ARQUITECTURAS MULTIMEDIA
3º ITI SISTEMAS

PRÁCTICA 1
PROGRAMACIÓN DE EXTENSIONES MULTIMEDIA

Marzo de 2010
Dpto. Ingeniería y Tecnología de Computadores
Universidad de Murcia

PROGRAMACIÓN DE EXTENSIONES MULTIMEDIA

Convocatoria de Junio

Fecha de entrega de las prácticas: Lunes 7 de Junio de 2010

REQUISITOS DE ENTREGA

- **Memoria impresa** que incluya:
  - Portada con el título de la práctica y los componentes del grupo.
  - Índice.
  - Explicación de aquellos aspectos **relevantes** de cada uno de los ejercicios.
  - **Listado impreso comentado** del código C/C++ para cada uno de los ejercicios.
  - Bibliografía, si la hubiere.
- Se adjuntará a la memoria un disquete de $3^{1/2}$ con el código fuente de los ejercicios. El disquete debe ir rotulado con el nombre de los componentes del grupo.
- Importante: Se valorará la claridad, modularidad y corrección del código.
- Las prácticas que no se ajusten a las condiciones de entrega **NO SERÁN EVALUADAS**.

VALORACIÓN DE LA PRÁCTICA

- Esta práctica supone el **90 %** de la nota global de prácticas de la asignatura.
- La calificación de esta práctica se realizará de acuerdo con la siguiente ponderación:
  - Ejercicio 1: 20 %.
  - Ejercicio 2: 30 %.
  - Ejercicio 3: 50 %.

EJERCICIO 1: Transformada de Wavelet

Escribir en C o C++, y después compilar con `icc` o `icpc`, respectivamente, una aplicación que obtenga la *Transformada de Wavelet 1D Daub-4* de un determinado número de valores reales generados aleatoriamente.

- Requisitos funcionales obligatorios:
  - Debe ser posible, a través de una interfaz de línea de comandos, establecer: *a)* el número de valores a los que aplicarles la transformada; y *b)* una indicación de si hay que mostrar o no por pantalla los valores originales y sus transformaciones correspondientes. La omisión de estos parámetros supondrá, respectivamente, la adopción de los dos siguientes valores por defecto: `10000000` y `false`.
  - La aplicación debe mostrar, a su término, el tiempo invertido en el cálculo de la transformada [1].
- Requisitos internos obligatorios:
  - Utilizar las instrucciones SIMD [2] que se consideren apropiadas para mejorar el rendimiento de la aplicación. Para los números reales a los que se les calcule la transformada es suficiente una codificación *IEEE 754 de simple precisión*.
  - Utilizando el mismo código fuente ha de ser posible compilar dos aplicaciones: una que realice sólo cálculos escalares y otra que sustituya los cálculos escalares por instrucciones SIMD.
- Requisitos internos opcionales [3]:
  - Aparte de las instrucciones SIMD, se valorará la aplicación de cualquier técnica de optimización adicional. Su inclusión deberá estar claramente delimitada en el código fuente (mediante directivas), con objeto de poder verificar fácilmente la mejora que sobre el rendimiento tiene la implantación de esa/s técnica/s de optimización.
- Requisitos para la memoria:
  - Descripción del escenario en el que se realizan las ejecuciones (contextos hardware y software).
  - Indicación de los ficheros fuente, de las directivas de compilación utilizadas y del comando empleado para la compilación de cada aplicación.
  - Breve manual de usuario.
  - Exposición tabular de las pruebas de ejecución realizadas. En cada prueba es necesario indicar, al menos: *a)* la aplicación con la que se realiza [4]; *b)* los parámetros de entrada; *c)* el tiempo de ejecución invertido; y *d)* el *speed-up* [5] logrado.
  - Exposición de conclusiones y argumentación razonada de los resultados, tanto si las mejoras en el rendimiento son las esperadas como si no.

