DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Programación SIMD

Organización del Computador II

Primer Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Barrios, Leandro Ezequiel	404/11	ezequiel.barrios@gmail.com
Benegas, Gonzalo Segundo	958/12	gsbenegas@gmail.com

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

Resumen

En el presente trabajo se describe la problemática de las palomas cuando nacen y todavía no aprenden a volar y el nido está muy alto y se quieren lanzar pero la madre no las deja porque corren el riesgo de estrellarse entonces tienen que esperar que la madre les traiga la comida y como es invierno hay poca comida entonces los pichones pasan hambre entonces se lanzan del nido entonces los perros los encuentran en el piso y entonces los comen porque los perros son animales carnívoros y eso significa que comen otros animales porque les gusta el sabor de los otros animales aunque quizás las palomas bebé no tanto porque todavía están “verdes” aunque decir verdes quizás sea apropiado porque cuando las palomas son bebés su color es amarillo como los pollitos...

Índice

Figura 1: Descripcion de la figura

1. Objetivos generales

El objetivo de este Trabajo Práctico es

2. Contexto

Titulo del parrafo Bla bla bla bla. Esto se muestra en la figura 1.

```
struct Pepe {

    ...

};
```

3. Enunciado y solucion

Filtro *tiles*

Programar el filtro *tiles* en lenguaje C y luego en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

aaa aa

Descripción

En esta función, el procesamiento de los pixeles es muy sencillo. Se debe copiar un fragmento de la imagen - $tile$ - repetidas veces. Se copian de a 5 pixeles, que es la máxima cantidad que entra en un registro xmm. Lo complejo de esta función fue llevar control de la parte a copiar.

Vamos copiando por filas en la imagen destino, en adelante dst. Es decir, se copia la primera fila del $tile$ repetidamente hasta llegar al final de la fila de dst. En la segunda fila de dst copiamos la segunda fila de $tile$ repetidamente, etc... Si pasamos la última fila del $tile$ volvemos a copiar desde su primera fila. Si pasamos la última fila del dst hemos finalizado.

Experimento 1 - análisis el código generado

Utilizar la herramienta `objdump` para verificar como el compilador de C deja ensamblado el código C. Como es el código generado, ¿cómo se manipulan las variables locales?¿le parece que ese código generado podría optimizarse?

El código queda escrito en 109 líneas, mientras que la versión de asm llevaba 202 líneas. Ésto se debe en parte al diferente enfoque al problema. tiles_c procesa de a un pixel a la vez, simplemente utilizando la función módulo. Es un ciclo muy simple. tiles_asm procesa de a cinco pixeles a la vez, pero tiene una estructura de control más compleja. Se puede ver que se guardan los parámetros de la función en la pila, según la convención de manejo de variables locales. Se efectúan demasiados accesos a memoria para acceder a éstas variables. Dada esta función en particular, se podrían mantener una mayor cantidad de variables en los registros, y no tener que buscarlos a memoria en cada iteración del ciclo.

Experimento 2 - optimizaciones del compilador

Compile el código de C con optimizaciones del compilador, por ejemplo, pasando el flag `-O1`[1]. ¿Qué optimizaciones realizó el compilador? ¿Qué otros flags de optimización brinda el compilador? ¿Para qué sirven?

Importante para los gráficos: con O2 y O3 dan el mismo archivo. No tiene sentido graficar nada.

Insertar gráfico de barras comparando tamaño del código.

Con la optimización O1, el compilador intenta reducir el tamaño del código y el tiempo de ejecución, sin aumentar demasiado el tiempo de compilación. Al mismo tiempo se va perdiendo la capacidad de $debugging$, ya que se puede alterar el orden de ejecución de las instrucciones. En este programa en particular, con la optimización, se pueden observar menos accesos a memoria. La optimización O1 activa el flag $-fif-conversion$, que intenta transformar saltos condicionales a otras estructuras de control. Los niveles de optimización O2 y O3 optimizan aún más, a razón de un mayor tiempo de complilación. Por ejemplo, activan el flag $-foptimize-sibling-calls$, que elimina la recursión en funciones con recursión a la cola. Si se quiere minimazar el tamaño del archivo objeto, lo mejor es utilizar la optimización Os.

Experimento 3 - secuencial vs. vectorial

Realice una medición de las diferencias de performance entre las versiones de C y ASM (el primero con -O1, -O2 y -O3).
¿Como realizó la medición?¿Cómo sabe que su medición es una buena medida?¿Cómo afecta a la medición la existencia de *outliers*[2]?¿De qué manera puede minimizar su impacto?¿Qué resultados obtiene si mientras corre los tests ejecuta otras aplicaciones que utilicen al máximo la CPU? Realizar un análisis **riguroso** de los resultados y acompañar con un gráfico que presente estas diferencias.

