SUB

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

En la primera entrega de este trabajo hicimos un programa que toma imágenes como entrada y genera nuevas imágenes efectuando sobre la primera un proceso conocido como *convolución*. La convolución es una técnica de detección de bordes, y el resultado es una imagen oscura del mismo tamaño que la original, con zonas claras donde la imagen fuente tiene "saltos de intesidad".

La detección de bordes en imágenes es una técnica utilizada tanto para compresión de archivos como para lograr efectos gráficos. La convolución consiste en recorrer los pixeles de la imagen (en nuestro caso en escala de grises) y aplicar 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.

Existen muchas matrices utilizadas para la convolución, con diferentes dimensiones y coeficientes. En la primera entrega nuestro trabajo fue realizar implementaciones de convolución para las matrices de Roberts (de 2x2), y de Prewitt y Sobel (de 3x3). La implementación fue realizada en assembler, para la arquitectura IA-32 básica, es decir usando los registros de propósito general.

Para esta segunda y última parte la tarea consiste en realizar nuevas implementaciones aprovechando las nuevas tecnología SIMD: las extensiones MMX, SSE, SSE2 y SSE3.

Estas extensiones se fueron agregando a los procesadores Intel de forma gradual (y en ese orden), por lo que algunas computadoras pueden no soportarlas todas. Todas ellas consisten en sets de instrucciones y hardware especializados para el procesamiento en paralelo. Los 8 registros **MMX** de 64 bits y los 8 **XMM** de 128 bits se comportan como vectores de varios datos ("empaquetados") que pueden operarse "verticalmente", es decir que entre dos registros se realiza cada operación elemento a elemento.

Esta tecnología es especialmente útil cuando se necesitan realizar cálculos sencillos (pues disponemos de pocos registros) de manera repetitiva a lo largo de grandes cantidades de datos "alineados". Esto es ideal (y está pensado para) el tratamiento de señales digitales como imágenes y sonidos.

En esta segunda parte rehicimos las implementaciones del primer trabajo y agregamos una nueva: el operador de Frei-chen, que requiere a diferencia de los otros un procesamiento en punto flotante. En todos los casos lo hicimos con los 8 registros XMM presentes desde la extensión SSE.

A continuación describiremos el trabajo realizado sin profundizar en las cuestiones ya analizadas en la parte anterior, sino enfocándonos principalmente en el nuevo mundo del cálculo en paralelo.

2. Desarrollo

2.1. El programa

El programa realizado es en realidad una extensión del presentado anteriormente. Permite aplicar los mismos operadores de derivación que aquella versión: de **Roberts**, **Prewitt** y **Sobel** realizados con la arquitectura de propósito general y la versión de *OpenCV* del operador de **Sobel**, además de las nuevas implementaciones agregadas de **Roberts**, **Prewitt**, **Sobel** y **Frei-chen** hechas con la tecnología *SIMD*.

Para cada operador solicitado el programa aplica la matriz correspondiente en X, luego en Y, y finalmente suma los resultados. La imagen resultante es grabada y se muestra por pantalla la cantidad aproximada de clocks de procesador insumida. Para el caso de Sobel, también permite aplicar convolución solamente en X o solamente en Y, tanto en la implementacióne básica como en la optimizada.

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.

2.2. Implementación

Dado que este trabajo consiste en una optimización de un algoritmo ya implementado, cuando comenzamos a hacerlo teníamos bastante claro *qué* teníamos que hacer. Lo que quedaba por averiguar era *cómo* hacerlo; nos referimos por supuesto al diseño de los algoritmos de cómputo en paralelo.

Tal es así que la tarea en un principio parecía sencilla: esta vez no nos trabaríamos con todas las incertidumbres y errores surgidos en el trabajo anterior, sólo tendríamos que reproducir los mismos algoritmos levantando los datos de a "paquetes" y realizando sobre ellos más o menos las mismas operaciones que antes.

