DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

SIMD: Filtros para Imágenes

Organización del Computador II

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Chiara Tarzia	368/17	dulcechiara@gmail.com
Maximiliano Martino	123/17	maxii.martino@gmail.com
Guido Lipina	XXX/XX	mail

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Resumen

El presente trabajo tiene como objetivo aprender a utilizar instrucciones que operan con m/'ultiples datos SIMD (Single Instruction Multiple Data), soportada por la familia de procesadores x84-64 de Intel. En general, este tipo de instrucciones se utilizan mucho en procesamiento digital de señales y procesamiento de gráficos. Utilizando estas herramientas implementamos cuatro filtros diferentes para imágenes. Estos filtros son: Tres Colores, Efecto Bayer, Cambia Color y Edge Sobel.

Índice

1. Introducción

En este trabajo práctico implementamos cuatro filtros gráficos: Tres Colores, Efecto Bayer, Cambia Color y Edge Sobel. Para esto utilizamos el modelo de procesamiento SIMD (Single Instruction Multiple Data) y programamos en lenguaje ensamblador, específicamente con el grupo de instrucciones SSE (Streaming SIMD Extensions) para optimizar la performance de nuestras implementaciones, ya que con estos podemos trabajar con más píxeles de la imágen al mismo tiempo.

1.1. Formato de imágen

Es importante para la comprensión de los filtros que implementaremos, detallar el formato de imagen sobre el que se trabajará: **BMP**. Según su especificación, se considera la imágen como una matriz de píxeles, almacenados uno tras otro en memoria en sucesión de filas, de izquierda a derecha pero de abajo hacia arriba; es decir, la imagen se guarda verticalmente espejada. Los píxeles, asimismo, se almacenan por sus cuatro componentes en el orden azul, verde, rojo y alfa (BGRA), o en el caso de que la imágen esté en escala de grises, un solo componente que representa el brillo, cada uno representado por un byte y con un valor comprendido en el intervalo $[0, 255]$.

2. Desarrollo

El objetivo de la presente sección es describir y explicar la implementación de cada uno de los filtros.

2.1. Tres Colores

El filtro Tres Colores toma una imágen y, a partir del valor del brillo de cada píxel, los pinta de rojo, crema o verde.

Para esta función, ya que usamos muchas máscaras precargadas, para hacer que el algoritmo sea más eficiente y no acceder a memoria en cada iteración, las cargamos todas a los XMMs antes de iniciar el ciclo. Luego, calculamos la posición de la matriz en la que vamos a trabajar haciendo la cuenta:

$$pos = [ColumnaActual + (FilaActual \times \#Columnas] \times TamDeDato$$

Donde $\#Columnas$ depende de la imágen y se pasa por parámetro y $TamDeDato$ es la cantidad de bytes que ocupa un píxel, es decir, 4. Para completar la iteración, le sumamos 4 al iterador de columna, ya que ese es el número de píxeles que procesamos en el ciclo.

En cuanto a las operaciones para aplicar el filtro en si, realizamos los siguientes pasos:

1. Primero accedemos a la matriz usando pos y lo copiamos a un registro XMM.
2. Desempaquetamos los valores de cada píxel de byte a word, y en dos XMMs tenemos un par de píxeles en cada uno.
3. Calculamos los brillos (w) de cada píxel:
   a) Ponemos en cero los valores de A.
   b) Sumamos horizontalmente dos veces en ambos registros y obtenemos dos valores de $s = R + G + B$ en cada uno.
   c) Unimos los XMMs en uno solo, es decir, tenemos los cuatro valores de s juntos, y los unpackeamos y los transformamos a float.
   d) Dividimos los cuatro valores por 3 y los convierto a int, obteniendo los valores de $w = \frac{R+G+B}{3}$.
4. Aplicamos los mismos pasos para cada colorear los píxeles a su color correspondiente. Como ejemplo de este proceso, presentamos los pasos de uno solo:
   a) Generamos la(s) mascara(s) necesaria(s) in-place con los valores de 85 y/o 170 para compararlas con los w, y guardamos la mascara que resulto de esa operación.

