DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

7 de mayo de 2017

Organización del Computador II
Primer Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Bonggio, Enzo	074/15	ebonggio@dc.uba.ar
?, ?	?	?
Tarrío, Ignacio	363/15	itarrio@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Convertir YUV a RGB y RGB a YUV

1.1. Implementación

Explicación general de la solución

Detalles de la implementación

1.2. Análisis preeliminar

Comparación de rendimiento de ASM vs C

Comparar para distintos tamaños, relaciones entre implementaciones

1.3. Hipótesis de trabajo

Conjunto de ideas de experimentos

Afirmaciones que buscan probar verdaderas

Deben ser concisas y claras

1.4. Diseño experimental

Explicación de como y que van a medir

Explicación del conjunto de datos de entrada

Detalles de la plataforma y la configuración de la misma

1.5. Resultados y Análisis

Resultados obtenidos, gráficos y tablas

Explicación e interpretación de los resultados obtenidos

1.6. Conclusiones

Relación entre las hipótesis de trabajo y resultados

2. Combinar

2.1. Introducción al problema

Este filtro consiste en cambiar la distribución de píxeles de una imagen tal que queden ordenados en cuatro cuadrantes diferentes. Se obtendrá mediante la aplicación de este filtro cuatro imágenes distintas de menor tamaño a la original, pero en donde los píxeles de la original se encuentran aun presentes en la imagen resultante. Es decir :

$$\forall\ pixel,\ cantidadPixeles(pixel, imagenOriginal) == cantidadPixeles(pixel, imagenResultante) \land imagenOriginal.length == imagenResultante.length$$

$$pixel \in imagenOriginal$$

El resultado visual que provoca la aplicación de este filtro es la sensación de que la imagen se dividió en cuatro pequeñas imágenes cuando verdaderamente ninguna de ella es igual a la otra. Se puede ver en el ejemplo de la figura 1 como es la distribución que va teniendo la aplicación de nuestro filtro allí podemos distinguir que los píxeles antes y luego de la aplicación del filtro son los mismos.

Figura 1: Distribución de píxeles tras aplicar el filtro de combinar

También podemos ver en la figura 2 cual es el impacto de nuestro filtro en una imagen real, podremos notar aqui lo parecidas que son imágenes resultantes en cada uno de los cuadrantes, a nuestro ojo es casi imperceptible la diferencia.

