Organización del Computador II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico Número 2

Procesamiento de imágenes

Integrante	LU	Correo electrónico
Alcain Pablo	485/07	pabloalcain@gmail.com
Gamarra Ignacio	792/11	gamarra_mi32@yahoo.com.ar
Santos Diego	874/03	diego.h.santos@gmail.com

Grupo

My Generation

Índice

1. Introducción

En este trabajo hemos realizado la implementación de funciones de procesamiento de imágenes en dos lenguajes de programación, *Assembler* y *C*. Además realizamos un análisis de la performance de las mismas a fin de determinar cual es mas eficiente respecto al tiempo de ejecución. Las funciones implementadas son:

- `Diferencia de Imágenes`, a partir de dos imágenes genera una tercera imagen que resalta los pixeles donde las dos imagenes fuentes difieren. Para obtener la tercer imagen se toma la norma infinito la norma infinito de la resta vectorial entre los píxel, ignorando el canal alfa. La fórmula a aplicar para obtener la tercer imágen es;
- `Blur Gaussiano`, tomando una imagen de fuente genera una segunda similar a la fuente pero con un aspecto desenfocado. La manera de lograr este efecto es calculando cada componente de la imagen de salida como un promedio ponderado con una gaussiana 2D de los píxeles que la circundan. Las formulas a aplicar para realizar el filtro son las siguientes:

En el caso de las implementaciones hechas en *Assembler* hemos usado el modelo de programación *SIMD*, a través del set de instrucciones `SSE`, pues el objetivo de este trabajo práctico es estudiar las ventajas y desventajas de usar ese modelo contra uno *SISD*.

En las siguientes secciones se explicará como se implementaron los filtros, se presentarán gráficos mostrando los tiempos de ejecución de cada filtro comparando ambas implementaciones y luego se presentará una conclusión acerca de los resultados obtenidos y el costo de usar un modelo *SIMD* para programar.

2. Desarrollo

2.1. Diferencia de Imagenes

Hicimos el filtro `diff` tanto en C como en ASM. Su implementación es bastante inmediata, al menos sin tener en cuenta los registros `XMM` de `SSE`. De esta manera, el filtro `diff` recorre todos los píxeles de ambas imágenes, de forma secuencial, resta los tres componentes de colores y toma el valor absoluto. Así, en cada píxel (i, j) [de 4 bytes cada uno], obtenemos tres bytes $(\Delta R, \Delta G, \Delta B)$. En este filtro `diff`, convenimos que la imagen diferencia fuera la norma infinito del vector en las tres posiciones (es decir, el máximo entre los tres). Así, una vez obtenidos $(\Delta R, \Delta G, \Delta B)$, calculamos el máximo entre los tres, ΔC, y en el píxel (i, j) de la imagen de salida anotamos los tres bytes $(\Delta C, \Delta C, \Delta C)$ y 0xFF en el canal alpha. Repetimos este procedimiento para todos los píxeles de la imagen.

En la implementación de ASM, aprovechamos los registros `SSE` disponibles en el procesador[1]. En cada registro entran 16 bytes: 4 píxeles que pueden procesarse a la vez. Ahora procesarlos no es tan trivial como antes, pero podemos ver que pensar algunas relaciones básicas de álgebra, ayuda a ver el camino a seguir. Recordemos los dos pasos:

Restar los tres componentes y tomar valor absoluto La clave para reducir este problema es considerando la siguiente igualdad:

$$\text{abs}(a - b) = \text{máx}(a, b) - \text{mín}(a, b) \tag{1}$$

Así, simplemente tenemos que asegurarnos de estar restando el máximo (byte a byte) menos el mínimo. Para eso, supongamos que tenemos las tiras de 16 bytes de ambas imágenes en `XMM0` y `XMM3`. Ahora, copiamos cualquiera de los dos (por caso, `XMM0`) a un registro extra `XMM1`, y guardamos `máx (XMM0, XMM3)` en `XMM0` [`pmaxub`]. Luego, guardamos `mín (XMM1, XMM3)` en `XMM1` [`pminub`]. Ahora, `XMM0` tiene los máximos y `XMM1` los mínimos, byte a byte, de los 4 píxeles. Los restamos byte a byte y guardamos el resultado en `XMM0` [`psubb`]. Hacemos aquí unas cuentas extras, ya que también estamos procesando el canal alpha. Esto no significa un detrimento en performance (ya que la operación está paralelizada), sino que si la información estuviera guardada en RGB sin canal alpha se podrían procesar más píxeles a la vez[2]. Estos pasos se ven esquematizados en la figura **??**

Obtener el máximo Ahora el desafío es encontrar, con operaciones `SSE`, una forma de encontrar el máximo de cada bloque de 4 bytes. Pensemos cada bloque de 4 bytes por separado primero. Tenemos tres valores R, G, B, A y queremos que esto se transforme en C, C, C, X [C es el máx(R, G, B) y X es el canal alpha, que por ahora puede tomar cualquier valor]. Si $\mathbf{t}$ es el vector de salida, queremos

$$\mathbf{t} = (\text{máx}(R, G, B), \text{máx}(R, G, B), \text{máx}(R, G, B)) \tag{2}$$