Poco a poco se fue haciendo evidente, sin embargo, que esto no era cierto; la programación con el modelo SIMD requiere un enfoque completamente distinto y tiene sus propias complicaciones. Al desarrollar un algoritmo de este tipo, se presentan muchas posibilidades de diseño, limitaciones y además requiere un trabajo adicional para acomodar los datos: empaquetarlos, desempaquetarlos, mezclarlos, etc...

El *qué*, aquello que conocíamos, era solamente el algoritimo a alto nivel. Lo que había que hacer era recorrer la imagen fuente y, por cada pixel, hallar el "producto interno" del entorno del punto con la matriz de convolución en X. Es decir, calcular la suma de productos elemento a elemento de la matriz contra una submatriz del dibujo del mismo tamaño. Ese resultado daría positivo o negativo dependiendo de la "dirección" del borde y de la matriz usada; inmediatamente lo saturaríamos a 0 quedándonos solamente con los bordes en una dirección y sentido.

Luego haríamos exactamente lo mismo con la matriz de convolución en Y. Una vez hecho esto tenemos dos valores no negativos que representan bordes en dos direcciones perpendiculares; entonces los sumamos y, saturando esa suma a 255 la escribiríamos en la imagen destino como un pixel en la misma posición en la que estábamos parado en la imagen fuente.

2.2.1. Programación con SIMD

En esta segunda parte asumimos que las imágenes tienen un ancho múltiplo de 16. Esto implica, por un lado, que ya no es necesario calcular el *widthstep* como hacíamos antes. Como *OpenCV* deja cada fila de pixeles alineada a 4 bytes en memoria, teníamos que calcular el mínimo múltiplo de 4 mayor o igual al ancho de la imagen. Ahora en cambio el width (ancho) y el widthstep coinciden.

Además esto nos evita accesos a memoria, pues en nuestras anteriores implementaciones guardábamos el widthstep en un registro para movernos por los mapas de bits pero accedíamos a memoria para consultar el width para saber cuándo terminar el bucle.

El hecho de que el ancho sea múltiplo de 16 también puede aprovecharse para operar con los registros **XMM**, donde caben justamente 16 pixeles (cada uno de un byte). Por ejemplo podemos dividir cada fila en bloques contiguos de 16 pixeles, levantando cada bloque de una vez hacia un registro **XMM**.

Esto es justamente lo que hicimos en el caso de **Roberts** y en el de **Frei-chen** (en este caso lo hicimos de a 8). Pero, como veremos más adelante, no es necesario basarse en esa condición: las implementaciones de **Prewitt** y **Sobel** barren cada fila de a 14 lugares, y están pensados para imágenes de cualquier ancho.

Para ello usamos la instrucción `MOVQ` en el caso de Frei-chen (para levantar cada bloque de 8 pixeles en la parte baja de un registro **XMM**) y la instrucción `MOVDQU` en los demás casos (para levantar bloques de 16 pixeles en registros **XMM**).

Una observación importante es que para operar con esos bytes sin signo (valores de 0 a 255) necesitamos extenderles el signo. Si operáramos usando bytes no podríamos representar valores negativos ni mayores a 255, lo cual es necesario para aplicar las matrices de convolución.

Para ello nos valimos de las instrucciones `PUNPCKHBW` y `PUNPCKLBW` que permiten desempaquetar la parte alta y baja respectivamente de un **XMM** transformando los 8 bytes en 16 words (completando con ceros). Esto se hace usando como operando fuente un registro con ceros.

Luego cada implementación realiza con esas words el procesamiento necesario, explicado en las próximas secciones, tratándolas como enteros **signados**. La excepción a esto es **Frei-chen**, que debe volver a desempaquetar los datos en dwords y castearlos a flotantes de precisión sencilla. El casteo se realiza con la instrucción `CVTDQ2PS`.

