Organización del Computador II

Segundo Cuatrimestre de 2009

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

Trabajo Practico N°1

Grupo RET

Integrante	LU	Correo electrónico
Mancuso Emiliano	597/07	emiliano.mancuso@gmail.com
Villaverde Bacskay Romina	740/07	romina.villaverde@gmail.com
Mataloni Alejandro	706/07	amataloni@gmail.com

Reservado para la catedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Enunciado

Hacer un programa para el procesamiento simple de imágenes. El programa debe permitir cargar una imagen directamente en escala de grises, aplicar los operadores de derivación de Roberts, Prewitt y de Sobel para realzar los bordes y guardar la imagen en algún formato que prefieran. Debe estar programado de manera modular, para que sea fácil agregar otros tipos de procesamiento en el futuro.

Se debe escribir la parte de interacción con el usuario y manejo de archivos en lenguaje C, utilizando la librería OpenCV. Las funciones de procesamiento de imágenes deben estar escritas en lenguaje ensamblador. El programa debe recibir en la línea de comandos el nombre del archivo de entrada y la operación:

- r1 = realzar los bordes con el operador de Roberts.
- r2 = realzar los bordes con el operador de Prewitt.
- r3 = realzar los bordes con el operador de Sobel, derivación sólo en x.
- r4 = realzar los bordes con el operador de Sobel, derivación sólo en y.
- r5 = realzar los bordes con el operador de Sobel, derivación en x y en y.

El prototipo de la función de realzar bordes escrita en lenguaje ensamblador debe ser:

```
void asmSobel(const char* src, char* dst, int ancho, int alto, int xorder, int yorder);
```

También se debe comparar los tiempos de ejecución entre el operador de Sobel implementado por el grupo y el operador de Sobel implementado en la librería.

2. Implementación de los algoritmos

Para resolver el problema de la detección de bordes lo que hicimos en un primer momento fue plantear un algoritmo que recorría la imagen a la que le queríamos detectar los bordes y para cada pixel realizabamos un ciclo en el cual multiplicábamos la matriz de derivación por los puntos de la imagen y luego de saturar el resultado lo guardábamos en el lugar correspondiente de la matriz de destino. Para esto teníamos definidas en C las matrices correspondientes para cada operador, las cuales eran globales por lo que las accedíamos desde el código de assembler.

En las primeras versiones recorríamos en un ciclo todos los valores de la matriz de convolución, y realizábamos las operaciones de manera "convencional", multiplicando los valores correspondientes de cada matriz (la de la imagen y la de los convolución) y sumándo el resultado a un acumulador.

Pero luego de hacer varias consultas nos dimos cuenta que estabamos realizando operaciones de más y desaprovechando espacio. Por un lado, si ya sabemos de antemano el tamaño de la matriz, no nos convenía hacer un ciclo para recorrerla y realizar los cálculos (gastando registros y haciendo comparaciones para ver si habíamos recorrido todos los valores), sino que accedíamos directamente a cada una de las posiciones de la matriz, hasta calcular todos los valores necesarios.

Otra mejora que hicimos fue que, aprovechándonos también que conocíamos los valores que tenían las matrices de convolución, dejamos de multiplicar los valores de la imagen con aquellas posiciones de la matriz de convolución en las que habían 0s y 1s; con este cambio, y dependiendo del operador, nos ahorramos entre un 50 y un 66 % de los cálculos.

Sin embargo, con las modificaciones mencionadas anteriormente seguíamos desperdiciando espacio, ya que al conocer los valores que tiene cada matriz, y al no tener un ciclo que la recorra, podíamos simplemente realizar las operaciones con los valores (que ya conocíamos) en lugar de guardar la matriz en memoria. Ésto también nos ahorraba accesos a memoria, ya que con esta nueva implementación, usamos parámetros inmediatos en las instrucciones.

¿Cómo realizamos las operaciones para entre los valores de cada matriz?:

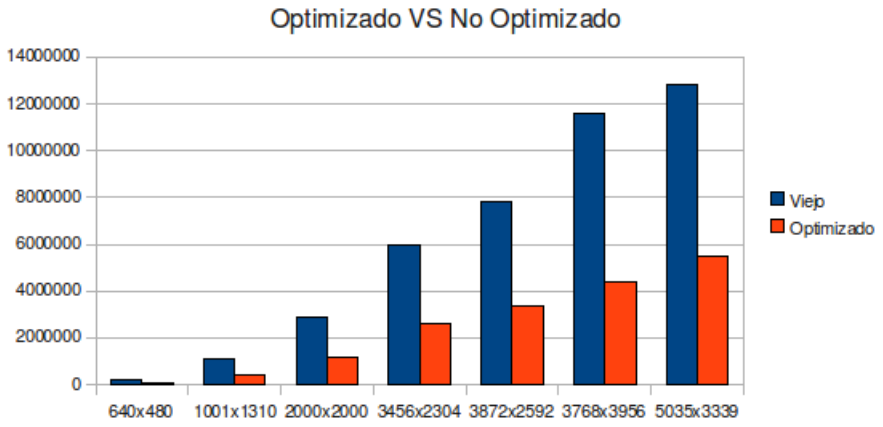
- Como dijimos anteriormente, obviamos todos aquéllas posiciones de la matriz de convolución en las que haya un 0. Es decir, en base a las modificaciones que realizamos, no tenemos en cuenta los valores nulos de la matriz, ya que tampoco incrementan el acumulador.
- Para multiplicar por 1, simplemente sumamos al acumulador el valor correspondiente de la matriz de la imagen.
- Para multiplicar por -1, simplemente restamos al acumulador el valor correspondiente de la matriz de la imagen.
- Para multiplicar por 2, utilizamos la instrucción SAL y sumamos el resultado al acumulador.
- Para multiplicar por -2, utilizamos nuevamente la instrucción SAL y restamos el resultado al acumulador. Podemos garantizar que esto funciona correctamente, ya que los valores de la matriz de la imagen son números positivos.

Como trabajamos con una matriz con números de ocho bits sin signo, luego de calcular el resultado por medio de la matriz de convolución, debíamos saturarlo. Esto significa que si el resultado que obtuvimos fue mayor que 255, entonces debíamos poner 255 en la matriz resultado (11111111 en binario). Por el contrario el resultado podia ser negativo, por lo que debíamos saturar a 0 si esto sucedía. Para esto usamos un macro al cual le pasabamos un registro (el número al que le queríamos aplicar la saturación) y nos devolvía el resultado en eax (en realidad en al que es lo que nesecitabamos).

```
% macro saturar 1                ;escribe en EAX el resultado de la saturacion

    cmp %1, 255                  ;comparamos con 255
    jnl %%poner255               ;si es mayor que 255 saturo con 255

    cmp %1, 0                    ;comparamos con 0
    jl %%ponerCero               ; si es menor a 0 saturo con 0

    mov eax, %1                  ;si esta entre 0 y 255 pongo el mismo registro
    jmp %%fin

%%ponerCero:
    mov eax, 0
    jmp %%fin

%%poner255:
    mov eax, 255

%%fin:

%endmacro
```

3. Comparación de nuestros algoritmos

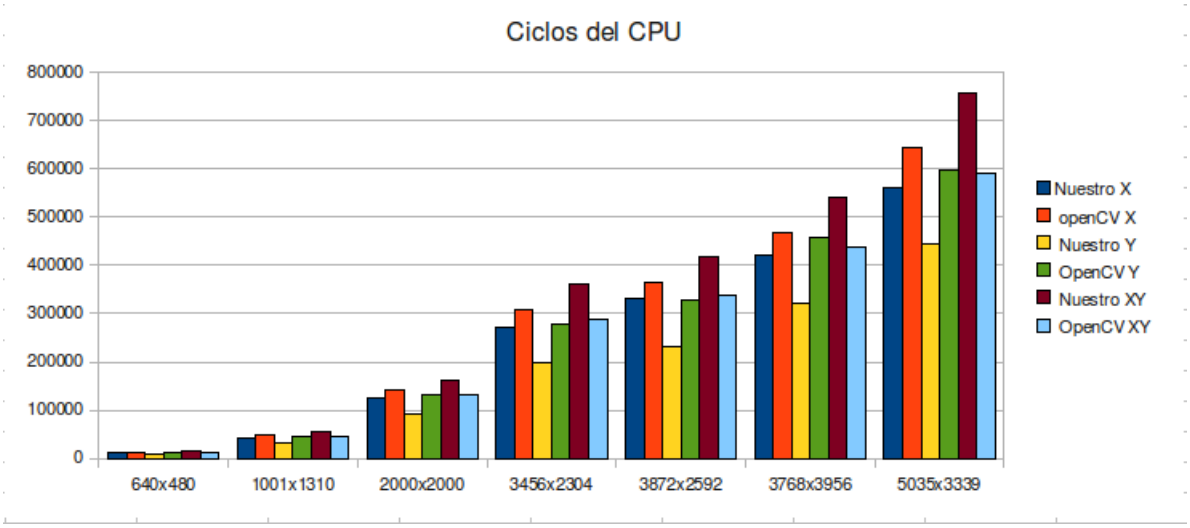
Despues de aplicar las optimizaciones, obviamente quisimos comprobar que realmente lo eran. Entonces utilizamos nuestro viejo algoritmo, que accedia a memoria, contra nuestro nuevo algoritmo que tiene las instrucciones inmediatas, y obtuvimos los siguientes resultados.



Es notable la diferencia entre las dos implementaciones, las optimizaciones que ralizamos fueron acertadas.

