Organización del Computador II

Trabajo Práctico 2

Emilio Almansi
ealmansi@gmail.com

Miguel Duarte
miguelfeliped@gmail.com

Federico Suárez
elgeniofederico@gmail.com

2^{do} cuatrimestre de 2013

Resumen

En este documento se analiza las diferencias en rendimiento encontradas para las implementaciones de algunos filtros de imágenes al utilizar instrucciones SIMD de la arquitectura Intel 64.

Índice

1. Introducción

A lo largo de este documento se realizará un análisis consiso sobre el procesamiento de datos SIMD mediante las intrucciones SSE de la arquitectura AMD-64. El mismo se hará mediante la comparación de diversas implementaciones de 3 filtros (2 de video y uno de imágen).

El procesamiento SIMD consiste en realizar una misma operación sobre varios datos de manera simultánea. Es decir que lo que se logra es paralelismo a nivel de datos. Esta clase procesamiento es ideal para realizar filtros sobre imágenes , porque ahí justamente lo que uno busca es que cada pixel reciba el mismo proceso.

Sin embargo para realizar este análisis es imperioso meterse un poco mas adentro. El comportamiento esperado a priori para estos procesos es una relación inversa entre el nivel de paralelismo y el tiempo consumido. Es decir, al procesar 4 datos a la vez uno esperaría obtener que el proceso tarde 4 veces menos tiempo, procesando 8 datos a la vez 8 veces menos tiempo, etc. Sin embargo esto no siempre ocurre. Durante este trabajo constantemente se va a intentar explicar que desviaciones se produjeron con respecto a este supuesto. Para eso vamos a analizar la arquitectura intel-64, la velocidad de acceso a memoria, el modo en que se usa la caché, el medio en el cuál se ejecutan los programas y los algoritmos utilizados entre otras cosas.

A lo largo del trabajo se va a ir mostrando como el uso de código de ensamblador optimizado para el uso de la tecnología SSE produce programas sumamente eficientes. Sin embargo producir ese código es sumamente trabajoso, mucho mas que usar lenguajes de mas alto nivel como c o c++. Por ese motivo se intentará hacer otro análisis (tal vez algo menos científico pero intentando justificar de la manera mas objetiva posible) de cuando vale la pena y cuando no.

2. Consideraciones generales

Método de trabajo

A la hora de implementar los filtros se tuvieron algunas consideraciones generales para poder explicar los resultados de manera correcta, para manetener la coherencia interna dentro del trabajo y para tener bases firmes en las que basarnos para sacar conclusiones. Siguiendo esta idea todo lo implementado sigue los siguientes criterios:

- Todas las implementaciones de C intentan ser lo mas intuitivas posibles. No se utilizaron optimizaciones demasiado extrañas. Intentan ser una traducción bastante fiel de la descripción en lenguaje natural del enunciado a C. Decimos esto para poder analizar de manera pura la diferencia entre los dos paradigmas. Además a la hora de analizar los archivos objeto creados por los compiladores tener un algorítmo simple ayuda a hacer un análisi mas simple y consistente. Por otra parte da libertad al compilador para meter tanta paralelización como pueda de tal manera que los diferentes compiladores puedan introducir ellos las optimizaciones en lugar de respetar un esquema previo.
- Todas las implementaciones en asm se escribieron intentando minimizar al máximo los accesos a memoria. Una vez dentro del ciclo sólo se accede a memoria para buscar datos y para escribir datos. La razón de esto es que los accesos a memoria son lentos, por lo que en general cuando se busca performance es buena idea evitarlos. Sin embargo en cada implementación se va a analizar si esto fue una decisión acertada o no.
- Todos los códigos de C se compilaron con 2 compiladores distintos:
  - GNU C Compiler (GCC): Se eligió por ser un compilador libre, conocido, popular , sumamente versatil y porque es capaz de realizar una gran gama de optimizacizaciones, probablemente todas aquellas que se pueden realizar sin acceder al micro código de intel.
  - Intel C++ Studio XE (ICC): Este compilador introduce de manera predeterminada código que aprovecha la tecnología SSE. Es decir que de manera predeterminada genera código objeto que realiza paralelismo a nivel de datos. Además realiza optimizaciones de altísima calidad aprovechando los detalles del micro-código.