[1] Se pide el tiempo de ejecución de únicamente la función que calcula la Transformada de Wavelet. Cualquier otra parte del programa, como por ejemplo la generación de números aleatorios, debe quedar excluida.
[2] Extensiones multimedia.
[3] La implementación de cualquier característica que sea considerada como opcional, tanto si es sugerida en este enunciado como si es propuesta por el propio grupo, puede suponer un incremento en la calificación del ejercicio de hasta un 33 %.
[4] De forma obligatoria se tendrán en cuenta dos aplicaciones: La que efectúa cálculos meramente escalares y la versión SIMD. No obstante, el uso de técnicas de optimización adicionales también deberá tener su reflejo en la tabla, para así poder valorar cuantitativamente la mejoras logradas con dichas optimizaciones suplementarias.
[5] Tiempo_Versión_Escalar / Tiempo_Versión_Optimizada

EJERCICIO 2: Cálculo de integrales definidas

Escribir en C o C++, y después compilar con `icc` o `icpc`, respectivamente, una aplicación que obtenga el área situada por debajo de una función real de una única variable real, entre dos valores dados (integral definida).

- Requisitos funcionales obligatorios:
  - La función a considerar debe ser $f(x) = |sen(x)|$.
  - La integral ha de obtenerse por métodos numéricos, aplicando la *Regla del Punto Medio* [6], con una anchura de intervalo, *step*, parametrizable y con unos límites para la integral también parametrizables.
  - El tipo de dato que se utilice tanto para x como para $f(x)$ debe garantizar una precisión aceptable en los resultados. Por ejemplo, sabiendo que $\int_0^{\pi/2} |sen(x)|dx = 1$, el resultado de la prueba $\int_0^{2000\pi/2} |sen(x)|dx$ debería ser 2000 (y no 2001 ni 1999, por ejemplo).
  - Debe ser posible, a través de una interfaz de línea de comandos, establecer: *a)* los límites de la integral definida; y *b)* el valor de *step*. La omisión de estos parámetros supondrá, respectivamente, la adopción de los dos siguientes valores por defecto: `[0,2000]` y `0.00001`.
  - La aplicación debe mostrar, a su término, el tiempo invertido en el cálculo de la integral definida. Es decir, sólo se "cronometrará" la función que calcule las sumas parciales de área, y no la parte previa de la aplicación que introduzca en un array los valores de $f(x)$ intervinientes.
- Requisitos funcionales opcionales:
  - Con el mismo código fuente, usando directivas para generar ejecutables distintos o con un mismo ejecutable que permita comportamientos adaptados, hacer factible la amplicación del cálculo de áreas a las dos siguientes funciones: $f(x) = x^2$ y $f(x) = \frac{1}{x+1}$.
- Requisitos internos obligatorios: Los mismos que en el ejercicio 1.
- Requisitos internos opcionales: Los mismos que en el ejercicio 1.
- Requisitos para la memoria: Los mismos que en el ejercicio 1.

[6] Dividimos el intervalo en el que hay que calcular el área, $[a, b]$, en un conjunto de n subintervalos de igual tamaño, para después ir acumulando, a través de un bucle que va de a a b, los resultados de los productos $step \times f(x)$, siendo *step* la anchura de los subintervalos considerados.

EJERCICIO 3: Detección de bordes en imágenes

Escribir en C o C++, y después compilar con `icc` o `icpc`, respectivamente, una aplicación que, a partir de una imagen de entrada dada, genere como salida otra imagen, en blanco y negro, en la que se muestren únicamente los bordes de la imagen de entrada original.

- Requisitos funcionales obligatorios:
  - A través de una interfaz de línea de comandos debe indicarse, obligatoriamente, la ruta de un archivo *bmp* de entrada. Este *bmp* ha de contener una imagen en escala de grises (8 bits de profundidad de color) [7]. La aplicación debe construir, a partir de esta imagen de entrada, una imagen de salida con fondo blanco en la que sólo se muestren, en negro, los píxeles que marcan las siluetas, o bordes, de la imagen de entrada. No se pide mostrar por pantalla ninguna imagen (ni la de entrada ni la de salida), sino sólo crear un fichero *bmp* con la imagen de salida [8].