Se midió el tiempo de ejecución de FIXME: **diez** corridas del programa, para cada tamaño de imagen. Se descartaron las mediciones menores que el percentil 25 o mayores que el percentil 75. Luego se tomó promedio. Se esperaba que los tiempos de ejecución de los filtros sean del orden del tamaño de entrada, lo que confirmamos viendo el carácter lineal de los gráficos.

Se detectó la presencia de outliers, específicamente tiempos de ejecución mayores que lo esperado para ese tamaño de entrada. Esto pudo haber sido causado, entre otras causas, por el uso del procesador debido a interrupciones o procesos de mantenimiento del sistema.

Mediante el procedimiento utilizado bastó para poder gráficar los tiempos de ejecución como una línea recta con respecto al tamaño de la imagen. Si se hubiese necesitado un mejor tratamiento de los outliers, se podría haber iterado más veces. Nuestro procedimiento es algorítmicamente más complejo que simplemente tomar el promedio, pero mucho más robusto frente a la presencia de outliers.

FIXME: **Insertar gráfico comparando tiempos de ejecución O, O1, O2, asm.**

Experimento 4 - cpu vs. bus de memoria

Se desea conocer cual es el mayor limitante a la performance de este filtro en su versión ASM.

¿Cuál es el factor que limita la performance en este caso? En caso de que el limitante fuera la intensidad de cómputo, entonces podrían agregarse instrucciones que realicen accesos a memoria y la performance casi no debería sufrir. La inversa puede aplicarse si el limitante es la cantidad de accesos a memoria.

Realizar un experimento, agregando múltiples instrucciones de un mismo tipo y realizar un análisis del resultado. Acompañar con un gráfico.

Experimento 5 (*opcional*) - secuencial vs. vectorial (parte II)

[1] agregando este flag a `CCFLAGS64` en el makefile

[2] en español, valor atípico: `http://es.wikipedia.org/wiki/Valor_atpico`

Si vemos a los pixeles como una tira muy larga de bytes, este filtro en realidad no requiere ningún procesamiento de datos en paralelo. Esto podría significar que la velocidad del filtro de C puede aumentarse hasta casi alcanzar la del de ASM. ¿ocurre esto?

Modificar el filtro para que en vez de acceder a los bytes de a uno a la vez se accedan como tiras de 64 bits y analizar la performance.

Filtro *Popart*

Programar el filtro *Popart* en lenguaje C y en en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 1 - saltos condicionales

Se desea conocer que tanto impactan los saltos condicionales en el código del ejercicio anterior con `-O1`.

Para poder medir esto, una posibilidad es quitar las comparaciones al procesar cada pixel. Por más que la imagen resultante no sea correcta, será posible tomar una medida del impacto de los saltos condicionales. Analizar como varía la performance.

Si se le ocurren, mencionar otras posibles formas de medir el impacto de los saltos condicionales.

Este filtro **reduce la imagen a un total de 5 colores** según el valor de la sumatoria de las componentes de cada pixel. Es decir, asigna el valor del pixel resultante según el intervalo en el que caiga la suma, con la particularidad de que estos cinco intervalos son *de igual tamaño*.

Por otro lado, la **suma mínima posible es** 0, y la **suma máxima posible es** $(255 * 3) = 765$, lo que significa que **para cualquier pixel la sumatoria estará contenida en** $\{0, \ \ldots, \ 765\}$. Dado que 765 es convenientemente divisible por 5, la longitud de cada intervalo es de 153. Así, el pixel P destino va a tomar un determinado valor $P_1 = (r_1, g_1, b_1)$ si la sumatoria de las componentes del pixel fuente está contenida en $I_1 = [0, 152]$, un valor $P_2 = (r_2, g_2, b_2)$ si la misma se encuentra en $I_2 = [153, 305]$, etc. Este último detalle es muy conveniente, ya que permite tener los pixeles en una tabla:

$$COLORES = \ < P_1, P_2, P_3, P_4, P_5 >$$

evitando así tener que efectuar el cálculo del valor del pixel en cada iteración. Más aún, desde el punto de vista implementativo esto es una gran ventaja, ya que es posible crear un vector de colores y acceder al mismo mediante los índices que resultan de **dividir**[3] **el valor de la sumatoria del pixel fuente por** 153. De esta forma, es fácil ver que el índice para cada intervalo coincidirá con el color que le corresponde:

$$INDICES = \ < I_1, I_2, I_3, I_4, I_5 >$$ [4]

.

