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. 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.

2. Discusión

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

A continuación se detallara lo escrito y pensado para los diferentes algoritmos, que fueron ajustados para mejor rendimiento según la matríz utilizada. Antes de empezar se definen nombres específicos (*macros*) para cada registro `xmm`, para facilitar la lectura del código. Los nombres definidos son: `srcl`, `srch` que contendrán las líneas leídas de la imagen original; `acul`, `acuh` que acumularán los resultados parciales del cálculo; y cuatro registros temporales `tmp1`, `tmp2`, `tmp3` y `tmp4` (en el caso del *Frei-Chen* el cuarto registro temporal se renombra a `sqrt2`).

Los algoritmos siguen una línea general, empiezan y terminan con lo definido en la convención C, se definen nombres de variables para los parámetros y se usan los registros generales de la siguiente manera:

- `eax`: Almacena las variables `xOrder` e `yOrder` que indican sobre qué derivada realizar el procesamiento.
- `ebx`: Contador para las filas.
- `ecx`: Contador para las columnas.
- `edx`: Ancho de la imagen.
- `esi`: Puntero a la imagen fuente.
- `edi`: Puntero a la imagen destino.

Luego para cada fila de la imagen se realiza el procesamiento sobre la derivada de `x` o de `y` según sea necesario procesando de a 14 píxeles, y antes de terminar la fila se revisa si se pueden calcular exactamente 14 píxeles, sino se retroceden los punteros para poder lograrlo y se pasa a la siguiente fila.

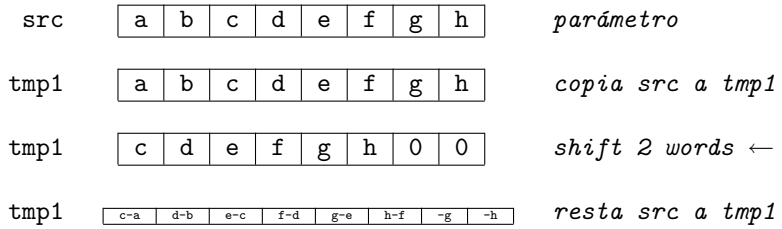
2.1. Estructura básica para *Sobel*, *Prewitt* y *Frei-Chen*

Los algoritmos de *Sobel*, *Prewitt* y *Frei-Chen* están estructurados de la misma manera. Cada uno tiene definidas sus propias *macros* denominadas `procesarLineaX`, `procesarLineaY`, `calcularX` y `calcularY`. De esta manera, el código de los tres filtros es bastante similar y para comprenderlos basta con entender la estructura básica de uno de ellos y luego los detalles particulares. La idea general del procesamiento se basa en las *macros* `procesarLineaX/Y`. Cada *macro* se encarga de leer 16 píxeles, hacer el procesamiento de los 14 píxeles procesables según la línea correspondiente de la matríz, y sumar el resultado de la operación en dos acumuladores. Las macros `calcularX` y `calcularY` son las encargadas de efectuar el procesamiento en paralelo de 6 píxeles convertidos a *word*. Cada algoritmo tiene sus versiones de ambas *macros*. Como estas *macros* no procesan todos los píxeles necesarios por limitaciones del método, cada algoritmo calcula individualmente los 2 píxeles centrales de los 16 levantados. Por último se empaquetan los datos y se guardan en la imagen resultante.

2.2. Sobel

2.2.1. Procesamiento en x

La *macro* `procesarLineaX` comienza cargando en los registros `srcl` y `srch` 16 píxeles de la imagen original, luego desempaqueta a ocho *words* adecuadamente con las instrucciones `punpcklbw` (*Unpack Low Bytes to Words*) y `punpckhbw` (*Unpack High Bytes to Words*). Mediante la *macro* `calcularX` se trabajan los ocho píxeles menos significativos contienidos en el registro `srcl`. Al aplicar convolución entre la primer línea de la matríz de *Sobel* (| -1 | 0 | 1 |) y una línea genérica | a | b | c | , el resultado esperado es | c-a | d-b | e-c | La *macro* `calcularX` obtiene el resultado esperado de 6 píxeles para *Sobel* en `x` de la siguiente manera:

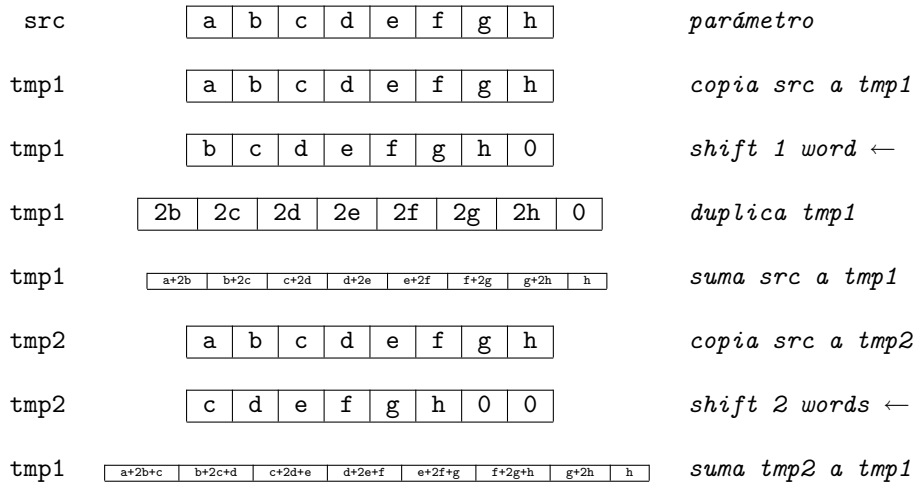


Luego de obtener el resultado mostrado en el esquema, se suman los 6 resultados válidos al acumulador correspondiente. Como la primer y tercer líneas son iguales y la segunda es el doble de la primera, se procesan todas las líneas con la misma *macro*. En el caso de estar procesando la segunda línea de la matríz (| -2 | 0 | 2 |), después de ejecutar la *macro* `calcularX`, se duplica el resultado obtenido.

Los valores intermedios se acumulan en `acul` y `acuh` y, una vez calculados, se procesan los dos píxeles centrales de los 14 píxeles originales que debían ser procesados.

2.2.2. Procesamiento en y

La idea detrás del procesamiento de Sobel en `y` es básicamente la misma al procesamiento en `x`. En este caso, al aplicar convolución entre la primer línea de la matríz (| -1 | -2 | -1 |) y una línea genérica (| a | b | c |), el resultado esperado es | -a-2b-c | -b-2c-d | -c-2d-e | . Para obtener este resultado, se realiza el siguiente procedimiento:



Luego de obtener el resultado mostrado en el esquema, se acumula el resultado de `calcularY` restando (en caso de estar calculando la primer línea de la matríz) o sumando (si se trata de la tercera). Esto se debe a que la matríz es igual en ambas líneas, salvo porque la primer línea es negativa. Como la matríz contiene sólo ceros en la segunda línea, no es necesario realizar los cálculos.

2.3. Prewitt

2.3.1. Procesamiento en `x`

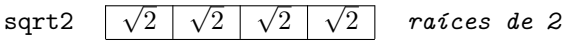
El cálculo de *Prewitt* en `x` es idéntico al de la primer línea de *Sobel*. Basta con repetir 3 veces el mismo procedimiento.

2.3.2. Procesamiento en `y`

El cálculo de *Prewitt* en `y` es similar al de *Sobel*. El único cambio en el algoritmo es que no se efectúa el 3er paso, es decir, no se duplica el registro temporal.