    A modo de ejemplo, se muestran a continuación una imagen de entrada y su salida correspondiente:

Figura 1: Imagen de entrada

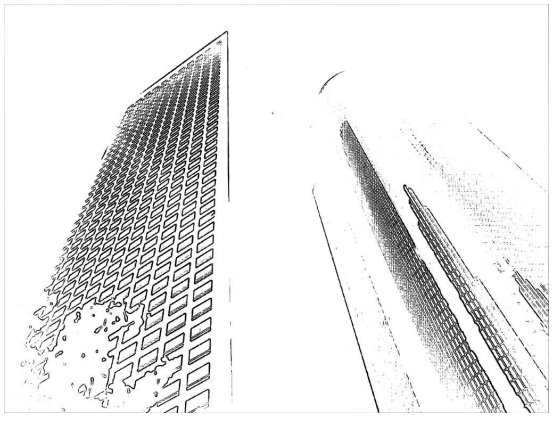

Figura 2: Imagen de salida

[7] Si no se dispone de imágenes de estas características, se puede recurrir a la aplicación *GIMP*, presente en todas las distribuciones Linux, para convertir cualquier imagen RGB de 24 bits de profundidad de color en otra imagen en escala de grises de 8 bits de profundidad de color.

[8] En el anexo 1 se expone el formato normalizado de los ficheros *bmp*. Una vez conocido dicho formato, resultará sencillo implementar la lectura de la imagen de entrada, así como el volcado final a disco de la imagen de salida.

El método para la detección de bordes es la fase inicial del *Detector de Bordes de Canny*. Concretamente, se pide cubrir las etapas siguientes:

1. Leer la imagen del fichero *bmp* de entrada, y alojarla en memoria de forma matricial, respetando la organización de los píxeles impuesta por el formato *bmp*.
2. Una vez leída la imagen de entrada, calcular su *Derivada en X según el Filtro de Sobel 3x3* y su *Derivada en Y según el Filtro de Sobel 3x3* [9].
3. Considerar, para cada píxel de la imagen de entrada, el módulo del vector gradiente asociado, cuyas componentes x e y son, respectivamente, las derivadas X e Y obtenidas en la etapa anterior. Si dicho módulo superase cierta cota, cuyo valor por defecto será **64** pero que en todo caso deberá ser parametrizable, pintar de negro el píxel correspondiente, y en caso contrario pintarlo de blanco [10].
4. Una vez actualizado el lienzo de salida, salvar en disco la imagen, respetando las normas del formato *bmp*.

- La aplicación debe mostrar, a su término, el tiempo estrictamente consumido por el proceso de detección de bordes. No se debe tener en cuenta el tiempo de lectura de la imagen de entrada, ni tampoco el tiempo de escritura de la imagen de salida [11].

- Requisitos funcionales opcionales:
  - Afinar la detección de bordes implementando las fases restantes del método de Canny (cálculo del argumento del vector gradiente, comparación con el nivel de gris de los píxeles "vecinos", etc).
- Requisitos internos obligatorios: Los mismos que en los ejercicios 1 y 2.
- Requisitos internos opcionales: Los mismos que en los ejercicios 1 y 2.
- Requisitos para la memoria: Los mismos que en los ejercicios 1 y 2.

[9] El anexo 2 contiene una explicación detallada acerca del filtro de Sobel y sobre el resto de fases del método de Canny.

[10] La expresión "pintar un píxel" no alude al hecho de representar gráficamente dicho píxel, sino a la actualización de su nivel de gris en el lienzo guardado en memoria.

[11] Para que la comparación de tiempos entre las distintas versiones de la aplicación resultante (con SIMD y sin SIMD, por ejemplo) sea lo suficientemente significativa, se recomienda trabajar con imágenes de entrada de gran tamaño, a partir de 3840 x 2880 (3840 columnas y 2880 filas).

ANEXO 1: Formato BMP

BMP, o *Windows Bitmpap*, es un formato de almacenamiento gráfico desarrollado por *Microsoft* en el que la imagen se guarda tal cual, es decir, sin recurrir a ningún algoritmo de compresión. No hay, por tanto, pérdida de calidad en las imágenes, y además su visualización por parte de los programas que las interpretan es muy rápida. Su gran tamaño, no obstante, las hace inadecuadas para transmisiones telemáticas.

