Trabajo Práctico 2

Organización del Computador II

Primer Cuatrimestre de 2019

1. Introducción

Este trabajo práctico consiste en implementar filtros gráficos utilizando el modelo de procesamiento SIMD. El mismo consta de dos componentes igualmente importantes. En primer lugar aplicaremos lo estudiado en clase programando de manera vectorizada un conjunto de filtros. Luego, considerando las características de microarquitectura del procesador, se pedirá realizar un análisis experimental del funcionamiento de los filtros. Estos serán codificados en lenguaje ensamblador y lenguaje C.

La etapa de experimentación se deberá realizar con un carácter científico y una metodología clara. Resulta fundamental entonces, la rigurosidad y exhaustividad del análisis que realicen.

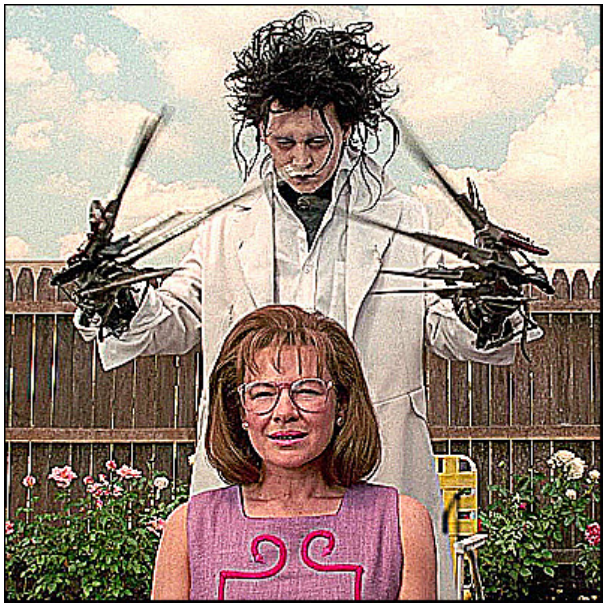
2. Filtros

Imagen original

Cuadrados

Manchas

Offset



Sharpen

Ruido

2.1. Preliminares

Consideramos a una imagen como una matriz de píxeles. Cada píxel está determinado por cuatro componentes: los colores azul (`b`), verde (`g`) y rojo (`r`), y la transparencia (`a`). En nuestro caso particular cada una de estas componentes tendrá 8 bits (1 byte) de profundidad, es decir, estarán representadas por números enteros en el rango $[0, 255]$.

Nombraremos a las matrices de píxeles `src` como la imagen de entrada y como `dst` a la imagen destino. Ambas serán almacenadas por filas, tomando el píxel *[0,0]* de la matriz como el píxel arriba a la izquierda de la imagen. [1]

En todos los filtros, el valor de la componente de transparencia debe ser 255.

[1] Notar que el formato BMP utilizado para almacenar las imágenes, invierte el orden de las filas en el archivo almacenado.

2.2. Cuadrados

Genera un efecto similar al desenfoque, replicando los píxeles de valor mayor alrededor de cada uno. Esta tarea la realiza en marcos de 4×4 píxeles por medio del siguiente pseudocódigo:

```
Para i de 4 a height - 5:
    Para j de 4 a width - 5:

        maxB = 0, maxG = 0, maxR = 0
        Para ii de 0 a 3:
            Para jj de 0 a 3:
                maxB = maximo(maxB, src[i+ii][j+jj].b)
                maxG = maximo(maxG, src[i+ii][j+jj].g)
                maxR = maximo(maxR, src[i+ii][j+jj].r)

        dst[i][j].b = maxB;
        dst[i][j].g = maxG;
        dst[i][j].r = maxR;
```

Las operaciones se realizan dejando un marco de 4 píxeles alrededor de toda la imagen. Estos píxeles deben ser completados con el color negro (0,0,0) sobre la imagen destino.