Luego del procesamiento correspondiente a la aplicación de las matrices, llegamos siempre a tener los valores de las derivadas en registros **XMM** como

8 words signadas. Para poder volcar los valores necesitamos saturarlos a bytes sin signo (a 0 y a 255) y además empaquetarlos de a 16 bytes en registros **XMM**. Todo esto se logra con la instrucción `PACKUSWB`, que justamente toma dos registros **XMM** con 8 words signadas cada uno y las empaqueta en un solo registro como bytes sin signo con saturación. Solo falta aplicar `MOVDQU` o `MOVQ` nuevamente para escribir los valores definitivos en el destino.

2.2.2. Operador de Roberts

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

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

Este operador se diferencia de los demás por el hecho de tener matrices de 2x2, con sólo dos valores significativos (no ceros). Con registros de propósito general esto no parecía marcar una gran diferencia, pues las operaciones se hacían secuencialmente y no nos afectaba mucho la cantidad que hubiera que efectuar.

Pero ahora que estamos implementándolo con paralelización, este operador resulta más simple que los demás.

En primer lugar, como el ancho de la imagen es múltiplo de 16, se optó por cargar los datos en columnas de a 16, que como se mencionó antes es el tamaño de los registros `xmm`. Entonces se tienen 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. Operadores de Prewitt y de Sobel

Matrices de Prewitt:

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

Matrices de Sobel:

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

La implementaciones de estos operadores son análogas entre sí, por eso las describimos juntas aquí.

Como podemos ver estos operadores tienen matrices de $3x3$ de números enteros. Esto presenta un panorama más complejo que el de Roberts. En aquel caso, bastaba con levantar dos filas de 16 pixeles (desplazadas una posición entre sí) para hallar el resultado de una fila completa también de 16 pixeles.

Ahora la situación es distinta, pues 3 filas de 16 pixeles alcanzan para calcular los valores de los 14 pixeles centrales. En un principio se nos ocurrieron dos opciones: una sería levantar bloques más grandes (de 18) tomando un pixel adicional en cada costado, y acomodarlos adecuadamente en los registros **XMM** para poder calcular el resultado no de 14 sino de 16 pixeles por iteración. Así vamos a poder completar de a 16 pixeles en la imagen destino.

Sin embargo optamos en este caso por la segunda opción: levantar los datos de a 16 y movernos de a 14. Esto obviamente no tiene en cuenta la restricción de que el ancho es múltiplo de 16. Por lo tanto cuando avancemos de a 14 es posible que en algún momento nos "pasemos de largo".

Al principio creímos que las últimas columnas (menos de 14) tenían que ser calculadas fuera del ciclo, porque si intentamos acceder a memoria más allá de

los límites de las imágenes podríamos estar escribiendo basura en algunos lugares del resultado e incluso en lugares de la memoria que no nos corresponden.

Pero luego nos dimos cuenta de que no había tantos problemas: donde termina cada fila, si nos pasamos escribiendo simplemente vamos a empezar a escribir datos en la fila siguiente (que luego pisaremos con la información válida cuando la hayamos calculado); por último, para la última fila que escribimos tampoco hay problema, pues recordemos que la última fila escrita es la anteúltima del dibujo.

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. Para cada las demás filas se realiza lo siguiente:

- 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 máscara está 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, desplazamos 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 guardados 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.

Como se ve, una vez que tenemos la derivada en X la escribimos en el destino como si ya hubiéramos terminado, y una vez que tenemos la derivada en Y levantamos aquellos valores y los sumamos. En el caso de Sobel, que permite aplicar convolución solamente en X y solamente en Y, se realiza sólo uno de los pasos.

Si bien parecería más eficiente guardarse las derivadas en X no en memoria sino en algún XMM hasta obtener las derivadas en Y, a la hora de hacer esas implementaciones no nos quedó ningún registro disponible a tal fin de manera que resolvimos dejarlo como se explicó.

2.2.4. Operador de Frei-Chen

Frei-Chen en x e y.