Figura 1: Mascaras RojoR, RojoG, RojoB

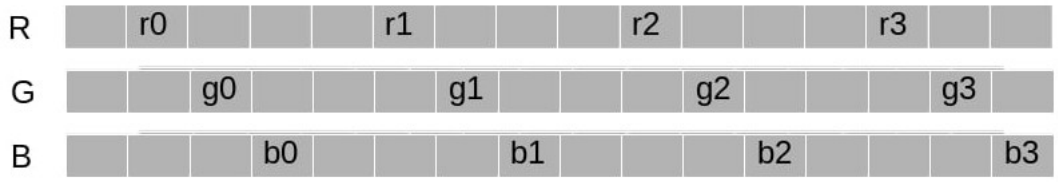

Figura 2: Registros después de shuffle.

b) Hacemos tres copias del registro con los w y a cada uno le sumamos una máscara que tiene los valores de R, G o B triplicados del color que estamos aplicando. Por ejemplo, si estamos aplicando el color rojo, nuestras mascaras se verían como la figura 1.

c) Dividimos cada uno de los valores en estos registros por 4, entonces tenemos para cada valor de w cuando estamos aplicando el color X en el registro que estamos calculando el canal C, los 4 resultados de la siguiente operación:

$$\tfrac{3}{4}ValorDeColor(X, C) + \tfrac{1}{4}w$$

Con $X \in \{Rojo, Verde, Crema\}$ y $C \in \{R, G, B\}$:

d) A estos tres registros les aplicamos la máscara que guardamos previamente para conservar sólo los valores que corresponden a este color y los empaquetamos dos veces a cada uno (d $\rightarrow$ w $\rightarrow$ byte).

e) A cada uno le hacemos un shuffle para que nos queden los valores de R, G y B en las posiciones correspondientes (figura 2).

f) Los sumamos al total.

5. Al registro que tiene el total le sumamos una máscara con $0xFF$ en los A, y con eso terminamos de armar nuestros píxeles.

6. Y para completar el proceso, movemos ese registro a la posición correspondiente.

2.2. Efecto Bayer

El Efecto Bayer construye un patrón de 4 $\times$ 4 píxeles sobre toda la imágen, en el que cada conjunto de píxeles representa solamente uno de los colores: rojo, verde o azul.

En este filtro, al igual que en Tres Colores, lo primero que hicimos fue cargar las mascaras para evitar hacer más accesos a memoria que los que necesitamos, los cuales harían que sea mucho menos eficiente. Además el modo de acceder a la posición de la matriz en donde vamos a trabajar es exactamente igual que en el filtro anterior.

Para recorrer la matriz decidimos usar un iterador para columnas y dos iteradores para filas: uno que avanza (fila y columna actuales, con los cuales recorremos la matriz) y uno que va en el sentido inverso (i, con el cual vemos de qué color hay que pintar ese conjunto de píxeles). Tomamos esta decisión porque para este filtro, la cuadrícula de colores debe empezar desde arriba a la izquierda en la foto, y

en el formato .bmp, los píxeles están ordenados de izquierda a derecha y de abajo hacia arriba, y no tenemos garantías de que el filtro es exactamente el pedido si calculamos el módulo de los iteradores que comienzan desde abajo. (Nota: los iteradores de columna avanzan de a cuatro píxeles).

En cuanto a operaciones con píxeles, en el ciclo modificamos cuatro por iteración. Los pasos que hacemos en el ciclo son los siguientes:

1. Dividimos el iterador de columna j por 8 y comparo el resto, es decir, $jmod8$, con 0. Aqui se abren varios casos:
   - $jmod8 \neq 0$*:* Entonces sé que en esa columna tenemos bloques azules y verdes, entonces chequeamos cual corresponde a esta posición haciendo $imod8$. Si es $<= 3$ es verde, y si no, es azul.
   - $jmod8 = 0$*:* En este caso, en esta columna tenemos bloques rojos y verdes, y con la misma cuenta que antes, veo cual corresponde al actual. Si es > 3 es verde, y es rojo en caso contrario.

Una vez que sabemos qué color tienen que ser los píxeles que estamos analizando, les pasamos con *pand* la máscara correspondiente que tiene $0xFF$ en la posición de los bytes que indican el ese color y el A. Así nos quedan todos los demás en 0.

