Organización del Computador II

Segundo Cuatrimestre de 2009

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

Trabajo Práctico 2

Procesamiento de imágenes para la detección de bordes en lenguaje ensamblador

Grupo XOR

Integrante	LU	Correo electrónico
Daniel Grosso	694/08	dgrosso@gmail.com
Nicolás Varaschin	187/08	nicovaras22@gmail.com
Mariano De Sousa	389/08	marian_sabianaa@hotmail.com

Índice

1. Introducción

En el presente trabajo, nos proponemos mejorar el programa de procesamiento de bordes realizado en el trabajo práctico anterior. La mejora sustancial consiste en utilizar el modelo de instrucciones SIMD con instrucciones SSE y diversas optimizaciones en el algoritmo para lograr un mejor rendimiento del programa.

A los algoritmos de Roberts, Prewitt y Sobel, que ya implementamos en el trabajo anterior, se le sumara el algoritmo de Frei-Chen. Para ello, asumiendo un ancho de imagen múltiplo de 16 píxeles, se procesaran en paralelo 14 píxeles en cada ciclo del programa logrando una considerable mejora en la performance.

El trabajo está orientado fuertemente a lograr velocidad, sacrificando necesariamente legibilidad del código, modularizacin y simpleza para cumplir el objetivo. Luego el resultado ser comparado con previas implementaciones para mostrar las diferencias de rendimiento de cada uno.

La interfaz con el usuario, escrita en lenguaje C, será similar a la del trabajo anterior y se detallar la nueva implementación en ensamblador en las siguientes secciones.

2. Desarrollo

El desarrollo de este trabajo se basó en modificar la implementación anterior con el fin de alcanzar los objetivos. Para lograr una mejor performance usando instrucciones del set SSE se propusieron diversos algoritmos posibles, siendo el último de los que describimos a continuación el elegido para la versión final. En todo momento se asumió un ancho de imagen múltiplo de 16 píxeles y cada algoritmo fue pensado independientemente de la matríz a usar.

El primer algoritmo pensado agarraba 16 píxeles, los 16 píxeles por encima y los 16 píxeles por debajo de esos de la siguiente manera:

```
xmm0  x  x  x  x  x  x  x  x  x  x  x  x  x  x  x  x
xmm1  x  p  p  p  p  p  p  p  p  p  p  p  p  p  p  x
xmm2  x  x  x  x  x  x  x  x  x  x  x  x  x  x  x  x
```

Sólo los píxeles indicados con una `p` se procesaban, usando como información adicional los que están marcados con una `x`.

Luego la matríz correspondiente se cargaba varias veces en otros registros `xmm`, de tal forma que pudieran ser procesados los 14 píxeles buscados en un solo ciclo del bucle principal. Luego, como la imagen es múltiplo de 16 píxeles, se avanzaba esa cantidad para seguir procesando. Esto creó un problema insalvable en el algoritmo: el último píxel de un ciclo y el primero del siguiente ciclo no se procesaban. Se pensó en salvar este problema manipulando esos dos píxeles con registros generales, pero la complejidad del algoritmo aumentaría y se optó por pensar un método más eficaz.

Un segundo algoritmo se basaba en cargar 16 píxeles a procesar, siendo 8 pertenecientes a una fila y los 8 restantes a la fila siguiente, cargando a su vez toda la información necesaria para procesar los dos conjuntos de 8 píxeles. La ventaja de este método está en la simplificación de cálculos, ya que al tratarse de *bytes* sin signo que tienen que ser multiplicados por algún posible valor signado en la matríz del operador, se tiene que extender el *byte* a *word*. Entonces entrando 8 *words* en cada registro `xmm`, tiene sentido procesar de a 8 píxeles. El problema de este método era la ineficiencia en accesos a memoria, y que al tener que procesar de a dos filas al mismo tiempo, el alto de la imagen tenia que ser múltiplo de dos, o manejar el caso impar de otra forma, y esto no era conveniente.