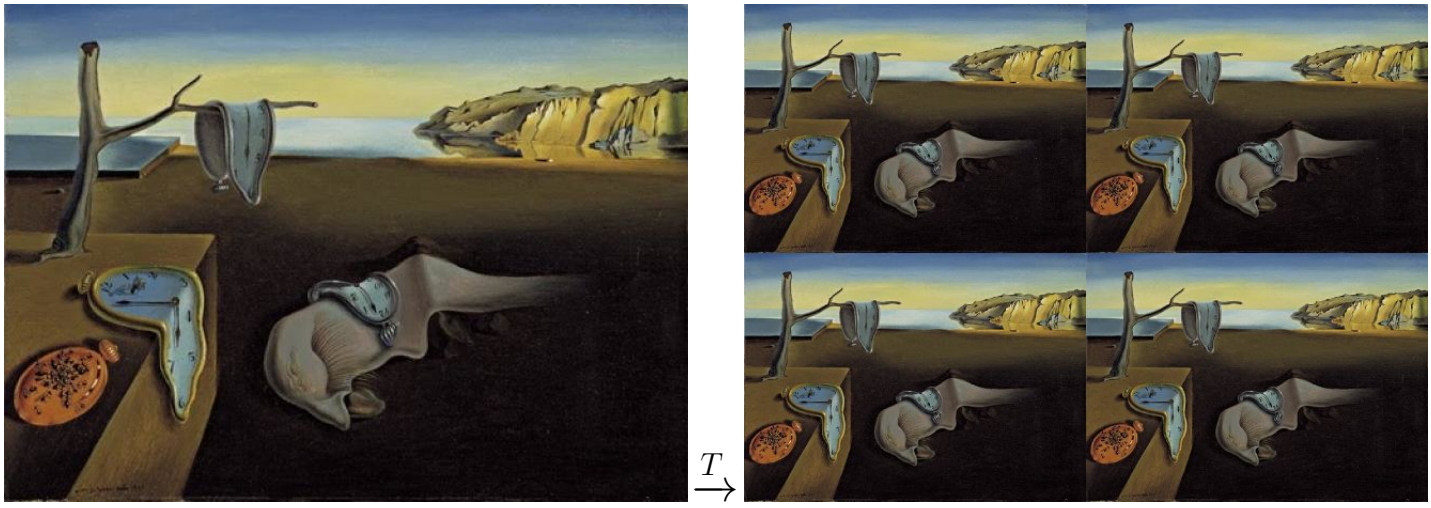

Figura 2: Imagen real antes y luego de la aplicacion del filtro

2.2. Implementación

Explicación general de la solución

Solución en codigo C La solución que fue planteada en c consiste en recorrer la matriz asociada a la imagen de entrada una sola vez píxel por píxel. Cada uno de estos píxeles posee una coordenada y por medio de esta se calcula la coordenada en la matriz asociada a la imagen de salida.

```
1  /* ************************************************************************ */
2  /* Organizacion del Computador II                                           */
3  /*                                                                          */
4  /*   Implementacion de la funcion fourCombine                               */
5  /*                                                                          */
6  /* ************************************************************************ */
7
8  #include "filters.h"
9  #include <math.h>
10
11 void C_fourCombine(uint8_t* src, uint32_t srcw, uint32_t srch,
12                    uint8_t* dst, uint32_t dstw, uint32_t dsth __attribute__((unused))) {
13     RGBA (*mSrc)[srcw] = (RGBA (*)[srcw]) src;
14     RGBA (*mDst)[dstw] = (RGBA (*)[dstw]) dst;
15     uint32_t halfHeight = srch>>1;
16     uint32_t halfWidth = srcw>>1;
17     for(uint32_t h = 0; h < srch ; h++) {
18         for(uint32_t w = 0; w < srcw; w++) {
19             uint32_t offsetH = (h % 2 == 0) ? 0 : halfHeight;
```

```
20              uint32_t offsetW = (w % 2 == 0) ? 0 : halfWidth;
21              mDst[offsetH + (h >> 1)][offsetW + (w >> 1)] = mSrc[h][w];
22          }
23      }
24  }
```

Solucion en ASM En cuanto a la implementación en código assembler se tuvo que pensar una solución totalmente diferente ya que al tener que hacerlo con registros de 128 bits nuestra solución de agarrar los píxeles uno por uno y calcular su lugar correspondiente no seria posible. Se tomaron las siguientes decisiones de modelado Decisiones de modelado:

- Decidir con cuanta información a la vez íbamos a trabajar , se llego a la conclusion que lo mejor seria trabajar con 8 pixeles a la vez. Esta decision se tomo puesto que eligiendo los 8 pixeles del lugar correcto podíamos llegar a armar 8 pixeles que iban a ser puestos en la imagen de salida. Esto nos deberia dar un ahorro en la cantidad de veces que vamos a pegarle a memoria.
- Decidir entre la posibilidad de agarrar 16 pixeles que este en la misma fila o que esten la misma columna. Nos parecio lo mas facil de implementar y lo mas efectivo a la hora de usar la cache era que tomaramos 8 pixeles contiguos. Se vera mas adelante en un experimento la diferencia entre ambos
- Ya que ibamos a tomar de a 8 pixeles contiguos pero el enunciado del trabajo practico solo aseguraba que la imagen a lo ancho iba a ser multiplo de 4 , teniamos que hacer algo con respecto a los casos donde la imagen no era multiplo de 8. En este caso tratamos de emular un poco la practica que realiza muchas veces el compilador de intel donde este separa el codigo en casos especiales para poder ahorrarse de preguntar en el medio del código . Por ende en nuestro codigo ASM solo preguntamos una vez si el ancho es multiplo de 4 o de 8 y dependiendo de eso el código se disfurca en dos