2.3. Cambia Color

Este filtro funciona recibe por parametro dos colores (C y N) y un entero (Lim) y reemplaza cada pixel por la siguiente funcion:

$$dst_{[}i,j]r,g,b = \begin{cases} src_{[}i,j]^r, src_{[}i,j]^g, src_{[}i,j]^b & \text{si } d \geq Lim \\ N^r \cdot (1-c) + src_{[}i,j]^r \cdot c, N^g \cdot (1-c) + src_{[}i,j]^g \cdot c, N^b \cdot (1-c) + src_{[}i,j]^b \cdot c & \text{si no} \end{cases}$$

Donde:
$d = \sqrt{2 \cdot \Delta R^2 + 4 \cdot \Delta G^2 + 3 \cdot B^2 + \frac{r \cdot (\Delta R^2 - \Delta B^2)}{256}}$
$r = \frac{C_1,r+C_2,r}{2} \Delta R = C_1,r - C_2,r$
$\Delta G = C_1,g - C_2,g$
$\Delta B = C_1,b - C_2,b$
$c = d^2/lim^2$
Para hacer esto recorremos toda la matriz y vamos levantando de a 4 pixeles en un registro xmm. Luego separamos las componentes rojo, verde y azul de cada pixel y hacemos las cuentas necesarias para cada $4 - tupla$ ($\Delta R, \Delta G, \Delta B, yr$). Desupues sumamos todo y calculamos c que luego utilizamos para calcular el componente de cada pixel que deberia ir en caso de deba ser reemplazado el color. Por ultimo obtenemos una mascara comparando la distancia con el limite pasado por parametro y la utilizamos con el resultado de la cuenta para ver que devolver (el valor original o la cuenta).

2.4. Edge Sobel

Este es un filtro de detección de bordes, su funcionamiento esta basado en detectar cambios entre píxeles. Si consideramos una fila de píxeles en una imagen como muestras de una señal, un cambio abrupto en la pendiente determinaría un borde. De esta forma se utilizan dos operadores, uno que detecta los cambios en el sentido X y con el otro los cambios en Y.

Operador X

-1	0	+1
-2	0	+2
-1	0	+1

Operador Y

-1	-2	-3
0	0	0
+1	+2	+1

Para la implementación de los mismos, hicimos una función auxiliar para cada uno. Ninguna de estas funciones auxiliares utiliza/respeta la convención C.

Para realizar el filtro, comenzamos desde la posición (2,2) de la imagen y calculamos el final de la matriz. Luego obtenemos los punteros a la fila anterior, actual y siguiente.

Posteriormente entramos en el ciclo con el cual recorreremos la matriz. Al comenzar cada iteración, nos traemos todos los píxeles necesarios y los guardamos en XMMs: esquina superior izquierda, píxel inmediatamente superior, esquina superior derecha, píxel inmediato anterior, píxel inmediato siguiente, esquina inferior izquierda, píxel inmediato inferior y esquina inferior derecha.

$$\begin{array}{ccc} \nwarrow \text{xmm1=xmm5} & \uparrow \text{xmm2} & \nearrow \text{xmm3} \\ \leftarrow \text{xmm4} & \cdot & \rightarrow \text{xmm6} \\ \swarrow \text{xmm7=xmm15} & \downarrow \text{xmm8} & \searrow \text{xmm9} \end{array}$$

Luego llamamos a la función auxiliar operadorX, que nos devuelve en los registros XMM1 y XMM4 el resultado de la parte baja, y el de la parte alta respectivamente:

$$\begin{aligned} \text{xmm1} &= \nwarrow 0 + \leftarrow 0 + \swarrow 0 + \nearrow 0 + \rightarrow 0 + \searrow 0 \\ \text{xmm4} &= \nwarrow 1 + \leftarrow 1 + \swarrow 1 + \nearrow 1 + \rightarrow 1 + \searrow 1 \\ \text{xmm5} &= \nwarrow 0 + \uparrow 0 + \nearrow 0 + \swarrow 0 + \downarrow 0 + \searrow 0 \\ \text{xmm5} &= \nwarrow 1 + \uparrow 1 + \nearrow 1 + \swarrow 1 + \downarrow 1 + \searrow 1 \end{aligned}$$

Finalmente, aplicamos módulo tanto a la parte baja como a la parte alta, para esto copiamos a un nuevo registro XMM, y lo restamos a un registro en 0, posteriormente obtenemos el máximo de cada registro. Finalmente empaquetamos ambos resultados en un solo registro y hacemos una suma con saturación sin signo, y el resultado lo guardamos en memoria.

