Optimización de la detección de bordes usando SSE

J. Enríquez - LU 36/08 - juanenriquez@gmail.com
N. Gleichgerrcht - LU 160/08 - nicog89@gmail.com
J. Luini - LU 106/08 - jluini@gmail.com

Resumen

En la primera parte de este trabajo hicimos implementaciones de detección y graficación de bordes para ciertos operadores de derivación usando la arquitectura básica de la IA-32. Ahora el objetivo es optimizar dichas implementaciones (prácticamente hacerlas de nuevo) usando características más avanzadas de la familia IA-32: el modelo SIMD y el set de instrucciones SSE. Además, añadiremos a nuestro programa un operador de derivación nuevo que requiere procesamiento en punto flotante.

En este informe presentamos el trabajo realizado en esta segunda parte y las conclusiones obtenidas, incluyendo los problemas surgidos y la comparación de eficiencia entre estas nuevas versiones y las anteriores, entre las nuestras y las de la biblioteca OpenCV, y entre las implementaciones SSE de aritmética entera y la de punto flotante.

Keywords

Detección de bordes - IA-32 - SIMD - SSE

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

Índice

1. Introducción

La detección de bordes en una imagen consiste en hallar las zonas de la misma en donde el color cambia .ªbruptamente". Esta herramienta es utilizada tanto para compresión de archivos como para lograr efectos sobre las imágenes.

Una manera simple y muy útil de detectar bordes es calculando cómo varía la intensidad o luminosidad de una imagen entorno a cada uno de sus pixeles. Esto se logra tomando la imagen en escala de grises y aplicando en cada punto un operador de derivación, que es una matriz de números cuyo producto interno con el entorno del punto mide la variación de la intensidad en alguna dirección. Esta técnica se llama **convolución**.

Existen muchas matrices utilizadas para la convolución, con diferentes dimensiones y coeficientes. Las de mayores dimensiones proveen una detección más suave y menos precisa en la que se reducen los efectos del ruido".

En el presente trabajo realizamos una implementación de los operadores de Roberts, Prewitt, Sobel y Frei-Chen en el lenguaje ensamblador correspondiente a la arquitectura IA-32. La misma contiene los algoritmos resueltos utilizando instrucciones de propósito general y cuenta además con las versiones de los mismos pero aplicando instrucciones específicas para procesamiento paralelo (SIMD).

Respecto de la técnica SIMD *(Single Instruction, Multiple Data)*, esta consiste en realizar operaciones a un vector de varios elementos de forma atómica, logrando esto mediante instrucciones y hardware especializado. De esta manera, se logra paralelizar a la hora de procesar información ya que como veremos más adelante, se puede llevar a cabo el cómputo de varios puntos de una imagen en un mismo paso, cosa que usando los registros de propósito general esto quedaba limitado a procesar de a uno por vez.

Como ventajas de este método nos gustaría destacar la capacidad del procesamiento paralelo que permite optimizar notablemente aplicaciones multimedia y de procesamiento de señales digitales. Si bien esto parece ser un gran y positivo cambio, cabe destacar que no todos los algoritmos son paralelizables en este sentido. Otro inconveniente importante es que el soporte para instrucciones SIMD en la arquitectura IA-32 se fue dando en forma progresiva ya que con el correr del tiempo se fueron agregando nuevas instrucciones. El problema que esto presenta es que al usar una determinada instrucción uno tiene que ver si ese procesador soporta esa instrucción en particular, lo cual, nuevamente, hace muy restrictivo y poco "portable" a los programas que utilizan SIMD.

En particular y para este trabajo, utilizamos instrucciones SIMD de la arquitectura IA-32 de Intel correspondientes a los subconjuntos de instrucciones denominados MMX, SSE1, SSE2 y SSE3. Consecuentemente, es requisito contar con un procesador de la familia Intel y similar que tenga soporte para estas instrucciones.