2.3. Manchas

Este filtro genera una serie de manchas sobre la imagen modificando los valores de cada píxel en función de una fórmula que calcula el tono. Ésta toma como parámetro el diámetro de las manchas, que en el pseudocódigo a continuación es identificado como `n`.

```
Para i de 0 a height - 1:
    Para j de 0 a width - 1:

        ii = 2 * PI * (i % n) / n
        jj = 2 * PI * (j % n) / n
        tono = sen(ii) * cos(jj) * 50 - 25;

        dst[i][j].b = SAT(src[i][j].b + tono)
        dst[i][j].g = SAT(src[i][j].g + tono)
        dst[i][j].r = SAT(src[i][j].r + tono)
```

Este filtro opera sobre todos los píxeles de la imagen.

2.4. Offset

El color de cada píxel está dado por sus tres componentes: rojo, verde y azul. En las impresiones por offsets, los colores se forman aplicando una a una cada componente sobre el papel. Cuando la aplicación de los colores se desfasa, se produce un efecto de estela de color alrededor de cada figura. El objetivo de este filtro es generar este efecto para un desfasaje fijo de las componentes de color de las imágenes, respetando el siguiente pseudocódigo:

```
Para i de 8 a height - 9:
    Para j de 8 a width - 9:
```

```
dst[i][j].b = src[i+8][j].b;
dst[i][j].g = src[i][j+8].g;
dst[i][j].r = src[i+8][j+8].r;
```

Las operaciones se realizan dejando un marco de 8 píxeles alrededor de toda la imagen. Estos píxeles deben ser completados con el color negro (0,0,0) sobre la imagen destino.

2.5. Sharpen

Existe un conjunto de efectos sobre imágenes que se realizan aplicando operaciones matriciales sobre los píxeles. Estas matrices se denominan operadores. Para este filtro se pide implementar el operador *Sharpen* o "realzado" respetando el siguiente pseudocódigo:

```
                        | -1, -1, -1 |
float sharpen[3][3] = | -1,  9, -1 |
                        | -1, -1, -1 |

Para i de 0 a height - 3:
    Para j de 0 a width - 3:

        totalB = 0, totalG = 0, totalR = 0

        Para ii de 0 a 2:
            Para jj de 0 a 2:
                totalB = totalB + sharpen[ii][jj] * src[i+ii][j+jj].b
                totalG = totalG + sharpen[ii][jj] * src[i+ii][j+jj].g
                totalR = totalR + sharpen[ii][jj] * src[i+ii][j+jj].r

        dst[i+1][j+1].b = SAT(totalB);
        dst[i+1][j+1].g = SAT(totalG);
        dst[i+1][j+1].r = SAT(totalR);
```

Las operaciones se realizan dejando un marco de 1 píxel alrededor de toda la imagen. Estos píxeles deben ser completados con el color negro (0,0,0) sobre la imagen destino.

2.6. Ruido

Este último filtro tiene carácter optativo. Busca agregar ruido a las imágenes generando información aleatoria para tal fin. El generador de números aleatorios utilizado es `rand()` de C. Por ende, para procesar cada píxel se debe llamar a una función de C. Realizar esta tarea todo el tiempo puede ser ineficiente, entonces la solución esperada utilizando SIMD de este filtro implica solicitar un espacio de memoria (buffer). Luego, se debe utilizar este buffer para almacenar los resultados de la generación de números aleatorios. Por último la solución tomaría los datos de este buffer para utilizarlos como entrada en el código del filtro. La estrategia tomada en la generación del buffer y su tamaño es parte de las decisiones de diseño para implementar este filtro.

Pseudocódigo:

```
srand(123);
Para i de 0 a height - 1:
    Para j de 0 a width - 1:
        a = (rand() % 255) - 127;
```

```
dst[i][j].b = SAT(str[i][j].b + a);
dst[i][j].g = SAT(str[i][j].g + a);
dst[i][j].r = SAT(str[i][j].r + a);
```

Este filtro opera sobre todos los píxeles de la imagen.