  Además si bien siempre se compiló indicándole al compilador que use instrucciones SSE4.3 además se hicieron 2 versiones distintas con cada compilador: Una con optimizaciones agresivas y otra sin ellas. Para la primera usó el flag -Ohard, mientras que para la segunda se dejó el comportamiento predeterminado.

Método de testeo y medición

A la hora de realizar las mediciones de tiempo se intentó armar el ambiente mas ameno posible. Para esto antes de realizar los tests se mataron todas las aplicaciones no vitales para el sistema operativo, incluída la interfaz gráfica, acceso a internet y toda esa clase de cosas (init 1, kill -9 -1). Además se desconectaro todos los periféricos innecesarios.

Las mediciones se repitieron entre 50 y 100 veces (según el filtro). Estos números se decidieron en base a prueba y error. Eliminando los valor atípicos de esa cantidad de valores hacia arriba la variación de la media era despreciable (menor al 0.5)

3. Filtro de color

3.1. Descripción del filtro

El filtro de color es una transformación sobre imágenes a color que tiene el efecto de decolorizar o pasar a escala de grises todos los píxeles de la entrada cuyo color no esté dentro de un rango de colores especificado. En la figura 1 se observa un ejemplo de funcionamiento típico.

Figura 1: Imagen antes y después de aplicar el filtro de color con color principal rojo.

La forma en la cual se especifica el rango de colores que deberá permanecer inmutado es mediante la elección de un color principal, cuya codificación RGB se denota con (rc, gc, bc), y un parámetro umbral *threshold*. Una vez determinados estos valores, un píxel de la imagen fuente será actualizado por el filtro únicamente si cumple:

$$||(r, g, b) - (rc, gc, bc)||_2 > threshold \tag{1}$$

donde (r, g, b) es la codificación en RGB del píxel. En particular, de cumplirse esta condición, los tres canales se actualizan de la siguiente forma:

$$r' = b' = g' = \frac{r + g + b}{3}$$

De esta última expresión se desprende que el color de los píxeles alterados pasa a estar en la escala de grises, ya que los tres canales toman igual valor. Como una observación adicional, queda claro mediante esta especificación que el filtro actúa de forma localizada en sobre cada píxel; su suceptibilidad a ser modificado y su nuevo valor dependen únicamente de su propio valor, y no del de sus vecinos.

3.2. Implementación en lenguaje C y lenguaje ensamblador

La implementación en C del filtro se realizó de la forma más sencilla e intuitiva posible; mediante un ciclo que visita una vez a cada píxel de la imagen, de izquierda a derecha y de arriba a abajo, evaluando la condición y modificando su valor de ser necesario. Como única optimización elemental, se modificó la condición (1) por la siguiente condición equivalente:

$$||(r, g, b) - (rc, gc, bc)||_2^2 = (r - rc)^2 + (g - gc)^2 + (b - bc)^2 > threshold^2$$

La modificación evita el cálculo de una raíz cuadrada, sin incurrir en el riesgo de exceder el rango del tipo de datos utilizado ya que el máximo *threshold* que no hace a la condición trivialmente falsa es $\sqrt{195075} \approx 441$ (el valor máximo que puede tomar la expresión de la izquierda es $255^2 + 255^2 + 255^2 = 195075$), por lo que cualquier valor mayor se puede reducir a 442.

La implementación en lenguaje ensamblador mantiene esencialmente el mismo procedimiento, con la salvedad de que fue adaptado para procesar cuatro píxeles en simultáneo mediante el uso de operaciones SIMD. Se recorre la matriz en el mismo órden descripto previamente, realizando lecturas de 16 bytes por iteración (equivalente a 5 píxeles y un byte sobrante).