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

Frei-chen es el operador que se agregó en esta segunda parte del trabajo. Además es el único que requiere procesamiento en punto flotante pues algunos de los coeficientes de sus matrices son números reales (raíces de dos). Con los registros de propósito general este procesamiento habría sido un tanto complicado.

Usando los registros XMM (de 128 bits) como cuatro valores de punto flotante de 32 bits (precisión simple) pudimos hacer una implementación con procesamiento paralelo de forma tan simple como lo hicimos con números enteros. La única diferencia importante es que en este caso entran 4 valores por registro XMM (con enteros usamos 8 valores de 16 bits).

Las matrices de **Frei-chen** son muy parecidas a las otras de 3x3 que usamos (**Prewitt** y **Sobel**), de manera que los algoritmos pueden pensarse de una forma parecida pese a la diferencia en la cantidad de valores empaquetados.

Siguiendo el mismo esquema podríamos cargar tres filas de 8 pixeles (en lugar de 16) usando dos registros XMM para cada fila. Con los valores de esos

3x8 pixeles podríamos calcular los valores de 6 pixeles (en lugar de 14), que son los 6 centrales de la fila del medio.

Sin embargo en este punto tomamos otro enfoque, para poder recorrer las imágenes realmente de a 8 y no de a 6. La idea es la siguiente: numeremos las columnas desde 0 y supongamos que en la primera iteración levantamos desde la columna 1 hasta la 8 (en lugar desde la 0 hasta la 7 como sería más típico). Estos datos nos van a permitir calcular las derivadas para 6 pixeles, desde la columna 2 hasta la 7. El pixel de la columna 8 no lo podemos calcular pues para ello necesitamos información de las columnas 7, 8 y 9 (nos falta la 9) así como tampoco podemos calcular el de la columna 9 (pues nos faltan los datos de la fila 9 y la 10).

Lo que pensamos fue, en esa iteración (además de escribir los datos de los 6 pixeles resueltos) guardarnos de alguna manera la información correspondiente a las dos últimas columnas en los registros **XMM**. Entonces, en el siguiente paso levantaremos (moviéndonos de a 8 posiciones) las columnas 9-16, pero además tendremos información de las columnas 7 y 8 (del paso anterior). En definitiva, vamos a poder resolver 8 puntos, aquellos dos que nos habían quedado "colgados" (el 8 y el 9) más los 6 pixeles centrales del bloque levantado (10-15).

Usando este mecanismo es claro que podemos efectivamente avanzar de a 8 filas. Ahora bien, ¿cómo nos guardamos la información de esas columnas? Usamos un mecanismo al que llamamos *comprimir columnas*. Mirando la matriz en X, observamos que podemos operar verticalmente, calculando paralelamente para cada columna el valor de la primera fila más raíz de dos por la segunda más la tercera. Como se ve, estos son los coeficientes por los que necesitamos multiplicar, más allá del signo.

En el código fuente y en adelelante nos referimos como (n) a la columna n comprimida. Al resultado buscado, es decir a la derivada en la columna i, sea en X o en Y, lo llamamos $[i]$. Una vez que las columnas están comprimidas de esta manera, la derivada de un pixel se halla como una resta de columnas comprimidas.

$$[i] = (i+1) - (i)$$

Asíque acomodando los datos de manera adecuada resulta sencillo hallar las derivadas en X a partir de las columnas así comprimidas. Para el caso de Y, la compresión por columnas es distinta: en este caso ni siquiera interesa el valor de la fila del medio; el valor comprimido es simplemente la tercera fila menos la primera.

Es decir que nos quedamos con la diferencia entre las filas sin importar cuál vamos a multiplicar luego por raíz de dos. El cálculo de cada derivada se reduce a...

$$[i] = (i-1) + \sqrt{2}(i) + (i+1)$$

La idea de todo esto es que de una iteración a la otra queremos guardarnos los valores comprimidos de las dos últimas columnas, tanto en X como en Y. Para esto nos alcanza un único registro **XMM** (`xmm5` en nuestro código) pues se trata de cuatro valores.