Para vectorizar esta expresión, tenemos que usar la siguiente identidad:

$$\text{máx}(R, G, B) = \text{máx}(R, \text{máx}(G, B)) = \text{máx}(G, \text{máx}(B, R)) = \text{máx}(B, \text{máx}(R, G)) \tag{3}$$

Definimos al vector $\mathbf{p} = (R, G, B)$ y al operador `rot` tal que:

[1] El procesador utilizado, intel i7, tiene disponible `AVX`, pero no utilizamos estas instrucciones

[2] Claro que ahora no podríamos garantizarnos esta hermosura de que entren *justo* 4 píxeles en un registro `SSE`

$$\text{rot}(R, G, B) = (G, B, R) \tag{4}$$

Las sucesivas aplicaciones de `rot` las definimos también:

$$\mathbf{q} = \text{rot}(R, G, B) = (G, B, R) \tag{5}$$
$$\mathbf{r} = \text{rot}(\text{rot}(R, G, B)) = (B, R, G) \tag{6}$$

Con esta notación, la identidad 3 queda

$$\text{máx}(R, G, B) = \text{máx}(p_1 \text{máx}(q_1, r_1)) = \text{máx}(p_2, \text{máx}(q_2, r_2)) = \text{máx}(p_3, \text{máx}(q_3, r_3)) \tag{7}$$

y, en consecuencia, $\mathbf{t}$ es

$$\mathbf{t} = \text{máx}(\mathbf{p}, \text{máx}(\mathbf{q}, \mathbf{r})) \tag{8}$$

Esta ecuación ahora está vectorizada y podemos usar `pmaxub` para calcular los máximos Sólo queda crear, a partir de $\mathbf{p}$ los vectores rotados $\mathbf{q}$ y $\mathbf{r}$; es decir, implementar el operador `rot`. Para esto, utilizamos una máscara que rota a la izquierda los tres bytes, es decir: $(R, G, B) \to (G, B, R)$, utilizando `pshufb` y una máscara del tipo (1, 2, 0).

Como, en rigor, en cada píxel tenemos también el canal alpha, la máscara para cada byte va a ser (1, 2, 0, 0xFF), poniendo un 0 en el canal alpha [esto también podría ser la posición 3 o cualquier otra, porque el canal alpha *tiene que* ser reescrito].

Esto explica el procedimiento general. Aumentarlo para procesar n píxeles en paralelo es directo, simplemente la máscara de rotación va a ser $\mathbf{m} = (1, 2, 0, 0\text{xFF}, 4+1, 4+2, 4+0, 0\text{xFF}, 8+1, 8+2, 8+0, 0\text{xFF}, \ldots)$. En general,

$$m_i = \begin{cases} 4 * \text{floor}(i/4) + \text{mod}(i+1, 3) & \text{mod}(i, 4) \neq 3 \\ 0\text{xFF} & \text{mod}(i, 4) = 3 \end{cases} \tag{9}$$

La implementación entonces es directamente una traducción de esto: Copiamos el registro `XMM0` ($\mathbf{p}$) a un registro `XMM1` y mantenemos la máscara de rotación en `XMM2`. Ahora rotamos `XMM1` con `pshufb` y obtenemos $\mathbf{q}$. Obtenemos el máximo de `XMM0` ($\mathbf{p}$) y `XMM1` ($\mathbf{q}$) con `pmaxub` y lo guardamos en `XMM0`. Rotamos nuevamente `XMM1` y resulta $\mathbf{r}$. Ahora calculamos el máximo de `XMM0` (máx($\mathbf{p}, \mathbf{q}$)) y `XMM1` ($\mathbf{r}$) y lo guardamos en `XMM0` (el objetivo final, $\mathbf{t}$).

En la figura **??** se puede observar un esquema de este procedimiento.

2.2. Blur Gaussiano

3. Resultados

En esta sección se muestran los tiempos de ejecución de cada algoritmo comparandose las implementaciones hechas en **C** contra las implementaciones hechas en **ASM**.
Dichos resultados se corresponden con la cantidad de ticks del procesador que cada algoritmo tomó, usando para ello el parámetro **-t** del programa principal, cuyo valor devuelto es justamente la cantidad de ticks para ejecutar el filtro de entrada una cantidad de veces equivalente a la pasada como parámetro.

Como medimos los ticks del procesador? Es algo que se hace internamente en el código que la cátedra nos proporcionó pero de todas formas sabemos que esto se lleva a cabo usando la instrucción **rdtsc**, la cual obtiene el Time Stamp Counter (TSC). Dicho registro se incrementa en uno con cada ciclo del procesador, de modo que la cantidad de ciclos total equivale a la diferencia del valor después y antes de ejecutar cada filtro.
Notar que este registro es global y por ende cuenta ticks que todos los procesos del sistema estan consumiendo, no solo el nuestro, de modo que sería incorrecto hacer solo una medición, en vez de eso hacemos 1000 y tomamos el promedio, para suavizar outliers (observación numéricamente muy distante al resto de los valores).