Una vez finalizado el ciclo, calculamos nuevamente el final de la matriz y apuntamos a los ultimos 4 pixeles y ciclamos una última vez para poder aplicarle el filtro a los mismos, que eran los que restaban. Por último nos encargamos de poner los bordes de la imagen en 0. Para esto, ponemos la última y primer fila toda en cero, recorriendo la matriz de manera lineal. Y posteriormente ponemos la primer y última columna en 0, avanzando el tamaño de una fila en cada iteración.

Implementación de funciones auxiliares:
Ambas funciones son semejantes en cuanto a la implementación, por lo tanto explicaremos como resolvimos el operador X, dado que el operador Y es análogo.
Comenzamos extendiendo precisión en los registros, para esto desempaquetamos la parte alta y la parte baja de los píxeles. Luego a cada píxel lo multiplicamos por el escalar correspondiente. Y finalmente sumamos todos los píxeles de la parte baja en un solo registro XMM, y lo mismo para la parte alta. Y estos son los registros que devolvemos.

3. Resultados

En esta sección compararemos cuánto tardó cada filtro en *asm* y *c* en ciclos de reloj. En particular, utilizamos las imágenes provistas por la cátedra y repetimos el experimento 1000 veces y promediamos cuánto tardo por iteración. Tomamos la decisión de analizar ese valor porque no tarda exactamente lo mismo cada corrida del algoritmo, pero no hay mucha variación entre la cantidad de ciclos de reloj por iteración, y el promedio de correr una gran cantidad de veces el algoritmo es un buen estimativo de cuanto tarda en general.

3.1. Tres Colores

Luego de correr nuestro experimento en Tres Colores, obtuvimos los siguientes resultados:

Imágen	Ciclos insumidos en ASM	Ciclos insumidos en C	Cuánto % menos tarda en ASM?
Puente	19 397 676	37 668 240	51.5
Paisaje	16 178 870	33 222 022	48.7
Colores 32	6 386 432	12 146 551	52.6

De esto podemos concluir que la implementación en ASM tarda aproximadamente la mitad que en C, es decir, es el doble de eficiente.

3.2. Efecto Bayer

Para Efecto Bayer, estos fueron nuestros resultados:

Imágen	Ciclos insumidos en ASM	Ciclos insumidos en C	Cuánto % menos tarda en ASM?
Puente	11 239 257	25 810 792	43.5
Paisaje	9 357 069	21 464 378	43.6
Colores 32	3 770 022	8 612 822	43.8

3.3. Cambia Color

Como en el programa siempre hace todos los calculos independientemente de si debe reemplazar los colores del pixel como si no, los tiempos de ejecucion no deberian cambiar mucho si lo ejecutamos sin importar los parametros que le pasemos.

Imágen	Ciclos insumidos en ASM	Ciclos insumidos en C	Cuánto % menos tarda en ASM?
Puente	11 380 419	35 578 484	31.9
Paisaje	9 835 728	30 023 961	32.7
Colores 32	8 303 447	18 559 974	44.7

3.4. Edge Sobel

Imágen	Ciclos insumidos en ASM	Ciclos insumidos en C	Cuánto % menos tarda en ASM?
Puente	2 023 756	52 628 556	96.2
Paisaje	1 643 739.750	43 266 152	96.2
Colores 32	639 565.562	16 976 804	96.2

Tardando poco menos del 4 % de lo que tarda C, observamos que la cantidad de ciclos insumidos en la implementación ASM es drásticamente inferior a la de C.

4. Conclusión

En resumen, en este trabajo tuvimos nuestra primera aproximación a las operaciones SSE, y pudimos comparar su eficiencia contra el procesamiento de datos individuales. Observamos que hay una clara ventaja en tiempo de cómputo con respecto a la implementación en lenguajes de alto nivel, C++ en nuestro caso, pero hay una clara desventaja a la hora de diseñar los algoritmos como así a la hora de encontrar errores y ”debugear”, lo cual contrajo grandes dificultades; a esto se le suma la complejidad y lo poco natural que resulta leer código en lenguaje ensamblador.