3. Implementación

Para facilitar el desarrollo del trabajo práctico se cuenta con un *framework* que provee todo lo necesario para poder leer y escribir imágenes, así como también compilar y probar las funciones a implementar.

3.1. Archivos y uso

Dentro de los archivos presentados deben completar el código de las funciones pedidas. Puntualmente encontrarán el programa principal, denominado **tp2**, que se ocupa de parsear las opciones ingresadas por el usuario y ejecutar el filtro seleccionado sobre la imagen ingresada. Los archivos entregados están organizados en las siguientes carpetas:

- doc: Contiene este enunciado y un *template* del informe en LaTeX.
- src: Contiene el código fuente, junto con el framework de ejecución y testeo. También tiene los fuentes del programa principal, junto con el **Makefile** que permite compilarlo. Además posee los siguientes subdirectorios:
  - build: Los archivos objeto y ejecutables del TP.
  - filters: Las implementaciones de los filtros
  - helper: Los fuentes de la biblioteca BMP y de la herramienta de comparación de imágenes.
  - img: Algunas imágenes de prueba.
  - test: Scripts para realizar tests sobre los filtros y uso de la memoria.

Compilación

Ejecutar `make` desde la carpeta `codigo`. Recordar que cada directorio tiene su propio Makefile, por lo que si se desean cambiar las opciones de compilación debe buscarse el Makefile correspondiente.

Uso

El uso del programa principal es el siguiente:

```
$ ./tp2 <nombre_filtro> <opciones> <nombre_archivo_entrada> [parámetros...]
```

Los filtros que se pueden aplicar y sus parámetros son los especificados en la sección 2.
Las opciones que acepta el programa son las siguientes:

- `-h, --help`
  Imprime la ayuda.

- `-i, --implementacion NOMBRE_MODO`
  Implementación sobre la que se ejecutará el proceso seleccionado. Las implementaciones disponibles son: `c`, `asm`.
- `-t, --tiempo CANT_ITERACIONES`
  Mide el tiempo que tarda en ejecutar el filtro sobre la imagen de entrada una cantidad de veces igual a `CANT_ITERACIONES`.
- `-o, --output DIRECTORIO`
  Genera el resultado en `DIRECTORIO`. De no incluirse, el resultado se guarda en el mismo directorio que el archivo fuente.
- `-v, --verbose`
  Imprime información adicional.

3.2. Código de los filtros

Para implementar los filtros descriptos anteriormente, tanto en C como en ASM se deberán implementar las funciones especificadas en la sección 2. Las imágenes se almacenan en memoria en color, en el orden B (blue), G (green), R (red), A (alpha).

Los parámetros genéricos de las funciones son:

- src : Es el puntero al inicio de la matriz de elementos de 32 bits sin signo (el primer byte corresponde al canal azul de la imagen (B), el segundo el verde (G), el tercero el rojo (R), y el cuarto el alpha (A)) que representa a la imagen de entrada. Es decir, como la imagen está en color, cada píxel está compuesto por 4 bytes.
- dst : Es el puntero al inicio de la matriz de elementos de 32 bits sin signo que representa a la imagen de salida.
- filas : Representa el alto en píxeles de la imagen, es decir, la cantidad de filas de las matrices de entrada y salida.
- cols : Representa el ancho en píxeles de la imagen, es decir, la cantidad de columnas de las matrices de entrada y salida.
- src_row_size : Representa el ancho en bytes de cada fila de la imagen incluyendo el padding en caso de que hubiese. Es decir, la cantidad de bytes que hay que avanzar para moverse a la misma columna de fila siguiente o anterior.

Consideraciones

Las funciones a implementar en lenguaje ensamblador deben utilizar el set de instrucciones **SSE**, a fin de optimizar la performance de las mismas.

