Organización del Computador II

Segundo Cuatrimestre de 2009

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

Trabajo Práctico 1

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	xxx/08	marian_sabianaa@hotmail.com

Índice

1. Introducción

En el presente trabajo, nos proponemos programar una aplicación de procesamiento de imágenes para la detección de bordes escrito mayormente en lenguaje ensamblador. Para ello implementaremos distintos algoritmos de detección, los cuales se basan en obtener para cada píxel, una matríz de las derivadas parciales de los píxeles alrededor, para luego aplicarla sobre la imagen resultante.

Usaremos los algoritmos de Roberts, Prewitt y Sobel, que difieren sólo en la matríz a utilizar para la transformación de los píxeles. Se procesarán, para cada algoritmo, la derivadas parciales respecto a X y respecto a Y, teniendo para Sobel la posibilidad de aplicar el filtro en base a cada variable por separado o ambas a la vez.

En cuanto a la implementación del programa, se utilizará la librería OpenCv para el manejo de entrada/salida de las imágenes y para comparar estadísticamente el tiempo de ejecución entre la implementación de los filtros escritos en lenguaje ensamblador y la función cvSobel propia de la librería. El sistema de interfaz con el usuario está escrito en lenguaje C, mientras que los filtros de bordes en lenguaje ensamblador.

En las siguientes secciones se explicará detalladamente el trabajo realizado, mostrando diferentes resultados intermedios y decisiones tomadas.

2. Desarrollo

El desarrollo de este trabajo se dividió en 3 partes: una implementación preliminar en lenguaje C, la implementación en lenguaje ensamblador cuyo desarrollo se basó en la versión de C, y la versión definitiva que contiene agregados y arreglos diversos. Esta separación de etapas fue importante, ya que nos permitió apoyarnos sobre un código completamente funcional en C, al momento de escribir las funciones en ensamblador. A continuación detallamos las etapas del trabajo.

2.1. Implementación preliminar en lenguaje C

Al comienzo del trabajo se realizó una versión del mismo en C que actúe adecuadamente de acuerdo a lo pedido. Esta versión buscaba implementar todas las funciones requeridas en lenguaje de alto nivel para luego utilizarla como guía para escribir el código en *assembler*.

Una vez escrito el código en C, éste se aprovechó para ir reemplazando función a función con código ensamblador, porque se podía asumir que el código de alto nivel era correcto. Esto nos permitió probar que las funciones reemplazantes se comportaban adecuadamente ya que debían igualar el comportamiento de las funciones reemplazadas.

Finalmente, todo el código en C fue cambiado por sus respectivas versiones en *assembler* a excepción del `main.c` el cual contiene todas las llamadas y la interfaz con el usuario que será detallada más adelante. El código C reemplazado esta incluido en los archivos `filters.c` y `filters.h`.

2.2. Implementación en lenguaje ensamblador

Se decidió separar la implementación *assembler* en varios archivos específicos: uno por cada operador a implementar y otro (`apply_mask.asm`) que aplica una mascara dada a la imagen a procesar. La idea es abstraerse de esa función en particular que es común a todos los operadores y programarla aparte, para facilitar el testeo del código.

Primero se escribió la función `apply_mask`, luego el código general que comparten los tres operadores a implementar y por último se utilizó para generar los tres archivos fuente finales de los operadores.

2.3. Versión definitiva

Finalmente, una vez terminado y testeado todos los archivos fuente, se agregó una interfaz con el usuario por consola. La interfaz permite abrir una imagen, aplicarle un algoritmo a elección y guardarla en un archivo separado o bien abrir la imagen a procesar en una ventana y aplicar los operadores en tiempo real con la posibilidad de guardar el resultado en un archivo separado.

Se tuvo especial cuidado en el manejo de los parámetros del programa para evitar errores y los errores correspondientes a problemas con la carga o guardado de la imagen también fueron considerados.

3. Discusión

3.1. Versión C

El código sigue la misma estructura que el código assembler final: una función por cada operador (`cRoberts`, `cPrewitt`, `cSobel`) que toman como parámetros los punteros a la imagen fuente y destino, el alto y ancho de la imagen y dos variables para indicar si se debe derivar respecto a x o respecto a y y la función `apply_mask` que toma un puntero a un píxel, el tamao de la imagen original incluyendo el relleno requerido por *OpenCv* para alinear la imagen, un puntero a la mascara a aplicar y el tamao de esa mascara y devuelve el píxel transformado. Los encabezados a modo de ejemplo son:

```
void cPrewitt(const char* src,char *dst,int width, int height,int xorder,int
yorder)
int apply_mask(const char* src, int line, const char* mask, int mask_sz)
```