En resumen, entonces, el método es el siguiente: en cada paso levantamos 8 columnas de datos (en 3 filas por supuesto); desde la columna $8 * i + 1$ hasta la $8 * i + 8$. Numerémoslas desde 1 hasta 8 sin importar si se trata del principio de la fila. Inmediatamente comprimimos en X esas columnas obteniendo $(1), (2), ..., (8)$. Además sabemos que en `xmm5` tenemos salvados los valores comprimidos de dos columnas anteriores del paso anterior; las llamamos (-1) y (0). Recordemos que las tenemos comprimidas tanto en X como en Y.

Con esas 10 columnas comprimidas vamos a poder encontrar 8 derivadas en X definitivas $([0], [1], ..., [7])$. Estos valores parciales no los volcamos en la imagen destino como hacíamos en el caso de Prewitt y Sobel, si no que los salvamos en un XMM como 8 enteros de 16 bits (con signo) (`xmm4` en nuestro código). Pero además es menester salvar en `xmm5` los valores (7) y (8), pues son quienes en la próxima iteración actuarán como (-1) y (0) para X. Entonces los salvamos sin perder los valores allí reservados para Y del paso anterior.

Con Y hacemos lo mismo, y finalmente también salvamos las columnas comprimidas en Y (7) y (8) para el siguiente paso. Una vez que tenemos las derivadas en X y en Y en sendos registros, debemos saturarlas a 0 para evitar cancelaciones. Esto lo logramos con `PMAXSW` contra un registro de ceros. Luego hacemos la suma y, como se explicó anteriormente, las instrucciones `PACKUSWB` y `MOVQ` hacen el resto.

3. Discusión

Lo primero que queremos destacar es todo el trabajo adicional que requiere el procesamiento con **SIMD**. Como vimos en varios casos se abrieron múltiples posibilidades, no quedando claro a veces cuál es la más conveniente.

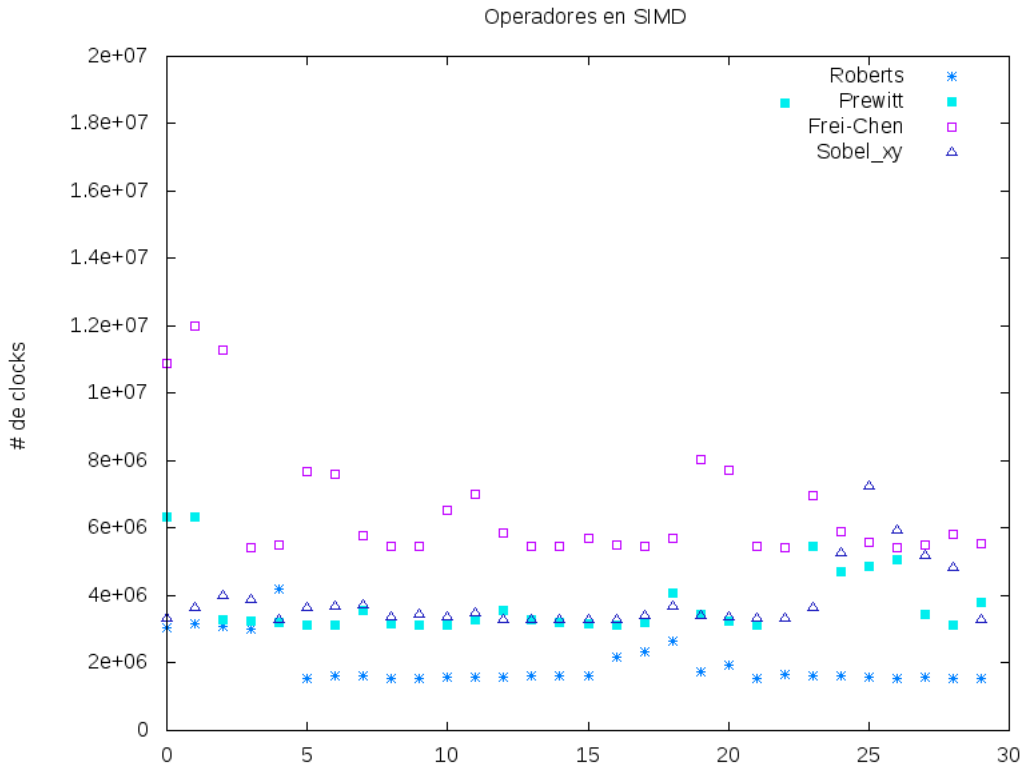
En numerosas ocasiones nos topamos con problemas; por ejemplo, a veces parecen no alcanzarnos los registros; o en otros casos parece que la falta de una instrucción en el set nos obliga a hacer más cuentas o movimientos de los necesarios.