A continuación, describiremos brevemente el programa realizado, discutiremos las cuestiones surgidas durante su desarrollo y expondremos los resultados y conclusiones extraídos.

2. Desarrollo

2.1. Programa

El programa que hicimos permite aplicar ciertos operadores de derivación a una imagen especificada por línea de comandos. Los operadores implementados son Roberts, Prewitt, Sobel y Frei-Chen. El primero tiene matrices de 2x2 y detecta bordes en diagonal. Los otros tres tienen matrices de 3x3 y detectan bordes verticales u horizontales. Se pueden invocar tanto las versiones de propósito general para los primeros tres, como las versiones de los operadores que utilizan instrucciones SIMD.

Para cada operador solicitado el programa aplica la matriz correspondiente en X, luego en Y, y finalmente suma los resultados. Para el caso de Sobel, también permite aplicar convolución solamente en X o solamente en Y.

Utilizamos la biblioteca **OpenCV** para manejar las imágenes convenientemente, es decir, dejamos la carga, el guardado de imágenes y el "aplanado" a escala de grises a dicha biblioteca. El grupo se limitó a crear una interfaz que contenga lo mencionado y a implementar los algoritmos de los operadores de detección de bordes en lenguaje ensamblador.

2.2. Implementación

2.2.1. Precisión numérica

Respecto de la forma de operar aritméticamente, surge un inconveniente si tenemos 16 pixeles (almacenados como bytes) cuando tenemos que hacer una resta por ejemplo, ya que podríamos estar perdiendo información al saturar. Por ejemplo:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \qquad \begin{pmatrix} 15 & 255 \\ 1 & -255 \end{pmatrix}$$

En este caso si la convolución se hace al a_{11} con la matriz de Roberts quedaría: $a_{11} = a_{22} - a_{21}$ que en nuestro caso se resume a...

Entonces para que en las restas de números no se pierda información en el medio lo que se hace es lo siguiente: traemos los bytes pero luego los desempaquetamos como *words* de forma tal de no perder información con las restas. Luego los empaquetamos saturados para grabarlos en la imagen destino convenientemente.

Gráficamente sería algo como:

Sin embargo, estas observaciones valen en caso que la operatoria se haga con valores enteros. Para el caso de Frei-Chen que tiene un $\sqrt{2}$ en la matriz esto deja de ser cierto ya que en ese caso, se opera directamente con números reales representados con precisión simple de 32-bits.

2.2.2. Operador de Roberts

El operador de Roberts presenta las siguientes matrices de convolución para x e y respectivamente.

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \qquad\qquad \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

Dada la precondición que dice que el ancho de las imágenes es múltiplo de 16 bytes y que cada pixel está representado con 1 byte, entonces, podemos visualizar a la imagen de forma tal que esta tiene n columnas de 16 bytes de ancho. Además al usar los registros `xmm` de 16 bytes de tamaño, podemos efectivamente para una fila de tamaño k pixeles hacer nada más que $\frac{k}{16}$ accesos a memoria para traer toda la fila en cuestión cuando antes no quedaba otra opción que traer todos los pixeles de uno en uno.

El algoritmo que se encarga de realizar el operador de Roberts está dividido en dos partes independientemente de la dirección en la que se está pasando el operador.

En primer lugar, como el ancho de la imagen es múltiplo de 16 bytes, se optó por dividir dicho ancho en múltiplos de 16 bytes, que como se mencionó antes es el tamaño de los registros `xmm`. Entonces se tienen por lo menos n columnas de 16 bytes de ancho y se procede trabajar con las $n-1$ primeras de la siguiente forma:

- Máscara X: Se cargan los 16 bytes de la fila actual en un registro y luego se cargan los 16 bytes de la fila siguiente pero desplazados uno. Es decir, si la fila actual es x y la siguiente es x' entonces se carga en un registro el $x_i, x_{i+1}, x_{i+2}, ..., x_{i+15}$ y en el otro $x'_{i+1}, x'_{i+2}, x'_{i+3}, ..., x'_{i+16}$ donde i es el número de la columna de la imagen.
- Máscara Y: Este caso es análogo al anterior solo que el desplazamiento se hace en la fila actual, osea, se carga el $x_{i+1}, ..., x_{i+16}$ en un registro y en el siguiente el $x'_i, ...x'_{i+15}$

En ambos casos la idea del algoritmo es mantener las cosas lo más simple posible, de forma tal de que en este punto lo único que se hace es restar cada uno de los 16 bytes (que están separados en grupos de 8 words) y luego reempaquetarlos y dejarlos en la imagen destino.

Luego se procesa la última columna de la siguiente manera:

- Mascara X: Se carga la fila actual y la siguiente sin desplazamiento pero a la fila siguiente se la *shiftea* a izquierda un byte de forma tal de tener

en un registro $x_i, x_{i+1}, x_{i+2}, ..., x_{i+15}$ para la fila actual y en otro para la siguiente: $x'_{i+1}, x'_{i+2}, x'_{i+3}, ..., x'_{i+15}, 0$. De esta manera logramos cargar la fila como corresponde (sin pasarnos del límite que tenemos que tener) y operarla correctamente luego de desplazarla dentro del registro.

- Máscara Y: Este caso es prácticamente igual al anterior. Se cargan sin desplazamiento tanto la fila actual como la siguiente. Luego se desplaza dentro del registro la fila actual con un *shift* y se pone un 0 en la última columna de la fila siguiente de forma tal de dejar el borde en 0 mediante una máscara de bits dado que ese último pixel no es procesable.

En ambos casos se realizan las operaciones aritméticas pertinentes y luego se guarda el resultado donde corresponde.

El volcado de datos a memoria se realiza solamente cuando ambas máscaras han sido pasadas. Antes se utiliza un registro `xmm` como acumulador para evitar un acceso que sería innecesario.

2.2.3. Operador de Prewitt

Matrices de convolución para el operador de Prewitt en x e y respectivamente.

$$\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

Como podemos ver, en el caso del operador de Prewitt consta, tanto para X como para Y, de matrices de $3x3$ toda de números naturales, así que podemos trabajar con operaciones sobre enteros. Utilizamos, para realizar este operación, los registros registros simd, para poder procesar en paralelo. Intentamos ir procesando de la máxima cantidad de pixeles posibles, así que terminamos realizando un algoritmo que llega a procesar de a 14 pixeles en paralelo, el cual se detalla a continuación.

Como la matriz de convolución es de $3x3$ y el valor a calcular es el del centro de la misma, lo primero que notamos es que todos los bordes de la imagen no se van a poder calcular, así que lo que se hace es poner 'a mano' un 0 en sus respectivas posiciones. Otro punto a tener en cuenta es que, como se comentó anteriormente, a pesar de poder levantar de a 16 bytes, las operaciones se realizan sobre word, así que en principio vamos a tener el doble de registros ocupados que la cantidad que levantamos.

La idea del algoritmo en si es, principalmente, recorrer toda la imagen fuente e ir guardando los resultados en la imagen destino. Para esto cabe destacar que la primer fila entera, se saltea y se pone todo en 0(negro) y lo mismo hacemos con la ultima fila. En cuanto el ciclo que se realiza, lo que hacemos es tener un bucle anidado, que simulan el recorrido de de la imagen en sobre cada fila y, para cada fila, sobre las columna. Dentro del bucle de las columnas, se aplica la convolución sobre x y sobre y, como se explica a continuación.