Tener en cuenta lo siguiente:

- El ancho de las imágenes es siempre mayor a 16 píxeles y múltiplo de 8 píxeles.
- No se debe perder precisión en ninguno de los cálculos, a menos que se indique lo contrario.
- La implementación de cada filtro deberá estar optimizada para el filtro que se está implementando. No se puede hacer una función que aplique un filtro genérico y después usarla para implementar los que se piden.
- Para el caso de las funciones implementadas en lenguaje ensamblador, se debe trabajar con la máxima cantidad de píxeles en simultáneo. De no ser posible esto para algún filtro, deberá justificarse debidamente en el informe.

- El procesamiento de los píxeles se deberá hacer **exclusivamente** con instrucciones **SSE**. No está permitido procesarlos con registros de propósito general, salvo para tratamiento de casos borde. En tal caso se deberá justificar debidamente y hacer un análisis del costo computacional.
- El TP se tiene que poder ejecutar en las máquinas del laboratorio.

3.3. Formato BMP

El formato BMP es uno de los formatos de imágenes más simples: tiene un encabezado y un mapa de bits que representa la información de los píxeles.

En este trabajo práctico se utilizará una biblioteca provista por la cátedra para operar con archivos en ese formato. Si bien esta biblioteca no permite operar con archivos con paleta, es posible leer tres tipos de formatos, tanto con o sin transparencia. Ambos formatos corresponden a los tipos de encabezado: BITMAPINFOHEADER (40 bytes), BITMAPV3INFOHEADER (56 bytes) y BITMAPV5HEADER (124 bytes).

El código fuente de la biblioteca está disponible como parte del material, deben seguirlo y entenderlo. Las funciones que deben implementar reciben como entrada un puntero a la imagen. Este puntero corresponde al mapa de bits almacenado en el archivo. El mismo está almacenado de forma particular: **las líneas de la imagen se encuentran almacenadas de forma invertida**. Es decir, en la primera fila de la matriz se encuentra la última línea de la imagen, en la segunda fila se encuentra la anteúltima y así sucesivamente. Dentro de cada línea los píxeles se almacenan de izquierda a derecha, y cada píxel **en memoria se guarda en el siguiente orden: B, G, R, A**.

3.4. Herramientas y tests

En el código provisto, podrán encontrar varias herramientas que permiten verificar si los filtros funcionan correctamente.

Diff

La herramienta diff permite comparar dos imágenes. El código de la misma se encuentra en `helper`, y se compila junto con el resto del trabajo práctico. El ejecutable, una vez compilado, se almacenará en `build/bmpdiff`.

La aplicación se utiliza desde la línea de comandos de la forma:

```
$ ./build/bmpdiff [opciones] <archivo_1> <archivo_2> <epsilon>
```

Compara los dos archivos según las componentes de cada píxel, siendo `epsilon` la diferencia máxima permitida entre píxeles correspondientes a las dos imágenes. Tiene dos opciones: listar las diferencias o generar imágenes blanco y negro por cada componente, donde blanco marca que hay diferencia y negro que no.

Las opciones soportadas por el programa son:

`-i`, `--image`	Genera imágenes de diferencias por cada componente.
`-v`, `--verbose`	Lista las diferencias de cada componente y su posición en la imagen.
`-a`, `--value`	Genera las imágenes mostrando el valor de la diferencia.
`-s`, `--summary`	Muestra un resumen de diferencias.

Tests

Para verificar el funcionamiento correcto de los filtros, además del comparador de imágenes, se provee de un binario con la solución de la cátedra y varios scripts de test. El binario de la cátedra se encuentra en la carpeta test/.

El comparador de imágenes se ubica en la carpeta src/helper, y debe compilarse antes de correr los scripts.

Para ejecutar los script, en primer lugar, se deben generar las imagenes de prueba. Esto se realiza ejecutando el script `1_generar_imagenes.py`. Para que este script funcione correctamente se requiere la utilidad convert que se encuentra en la biblioteca imagemagick.[2].

