Trabajo Práctico 2

Organización del Computador II

Segundo Cuatrimestre 2013

1. Introducción

El objetivo de este trabajo práctico es explorar el modelo de programación **SIMD**. Una aplicación popular del modelo SIMD es el procesamiento de imágenes y video.

En este trabajo práctico se implementarán filtros para estas aplicaciones, utilizando lenguaje ensamblador (ASM) e instrucciones **SSE**, y se analizará la performance del procesador al hacer uso de **SIMD** para estas aplicaciones, comparándolas con sus implementaciones respectivas en lenguaje **C**.

Los filtros a implementar se describen a continuación.

1.1. Filtro de Color

El filtro de color se aplica pixel a pixel en una imágen en color. La función recibe como parámetros: rc, gc, bc.

Usando esto, el filtro de color se puede describir mediante la siguiente función partida:

r, g, y b representan el valor del canal rojo, verde y azul, respectivamente en, el i-ésimo pixel.

$$\mathsf{dst}_{[i]} = \begin{cases} \dfrac{r+g+b}{3} & \text{si } \mathsf{distancia}((r,g,b),(rc,gc,bc)) > \mathsf{threshold} \\ \mathsf{src}_{[i]} & \text{en otro caso} \end{cases}$$

$$\text{donde } \mathsf{distancia}((r,g,b),(rc,gc,bc)) = \sqrt{(r-rc)^2+(g-gc)^2+(b-bc)^2}$$

y threshold representa un valor de tolerancia máxima.

Esto quiere decir que los píxeles cuyo color sea "cercano" a los valores pasados como parámetros (es decir, la distancia es menor o igual al threshold) se mantendrá del mismo color, y el resto se convertirá a blanco y negro.

Notar, para las optimizaciones, que no es necesario seguir el procedimiento al pie de la letra. Lo importante es que el resultado final sea el mismo.

1.2. Miniature

El filtro de Miniature (*tilt-shift*) toma una imagen de un video y aplica un efecto que hace que parezca una imagen de una maqueta de juguete. La intensidad del efecto se basa en la cantidad de veces que se aplica el filtro.

La imagen se separa en 3 bandas horizontales, la superior, la media y la inferior, donde cada una abarca ciertas filas consecutivas f. Los límites de las bandas son configurables por parámetro.

$$\begin{aligned}
\textbf{banda superior:} \quad & 0 \leq f < \mathsf{topPlane} * \mathsf{height} \\
\textbf{banda media:} \quad & \mathsf{topPlane} * \mathsf{height} \leq f < \mathsf{bottomPlane} * \mathsf{height} \\
\textbf{banda inferior:} \quad & \mathsf{topPlane} * \mathsf{height} \leq f < \mathsf{height}
\end{aligned}$$

El efecto se compone de dos desenfoques gaussianos aplicados a la imagen, uno en la banda superior y otro en la banda inferior.

Definimos la matriz de transformación M como:

$$M = \begin{bmatrix} 0{,}01 & 0{,}05 & 0{,}18 & 0{,}05 & 0{,}01 \\ 0{,}05 & 0{,}32 & 0{,}64 & 0{,}32 & 0{,}05 \\ 0{,}18 & 0{,}64 & 1 & 0{,}64 & 0{,}18 \\ 0{,}05 & 0{,}32 & 0{,}64 & 0{,}32 & 0{,}05 \\ 0{,}01 & 0{,}05 & 0{,}18 & 0{,}05 & 0{,}01 \end{bmatrix}$$

Para cada píxel en la posición i, j, se hace una cuenta (por separado) para cada canal de color, que es la siguiente:

$$\begin{aligned}
&\mathsf{S}_{[i-2][j-2]}\mathsf{M}_{[0][0]} + \mathsf{S}_{[i-2][j-1]}\mathsf{M}_{[0][1]} + \mathsf{S}_{[i-2][j]}\mathsf{M}_{[0][2]} + \mathsf{S}_{[i-2][j+1]}\mathsf{M}_{[0][3]} + \mathsf{S}_{[i-2][j+2]}\mathsf{M}_{[0][4]} + \\
&\mathsf{S}_{[i-1][j-2]}\mathsf{M}_{[1][0]} + \mathsf{S}_{[i-1][j-1]}\mathsf{M}_{[1][1]} + \mathsf{S}_{[i-1][j]}\mathsf{M}_{[1][2]} + \mathsf{S}_{[i-1][j+1]}\mathsf{M}_{[1][3]} + \mathsf{S}_{[i-1][j+2]}\mathsf{M}_{[1][4]} + \\
&\mathsf{S}_{[i][j-2]}\mathsf{M}_{[2][0]} + \mathsf{S}_{[i][j-1]}\mathsf{M}_{[2][1]} + \mathsf{S}_{[i][j]}\mathsf{M}_{[2][2]} + \mathsf{S}_{[i][j+1]}\mathsf{M}_{[2][3]} + \mathsf{S}_{[i][j+2]}\mathsf{M}_{[2][4]} + \\
&\mathsf{S}_{[i+1][j-2]}\mathsf{M}_{[3][0]} + \mathsf{S}_{[i+1][j-1]}\mathsf{M}_{[3][1]} + \mathsf{S}_{[i+1][j]}\mathsf{M}_{[3][2]} + \mathsf{S}_{[i+1][j+1]}\mathsf{M}_{[3][3]} + \mathsf{S}_{[i+1][j+2]}\mathsf{M}_{[3][4]} + \\
&\mathsf{S}_{[i+2][j-2]}\mathsf{M}_{[4][0]} + \mathsf{S}_{[i+2][j-1]}\mathsf{M}_{[4][1]} + \mathsf{S}_{[i+2][j]}\mathsf{M}_{[4][2]} + \mathsf{S}_{[i+2][j+1]}\mathsf{M}_{[4][3]} + \mathsf{S}_{[i+2][j+2]}\mathsf{M}_{[4][4]}
\end{aligned}$$

donde S corresponde a la imagen fuente para el canal de color dado. Finalmente, el valor del pixel deberá ser normalizado (es decir, el producto de la subimagen por la matriz M no deberá aumentar o disminuir la cantidad de brillo total de la submatriz).

El algoritmo realizará *cantidadDeIteraciones* de procesamiento sobre las bandas. En cada iteración deben procesarse gradualmente menos filas. Más especificamente, desde el centro deben restarse a la banda un total de

$$\frac{\text{iteracion actual} \cdot \text{cantidadDeFilasDeLaBanda}}{\text{cantidadDeIteraciones}}$$

Donde el contador de iteraciones debe ir de 0 a cantidadDeIteraciones-1. Además deberá copiarse la imagen *dst* en la imagen *src* al final de cada iteración con el fin de incrementar el efecto de blur en el resto de la banda.

Las 2 primeras y 2 últimas filas y columnas no se procesan.

Nota: Para la implementación en ASM, plantear y experimentar que es lo más conveniente para guardar en memoria, en que parte de la memoria, y que conviene guardar en registros. Documentar toda decisión de diseño tomada, dejando bien en claro como operará el filtro y las distintas partes del algoritmo.

Nota: Quizás no sea necesario operar con la matriz por completo.

1.3. Decodificación Esteganográfica

Es posible enviar mensajes ocultos embebidos en las imágenes y videos. Se sabe que el ojo humano no puede percibir fácilmente diferencias pequeñas en los distintos colores de una imágen. Por este motivo, se pueden utilizar los bits menos significativos de los valores de los píxeles de color para poder embeber algún mensaje secreto.

Para realizar la decodificación, se procesará de a 4 bytes de la imagen por cada caracter. De cada uno de esos 4 bytes, deberán tomarse los dos bits menos significativos (code) para formar un byte. O sea que se utilizarán 4 colores por cada byte decodificado.