Es notorio también que en esta arquitectura, pequeños cambios en lo que se quiere lograr pueden repercutir en el algoritmo de manera muy fuerte, ya que los datos al estar empaquetados de alguna manera están "atados". Por ejemplo, si a las matrices de 3x3 les quisiéramos poner un 1 en la posición central (aunque esto no tenga mucho sentido para la detección de bordes) el algoritmo debería transformarse completamente, ya que los registros se van a agotar y la forma de acomodar los valores no va a poder ser la misma.

Otra cuestión que notamos es que haciendo este tipo de códigos, cuyo diseño se basa fuertemente en la eficiencia, uno corre el riesgo de querer ir demasiado lejos; es decir, uno empieza a tratar de sacar el máximo jugo a cada operación no conformándose nunca.

En este aspecto creemos que es importante enfocarse en la optimización "a gran escala" (es decir cómo diseñar las cosas para operar aprovechando la paralelización) y no preocuparse tanto por hacer "la mímina cantidad de operaciones". Lo que queremos decir es que, quizás por querer evitar un par de operaciones, se puede caer en complicaciones importantes, cuando en realidad el grueso del trabajo ya está paralelizado y es terriblemente eficiente.

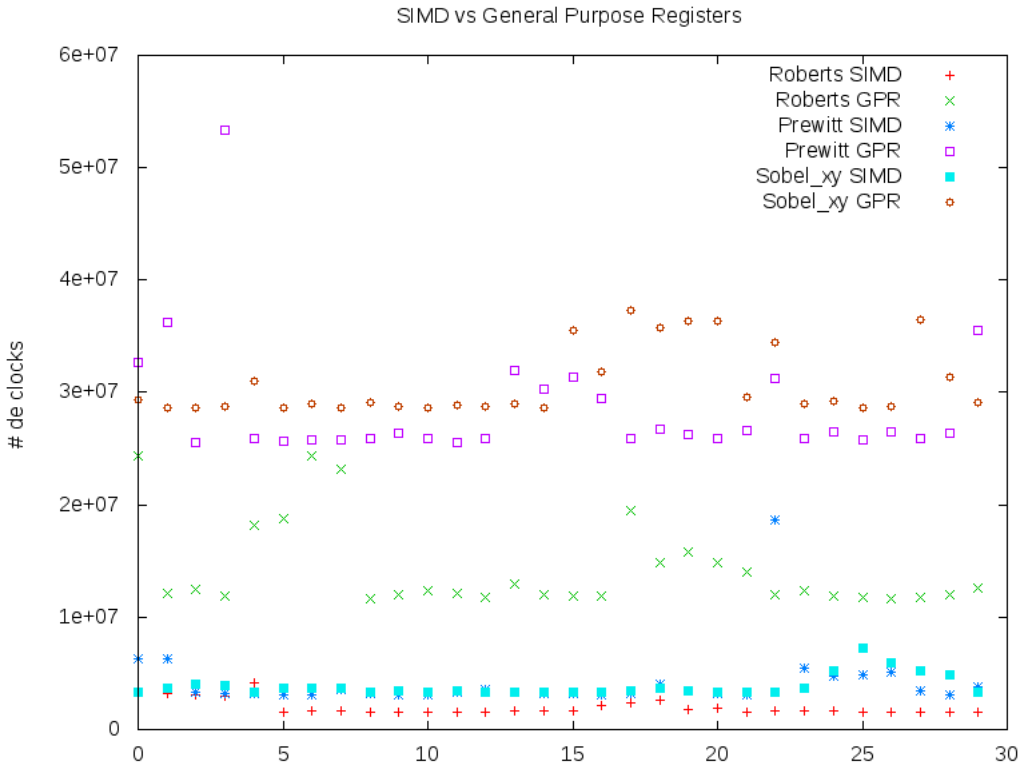
3.1. Comparación entre operadores SIMD