3.1. SpeedUp

En computación paralela el *speedup* refiere a cuánto más rápido es un algoritmo paralelo (en nuestro caso refieren a las implementaciones en Asm que hacen uso de las instrucciones SSE) que el correspondiente algoritmo secuencial (lo que sería cada implementación en C).

Porqué consideramos importante medir que tanto más rápido es la versión paralela que la secuencial? Si bien este tipo de análisis excede originalmente lo pedido por la cátedra, nos pareción razonable dar una idea de la magnitud de que tantas veces es mejor una implementación paralela que secuencial. Si bien este valor se desprende de la cantidad de ticks insumidos por cada implementación, nos pareció adecuado formalizarlo usando el concepto de *speedup*. Se calcula con la fórmula:

$$S_p = \frac{T_1}{T_p} \tag{10}$$

donde:

- T_1 : cantidad de ticks del algoritmo secuencial
- T_p : cantidad de ticks del algoritmo paralelo con p procesadores (en nuestro caso el número de procesadores equivaldría a cantidad de pixels procesados en paralelo)

Se considera speedup lineal cuando:

$$S_p = p \tag{11}$$

3.2. Imágenes y consideraciones sobre los gráficos

Para el analisis generamos un script en python que genera imagenes al azar, de distintas dimensiones, las imagenes son cuadradas y las dimensiones usadas para la comparacion fueron: ACA REEMPLAZAR POR LAS QUE USAMOS100x100, 150x150, 200x200, 250x250, 300x300, 350x350, 400x400, 450x450, 500x500, 550x550, 600x600, 650x650.

Dado que en todos los casos consideramos imágenes cuadradas, en el eje de abscisas de los gráficos sólo especificamos la cantidad de pixels por fila (o columna, que es igual)

Las mediciones se repitieron 1000 veces y los parámetros particulares a cada filtro se especifican previo a su correspondiente gráfico. En las siguientes secciones mostramos gráficos con cierto análisis particular para cada filtro según corresponda y al final, conclusiones que aplican a todos los filtros.

3.3. Comparaciones entre las optimizaciones de C

Aca comentar que diferencias hay entre una y otra

3.4. Diferencia de Imagenes

3.5. Blur Gaussiano

A partir de los resultados y tablas vistas podemos mencionar las siguientes conclusiones generales que aplican a todos los filtros:

- Las implementaciones hechas en *Assembler* son efectivamente más rápidas que las implementaciones hechas en *C*. Esto era lo esperado pueés las implementaciones en *Assembler* hacen uso de las instrucciones SSE y se procesan mas de un pixel simultaneamente.
- En los algoritmos en los que no hubo conversión a Float de los datos, la velocidad de resolución fue mayor porque al no ser necesario usar floats, cada pixel ocupaba menos bytes y por ende podíamos procesar mayor cantidad simultaneamente.
- El SpeedUp de las funciones no cambió demasiado entre las funciones. Y todas estas operaciones se pueden obtener directamente con instrucciones SIMD, mientras que en C hubo que programarlas, realentizando el algoritmo.
- En los gráficos de SpeedUp se pueden ver picos, que si bien no son muy marcados, llaman la atención. Pensamos que esto puede suceder porque medimos los ciclos que tarda el procesador en correr las funciones, y estos se pueden ver afectados por otro uso en simultáneo que se le esté dando al mismo.
- En todos los gráficos de comparacion entre C y ASM las curvas presentan un crecimiento semejante al de una funcion cuadrática. Esto se debe a que la escala tomada en el eje de abscisas hace referencia al tamaño de un lado de las imagenes (todas las imagenes son cuadradas). Como la cantidad total de pixeles en la imagen es el cuadrado de este valor, la cantidad de ciclos de cpu aumenta linealmente respecto a la cantidad de pixeles.

4. Conclusión

Básicamente, este trabajo práctico nos ha hecho pensar como implementar ciertas funciones de una manera distinta a la que estamos acostumbrados. Además, nos ha hecho conocer en profundidad el modelo *SIMD*.

La programación vectorial es muy útil para optimizar funciones. Creemos que nuestras implementaciones muestran claramente la ganancia en performance que hay al programar así. Sin embargo, hay un costo. Los algoritmos son más complejos debido a que el número de instrucciones aumenta significativamente, se debe pensar cuidadosamente como se cargan los datos de memoria y como se guardan para evitar errores de *Violación de segmento*. Muchas veces los datos deben ser transformados y reacomodados para poder usarlos con las operaciones *SIMD*. También hay que analizar bien cuando finalizar los ciclos y si se debe tratar aparte y como se deben tratar los últimos datos, esos que no pudieron ser tratados en el ciclo debido a que no eran suficientes como para poder cargarlos en un registro `xmmx`. La otra desventaja que tiene este modo de operación es que nos ata a una arquitectura específica haciendo nuestro programa no portable. Nos ha pasado tener que repensar ciertos partes del código por no contar, por ejemplo, con la extensión `SSE3`.

Para cerrar, nos resultó interesante ver la aplicación de la programación vectorial en un tema concreto como el tratamiento de imágenes.