Ahora hablemos un poco mas de la iteración dentro del un ciclo de nuestro código ASM , podemos separar lo que hace dentro de una columna de lo que hace al cambiar de fila. Dentro de una columna nuestro objetivo es agarrar 8 pixeles llamense Pi con $1 \leq i \leq 8$ y trabajarlos de forma tal que queden listos para ser pegados en la memoria de la imagen destino. El siguiente grafico nos mostrara como ocurre la transformación de los 8 pixeles desde que son extraidos desde la imagen fuente hasta que estan listos para ser puestos en la imagen destino :

P8	P7	P6	P5	P4	P3	P2	P1

Separo en pares e impares

0	P7	0	P5	0	P3	0	P1

P8	0	P6	0	P4	0	P2	0

Shifteo pre juntar

P7	0	P5	0	0	P4	0	P2

Los uno cruzados con un or

P7	P3	P5	P1	P8	P4	P6	P2

Hago shuffle

P7	P5	P3	P1	P8	P6	P4	P2

Una vez que tenemos los 8 píxeles ordenados según los quiero procedo a guardarlos en la posiciones de memoria a las que apuntan el registro $R8$ y $R9$ sin preocuparme en este caso del cuadrante donde estos dos estan apuntando.

En cuando a que pasa cuando la iteración sobre la columna se termina y se pasa a una nueva columna lo vamos a explicar a continuación: Primero movemos $R8$ y $R9$ hacia su próxima fila recordando que la próxima fila sea cual sea el cuadrante donde se encuentro se realiza sumándole una fila a cada uno.

El tema viene ahora en que hay que ir switcheando los cuadrantes donde voy insertando los pixeles resultantes cada uno fila. Siendo esta parte del codigo la que se encarga de swtichear $R8$ y $R9$ entre los distintos cuadrantes se utiliza para ello una variable puente para no sobrescribir ni pisar ningun valor.

Snippet del codigo :

```
1           mov rax, r8
2           mov r8, r10
3           mov r10, rax
```

```
4           mov rax, r9
5           mov r9, r11
6           mov r11, rax
```

Lo único que cambia de esta explicación con respecto a cuando el ancho no es multiplico de 8 es en este punto donde tenemos que cambiar de fila, ya que nos quedaron 4 píxeles sin procesar y como seria demasiado trabajoso quedarselos para poder trabajarlos en otra iteración posterior, lo mejor que nos ocurrió fue trabajarlos manualmente e insertarlos en el lugar donde les corresponde. Luego de esto se sigue con el paso recién explicado.

2.3. Análisis preliminar

Comparación de rendimiento de ASM vs C

Comparar para distintos tamaños, relaciones entre implementaciones

2.4. Hipótesis de trabajo

Conjunto de ideas de experimentos

Ideas sobre los experimentos a realizar

Afirmaciones que buscan probar verdaderas

Deben ser concisas y claras

2.5. Diseño experimental

Explicación de como y que van a medir

Explicación del conjunto de datos de entrada

Detalles de la plataforma y la configuración de la misma

2.6. Resultados y Análisis

Resultados obtenidos, gráficos y tablas

Explicación e interpretación de los resultados obtenidos

2.7. Conclusiones

Relación entre las hipótesis de trabajo y resultados

3. Zoom

3.1. Implementación

Explicación general de la solución

Detalles de la implementación

3.2. Análisis preeliminar

Comparación de rendimiento de ASM vs C

Comparar para distintos tamaños, relaciones entre implementaciones

3.3. Hipótesis de trabajo

Conjunto de ideas de experimentos

Afirmaciones que buscan probar verdaderas

Deben ser concisas y claras

3.4. Diseño experimental

Explicación de como y que van a medir

Explicación del conjunto de datos de entrada

Detalles de la plataforma y la configuración de la misma

3.5. Resultados y Análisis

Resultados obtenidos, gráficos y tablas

Explicación e interpretación de los resultados obtenidos

3.6. Conclusiones

Relación entre las hipótesis de trabajo y resultados

4. Maximo cercano

4.1. Implementación

Explicación general de la solución

Detalles de la implementación

4.2. Análisis preeliminar

Comparación de rendimiento de ASM vs C

Comparar para distintos tamaños, relaciones entre implementaciones