Como se ve en este primer gráfico, la implementación más veloz es la de Roberts. Esto por supuesto se explica por la simplicidad de ese operador, sobre lo que hablamos anteriormente. En el otro extremo, la que más tiempo toma es la implementación de Frei-chen, lo que se debe (al menos en parte) a que opera en punto flotante, donde todas las operaciones son por naturaleza más costosas.

En ese sentido habríamos esperado encontrarnos con más diferencia aún. Nos parece notable que el hardware especializado permita con punto flotante hacer cálculos no mucho más ineficientemente que con enteros.

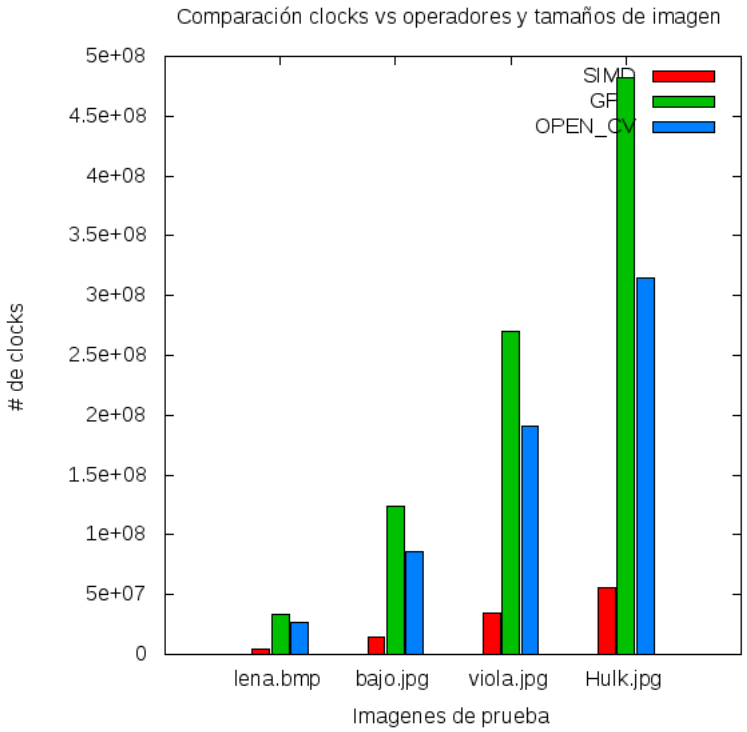
3.2. SIMD vs. GPR



Como vemos aquí el procesamiento **SIMD** realmente es **otra cosa**. Las implementaciones que logramos son algo así como 10 veces más eficientes que las anteriores (¡que ya de por sí corrían en un parpadeo!).

Hacer este tipo de funciones con estas tecnologías realmente vale la pena, pues, pese al esfuerzo adicional mencionado, finalmente se logró un resultado equivalente de manera relativamente sencilla, y con un enorme beneficio en relación a la eficiencia. Además, los procesadores actuales de uso general contienen siempre estas extensiones, por lo que la portabilidad en la práctica no suele ser un problema.