Utilizando este último concepto, es posible evitar el uso de condicionales en la versión de C del filtro, ya que para cada iteración simplemente:

1. Se obtienen los valores de las componentes (R, G, B) del pixel fuente.
2. Se obtiene la suma **Sum** $= R + G + B$.
3. Se divide con truncamiento, obteniendo el índice **i** $=$ **Sum**$/153$.
4. Se obtiene el color resultante **c** $=$ `COLORES[i]`.
5. Se asigna **c** al pixel destino.

[3] División entera (es decir, con truncamiento).

[4] Abuso de notación: se busca mostrar un "paralelismo" entre la tabla (vector) de colores expuesta algunos renglones más arriba y el índice en donde caerá la sumatoria de cada intervalo.

Para realizar esto, es necesario definir previamente el array:

```
rgb_t colors[] =
{
  {255,   0,   0},
  {127,   0, 127},
  {255,   0, 255},
  {  0,   0, 255},
  {  0, 255, 255},
  {  0, 255, 255}
};
```

Es importante notar que dado que $765/153 = 5$ se sale del rango de índices [5] necesarios para 5 colores, $[0, \ldots, 4]$, fue necesario repetir el último color.

Para la versión de Assembler se aprovechan las características de SIMD para poder paralelizar las operaciones de lectura y cálculos con los datos. En este caso, por cada iteración se leen 4 pixeles a la vez, es decir 12 bytes, tal y como se ve en ***Figura 2***. Estos pixeles se guardan en `xmm0`. Posteriormente se los replica en `xmm1` y `xmm2`.

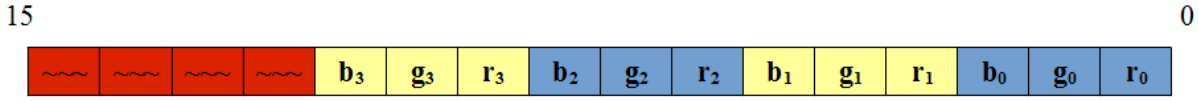

Figura 2: Estado inicial de XMM0

Luego se aplica la operación `PSHUFB` para cada registro, con las máscaras «**MÁSCARA R**», «**MÁSCARA G**» y «**MÁSCARA B**», tal y como se puede apreciar en ***Figura 3*** para rojo, ***Figura 4*** para verde y ***Figura 5*** para azul respectivamente. Luego de aplicar estas máscaras, como los valores fuente eran bytes sin signo, se puede extenderlos simplemente agregando ceros adelante. Por la forma en que están dispuestos los valores, esta extensión del tamaño del dato es trivial, tal y como se ejemplifica en ***Figura 6*** para el color rojo. La misma "operación" (trivial e implícita) se realiza para el resto de los registros.

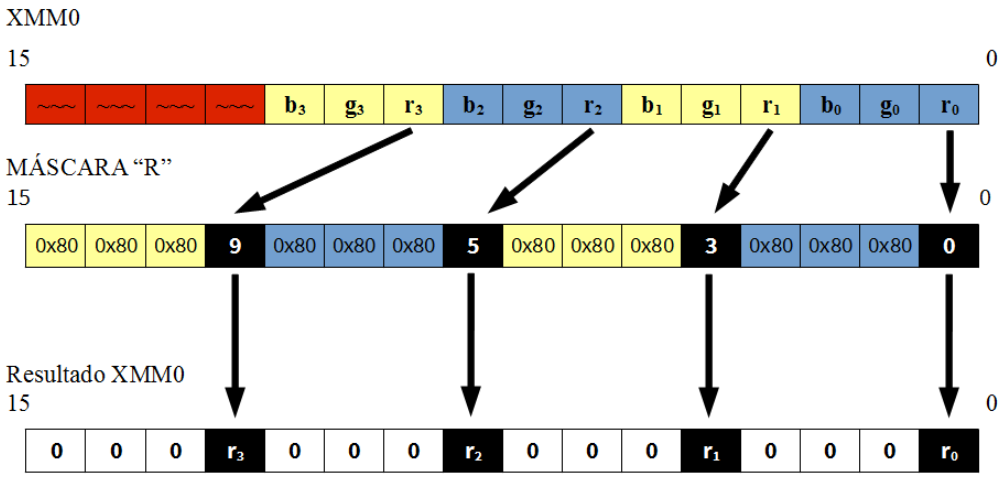

Figura 3: «**MÁSCARA R**»