La necesidad de limitar el procesamiento simultáneo a cuatro píxeles se desprende del hecho de que, como se dijo antes, la expresión que mide la distancia entre un píxel y el color principal puede tomar valores en el rango

$0, ... 195075$. Este último valor no cabe en un entero de 16 bits, precisándose un **double word** para almacenar ese resultado temporal; esto implica un total de hasta cuatro valores de distancia en un registro XMM. Por esta razón, además sería similar trabajar con el tipo de datos **float** ya que también caben hasta cuatro por registro de 128 bits.

De esta forma, el procedimiento realizado en cada iteración consiste en (para cada uno de los cuatro píxeles en simultáneo) comparar el valor de la distancia con el *threshold*, y calcular el promedio de los tres canales, actualizando luego sus valores según la siguiente expresión informal:

$$\text{valor\_original} \wedge \neg\text{cumple\_condicion} + \text{promedio} \wedge \text{cumple\_condicion}$$

Esto permite expresar el equivalente a una expresión del tipo **if-then-else**, en el lenguaje del procesamiento simultáneo. El cómputo de los flags con el resultado de las comparaciones y del promedio se puede describir mediante el pseudo-código de la figura **??**.

```
prom := [0,0,0,0]                              // enteros doubleword
dist := [0,0,0,0]

desempaquetar el rojo de cada pixel a un double word
rojos := [r4, r3, r2, r1]

prom += rojos
rojos -= [rc, rc, rc, rc]
rojos *= rojos
dist += rojos

repetir para verdes
repetir para azules

prom := prom / 3
dist := dist > [threshold, threshold, threshold, threshold]
empaquetar prom y dist a formato pixeles

resultado := datos AND NOT dist
resultado += prom AND dist
```

Figura 2: Pseudo-código de la implementación en lenguaje ensablador del filtro color.

Para simplificar la implementación, se asumió que la imagen tiene una cantidad de píxeles *múltiplo de 4* ($ancho * altura = 4 * k$); de esta forma, si en cada iteración se procesan 4 píxeles, no es necesario realizar una adaptación en función del tamaño de la imagen. Esta decisión es razonable porque si se admitieran imágenes sin esta característica, habría que procesar solamente 1, 2 o 3 píxeles fuera del ciclo principal, sin alterar significativamente la performance del filtro.

Sin embargo, incluso para una cantidad de píxeles múltiplo de 4, es importante contemplar el *caso borde* que sucede en la última iteración, cuando faltan procesar los últimos cuatro píxeles de la imagen. De realizarse una lectura desde el comienzo del píxel, se leería junto a los píxeles restantes un total de 4 bytes de memoria posteriores al fin de la imagen. Por esta razon, la última iteración se dejó fuera del ciclo principal, realizando un retroceso de 4 bytes en el puntero de lectura, y ejecutando una versión levemente modificada del cuerpo de ciclo de forma tal que procese los píxeles en los últimos 12 bytes del registro, en vez de los primeros.

3.3. Optimizaciones en código C

El código de la implementación en lenguaje C se compiló con los distintos flags de optimización descriptos en la sección **??**. Los resultados en cuestión de performance se pueden ver en la figura **??**, la cual muestra una comparación entre la cantidad de clocks consumidos durante la ejecución de cada versión.

TODO:: Discutir un poco la figura.

Adicionalmente, se analizaron las diferencias entre el código generado con compilación estándar y el código generado con optimización de tipo O1. La diferencia entre ambos evidencia inmediatamente una característica subóptima del código estándar; *todas* las variables locales se alojan en la pila, incluso habiendo disponibilidad de registros. Los registros se utilizan únicamente como variables temporales, y al final de cada extracto de código se guarda el resultado

en un espacio de la pila. El código generado con O1, en cambio, ahorra gran cantidad de esos accesos a memoria. Este comportamiento se ejemplifica con el siguiente extracto (figura **??**).

```
mov   eax, DWORD PTR[rbp-0x10]    | imul   r15d,r15d
mov   edx,eax                     | imul   r14d,r14d
imul  edx, DWORD PTR[rbp-0x10]    | add    r14d,r15d
mov   eax, DWORD PTR[rbp-0xc]     | imul   ebx,ebx
imul  eax, DWORD PTR[rbp-0xc]     | add    ebx,r14d
add   edx,eax                     |
mov   eax, DWORD PTR[rbp-0x8]     |
imul  eax, DWORD PTR[rbp-0x8]     |
add   eax,edx                     |
mov   DWORD PTR [rbp-0x4],eax     |
```

Figura 3: Código objeto generado con compilación por default (izquierda), y con optimización de tipo O1 (derecha).