El algoritmo para decodificar tambien utiliza el tercer y cuarto bit menos significativo de cada byte (op) para saber como interpretar los dos primeros bits. Por esto, dependiento del valor de estos dos bits se realiza lo siguiente:

- 00 : Quedan iguales
- 01 : Se suma uno
- 10 : Se resta uno
- 11 : Se niega el valor

La formulación matemática es la siguiente:

$$\text{char}(\text{src}, i) = \phi_0(\text{src}[i * 4]) \mid \phi_1(\text{src}[i * 4 + 1]) \mid \phi_2(\text{src}[i * 4 + 2]) \mid \phi_3(\text{src}[i * 4 + 3])$$

donde

$$\phi_i(\text{byte}) = (\delta(\text{byte}_{\text{code}}, \text{byte}_{\text{op}})) << (i \cdot 2)$$

y

$$\delta(\text{code}, \text{op}) = \begin{cases} (code & \text{si op } = 0 \\ (code + 1 \,\%4 & \text{si op } = 1 \\ (code - 1 \,\%4 & \text{si op } = 2 \\ (not(code) & \text{si op } = 3 \end{cases}$$

2. Enunciado

Además de explorar el modelo de programación **SIMD**, el objetivo será analizar la performance del procesador al hacer uso de las intrucciones **SSE**.

Para esto, cada uno de los filtros deberá ser implementado al menos en dos versiones: una en lenguaje **C**, y una en ASM haciendo uso de las instrucciones **SSE**.

Los ejercicios que se enumeran a continuación son obligatorios y sirven como guia mínima para que realicen las optimizaciones y el analisis de las mismas. Tambien deberán plantear optimizaciones que surjan del desarrollo y decisiones puntuales del grupo en cada ejercicio y acompañarlas de un respectivo analisis.

Finalmente, se compararán ambas versiones analizando las mejoras de performance obtenidas.

2.1. Ejercicios

Ejercicio 1
Programar el *Filtro de Color* en lenguaje C.

Ejercicio 2
Utilizando el código C del ejercicio anterior como pseudocódigo, programar el *Filtro de Color* en ASM haciendo uso de las instrucciones **SSE**.

Nota: No intentar realizar el código ASM directamente, utilizar el código C anterior como pseudocódigo para realizar el código ASM. Si el código C no parece servir para este objetivo, es fuertemente recomendado replantearse el código realizado.

Ejercicio 3
¿Cuáles son las diferencias estructurales entre la versión C y la de ASM de los ejercicios anteriores? ¿Qué cambia escencialmente entre las dos versiones? Utilice la herramienta `objdump` para verificar como el compilador de C deja ensamblado el código C. Como es el código generado, ¿como se manipulan las variables locales?¿le parece que ese código generado podria optimizarse?

Ejercicio 4
Compile el código del ejercicio 1 con optimizaciones del compilador, por ejemplo, pasando el flag `-O1`. ¿Que optimizaciones encuentra?¿Que otras flags de optimización brinda el compilador?¿Para que sirven?

Ejercicio 5
Realice una medición de las diferencias de performance entre las versiones de los ejercicios 1, 2 y 4 (este último con -O1, -O2 y -O3).
¿Como realizó la medición?¿Como sabe que su medición es una buena medida?¿Como afecta a la medición la existencia de outliers[1]?¿De que manera puede minimizar su impacto?¿Que resultados obtiene si mientras corre los tests ejecuta otras aplicaciones que utilicen al máximo la CPU? Realizar un gráfico que represente estas diferencias y un análisis científico de los resultados.

Ejercicio 6
Se desea conocer que tanto impactan los saltos condicionales en el código del ejercicio 4 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

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

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

Ejercicio 7

La técnica para *desenrollar ciclos* (*loop unrolling*, *loop unwinding*) es muy útil a la hora transformar los ciclos para optimizar la ejecución del código.

Estudiar esta técnica y proponer un aplicación al código del Ejercicio 2.

Utilizando el código ASM del Ejercicio 2, programar el *Filtro de Color* en ASM haciendo uso de las instrucciones **SSE** y la técnica de *desenrollado de ciclos*.

Ejercicio 8

¿Cuáles son las diferencias de performace entre las versiones de los Ejercicios 1, 2 y 7?

Realizar gráficos que representen estas diferencias y permitan realizar este análisis.

Ejercicio 9

Debido a las operaciones necesarias para llevar adelante el filtro, quizás deban transformarse los tipos de datos, ya sea por extensión/compresión de la representación, por ejemplo char/int, o por conversión de la precisión, por ejemplo int/float.

Indicar en qué momento del código del filtro implementado en el Ejercicio 2 es más conveniente realizar estas conversiones, tanto para manejar los datos de entrada como para devolver los resultados.

¿Qué otras optimizaciones proponen?

