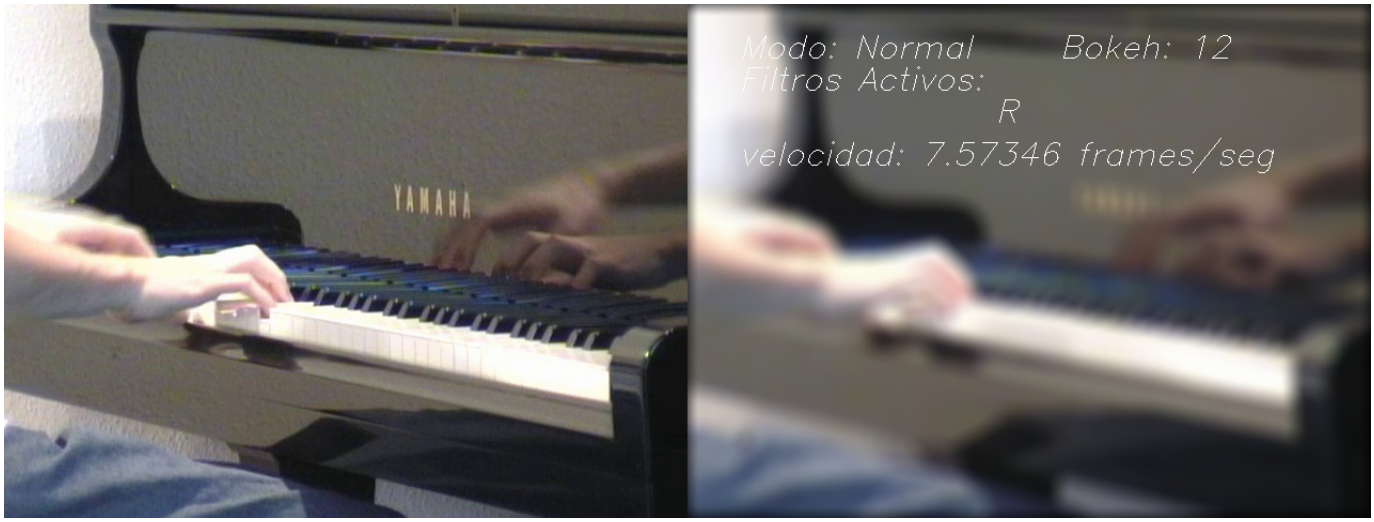
3.1.13. Filtro (R) Bokeh

Se realiza una simulación del bokeh, mediante la convolución de la imagen con un kernel que aproxima un disco uniforme (contiene únicamente dos valores, 0 y c, donde la distribución de los c aproxima un disco circular).

Se toma el valor de c de tal forma que la intensidad de color promedio no se modifique, es decir, se toma $c = \frac{1}{A}$, siendo A la cantidad de celdas de valor c en el kernel a utilizar. Para manejar los bordes de la imagen, se asume que la intensidad de color fuera de la misma es 0 (cuando el valor de desenfoque es suficientemente grande, se puede apreciar un oscurecimiento progresivo hacia los bordes de la imagen).

La intensidad del desenfoque se puede aumentar con la tecla `P` y disminuir con la tecla `O`.