Luego, el script `2_test_diff_cat_asm.py`, chequea diferencias entre las imagenes generadas por el binario de la cátedra y su binario. Este script prueba solamente por diferencias el resultado final.

Por último, el script `3_correr_test_mem.sh`, chequea el correcto uso de la memoria. Al demorar mucho tiempo, este test se ejecuta solamente para los casos más chicos.

3.5. Mediciones de rendimiento

La forma de medir el rendimiento de nuestras implementaciones se realizará por medio de la toma de tiempos de ejecución. Como los tiempos de ejecución son muy pequeños, se utilizará uno de los contadores de performance que posee el procesador.

La instrucción de assembler `rdtsc` permite obtener el valor del Time Stamp Counter (TSC) del procesador. Este registro se incrementa en uno con cada ciclo del procesador. Obteniendo la diferencia entre los contadores antes y después de la llamada a la función, podemos obtener la cantidad de ciclos de esa ejecución. Esta no es siempre igual entre distintas invocaciones de la función, ya que el registro es global al procesador y se ve afectado por una serie de factores.

Existen principalmente distintas problemáticas a solucionar:

a) La ejecución puede ser interrumpida por el *scheduler* para realizar un cambio de contexto. Esto implicaría contar muchos más ciclos (*outliers*) que si nuestra función se ejecutase sin interrupciones.

b) Los procesadores modernos varían su frecuencia de reloj, por lo que la forma de medir ciclos cambia dependiendo del estado del procesador.

c) El comienzo y fin de la medición deben realizarse con la suficiente exactitud como para que se mida solamente la ejecución de los filtros, sin ser afectada por ruidos como la carga o el guardado de las imágenes.

Para medir tiempos deberán idear e implementar una metodología que les permita evitar estos tres problemas. En el archivo `tp2.c` se provee código para realizar una medición de tiempo básica. El mismo podrá ser modificado para mejorar y automatizar las mediciones.

[2]Para instalar sudo apt-get install imagemagick

4. Ejercicios

Se deberá implementar el código de los filtros, realizar un análisis de su performance y presentar un informe de los resultados.

4.1. Implementación

Deberán implementar todos los filtros mencionados en lenguaje ensamblador, utilizando instrucciones SSE y respetando para cada uno una cantidad mínima a procesar de píxeles en simultáneo:

- **Cuadrados**: 4 píxeles en simultáneo.
- **Manchas**: 2 píxeles en simultáneo.
- **Offset**: 4 píxeles en simultáneo.
- **Sharpen**: 2 píxeles en simultáneo.
- **Ruido**: 4 píxeles en simultáneo (optativo).

La implementación de los filtros que sólo realicen cálculos con números enteros no podrá perder precisión. Es posible que se deseen realizar optimizaciones a costo de perder algo de precisión. Esto será aceptable siempre y cuando se analice también la calidad de la imagen resultante en la versión optimizada contra la no optimizada.

4.2. Análisis

Las siguientes preguntas pueden ser usadas como guía. La evaluación del trabajo práctico no sólo consiste en responderlas, sino en desarrollar y responder nuevas sugeridas por ustedes mismos, buscando entender y razonar sobre el modelo de programación SIMD y la microarquitectura del procesador.

- ¿Cuál implementación es "mejor"?
- ¿Qué métricas se pueden utilizar para calificar las implementaciones y cuantificarlas?
- ¿En qué casos? ¿De qué depende? ¿Depende del tamaño de la imagen? ¿Depende de la imagen en sí? ¿De los parámetros?
- ¿Cómo se podrían mejorar las métricas de las implementaciones propuestas? ¿Cuáles no se pueden mejorar?
- ¿Es una comparación justa? ¿De qué depende la velocidad del código C? ¿Cómo puede optimizarse?
- ¿Cuál es la cantidad de instrucciones ejecutadas por píxel en cada implementación? ¿Y de accesos a memoria? ¿Se condice empíricamente esta diferencia en la performance de los filtros?
- ¿Hay diferencias en operar con enteros o punto flotante? ¿La imagen final tiene diferencias significativas?
- ¿El overhead de llamados a funciones es significativo? ¿Se puede medir?