- Mascara X: Para esta mascara, lo que hacemos es en en donde estamos parado, que es el valor centrar del primer pixel a calcular, tomamos la los 16 pixeles de la fila de arriba, la actual y la de abajo, desplazados 1 a la izquierda, ya que ese dato es necesario para calcular los pixeles. Una vez que se tienen estas 3 lineas cargadas de 16 bytes cada una, se pasan a 6 lineas de 8 words cada una, para así poder hacer los cálculos sin perder precisión. De esta forma, y para reducir significativamente los accesos a memoria, utilizamos 6 de los 8 registros xmm para guardar datos, con lo cual debemos adaptar el algoritmo para que se puedan hacer las cuentas con dos registros. Esta mascara esta dividida en dos partes

  - Los 8 primeros pixeles: para poder calcular estos pixeles lo que hicimos es, utilizando los tres registros xmm que contienen estos pixeles, primero hacemos las tres restas, que esto es restar los tres registros así como están (ie. $res = -a1 - a2 - a3$) y luego, para poder realizar las sumas, despasamos cada registro en dos posiciones, poniendo donde estaba el primer pixel, el tercero, el segundo el cuarto, etc. De esta manera, quedan las ultimas dos posiciones libres, las cuales se completan con los dos primeros pixeles de la parte alta. Una vez que se tiene esto, se suman al resultando anterior.
  - Los 8 últimos pixeles: de estos pixeles los que vamos a calcular son las primeras 6 posiciones, para esto primero hacemos la resta igual con en el item anterior, y después lo que hacemos es desplazar cada registro dos pixeles, para poder hacer la suma correspondiente. Si es como obtenemos 6 valores calculados.

  Por ultimo lo que hacemos es poner todos estos pixeles un un solo registro, saturando a byte sin signo, y por ultimo moverlos a la posición correspondiente en la imagen destino.

- Máscara Y: Para esta mascara, solamente necesitamos la linea de arriba y la de abajo del pixel actual. Si que cargamos las dos lineas y dividimos en 4 de 8 words cada una. Nuevamente dividimos esta parte en dos

  - Los 8 primeros pixeles: Lo que hacemos acá es sumar y restar los registros correspondientes, 3 veces pero en cada paso vamos desplazando cada registro para sacar el pixel en la primer posición, y poniendo en la ultima posición el primer pixel del registro que contiene a los 8 posteriores.
  - Los 8 últimos pixeles: en este paso se realiza los mimos pasos que en los primeros 8, con la salvedad que no se rellena en cada desplazamiento, ya que solo necesitamos los primeros 6 pixeles.

  Una vez hecho esto, se procede a unir los resultados, saturando sin signo para que queden valores entre 0 y 255. Para terminar, lo que se hacer es levantar los valores guardaos con la mascara x, sumarlos al actual y volverlos a guardar.

Ya habiendo calculado estos pixeles, lo que se hace es avanzar el iterador de la columna unas 14 posiciones, e iterar hasta que nos pasemos del ancho de la imagen.

2.2.4. Operador de Sobel

Matrices de Sobel en x e y

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

En esta parte no vamos a detallar mucho, ya que es muy similar a la forma de calcular Prewitt, las únicas diferencias a destacar son, que para poder ver si se debe hacer en realizar la mascara de X o la de Y, se va preguntando, basado en los parámetros XOrder e YOrder, si hay que entrar en ese calculo, o si hay que saltarlo. Y por otro lado, la otra modificación es que se suma o resta dos veces (en lugar de una) en los lugares correspondiente para que cumpla con la matriz de convolución correspondiente. Salvando estas diferencias, el método es casi igual al de Prewitt y es por eso que no se entra en detalle.

2.2.5. Operador de Frei-Chen

Frei-Chen en x e y.

$$\begin{pmatrix} -1 & 0 & 1 \\ -\sqrt{2} & 0 & \sqrt{2} \\ -1 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} -1 & -\sqrt{2} & -1 \\ 0 & 0 & 0 \\ 1 & \sqrt{2} & 1 \end{pmatrix}$$

3. Discusión

Tratándose de implementaciones en lenguaje en ensamblador, uno de las cuestiones que más tuvimos en cuenta y analizamos fue, por supuesto, la eficiencia de los algoritmos. Como ya dijimos, lo que hicimos fue medir la cantidad de clocks de procesador que cada función (incluso la de OpenCV y la que escribimos en C) consumen. Esto se logró mediante invocaciones desde C a la función `rdtsc` de lenguaje ensamblador.