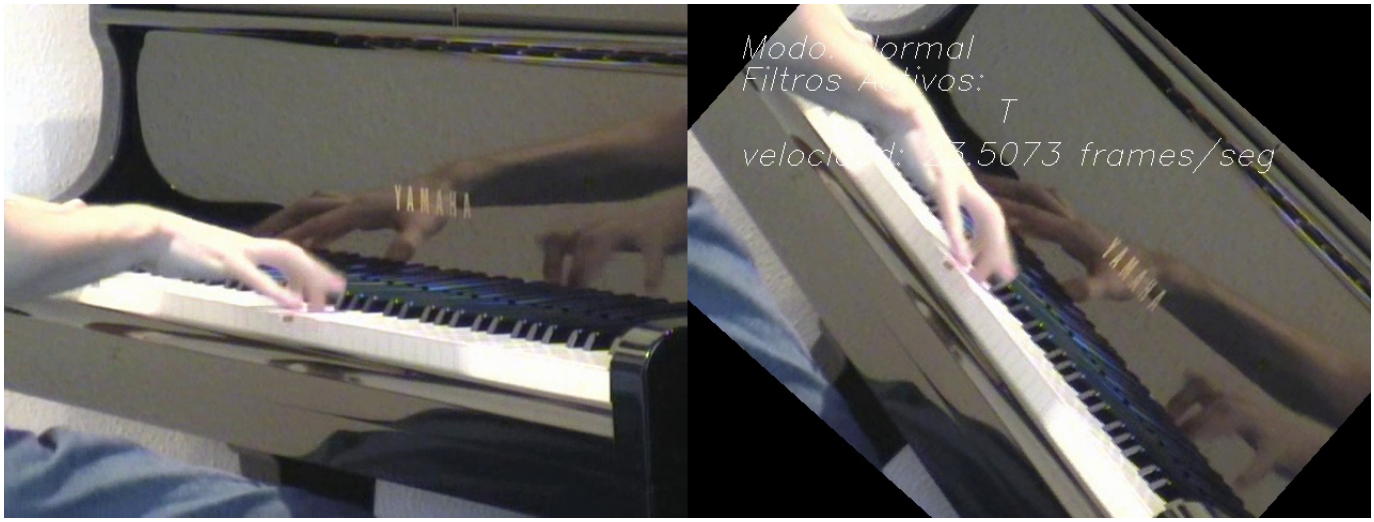




3.1.14. Filtro (T) Rotación por nearest neighbor

Se aplica una rotación a la imagen, aplicando la ya mencionada técnica de nearest neighbor [11].

El método es simplemente calcular, para cada píxel destino, la coordenada correspondiente en la imagen original previo a la rotación, y samplear su valor. Si el píxel más cercano resulta estar fuera de la imagen, se asume un píxel negro.



[11] http://en.wikipedia.org/wiki/Nearest-neighbor_interpolation

4. Resultados

Para correr el programa usamos una CPU " Intel i5 @ 2.50GHz " y una placa de video "GFORCE 9600 GSO". Como medida de velocidad tomamos los frames por seg que puede generar la computadora.

Se corrieron los filtros en 3 resoluciones de videos 360p (640x360), 720p (1280x720) y 1080p (1920x1080).

La información será presentada en este estilo de tablas:

Dimensión de la imagen	Sin filtros	Llamada a CUDA
360p	75	62
720p	53	38
1080p	30	23

Donde que por ejemplo la velocidad de frames alcanzadas con 360p sin ningún tipo de filtros es de aproximadamente 75.

La tercer columna indica los frames que se alcanzan si se copia la imagen desde la memoria principal a la memoria de video y de vuelta sin hacer ningún tipo de proceso sobre ella. Dando este una velocidad máxima que podría llegar a alcanzar un filtro que es aplicado por la placa de video.

4.1. Performance de los filtros

Por cada resolución se hizo una tabla similar, mostrando la eficiencia de las implementaciones de cada uno de los filtros con los 4 lenguajes elegidos para este tp.

Todas las medidas fueron tomadas en Frames/Segundos.

4.1.1. 360p

Tipo de Filtro	c++	Assembler	SSE	CUDA
1	52	58-60	65-69	55
2	60	72	73	55
3	13-19	45-53	49-54	19-25
4	38	65	67	55
5	6	10	26	6
6	39	60-62	70	49
7	19	13	42	36
8	12	22	34	20
9	19-21	29-30	50	17
Q	54	60	67	55
W	43	65	64	55
E	55	68	70	55
R	9	33	36	22
T	27	60	65	55

Graficando cada uno de los filtros:

