DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

subtitulo del trabajo

Organización del Computador II

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Nombre	XXX/XX	`mail`
Nombre	XXX/XX	`mail`

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 ...

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

3.1. Filtro cropflip

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

Experimento 1.1 - análisis el código generado

En este experimento vamos a utilizar la herramienta `objdump` para verificar como el compilador de C deja ensamblado el código C.

Ejecutar

```
objdump -Mintel -D cropflip_c.o
```

¿Cómo es el código generado? Indicar *a*) Por qué cree que hay otras funciones además de `cropflip_c` *b*) Cómo se manipulan las variables locales *c*) Si le parece que ese código generado podría optimizarse hola

Experimento 1.2 - optimizaciones del compilador

Compile el código de C con flags de optimización. Por ejemplo, pasando el flag `-O1`[1]. Indicar 1. Qué optimizaciones observa que realizó el compilador 2. Qué otros flags de optimización brinda el compilador 3. Los nombres de tres optimizaciones que realizan los compiladores.

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

3.2. Mediciones

Realizar una medición de performance *rigurosa* es más difícil de lo que parece. En este experimento deberá realizar distintas mediciones de performance para verificar que sean buenas mediciones.

En un sistema "ideal" el proceso medido corre solo, sin ninguna interferencia de agentes externos. Sin embargo, una PC no es un sistema ideal. Nuestro proceso corre junto con decenas de otros, tanto de usuarios como del sistema operativo que compiten por el uso de la CPU. Esto implica que al realizar mediciones aparezcan "ruidos" o "interferencias" que distorsionen los resultados.

El primer paso para tener una idea de si la medición es buena o no, es tomar varias muestras. Es decir, repetir la misma medición varias veces. Luego de eso, es conveniente descartar los outliers [2], que son los valores que más se alejan del promedio. Con los valores de las mediciones resultantes se puede calcular el promedio y también la varianza, que es algo similar el promedio de las distancias al promedio[3].

Las fórmulas para calcular el promedio μ y la varianza σ^2 son

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}$$

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

[3] en realidad, elevadas al cuadrado en vez de tomar el módulo

Experimento 1.3 - calidad de las mediciones

1. Medir el tiempo de ejecución de cropflip 10 veces.
2. Implementar un programa en C que no haga más que ciclar infinitamente sumando 1 a una variable. Lanzar este programa tantas veces como *cores lógicos* tenga su procesador. Medir otras 10 veces mientras estos programas corren de fondo.
3. Calcular el promedio y la varianza en ambos casos.
4. Consideraremos outliers a los 2 mayores tiempos de ejecución de la medicion a) y también a los 2 menores, por lo que los descartaremos. Recalcular el promedio y la varianza después de hacer este descarte.
5. Realizar un gráfico que presente estos dos últimos items.

A partir de aquí todos los experimentos de mediciones deberán hacerse igual que en el presente ejercicio: tomando 10 mediciones, luego descartando outliers y finalmente calculando promedio y varianza.

Experimento 1.4 - secuencial vs. vectorial

En este experimento deberá realizar una medición de las diferencias de performance entre las versiones de C y ASM (el primero con -O0, -O1, -O2 y -O3) y graficar los resultados.

Experimento 1.5 - 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 extra y la performance casi no debería sufrir. La inversa puede aplicarse, si el limitante es la cantidad de accesos a memoria. [4]

Realizar un experimento, agregando 4, 8 y 16 instrucciones aritméticas (por ej `add rax, rbx`) analizando como varía el tiempo de ejecución. Hacer lo mismo ahora con instrucciones de acceso a memoria, haciendo mitad lecturas y mitad escrituras (por ejemplo, agregando dos `mov rax, [rsp]` y dos `mov [rsp+8], rax`).[5]

Realizar un único gráfico que compare: 1. La versión original 2. Las versiones con más instrucciones aritméticas 3. Las versiones com más accesos a memoria

Acompañar al gráfico con una tabla que indique los valores graficados.

Filtro *Sierpinski*

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

Experimento 2.1 - secuencial vs. vectorial

Analizar cuales son las diferencias de performace entre las versiones de C y ASM de este filtro, de igual modo que para el experimento 1.4.

Experimento 2.1 - cpu vs. bus de memoria

¿Cuál es el factor que limita la performance en este filtro? Repetir el experimento 1.5 para este filtro.

[4] también podría pasar que estén más bien balanceados y que agregar cualquier tipo de instrucción afecte sensiblemente la performance

[5] Notar que en el caso de acceder a `[rbp]` o `[rsp+8]` probablemente haya siempre hits en la cache, por lo que la medición no será de buena calidad. Si se le ocurre la manera, realizar accesos a otras direcciones alternativas.

Filtro *Bandas*

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

Experimento 3.1 - saltos condicionales

Se desea conocer que tanto impactan los saltos condicionales en el código de filtro Bandas con `-O1` (la versión en C).
Para poder medir esto de manera aproximada, remover el código que detecta a que banda pertenece cada pixel, dejando sólo una banda. Por más que la imagen resultante no sea correcta, será posible tomar una medida aproximada del impacto de los saltos condicionales. Analizar como varía la performance.

Experimento 3.2 - secuencial vs. vectorial

Repetir el experimento 1.4 para este filtro.

Filtro *Motion Blur*

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

Experimento 4.1

Repetir el experimento 1.4 para este filtro

4. Conclusiones y trabajo futuro