El resultado que obtuvimos en ese sentido fue, como esperábamos, que nuestras implementaciones en ensamblador no alcanzaron la rapidez de la implementación de OpenCV, y que la implementación hecha en C fue aún más lenta. Los siguientes gráficos reflejan estas diferencias para imágenes de distintos tamaños.[1]

	Roberts	Prewitt	Sobel
Media	12618185	27276745	30886607
Desvío	1511854	2448652	2598392
Max	17743236	33694296	37954836
Min	11549460	25707576	28905552

	OpenCV	C	ASM
Media	26412756	65944312	30886607
Desvío	10563470	4921205	2598392
Max	68276028	80518500	37954836
Min	21301152	63370956	28905552

La diferencia de eficiencia entre la implementación en C y las hechas en ensamblador tiene un motivo claro y conocido. Por naturaleza, los lenguajes de programación de más alto nivel nos abstraen de muchas cuestiones de implementación, pero a la vez nos quitan control sobre esa implementación. Mientras en ensamblador usamos directamente los registros del procesador (memoria más rápida de la máquina) en C éstos se usan de manera interna y el programador trabaja con datos en memoria RAM.

Esta razón bastaría para justificar las diferencias de rendimiento, pero no es la única: además, el sólo hecho de estar programando en un lenguaje de nivel superior hace que el programador ponga menos atención en los detalles de eficiencia y más en otros aspectos como la legibilidad del código o su reutilizabilidad.

En cambio, el motivo de la diferencia de rendimiento entre nuestras implementaciones en ensamblador y la de OpenCV resulta menos evidente. De

[1] *Las ejecuciones fueron realizadas en una PC: MSI Wind U100 con procesador Intel Atom N270 - 1Gb de RAM y bajo el S.O. Ubuntu Netbook Remix 9.04*

hecho no estudiamos el código que utiliza OpenCV. Sabemos, sin embargo, que nuestras implementaciones no aprovechan al máximo las posibilidades de los procesadores, a diferencia de las de OpenCV que seguramente utilizan capacidades más avanzadas y/o específicas de los mismos como por ejemplo el uso de MMX, SSE, etc..

Un ejemplo que se nos ocurre es la saturación: nosotros, para cada píxel, verificamos si el resultado está por debajo de 0 o por encima de 255 para ajustarlo. Estos chequeos y ajustes consumen un tiempo significativo de ejecución; es claro que usando aritmética saturada nativa del procesador se mejoraría mucho la performance.

Otra característica que imaginamos se podría aprovechar es la paralelización: al tratarse de cálculos sencillos y repetitivos podrían venir bien las instrucciones de datos empaquetados.

También resulta interesante observar en qué medida se aceleró el algoritmo cuando eliminamos los accesos a memoria innecesarios que realizábamos al utilizar la pila en lugar de sólo los registros. El gráfico que sigue compara el rendimiento de la primera implementación en ensamblador que hicimos (que aplicaba Roberts en X usando la pila) con la definitiva (que minimiza los accesos a memoria y aplica Roberts en ambas direcciones). Es notorio que, aunque la primera implementación realiza menos cálculos pues aplica una sola matriz en lugar de dos, sea tanto menos eficiente por no aprovechar al máximo los registros del procesador.

	"Reg"	"Push"
Media	12618185	14692721
Desvío	1511854	1105330
Max	68276028	19269852
Min	11549460	13878600

4. Conclusiones

Con la realización del presente trabajo pudimos apreciar la mejora que se tiene al implementar un algoritmo con instrucciones del tipo SIMD.