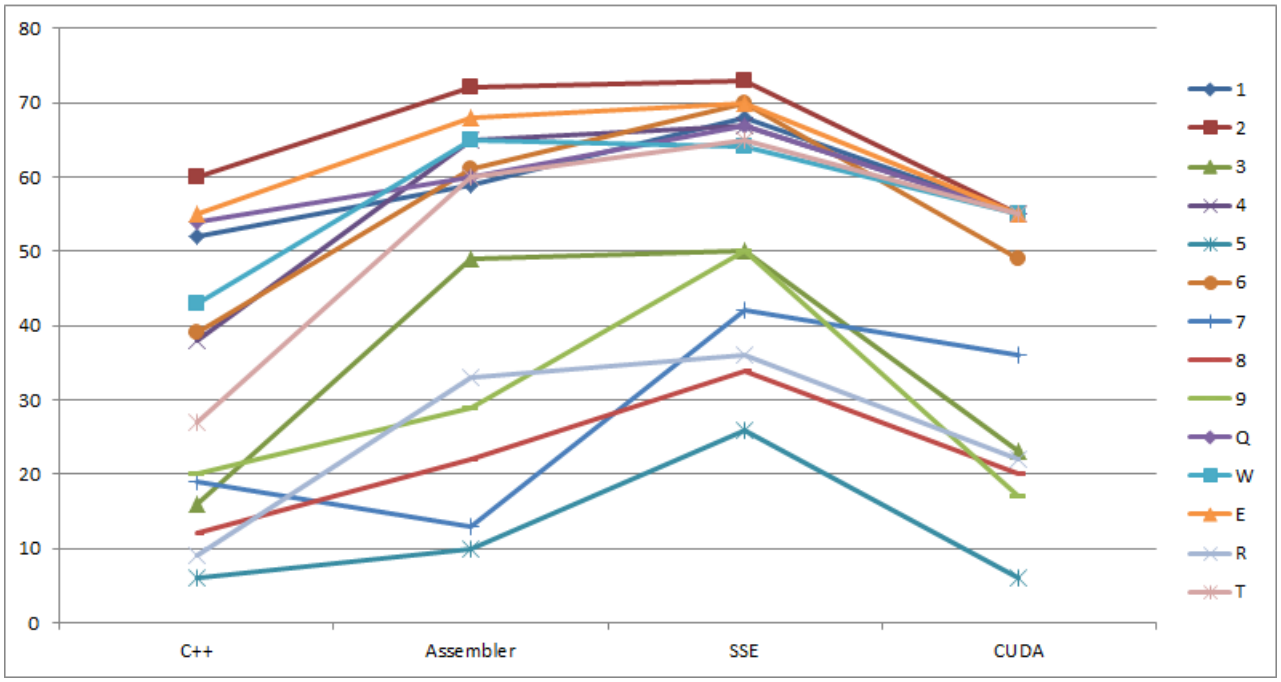
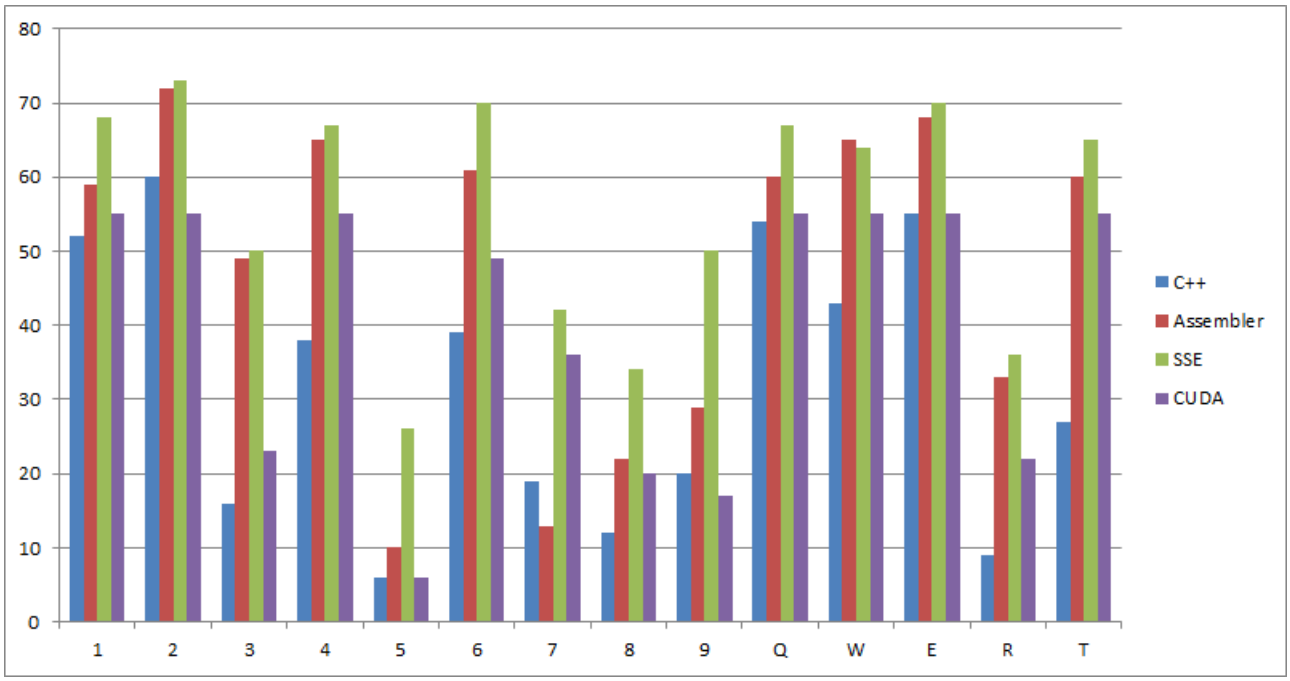