[5] Es decir que el último color tiene de 154 valores de largo, y no 153 como el resto

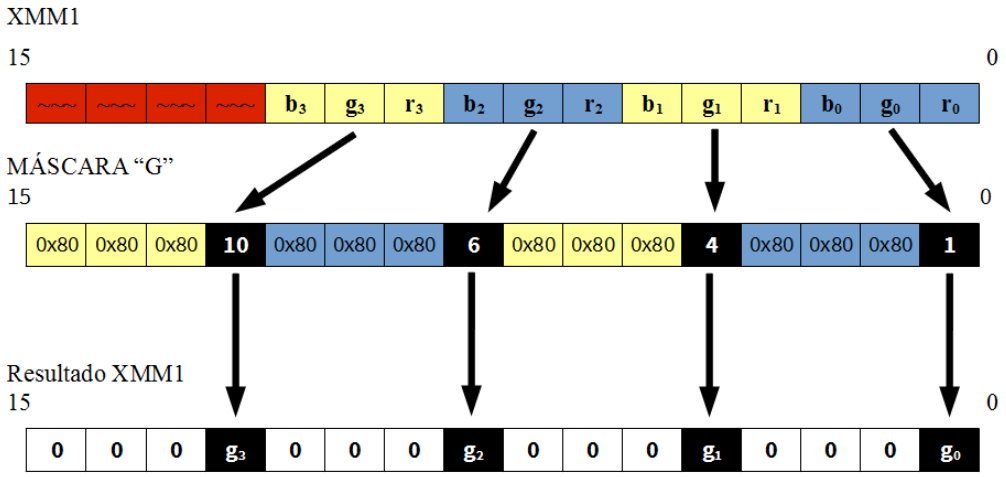

Figura 4: «**MÁSCARA G**»

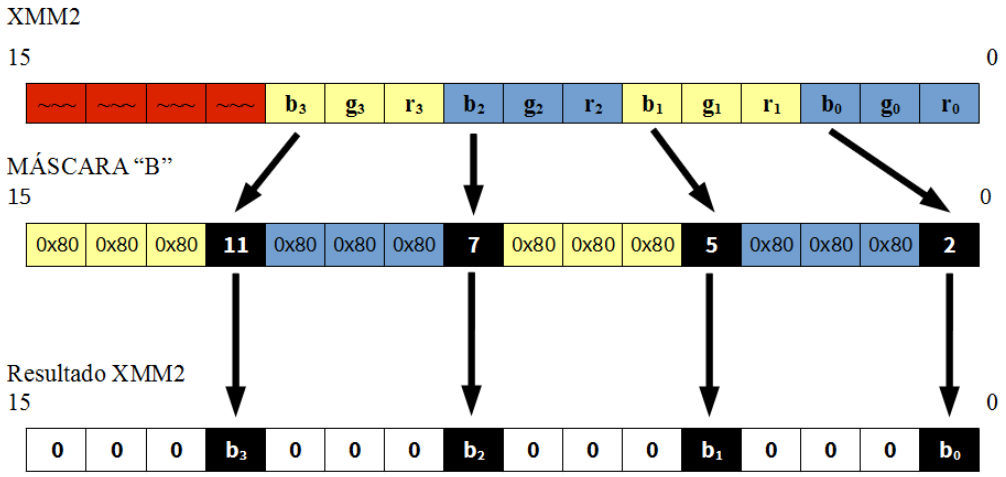

Figura 5: «**MÁSCARA B**»

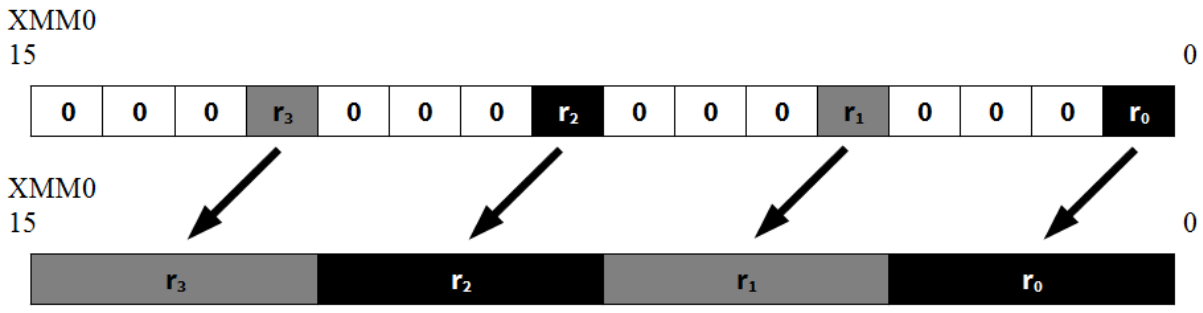

Figura 6: Extiende el registro trivialmente

Experimento 2 - cpu vs. bus de memoria

¿Cuál es el factor que limita la performance en este caso?