De acuerdo a la experimentación realizada, pudimos observar que en promedio la mejora de utilizar SIMD es de alrededor de un XXXXX % en promedio para las implementaciones de los operadores de Roberts, Prewitt y Sobel.

Si bien los resultados obtenidos son óptimos es evidente la limitación que en principio tienen estas instrucciones ya que como se dijo en la introducción, no todos los algoritmos son paralelizables y tratables bajo el concepto de la filosofía SIMD.

A pesar de los beneficios de este tipo de instrucciones, el grupo encontró particular dificultad a la hora de *debuggear* el programa ya que por un lado, no logramos que el compilador genere los símbolos de *debugging* correctamente a pesar de estar configurado para esto y además,la programación con MMX y SSE hace el trabajo un poco más dificil ya que en general no se acostumbra a pensar en paralelo cuando uno programa con lenguajes de nivel más alto. Vale mencionar también que el código se vuelve un poco más críptico cuando se emplean este tipo de técnicas.

Creemos que es fundamental ahondar en este tipo de técnicas de programación en paralelo ya que si bien su uso en principio puede estar restringido a un conjunto bastante acotado de problemas, estamos convencidos que es positivo hacer hincapie en este tema ya que la manera de pensar este tipo de problemas es distinta a lo habitual y consecuentemente, es un recurso nuevo y que puede ser muy útil para encarar diversos problemas.

Como discutimos anteriormente, verificamos el hecho esperado de que nuestras implementaciones no alcanzarían la rapidez de las de OpenCV.

5. Apéndice A: Manual de usuario

El programa realizado permite aplicar diferentes implementaciones de detección de bordes a imágenes. Tanto el nombre de las imagen fuente como los operadores se indican por línea de comandos de la siguiente manera (suponiendo que el ejecutable está en .exe/bordes"):

```
$ exe/bordes {fuente} {destino} {operadores}
```

Donde:

* `fuente`: es la imagen de origen; puede tener cualquier formato soportado por la función cvLoadImage de OpenCV; los formatos más comunes están soportados;

* `destino`: es el nombre de las imágenes de salida (generadas por el programa) SIN INCLUIR LA EXTENSIÓN; los archivos generados tendrán ese nombre más un sufijo que indica el operador utilizado, más la extensión; la extensión, el formato y el tamaño son obtenidos de la imagen de entrada;

* `operadores`: una o más claves separadas por espacio; cada clave representa una implementación particular de detección de bordes; las claves posibles y sus significados son los que siguen:

CLAVE	OPERADOR	DIRECCION	IMPLEMENTACIÓN	SUFIJO
r1	Roberts	XY	Ensamblador	_asm_roberts
r2	Prewitt	XY	""	_asm_prewitt
r3	Sobel	X	""	_asm_sobelX
r4	Sobel	Y	""	_asm_sobelY
r5	Sobel	XY	""	_asm_sobelXY
cv3	Sobel	X	OpenCV	_cv_sobelX
cv4	Sobel	Y	""	_cv_sobelY
cv5	Sobel	XY	""	_cv_sobelXY
c3	Sobel	X	C	_c_sobelX
c4	Sobel	Y	""	_c_sobelY
c5	Sobel	XY	""	_c_sobelXY
push	Roberts	X	Ensamblador usando pila	_asm_roberts(push)
byn	Escala de grises			_byn

Por ejemplo, si tenemos el ejecutable en `"exe/bordes"` y una imagen en `"pics/lena.bmp"`, llamando al programa como

```
$ exe/bordes pics/lena.bmp pics/lena byn r1 r2 r3 r4 r5 cv3 cv4 cv5 c3 c4 c5 push
```

se aplicarán todas las implementaciones disponibles y se generarán los archivos `"pics/lena_asm_roberts.bmp"`, `"pics/lena_asm_prewitt.bmp"`, etc... Además el programa mostrará por salida estándar la cantidad aproximada de clocks de procesador insumida por cada implementación.