Gráfico de barras con los filtros graficados individualmente:



4.1.2. 720p

Tipo de Filtro	c++	Assembler	SSE	CUDA
1	22	27	35	27
2	27-29	41-45	44-47	27
3	4	20-25	21-26	8
4	12	31-35	33-37	26
5	1.5	2.8	6	2.1
6	14	31	38	17
7	5	3.7	12	10
8	2.9	7	9	7.3
9	6	9	16	2.5
Q	23	30	35	27
W	17	35-37	32	25
E	24	39	42	25
R	2.3	11	12	7
T	9	36-31	36	28-29

Graficando cada uno de los filtros:

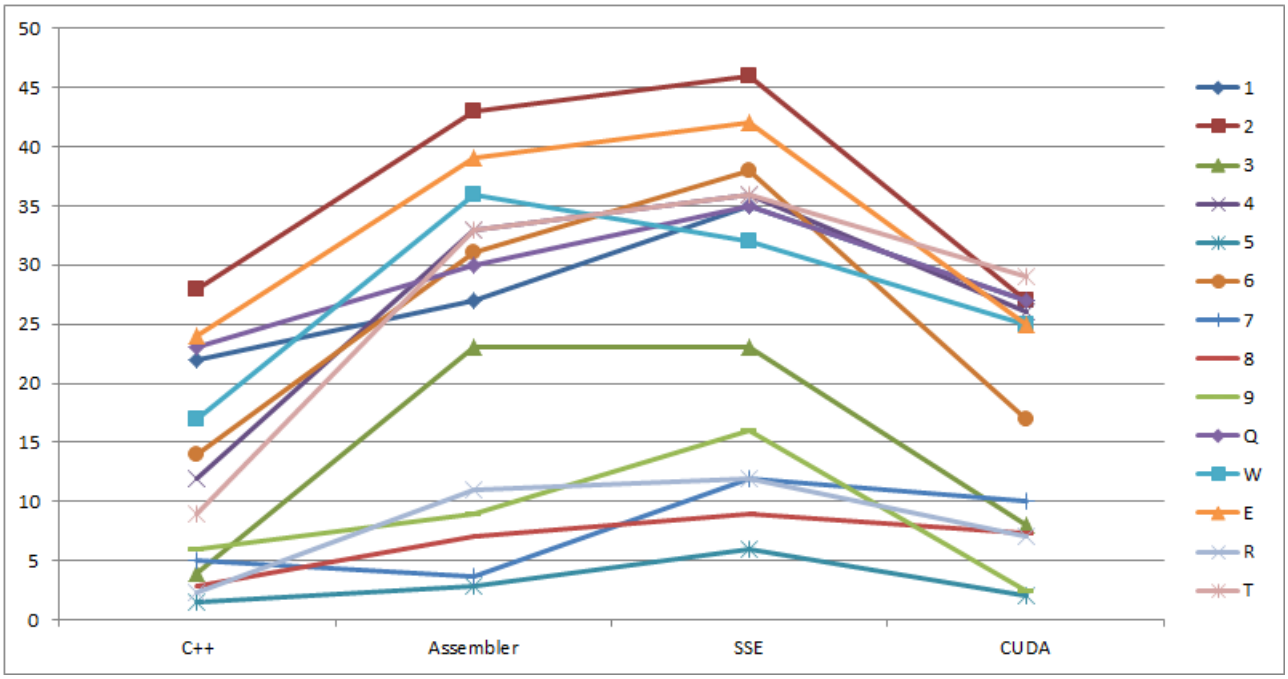