Realizar un experimento, agregando múltiples instrucciones de un mismo tipo y realizar un análisis del resultado. Acompañar con un gráfico.

Experimento 3 - prefetch

La técnica de *prefetch* es otra forma de optimización que puede realizarse. Su sustento teórico es el siguiente:

Suponga un algoritmo que en cada iteración tarda n ciclos en obtener un dato y una cantidad similar en procesarlo. Si el algoritmo lee el dato i y luego lo procesa, desperdiciará siempre n ciclos esperando entre que el dato llega y que se comienza a procesar efectivamente. Un algoritmo más inteligente podría pedir el dato $i+1$ al comienzo del ciclo de proceso del dato i (siempre suponiendo que el dato i pidió en la iteración $i-1$. De esta manera, a la vez que el procesador computa todas las instrucciones de la iteración i, se estáran trayendo los datos de la siguiente iteración, y cuando esta última comience, los datos ya habrán llegado.

Estudiar esta técnica y proponer una aplicación al código del filtro en la versión ASM. Programarla y analizar el resultado. ¿Vale la pena hacer prefetching?

Experimento 3 - secuencial vs. vectorial

Analizar cuales son las diferencias de performace entre las versiones de C y ASM. Realizar gráficos que representen estas diferencias.

Filtro *Temperature*

Programar el filtro *Temperature* en lenguaje C y en en ASM haciendo uso de las instrucciones vectoriales (**SSE**).

Experimento 1

Analizar cuales son las diferencias de performace entre las versiones de C y ASM. Realizar gráficos que representen estas diferencias.

Filtro *LDR*

Programar el filtro *LDR* en lenguaje C y en ASM haciendo uso de las instrucciones **SSE**.

Descripción

En este filtro se procesa de a un solo pixel en cada iteración del ciclo principal. Esto se debe a que para cada pixel no alcanza una lectura a memoria; se deben leer los 24 pixeles a su alrededor, lo que insume 5 lecturas a memoria. Primero se calcula la suma de las componentes r, g, y b de los pixeles circundantes. El valor máximo de esta suma es 5 * 5 * 255 * 3 = 19125 que ocupa 15 bits. Ésto nos obliga a extender cada componente r, g, b a word antes de sumar, para prevenir un overflow.

El valor máximo de una componente r, g, o b, es 255. El módulo de alfa tiene como valor máximo 255. Luego el dividendo tiene como valor máximo absoluto 1243603125, que ocupa 31 bits. Como la mantisa de un float es de 24 bits, debemos guardarlo en un double, cuya mantisa es de 53 bits. Debemos realizar tres divisiones, una para cada componente del pixel procesado. Como estamos trabajando con doubles, vamos a poder hacer dos divisiones en una sola instrucción, y la tercera en otra instrucción. Para podr hacer las dos divisiones, vamos a replicar ciertos valores que nos interesan. En registros xmm, como double, vamos a tener:

```
alfa | alfa

max | max

sumargb | sumargb
```

```
- | src_r

src_g | src_b
```

Luego procedemos a hacer las respectivas multiplicaciones y divisiones. Ahora, para sumar $src_{(i,j)}$ y $var_{(i,j)}$, los voy a tener que considerar como $signed$. Los voy a usar como signed word. $src_{(i,j)}$ lo voy a extender con ceros, y $var_{(i,j)}$, que era de punto flotante, lo voy a truncar con signo. No se va a perder información relevante al truncar, ya que valores mayores que 255 o menores que -255 no van a hacer diferencia, si después se va a saturar entre 0 y 255. Se efectua una suma con signo, ahora podemos trabajar con r, g, y b al mismo tiempo. El resultado se empaqueta de vuelta a byte, con saturación sin signo. Ahora se escribe el pixel procesado a memoria.

Experimento 1

Analizar cuales son las diferencias de performace entre las versiones de C y ASM. Realizar gráficos que representen estas diferencias.

4. Conclusiones y trabajo futuro