El código incluye matrices de `char` definiendo cada operador y un macro para la saturación. En cuanto al código en sí, fue escrito pensando en como sería mas parecido a lo que se llegaría a escribir en ensamblador, por ejemplo se optó por:

```
for( x = 1 ; x<width-1 ; x++ ) {
   if( xorder != 0 )
      k =apply_mask(&src[line*(y-1)+x-1], line, OPERADOR_PREWITT_X, 3);
   if( yorder != 0 )
      k+=apply_mask(&src[line*(y-1)+x-1], line, OPERADOR_PREWITT_Y, 3);
   ...
}
```

en lugar de:

```
for( x = 1 ; x<width-1 ; x++ ) {
   k =xorder*apply_mask(&src[line*(y-1)+x-1], line, OPERADOR_PREWITT_X, 3);
   k+=yorder*apply_mask(&src[line*(y-1)+x-1], line, OPERADOR_PREWITT_Y, 3);
   ...
}
```

o también en la decisión de aplicar, en los operadores de *Roberts* y Prewitt, las dos máscaras correspondientes por separado, como se haría con el operador de *Sobel*, aunque no fuera necesario dado que en esos operadores se aplican ambas mascaras necesariamente.

3.2. Versión asm

Pasaremos a detallar el código para la función `apply_mask` y luego el código de la implementación de los operadores, en todos los archivos se usa una macro para la convención C.

3.2.1. apply_mask

La función recibe los siguientes parámetros:

- El puntero a la imagen a transformar que se define como `imgSrc`.
- El ancho de la imagen incluyendo los bytes de alineamiento que se define como `line`.
- El puntero a la máscara a aplicar que se define como `ptr_mask`.
- El tamao de esa máscara que se define como `mask_size`.

Resumen de los registros utilizados:

- `eax`: Sólo utilizado como resultado de las multiplicaciones y como valor de retorno.
- `ebx`: Se utiliza como contador tanto como para filas como para columnas. Se asume que la mascara a aplicar es de tamao menor a 255x255, con lo cual utilizamos `bl` para las columnas y `bh` para las filas, ahorrando así el uso de otro registro para contador.
- `ecx`: Contiene el puntero a la mascara, `ptr_mask`
- `edx`: No utilizado.
- `esi`: Contiene el puntero a los píxeles a transformar avanzando sobre `imgSrc`.
- `edi`: Guarda los resultados parciales de las operaciones que realiza el algoritmo.

El algoritmo simplemente recorre la máscara sobre los píxeles alrededor del píxel pasado como parámetro (`imgSrc`) multiplicando y sumando y guardando el resultado intermedio en `edi`, para finalmente, devolver ese resultado en `eax`. Se requiere asumir siempre que la máscara es aplicable (esto es, que no intente calcular sobre valores fuera del rango de la imagen) pero esto se cumple como se verá en la discusión de las funciones de los operadores.

Al momento de escribir el algoritmo, surgieron algunas complicaciones y obstáculos. Se pensó en un principio ahorrar registros, especialmente en el problema de cómo no usar dos registros para dos contadores. Las ideas propuestas fueron: tener ambos contadores en memoria, tener un contador en memoria y el otro en un registro, tener un solo registro que guarde el valor del contador exterior (el de las filas) en el *stack*, luego se use para el contador interior (el de columnas) y finalmente se recupere del *stack* el valor guardado anteriormente, y la variante que fue elegida: asumir que la mascara no podría ser muy grande y entonces usar dos partes de un mismo registro.

Otro problema que surgió estuvo relacionado con la multiplicación de bytes que se realizaba. Se necesita multiplicar un byte no signado guardado en memoria (`[esi]`) por un byte signado perteneciente a la máscara a aplicar. Si se usaba directamente la instrucción `mul`, no se tenia en cuenta el byte signado de

la máscara, en cambio si se usaba la operación `imul`, se esta leyendo el byte en memoria como signado dando como resultado la siguiente imagen:

La idea de la solución es extender el signo de la máscara y realizar una multiplicación signada. Al principio se usó:

```
mov al,[ecx] ;extensión de signo de mascara
cbw
mov edx,eax
```

Pero luego se cambió por una instrucción que hace exactamente lo que necesitábamos:

```
movsx dx,[ecx]
```

La última cuestión con la función es que se demoraba algunos segundos en procesar y la imagen resultaba de la siguiente manera:

El problema estaba en que el contandor de columnas no volvía a 0 al final de cada ciclo, por lo cual cada aplicación de la máscara se realizaba 256 veces por vuelta. Bastó sólo con limpiar el registro al final del ciclo (`xor bl,bl`).

3.2.2. Operadores: asmSobel, asmRoberts y asmPrewitt

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.asm` y luego los cambios hechos para llegar a la implementación de los otros archivos.

La función recibe los siguientes parámetros:

- El puntero a la imagen a transformar que se define como `ptr_src`.
- El puntero a la imagen destino que se define como `ptr_dst`.
- El ancho real de la imagen, sin bytes de alineamiento, que se define como `width`.
- La altura de la imagen que se define como `height`.
- Dos parámetros que indican si se aplicará el operador derivando respecto a x o a y y que se definen respectivamente como `xorder` e `yorder`.