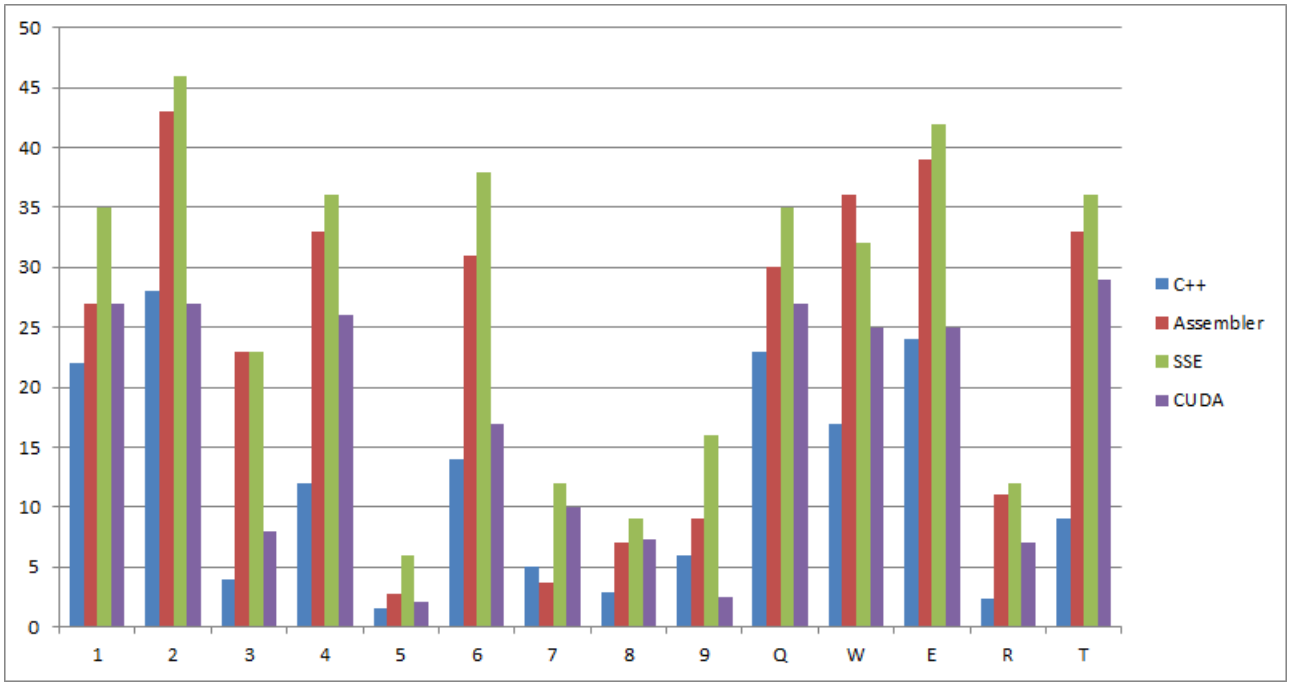


Gráfico de barras con los filtros graficados individualmente:



4.1.3. 1080p

Tipo de Filtro	c++	Assembler	SSE	CUDA
1	10	14	17	8
2	14	23	24	8
3	1.7	11	10	2.2
4	5.8	18	20	10
5	1	1.2	2.5	1
6	6.8	16	23	5
7	2.2	1.7	5	2.5
8	1.3	3	5	2.3
9	2.7	4	7	2.58
Q	12	17	20	10
W	8.3	19	17	8
E	12	21	22	8
R	1.1	5	5	2
T	4	15	17	11

Graficando cada uno de los filtros:

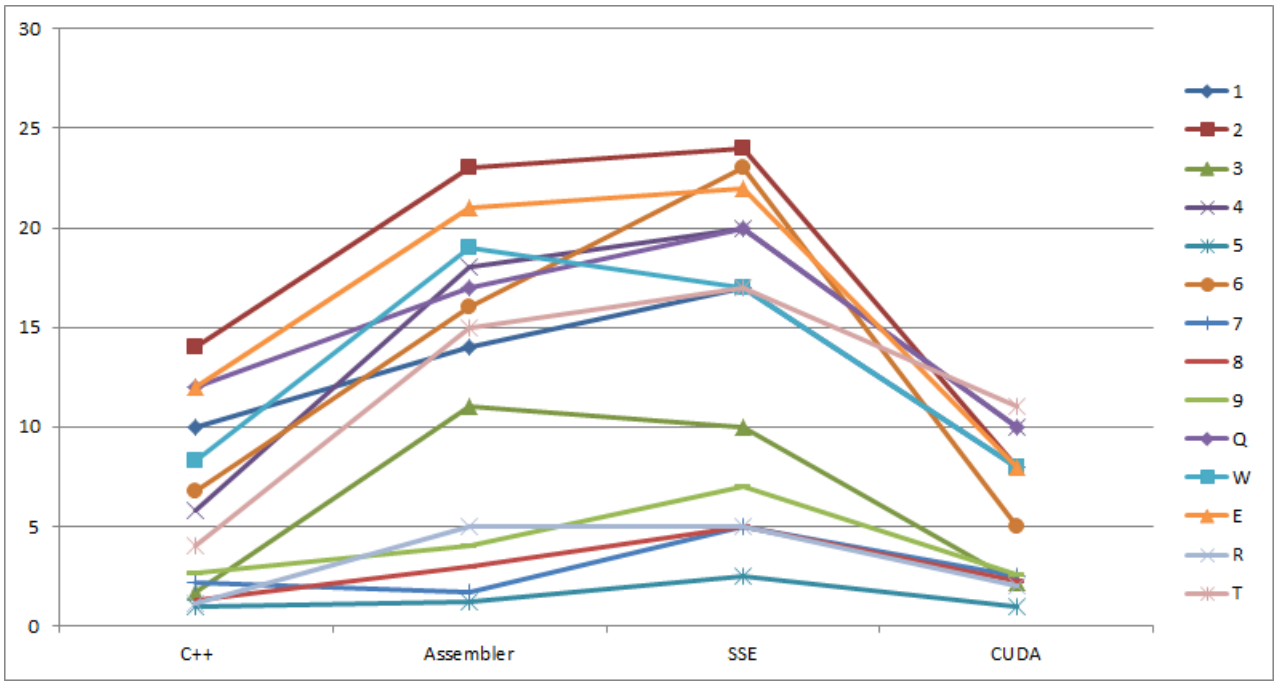
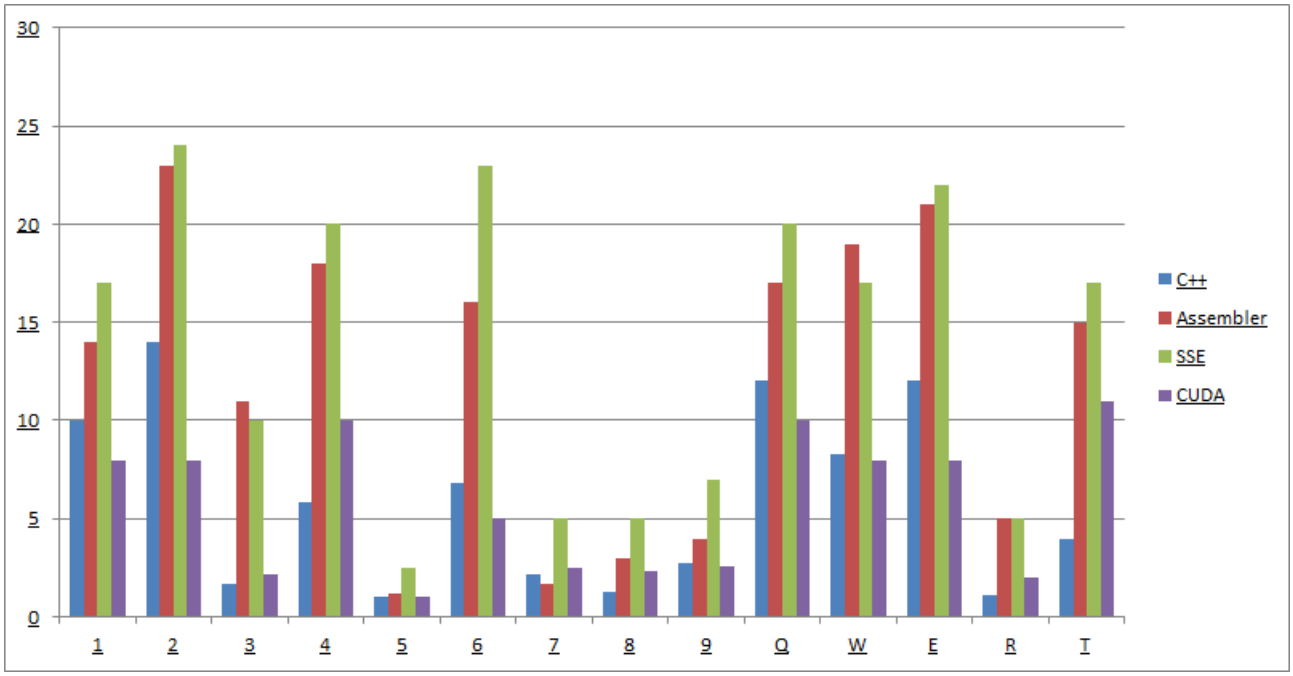