Ejercicio 10

Programar el filtro *Miniature* en lenguaje C.

Ejercicio 11

Utilizando el código C del ejercicio anterior como pseudocódigo, programar el filtro *Miniature* en ASM haciendo uso de las instrucciones **SSE**.

Ejercicio 12

¿Cuáles son las diferencias estrucutrales entre la versión en C y la de ASM de los Ejercicios 10 y 11?

¿Qué cambia escencialmente entre las dos versiones?

¿Cuáles son las diferencias de performace entre esta dos versiones?

Realizar gráficos que representen estas diferencias y permitan realizar este análisis.

Ejercicio 13

¿Las diferencias anteriores se mantienen con distintos tamaños de entrada?

¿Influye en estas diferencias la multiplicidad de las filas y columnas de la entrada?

¿Cuales son los aspectos del filtro que encontrarón más prioritarios para optimizar y porqué? ¿Los analisis finales lo confirman?

Ejercicio 14

En el caso de existir diferencias, ¿a qué se deben? ¿cuál es su origen? ¿Tienen alguna relación con las diferencias estrucutrales entre la versión C y ASM de los Ejercicios 12 y 13? Justificar.

Ejercicio 15

Programar el filtro *Decodificación Esteganográfica* en lenguaje C.

Ejercicio 16

Utilizando el código C del ejercicio anterior como pseudocódigo, programar el filtro *Decodificación Esteganográfica* en ASM haciendo uso de las instrucciones **SSE**.

Ejercicio 17

¿Cuáles son las diferencias estructurales entre la versión C y la de ASM de los Ejercicios 15 y 16?
¿Qué cambia escencialmente entre las dos versiones?
¿Cuáles son las diferencias de performace entre esta dos versiones?
¿Estas diferencias se mantiene con distintos tamaños de entrada?
¿Influye la multiplicidad de las filas y columnas de la entrada?

Realizar gráficos que representen estas diferencias y permitan realizar este análisis.

Ejercicio 18

La técnica de *entubado de código* (*software pipelining*, *out-of-order execution*) es otra técnica muy utilizada para optimizar ciclos realizando ejecución fuera de orden y aprovechando los recursos de harware del procesador.

Estudiar esta técnica y proponer una aplicación al código del Ejercicio 16.

Utilizando el código ASM del Ejercicio 16, programar el filtro *Decodificación Esteganográfica* en ASM haciendo uso de las instrucciones **SSE** y la técnica de *entubado de código*.

Proponer optimizaciones que surjan de su desarrollo y decisiones puntuales que hayan tomado. Hacer un analisis de las mismas.

Ejercicio 19 ¿Cuáles son las diferencias de performace entre las versiones de los Ejercicios 15, 16 y 18?

Realizar gráficos que representen estas diferencias y permitan realizar este análisis.

Ejercicios opcionales

Las limitaciones de performance suelen dividirse en dos grandes áreas: capacidad de cómputo y ancho de banda. Es decir, la limitación que hace que un programa no se ejecute más rápidamente puede ser el tiempo que se tarda en procesar cada dato o el tiempo en que tarda en recibir o enviar los datos desde y hacia la memoria. Si un algoritmo realiza muchos cómputos y accede poco a memoria, tardará mas en procesar cada dato que en recibirlo de memoria, y por lo tanto estará limitado por la capacidad de cómputo. Si en cambio, realiza poco procesamiento pero con grandes cantidades de datos, tardará más en la transmisión de los datos que en su procesamiento, y por lo tanto estará limitado por el ancho de banda de la conexión con la memoria.

Determinar para cada uno de los tres filtros si está inherentemente limitado por la capacidad de cómputo o por el ancho de banda disponible, y en que grado.

Determinar como afecta la performance la cache del procesador en el que corre los tests.