Sin embargo, la lógica de flujo del código generado es equivalente. Es decir, las operaciones realizadas y el orden en que se realizan son iguales; se mantiene la forma de recorrer la imagen y el procesamiento píxel por píxel.

3.4. Optimización en código ASM - Desenrollado de ciclos

Sobre la implementación en lenguaje ensamblador ya descripta, se realizaron dos sucesivas modificaciones para estudiar la técnica conocida como desenrollado de ciclos.

4. Filtro Miniature

4.1. Descripción del filtro

El filtro miniature es un caso particular de la familia de filtros por convolución, en los cuales se procesa una imagen realizando una convolución entre esta y una matriz determinada denominada *kernel*. Las características del *kernel* determinan el efecto resultante sobre la matriz; en este caso, el efecto logrado se conoce como desefoque gaussiano (se puede ver un ejemplo en la figura **??**), y se obtiene mediante el siguiente *kernel*:

$$\frac{1}{600} * \begin{pmatrix} 1 & 5 & 18 & 5 & 1 \\ 5 & 32 & 64 & 32 & 5 \\ 18 & 64 & 100 & 64 & 18 \\ 5 & 32 & 64 & 32 & 5 \\ 1 & 5 & 18 & 5 & 1 \end{pmatrix}$$

Operativamente, la imagen es procesada mediante una actualización píxel a píxel, reemplazando el valor de cada canal de color del píxel por una combinación lineal del valor de sus vecinos, cada uno ponderado por un coeficiente determinado por un elemento de la matriz. Se toma el elemento central del *kernel* como coeficiente del píxel que se está procesando, y la posición relativa a los demás elementos de la matriz determina a cuál vecino corresponde en la combinación lineal. El factor $\frac{1}{600}$ es una constante de normalización que garantiza que el resultado sea un valor en el rango $[0, 255]$.

Figura 4: Imagen antes y después de aplicar el filtro miniature con parámetros de banda 0,08 y 0,25, y un total de 20 iteraciones.

Se incorpora una leve modificación al modelo típico de filtro por convolución, limitando el efecto del filtrado a dos bandas dentro de la imagen; una banda superior, y una banda inferior, dejando la banda central de la imagen inalterada. Se especifican dos parámetros $0 < topPlane < bottomPlane < 1$, de forma tal que las bandas quedan determinadas de esta forma:

banda superior: filas 0 ... $topPlane * altura$
banda media: filas $topPlane * altura + 1$ ... $bottomPlane * altura - 1$
banda baja: filas $bottomPlane * altura$ ... $altura - 1$

Adicionalmente, se realizan múltiples iteraciones de filtrado, reduciendo el ancho de cada banda luego de cada iteración por:

Δbanda superior: $topPlane * altura/cantIteraciones$
Δbanda baja: $(1 - bottomPlane) * altura/cantIteraciones$

En el caso de los píxeles del borde, donde no existe el vecindario completo, se optó por exceptuarlos del filtrado, simplificando el procedimiento al saltear las dos primeras y dos últimas filas y columnas.

4.2. Implementación en lenguaje C y lenguaje ensamblador

El filtro se implementó en C mediante un ciclo que visita una vez por iteración a cada píxel de la banda superior y de la banda inferior, y actualizando el valor de sus tres canales de color por la combinación lineal descripta previamente.

Para este filtro, la implementación intuitiva en C resultó particularmente sugerente a la necesidad de buscar formas de optimizar el procedimiento, dado que cada píxel se lee de la memoria hasta 25 veces (una vez por cada vecino) resultando en un tiempo de ejecución prolongado; y además, el tipo de procesamiento realizado por cada píxel es el cómputo de una combinación lineal, para lo cual los procesadores están altamente optimizados, desperdiciando los beneficios del hardware.

5. Decodificación esteganográfica

6. Conclusiones