El tercer algoritmo volvió a la idea del primero, procesar de a 14 píxeles. La diferencia está en que procesaba 14 píxeles del principio de la fila y a la vez, 14 del final de la fila y en vez de avanzar de a 16 píxeles, avanzaba de a 14 tanto desde el comienzo hacia el final de la fila como del final hacia el comienzo y cuando llegaba a la mitad de la fila, pasaba a la siguiente. La principal ventaja es que nos librábamos del problema de saber cuando terminó una fila, ya que, recorriendo de ambos lados a la vez, a lo sumo se procesarían dos veces algunos píxeles del medio. El problema fue que este algoritmo necesitaba una mayor cantidad de registros e implicaba un difícil manejo de punteros.

El algoritmo final es una modificación del anterior: se procesan 14 píxeles y

se avanza de a 14 píxeles de izquierda a derecha solamente, y cuando el ancho restante de la imagen es menor que 14 píxeles, se procesan 14 de derecha a izquierda. Otra vez, a lo sumo se procesarán dos veces algunos píxeles del final, pero se justifica, ya que cuesta mas ciclos revisar el ancho y procesar esos píxeles restantes de alguna otra forma.

En la siguiente sección se detalla el funcionamiento del algoritmo junto con diferentes problemas que surgieron al programarlo.

3. Discusión

El código de este trabajo utiliza las funciones en lenguaje `C` del trabajo anterior y nuevas funciones en *assembler*. El trabajo en *assembler* esta separado en un archivo por filtro a aplicar y un archivo para las macros en común, utilizadas por los filtros.

A continuación se detallará lo escrito y pensado para el algoritmo de Sobel, que es análogo para el resto de los operadores, ya que con cambios mínimos se logró implementarlos, salvo el Frei-Chen que será descripto mas adelante.

3.1. Operadores

Al comienzo se programó la función para el operador de Sobel ya que podíamos comparar los resultados con el método `cvSobel` de la librería. Una vez programado, se aprovechó lo escrito para implementar los otros operadores, realizando cambios mínimos al código. Pasaremos entonces a detallar lo trabajado sobre el archivo asmSobel y luego los cambios hechos para llegar a la implementación de los otros archivos.

La operación consiste en un ciclo que comienza levantando los primeros 16 píxels de la imagen. Para poder operar las 3 líneas de la matríz sin saturar antes de tiempo, se convierten los bytes (píxeles) en words y las dos partes de 8 píxels son almacenadas en registros distintos. Luego se procede a procesar cada parte. Para poder aplicar las líneas a todos los píxels obtenidos, la matríz de Sobel fue extendida a 8 words, quedando cada línea definida de la siguiente manera:

```
Sobel X línea 1  -1   0   1  -1   0   1  -1   0
Sobel X línea 2  -2   0   2  -2   0   2  -2   0

Sobel Y línea 1  -1  -2  -1  -1  -2  -1  -1  -2
Sobel Y línea 3   1   2   1   1   2   1   1   2
```

Para facilitar la comprensión del algoritmo se utilizarán las siguientes representaciones:

Línea de la matríz	x	y	z	x	y	z	x	y

Primeros 8px de src	a	b	c	d	e	f	g	h

Últimos 8px de src	i	j	k	l	m	n	o	p

Para procesar cada parte, se utilizó el siguiente algoritmo:

- Desplaza el registro que contiene los píxeles las veces necesarias para dejar el primer píxel a calcular al principio. Por ejemplo, para procesar `a` y `d` no hace falta desplazar, pero para procesar `b` y `e` desplaza una vez hacia la derecha.
- Multiplica los 8 píxels correspondientes con la línea de la matríz que está siendo procesada.

a∗x	b∗y	c∗z	d∗x	e∗y	f∗z	g∗x	h∗y

- Copia el resultado a un registro auxiliar y luego se alínean los píxeles que serán operados en breve.

Temporal (1ro)	a∗x	b∗y	c∗z	d∗x	e∗y	f∗z	g∗x	h∗y

Temporal (2do)	b∗y	c∗z	d∗x	e∗y	f∗z	g∗x	h∗y	0

- Hace la suma entre los dos registros.

Resultado Parcial	a∗x+b∗y	-	-	d∗x+e∗z	-	-	-	-

- Alinea al principio del registro temporal el tercer pixel.