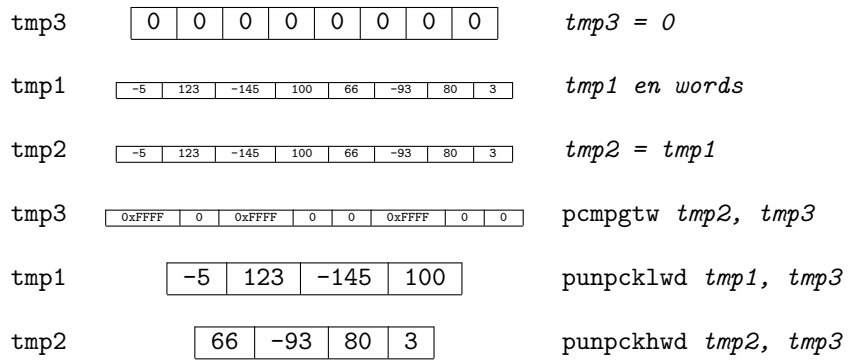
2.4. Frei-Chen

La matríz de *Frei-Chen* contiene números reales además de números enteros. Para hacer de forma más eficiente el algoritmo, primero se precalcula $\sqrt{2}$ en un registro temporal (`sqrt2`) mediante la instrucción `sqrtps` (*Square Root Parallel Single*) ya que es el único número real presente en la matríz.



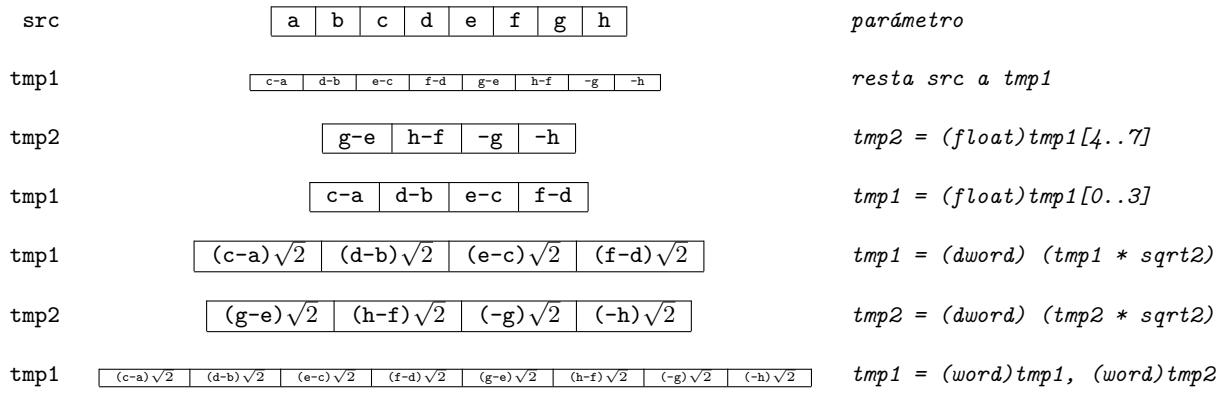
Al tener que multiplicar por un número real, los cálculos intermedios son convertidos a *float*. Por limitaciones del set de instrucciones `IA-32`, a la hora de

realizar la conversión, se extienden los enteros de 16 *bits* a 32 *bits*. Para realizar la extensión signada, se utilizó la instrucción `pcmpgtw` junto con `punpckhwd` y `punpcklwd` comparando el registro a extender con un registro en 0 como se muestra en el siguiente ejemplo:



2.4.1. Procesamiento en `x`

El algoritmo realizado para *Frei-Chen* en `x` es similar al de *Prewitt*. En este caso, la diferencia reside únicamente en el cálculo de la segunda línea de la matríz ($\boxed{-\sqrt{2} \mid 0 \mid \sqrt{2}}$). Para calcular esta línea, se aprovecha la propiedad distributiva de la multiplicación respecto de la suma y la resta, calculando primero la resta en enteros, luego convirtiendo a *float* y multiplicando el resultado por $\sqrt{2}$. Una vez hechas las operaciones, se convierte el resultado de *float* a *dword*, para finalmente convertirlo a *word*.



Luego el algoritmo continúa igual que el algoritmo de *Prewitt*, salvando los cálculos individuales de los píxeles centrales que vuelve a realizar un procedimiento similar al ya mostrado.

2.4.2. Procesamiento en `y`

En este caso, la única diferencia entre la matríz de *Prewitt* y *Frei-Chen* es en la columna central, la cuál está multiplicada por $\sqrt{2}$.

3. Manual de usuario

3.1. 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'

3.2. 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*)

3.3. Instrucciones de compilación

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