4. OpenCV vs. RET

La librería OpenCV implementa su propio algoritmo para el operador de Sobel. Como nosotros también implementamos uno utilizamos imágenes de distintos tamaños, y medimos los ciclos del CPU para comparar la eficiencia de las distintas implementaciones. Para esta comparación utilizamos nuestro algoritmo optimizado.



Para realizar este gráfico, seleccionamos distintas imágenes con distinta resolución, y corrimos cada algoritmo un total de 30 veces para luego calcular un promedio. Para ajustar la escala del gráfico, la cantidad de ciclos esta representada en miles, pues sino se hacía despreciable respecto a las imágenes más pequeñas.

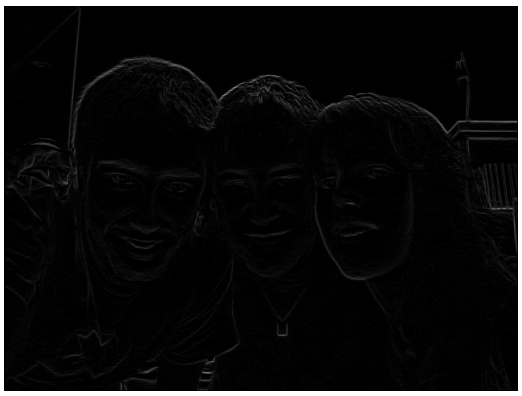
Con esté gráfico observamos que en imágenes pequeñas las dos implementaciones eran similares, pero a medida que el tamaño de las imágenes aumenta, la diferencia entre las dos se hace más grande. Si bien nuestra implementación para la derivación en X y en Y por separado consume muchos menos ciclos de CPU que la implementación de OpenCV, la librería desarrollo un algoritmo mucho más eficiente para la detección de los bordes en XY.

5. Imagenes procesadas

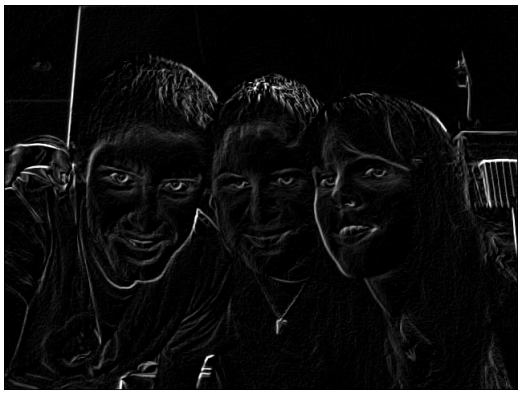
Para probar los algoritmos, y los distintos operadores, tomamos la siguiente fotografía.

Conforme a los operadores especificados, obtuvimos los siguientes resultados.

5.1. Operador de Roberts



5.2. Operador de Prewitt



5.3. Operador de Sobel

5.3.1. Derivación sólo en x

5.3.2. Derivación sólo en y

5.3.3. Derivación en x y en y

En esta última comparación existe una gran diferencia entre la implementación de OpenCV (izquierda) y la nuestra (derecha). Esto se debe a que la implementación de la librería no calcula la norma para la detección de bordes en XY, y eso hace que la saturación final de la imágen sea bastante más oscura que la nuestra, y muy alejada a los bordes que la misma libreria detecto analizando particularmente.

6. Conclusiones

Como conclusión de este trabajo podemos destacar la importancia de optimizar un código y las grandes diferencias que se aprecian al hacerlo. En este caso al procesar sobre una matriz y sabiendo los valores y el tamaño de la misma pudimos ahorrar muchas operaciones gracias a no tener que realizar ningun ciclo. Por otro lado, cuando podiamos predecir una multiplicacion por 0, 1 o -1, aplicamos ciertas instrucciones que optimizaban el código obteniendo el mismo resultado. Se pierde portabilidad del algoritmo, pero se gana en velocidad.

7. Manual de usuario

1. Ejecutar el comando `make` o `make install` desde el directorio `tp1`.
2. Acceder al directorio `exe` y ejecutarlo
   a) Directo `./tp r1 images/lena.bmp`
   b) Interactivo `./tp`
3. Las imágenes procesadas se encuentran en el directorio `resultados`.
4. En el directorio `src/images/` se encuentran algunas imágenes de ejemplo para procesar.
5. Para agregar imágenes a procesar, estas deben ser agregadas al directorio `src/images/`.

Archivos

```
tp1/
|-- Makefile
|-- enunciado
|   `-- Enunciado.pdf
|-- exe
|-- informe
|   `-- Informe.pdf
|-- resultados
`-- src
    |-- Makefile
    |-- asmPrewitt.asm
    |-- asmRoberts.asm
    |-- asmSobelXY.asm
    |-- auxiliares.asm
    |-- images
    |   |-- 10mb.jpg
    |   |-- 1mb.jpg
    |   |-- 2mb.jpg
    |   |-- 5mb.jpg
    |   |-- emma.jpeg
    |   |-- foto3.jpg
    |   |-- foto_tp.jpg
    |   `-- lena.bmp
    `-- main.c
```