3.3. SIMD vs. OpenCV



En este último gráfico en el que comparamos la eficiencia de las nuevas implementaciones de Sobel contra las de **OpenCV**, podemos observar que se superó el rendimiento de esa biblioteca. Esto parecería indicar que las implementaciones de **OpenCV** no usan XMM sino los registros de propósito general.

Las imágenes de este gráfico son todas "múltiplos" en tamaño de `lena.bmp` en el siguiente sentido: `bajo.jpg` tiene exactamente el doble de pixeles (1024x1024), `viola.jpg` tiene 3 × "lena" (1536x1536) y, por último, `hulk.jpg` que es 4 veces "lena" (2048x2048).

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 esta mejora consiste en una eficiencia aproximadamente 10 veces más grande.

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 al estilo SIMD. Sin embargo y como se notó en este trabajo, son ideales para el procesamiento de imágenes. Pareciera que toda transformación que se quiere aplicar a imágenes (o a sonidos) encontrará en SIMD una herramienta poderosa.

Como ya dijimos, pese a la relativa simplicidad con la que se pueden llevar a cabo este tipo de algoritmos en esta arquitectura, queremos mencionar nuevamente que esto requiere un esfuerzo adicional, y en muchos casos trabas importantes. Además, el código resultante es mucho menos declarativo, así como también mucho más sensible a cambios.

Por último queremos observar que no hay motivo para conformarse con los registros **XMM**. De la misma manera, nuevas arquitecturas podrían proveer no 8 sino 16 o 32 registros, no de 128 bits sino de 256 o 1024. En ese caso se manifestará la poca reusabilidad de la paralelización, pues las implementaciones hechas para 8 registros de 128 bits claramente no van a aprovechar los nuevos 16 o 32 registros.

A priori podría pensarse en un esquema en el que el código no esté tan atado a cómo acomodar los datos, sino que tenga sentido en diferentes arquitecturas de cálculo en paralelo, aprovechando en cada caso la cantidad provista de registros y bits. Para ello haría falta trabajar con código de más alto nivel, pensando en qué operaciones se deben paralelizar pero no de a cuántas. Las cuestiones de acomodamiento de datos quedaría en manos de un compilador. Es una posibilidad muy complicada pero también muy interesante, y sólo con el paso del tiempo conoceremos su factibilidad.

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:

```
USO:    ./bordes $src $dest ${lista de operadores}

- $src: nombre del archivo de origen
- $dest: nombre del archivo de salida SIN EXTENSIÓN
- ${lista de operadores}: una o más de las siguientes claves:

         CLAVE | OPERADOR  | DIRECCION | IMPLEMENTACIÓN
        -------+-----------+-----------+----------------
          r1   |  Roberts  |    XY     |  asm + sse
          r2   |  Prewitt  |    XY     |      "
          r3   |   Sobel   |    X      |      "
          r4   |   Sobel   |    Y      |      "
          r5   |   Sobel   |    XY     |      "
          r6   | Frei-chen |    XY     |      "
               |           |           |
          a1   |  Roberts  |    XY     |asm g. purpose
          a2   |  Prewitt  |    XY     |      "
          a3   |   Sobel   |    X      |      "
          a4   |   Sobel   |    Y      |      "
          a5   |   Sobel   |    XY     |      "
               |           |           |
          cv3  |   Sobel   |    X      |    OpenCV
          cv4  |   Sobel   |    Y      |      "
          cv5  |   Sobel   |    XY     |      "
               |           |           |
        c1...c6|   (idem)  |   (idem)  |      C
               |           |           |
          byn  |   Escala de grises    |
               |           |           |
El programa generará una imagen por cada operador solicitado. El
```

```
nombre de los archivos de salida será $dest más un guión bajo más el
operador utilizado más la extensión (obtenida del archivo fuente).
```

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 r6
```

Se aplicará el operador de Frei-Chen en SIMD y se generará el archivo `"pics/lena_r6.bmp"` . Además el programa mostrará por salida estándar la cantidad aproximada de clocks de procesador insumida por cada implementación.