Se definen macros para la convención C, para el manejo de la saturación (`eax_to_char_sat`), y para conseguir el ancho de la imagen con los bytes de alineamiento (`getLineSize`).

Resumen de los registros utilizados:

- `eax`: Se usa dentro de las macros y para guardar el píxel que luego sera saturado y escrito en la imagen de destino.
- `ebx`: Guarda el puntero a la imagen fuente y avanza sobre él para obtener todos los píxeles de la imagen.

- `ecx`: Tiene el puntero a la imagen destino que avanza junto con el `ebx`, y también es usado para pasar la posición de un píxel de la imagen original a la función `apply_mask`.
- `edx`: Contiene temporalmente el resultado de la aplicar la máscara en x.
- `esi`: Guarda el puntero al inicio de la imagen de destino que luego avanzará junto con ebx y sirve para definir a ecx.
- `edi`: Contador de columnas.

También se definieron dos variables locales:

- `line`: Guarda el ancho de la linea con los bytes de alineamiento.
- `yindex`: Contador para las filas.

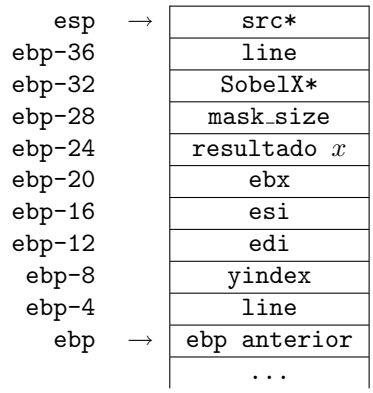
La operación consiste en recorrer toda la imagen, menos los bordes, aplicando en cada píxel la matríz de *Sobel* respecto a x o a y según lo indiquen las variables `xorder` e `yorder` y saturando el resultado de cada llamada a `apply_mask`. Luego esos resultados se suman y saturan otra vez obteniendo la transformación buscada.

Respecto a las macros definidas, la función `getLineSize` toma el ancho real de la imagen (`width`) y la cantidad a alinear (`align`, generalmente 4, para alinear a dword) y computa line de la siguiente manera:

```
line = width - (witdh mod align) + align
```

Por último, la macro `eax_to_char_sat` simplemente satura eax entre 0 y 255. En las primeras implementaciones de `asmSobel`, nos encontramos con el mismo problema de como manejar la poca cantidad de registros que apareció cuando se trabaja en `apply_mask`, pero esta vez se optó por guardar uno de los contadores en memoria. Se tuvo especial cuidado en asegurarnos de guardar cada registro necesario antes de las llamadas a `apply_mask` ya que ésta modifica la mayoría de los registros. Antes de la primer llamada a `apply_mask` se reserva un lugar (`push dword 0`) en el *stack* donde luego se almacenará el resultado de aplicar la máscara en x.

El programa llama a `apply_mask` a lo sumo dos veces por píxel, pero le pasa los parámetros al *stack* una sóla vez. Esto funciona porque al derivar respecto a y, se modifica directamente sobre la pila el parámetro del puntero a la matríz a usar, dejando el resto intacto, como se puede ver en el siguiente esquema del estado del *stack* justo antes de llamar a `apply_mask` por primera vez:



Lo único reemplazado si se quiere derivar respecto a y, es el dato en `ebp-32`. De esta manera ahorramos instrucciones y movimiento de datos en memoria.

Para los operadores de *Prewitt* y *Roberts* se reutilizó el mismo código, teniendo que cambiar sólamente las matrices a utilizar y, en el caso de *Roberts*, el `mask_size`.

Ejemplo del programa aplicado a diferentes imágenes:

3.3. Medición de performance

Dado por enunciado, además de la implementación de las funciones previamente explicadas, fue necesario realizar *tests de performance* sobre nuestro trabajo, para estos resultados ser comparados con los métodos dados por la librería *OpenCv* (función `cvSobel`). También nos pareció importante saber qué tan grande era la diferencia de *performance* entre las implementaciones en *assembler* y C. Al tener todas las funciones ya implementadas en ambos lenguajes, vimos interesante la idea de compararlas entre sí. 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
Roberts		
Assembler	34.714.238	42.746.947
C	320.676.453	336.808.912
Prewitt		
Assembler	60.428.397	63.629.648
C	634337521	664.132.063

Dado a que los algoritmos no fueron pensados para optimizar al máximo en velocidad, se puede ver que `cvSobel` realiza aproximadamente 6 veces menos de ciclos, obteniendo un mayor rendimiento que la versión en *assembler*. Por otra parte, al comparar los resultados entre las implementaciones de *assembler* y C, las diferencias son estremecedoras: la *performance* de la versión en *assembler* es aproximadamente 8 veces mejor que la versión en C.