2.2. Código

Para implementar los filtros descriptos anteriormente, tanto en C como en ASM se deberán implementar las siguientes funciones para imágenes en color (24 bits):

- *void color_filter (unsigned char *src, unsigned char *dst, unsigned char rc, unsigned char gc, unsigned char bc, int width, int height);*
- *void miniature (unsigned char *src, unsigned char *dst, int width, int height, float topPlane, float bottomPlane, int iters);*
- *void decode (unsigned char *src, unsigned char *code, int width, int height);*

- *src*: Es el puntero al inicio de la matriz de elementos de 24 bits sin signo (el primer byte corresponde al canal azul de la imagen (B), el segundo el verde (G) y el tercero el rojo (R)) que representa a la imagen de entrada. Es decir, como la imagen está en color, cada píxel está compuesto por 3 bytes.
- *dst*: Es el puntero al inicio de la matriz de elementos de 24 bits sin signo que representa a la imagen de salida.
- *height*: Representa el alto en píxeles de la imagen, es decir, la cantidad de filas de las matrices de entrada y salida.
- *width*: Representa el ancho en píxeles de la imagen, es decir, la cantidad de columnas de las matrices de entrada y salida.
- *rc*: Representa un valor arbitrario para canal rojo.
- *gc*: Representa un valor arbitrario para canal verde.
- *bc*: Representa un valor arbitrario para canal azul.
- *topPlane*: Es un numero flotante entre 0.0 y 1.0, que representa el porcentaje de la imagen que va a ser considerada banda superior para la función Miniature.
- *bottomPlane*: Es un numero flotante entre 0.0 y 1.0, que representa el porcentaje de la imagen que va a ser considerada banda inferior para la función Miniature.
- *iters*: Cantidad de veces que se va a aplicar el desenfoque gaussiano para la parte superior e inferior de la imagen.
- *code*: Es el puntero a un arreglo de caracteres donde se va a obtener el mensaje decodificado de la imagen.

2.2.1. 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.
- No se debe perder precisión en ninguno de los cálculos.
- 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, deberán trabajar con **al menos 2 píxeles simultámeamente**.

  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.
- El TP se tiene que poder ejecutar en las máquinas del laboratorio.

2.3. Desarrollo

Para facilitar el desarrollo del trabajo práctico se cuenta con todo lo necesario para poder compilar y probar las funciones que vayan a implementar.

Dentro de los archivos presentados deben completar el código de las funciones pedidas. Puntualmente encontrarán el programa principal (de línea de comandos), denominado **tp2**, que se ocupa de parsear las opciones ingresadas por el usuario y ejecutar el filtro seleccionado sobre la imagen ingresada.

Para la manipulación de los videos (cargar, grabar, etc.) el programa hace uso de la biblioteca **OpenCV**, por lo que no se requiere implementar estas funcionalidades.

Para instalar las dependencias necesarias, en las distribuciones basadas en **Debian** basta con ejecutar:

```
$ sudo apt-get install libcv-dev libhighgui-dev libcvaux-dev libc6-dev
```

Los archivos entregados están organizados en las siguientes carpetas:

- *bin* : Contiene el ejecutable del TP.
- *data* : Contiene videos de prueba.
- *enunciado* : Contiene este enunciado.
- *src* : Contiene los fuentes del programa principal, junto con su respectivo **Makefile** que permite compilar el programa.
- *test*: Contiene scripts para realizar tests sobre los filtros y uso de la memoria.

El uso del programa principal es el siguiente:

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

Soporta los tipos de videos más comunes y acepta las siguientes opciones:

- Los filtros que se pueden aplicar son:

  - **fcolor**
    Parámetros: `r` intensidad del color Rojo [0, 255]
    `g` intensidad del color Verde [0, 255]
    `b` intensidad del color Azul [0, 255]
    `threshold` distancia tolerable hacia ese color
    Ejemplo de uso: *fcolor -i c tree.avi 0 230 0 4000*
  - **miniature**
    Parámetros: `topPlane` parte superior de la pantalla a blurear [0, 1)
    `bottomPlane` parte inferior de la pantalla a blurear [0, 1)
    `iteraciones` cantidad de Iteraciones
    Ejemplo de uso: *miniature -i c tree.avi 0.3 0.8 3*
  - **decode**
    Parámetros:
    Ejemplo de uso: *decode -i c tree.avi*

- *-h, –help*
  Imprime la ayuda
- *-i, –implementacion NOMBRE_MODO*
  Implementación sobre la que se ejecutará el proceso seleccionado. Los 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
- *-f, –frames CARPETA*
  Genera frames independientes en CARPETA en vez de armar un archivo de video. Es utilizado para testing.
- *-v, –verbose*
  Imprime información adicional

Por ejemplo:

$ ./tp2 -v fcolor -i c tree.avi 0 230 0 30000

Aplica el **Filtro de Color** al video tree utilizando la implementación en lenguaje c del filtro, pasándole como parámetro el color representado por las componentes rgb iguales a (0, 230, 0) y teniendo una tolerancia máxima de 30000.