- ¿Las limitaciones de performance son causadas por los accesos a memoria? ¿O a la memoria cache? ¿Se podría acceder mejor a esta?
- ¿Y los saltos condicionales? ¿Afectan la performance? ¿Es posible evitarlos total o parcialmente?
- ¿El patrón de acceso a la memoria es desalineado? ¿Hay forma de mejorarlo? ¿Es posible medir cuánto se pierde?

4.3. Informe

El informe debe incluir las siguientes secciones:

a) **Carátula**: Contiene

- Número / nombre del grupo
- Nombre y apellido de cada integrante
- Número de libreta y mail de cada integrante

b) **Introducción**: Describe lo realizado en el trabajo práctico.

c) **Desarrollo**:
Describe cada una de las funciones que implementaron. Para la descripción de cada función deberán decir cómo opera una iteración del ciclo de ella. Es decir, cómo mueven los datos a los registros, cómo los reordenan para procesarlos, las operaciones que se aplican a los datos, etc. Además se agregará un detalle más profundo de las secciones de código que consideren más importantes. Para esto pueden utilizar pseudocódigo, diagramas (mostrando gráficamente el contenido de los registros `XMM`) o cualquier otro recurso que le sea útil para describir la adaptación del algoritmo al procesamiento simultáneo SIMD. No se deberá incluir el código assembler de las funciones (aunque se pueden incluir extractos en donde haga falta).

d) **Resultados**:
Deberán **analizar** y **comparar** las implementaciones de las funciones en su versión **C** y **assembler** y mostrar los resultados obtenidos a través de tablas y gráficos. Para esto deberán plantear experimentos que les permitan comprobar las diferencias de performance e hipotetizar sobre sus causas.
Deberán además explicar detalladamente los resultados obtenidos y analizarlos. En el caso de encontrar anomalías o comportamientos no esperados deberán construir nuevos experimentos para entender qué es lo que sucede.
Utilizar como guía para la realización de experimentos las preguntas de la sección anterior. Al responder estas preguntas (y otras que vayan surgiendo), se deberán analizar y comparar las implementaciones de cada función en su versión `C` y `ASM`, mostrando los resultados obtenidos a través de tablas y gráficos. También se deberán *comentar* los resultados obtenidos. En el caso de que sucediera que la versión en C anduviese más rápido que su versión ASM, **justificar** a qué se debe esto.

> Deberan entregar comparaciones de tiempo entre la versión C proporcionada por la cátedra, compilada con máximas optimizaciones (o3) y la versión ASM.

En caso de considerarlo necesario, pueden alterar las implementaciones en C de la cátedra por sus propias implementaciones buscando una solución más eficiente.

e) **Conclusión**: Reflexión final sobre los alcances del trabajo práctico, la programación vectorial a bajo nivel, problemáticas encontradas, y todo lo que consideren pertinente.

El informe no puede exceder las **20** páginas, sin contar la carátula.

Importante: El informe se evalúa de manera independiente del código. Puede reprobarse el informe, y en tal caso deberá ser reentregado para aprobar el trabajo práctico.

5. Entrega y condiciones de aprobación

El presente trabajo es de carácter **grupal**, siendo los grupos de **3 personas**, pudiendo ser de 2 personas en casos excepcionales con previa consulta y confirmación del cuerpo docente.

La fecha de entrega de este trabajo es **Jueves 9/5**. Deberá ser entregado a través de un repositorio GIT almacenado en `https://git.exactas.uba.ar` respetando el protocolo enviado por mail y publicado en la página web de la materia. La fecha límite para la **reentrega** es el día **Martes 11/6.**

Ante cualquier problema con la entrega, comunicarse por mail a la **lista de docentes**.