Gráfico de barras con los filtros graficados individualmente:



5. Discusión

5.1. Performance de los filtros

5.1.1. C++

Se puede ver en los resultados que el código de C++, como era de esperarse, no ha sido la más rápida en ninguno de los casos, siempre es esperable que el Assembler corra más rápido (cosa que ocurrió, menos en el filtro 7).

5.1.2. Assembler

Como se puede apreciar le ha sacado una ventaja importante, en casi todos los filtros, a la implementación en C++ y CUDA. Acentuando cada vez más a medida que el tamaño de la imagen aumenta. Se puede ver en el primer grafico de cada resolución como la pendiente de la línea entre C++ y assembler como así también de SSE a CUDA es más acentuada a medida que la resolución aumenta.

5.1.3. SSE

SSE como era de esperarse tuvo una performance aún mejor que el assembler sin utilización de este set de instrucciones. Gano en velocidad en casi todos los filtros y en algunos cosas (especialmente los filtros con necesidades de procesamiento altas) pudo sacar una ventaja enorme. Por ejemplo en el filtro (5) duplica la cantidad de frames que puede generar el programa.

Si bien es muy rápido fue realmente muy complicada su implementación, es poco factible hacer cosas complejas con este set de instrucciones que cuesta muchísimo pensar como poder sacarle la mayor ventaja posible.

5.1.4. CUDA

A diferencia de las expectativas iniciales, la performance de CUDA no fue la mejor de las 4. Perdiendo contra Assembler y SSE.

Las razones más significativas por la que creemos que esto ocurre son la demora en la trasferencia de la imagen desde la memoria principal a la memoria de video y que el código CUDA compilado normalmente no es tan optimizado como lo que se puede lograr en Assembler (por la misma razón por la que el Assembler le saca ventaja a C++). También hay que tener en cuenta que la placa de video en la que se corrió, dentro de las que corren CUDA, es de las más lentas.

Por ejemplo en el filtro uno donde lo único que hay que hacer en cada pixel es invertir los colores, CUDA difícilmente puede competir contra Assembler que únicamente recorre la matriz haciendo una cuenta sencilla por casillero, mientras que CUDA tiene que tomar una cantidad considerable de tiempo en mandar la información a la placa de video y luego retornarla.

5.2. Cambio en la resolución de la imagen

Con el cambio de resolución la diferencia entre los códigos se acentúa bastante y se aprecia más la diferencia que SSE y Assembler le sacan en casi todos los filtros a las otras dos implementaciones.

6. Conclusiones

CUDA no es eficiente para el proceso de este tipo de filtros de video (probablmente algunos filtros mas pesados y paralelizables sea mas util) ya que toma demasiado tiempo el transpasar la imagen siendo que se puede usar ese tiempo para hacer los filtros en Assembler o SSE.