Si hacemos:

$ ./tp2 -t 1000 -i asm miniature tree.avi 0.2 0.8 1

Aplica el filtro **Miniature** al video tree dentro de un ciclo de **1000** iteraciones y devuelve la cantidad de **ticks** (ciclos de reloj del procesador) que insumió la aplicación del filtro. Esto será utilizado para comparar la performance de las versiones en C y assembler.

El ejecutable genera un video con los frames filtrados para poder apreciar el resultado más fácilmente. Sin embargo, el testing se hará sobre los frames generados y no sobre el video final.

2.3.1. Tests

Para verificar el correcto funcionamiento de los filtros, se provee el script `run_tests.sh` que se encuentra en la carpeta `solucion/tests`. El mismo verifica que los resultados de las versiones de C y Assembler sean iguales, que no haya problemas en el uso de la memoria y, por último, que los frames de los videos sean iguales a los generados por la cátedra.

En particular, para probar la funcion `decode`, se debe utilizar con la imagen `encoded.bmp`. Esto generará un archivo de texto plano con los datos desencriptados. En los scripts de testing esta el hash MD5 de la salida de la cátedra. Si esta coincide con el hash MD5 de la salida generada por el TP, los datos obtenidos serán considerados correctos.

2.3.2. Mediciones de tiempo

Utilizando la instrucción de assembley `rdtsc` podemos obtener el valor del Time Stamp Counter (TSC) del procesador. Este registro se incrementa en uno con cada ciclo del procesador. Restando el valor del registro antes de llamar a una función a su valor luego de la llamada, podemos obtener la duración en ciclos de esa ejecución.

Las macros para medir tiempo se encuentran en el archivo `tiempo.h`. Para usarlas, se debe determinar como y donde implementarlas en el `tp2.c` para poder obtener las mediciones más exactas de tiempo con granularidad de ciclo de clock.

Esta cantidad de ciclos no es siempre igual entre invocaciones de la función, ya que este registro es global del procesador y, si el programa es interrumpido por el *scheduler* para realizar un cambio de contexto, contaremos muchos más ciclos que si la función se ejecutara sin interrupciones. Por esta razón el programa principal del TP permite especificar una cantidad de iteraciones para repetir el filtro, con el objetivo de suavizar este tipo de outliers.

2.4. 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 *en profundidad* cada una de las funciones que implementaron, respondiendo cada una de las preguntas de los *Ejercicios*.

Para la descripción de cada función deberán decir cómo opera una iteración del ciclo de la función. Es decir, cómo mueven los datos a los registros, cómo los reordenan para procesarlos, las operaciones que se aplican a los datos, etc. 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).

Las preguntas en cada ejercicio son una guía para la confección de los resultados obtenidos. Al responder estas preguntas, se deberán analizar y comparar las implementaciones de cada funciones en su versión `C` y `ASM`, mostrando los resultados obtenidos a través de tablas y gráficos.

También se deberá comentar acerca de los resultados obtenidos. En el caso de que sucediera que la versión en C anduviese más rápidamente que su versión ASM, **justificar fuertemente** a qué se debe esto.

d) **Conclusión** Reflexión final sobre los alcances del trabajo práctico, la programación en el modelo **SIMD** a bajo nivel, problemáticas encontradas, y todo lo que consideren pertinente.

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.

3. Entrega

El presente trabajo es de carácter **grupal**, siendo los grupos de **3 personas**, pudiendo ser de 2 personas en casos excepcionales previa consulta y confirmación del cuerpo docente. Se deberá entregar un archivo comprimido con el mismo contenido que el dado para realizarlo, habiendo modificado solo los archivos que tienen como nombre las funciones a implementar.

La fecha de entrega de este trabajo es **Martes 24 de Septiembre** y deberá ser entregado a través de la página web. El sistema solo aceptará entregas de trabajos hasta las **17:00hs** del día de entrega.

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