Temporal	c∗z	d∗x	e∗y	f∗z	g∗x	h∗y	0	0

- Hace la suma entre los dos registros.

Resultado Parcial	a∗x+b∗y+c∗z	-	-	d∗x+e∗z+f∗z	-	-	-	-

- Pone en 0 todas los bytes que contienen información no útil.

Resultado Parcial	a∗x+b∗y+c∗z	0	0	d∗x+e∗z+f∗z	0	0	0	0

- Desplaza el resultado parcial a la ubicación del destino donde se tiene que acumular. Por ejemplo, si se procesaron `a` y `d`, desplaza una vez hacia la izquierda.

- Suma el resultado parcial en el acumulador correspondiente.

- Repite el procedimiento hasta calcular los 6 píxeles centrales.

0	a∗x+b∗y+c∗z	b∗x+c∗y+d∗z	c∗x+d∗y+e∗z	d∗x+e∗z+f∗z	e∗x+f∗y+g∗z	f∗x+g∗y+h∗z	0

- Calcula por separado `g∗x+h∗y+i∗z`, que había quedado sin procesar
- Calcula por separado `h∗x+i∗y+j∗z`
- Repite el procesamiento (que se hizo con la parte baja) con la parte alta

- Repite el procesamiento de la línea con las 2 siguientes líneas, cambiando a las líneas de matríz correspondientes
- Empaqueta a bytes los acumuladores
- Copia el resultado al destino
- Avanza 14 píxels
- Repite todo el procedimiento hasta terminar la imagen

3.2. Medición de Performance

La siguiente tabla muestra la cantidad de ciclos mínima y promedio de cada implementación de los filtros, obtenidos de una muestra de 1000 ejecuciones de cada uno sobre la imagen de prueba `lena.bmp`:

Implementación	Ciclos de reloj	
	Mínimo	Promedio
Sobel		
Assembler	58.720.540	60.010.675
C	393.586.848	416.838.429
OpenCv	9.338.589	9.797.886
SSE	7.845.504	7.991.641
Roberts		
Assembler	34.714.238	42.746.947
C	320.676.453	336.808.912
SSE	1.749.548	1.915.205
Prewitt		
Assembler	60.428.397	63.629.648
C	634.337.521	664.132.063
SSE	8.024.988	8.276.912

4. Manual de usuario

4.1. Ayuda rápida

Uso:
`./bordes [opciones] [archivo]`

Opciones:
`-r#` Aplica el operador #
`-g` Modo gráfico

Operadores posibles:
1: Operador de Roberts
2: Operador de Prewitt
3: Operador de Sobel derivando por X
4: Operador de Sobel derivando por Y
5: Operador de Sobel derivando por X e Y

Si no se especifica un archivo de entrada, se usará 'lena.bmp'

4.2. Descripción

El programa se puede invocar en modo gráfico (`-g`) o directo (`-r#`) y opcionalmente una imagen. En modo directo, se leerá una imagen pasada como parámetro o la imagen por defecto (`lena.bmp`), se le aplicará el filtro seleccionado y se guardará con el nombre original mas un postfijo que indica que filtro fue aplicado y con la extensión original.

En modo gráfico, se puede abrir una imagen y aplicar los filtros en tiempo real, con las teclas del `1` al `5`, restaurar la imagen en escala de grises con la tecla `0`, y guardar el resultado actual con la tecla `s`. Se puede salir de este modo con la tecla `ESCAPE`.

El orden de los parámetros no importa, puede pasarse primero la dirección de la imagen seguida por el modo a usar o viceversa.

Tipos de imágenes soportados:

- `Windows bitmaps - BMP, DIB`
- `JPEG files - JPEG, JPG, JPE`
- `Portable Network Graphics - PNG`
- `Portable image format - PBM, PGM, PPM`
- `Sun rasters - SR, RAS`
- `TIFF files - TIFF, TIF`

(extraído de la documentación de la librería *OpenCv*)

4.3. Instrucciones de compilación

Dirigirse a la carpeta del código (`src/`) y ejecutar el comando `make`.