Un fichero *bmp* consta de dos partes: cabecera y datos. La cabecera describe las características de la zona de datos (dimensiones de la imagen, puntero al primer píxel, número de colores, etc); la zona de datos almacena los *píxeles* que conforman el lienzo, o *canvas*, de la imagen. Estos píxeles se guardan de forma contigua, de abajo a arriba y de izquierda a derecha, siendo el primer píxel la esquina inferior izquierda, y el último la superior derecha.

A continuación se listan los campos más significativos de una cabecera *bmp*:

Bytes	**Contenido**
0, 1	Identificación del tipo de fichero *bmp*. Su contenido es, generalmente, `BM`.
10, 11, 12, 13	Nº byte en donde comienza el primer píxel de la imagen. A la hora de obtener ese puntero, debe considerarse como valor menos significativo el byte 10. Por tanto, el puntero sería $[10] + 256 \times [11] + 65536 \times [12] + 16777216 \times [13]$. Este principio es aplicable a todos aquellos campos que consuman más de 1 byte y que almacenen un valor numérico.
18, 19, 20, 21	Anchura del *bmp*, medida en número de píxeles.
22, 23, 24, 25	Altura del *bmp*, medida en número de píxeles.
28, 29	Nº de bits necesarios para guardar el color de cada píxel. Valores más habituales: 8 (escala de grises, un único canal) y 24 (entrelazado de los canales *Blue*, *Green* y *Red*, con 1 byte para cada color de cada píxel).

Cuadro 1: Cabecera de un fichero *bmp*

Los posibles colores de un píxel se codifican dentro del intervalo $[0, 255]$. En imágenes en escala de grises, con un único canal, el valor 0 representa el color negro y el valor 255 el color blanco.

En imágenes de 24 bits de profundidad de color, esto es, imágenes con tres canales, el color de cada píxel requiere tres bytes, siendo el primero de ellos su nivel de *blue*, el segundo su nivel de *green* y el tercero su nivel de *red*. De este modo, por ejemplo, la terna $(0, 0, 0)$ representa el color negro, $(255, 0, 0)$ el azul puro, $(0, 255, 0)$ el verde puro, $(0, 0, 255)$ el rojo puro y $(255, 255, 255)$ el color blanco.

Ejemplo 1

Imagen en escala de grises, de dimensiones 640×480 (640 columnas y 480 filas). La zona de datos comienza en el byte señalado por el puntero `lienzo` [12].

- Valor de gris del píxel (0,0) -columna 0 y fila 0- = lienzo[0].
- Valor de gris del píxel (105,4) -columna 105 y fila 4- = lienzo[$(4 \times 640) + 105$] = lienzo[2665].
- Valor de gris del píxel (639,479) -esquina superior derecha- = lienzo[$(479 \times 640) + 639$] = lienzo[307199].

Ejemplo 2

Imagen de 24 bits, de dimensiones 640×480. La zona de datos comienza en el byte señalado por el puntero `lienzo`.

- Valores de color del píxel (0,0) = lienzo[0] para el azul, lienzo[1] para el verde y lienzo[2] para el rojo.
- Valores de color del píxel (105,4) = lienzo[$3 \times ((4 \times 640) + 105)$] = lienzo[7995] para el azul; lienzo[$(3 \times ((4 \times 640) + 105)) + 1$] = lienzo[7996] para el verde; y lienzo[$(3 \times ((4 \times 640) + 105)) + 2$] = lienzo[7997] para el rojo.
- Valores de color del píxel (639,479) = lienzo[$3 \times ((479 \times 640) + 639)$] = lienzo[921597] para el azul; lienzo[$(3 \times ((479 \times 640) + 640)) + 1$] = lienzo[921598] para el verde; y lienzo[$(3 \times ((479 \times 640) + 640)) + 2$] = lienzo[921599] para el rojo.

[12] Declarado como `unsigned char *lienzo`

ANEXO 2: Detector de Bordes de Canny

El algoritmo descrito por Jhon Canny en 1986 para detectar bordes sigue siendo actualmente uno de los métodos de localización de bordes más eficaz [13] y eficiente. Toma como entrada una imagen en escala de grises [14] y proporciona como salida una imagen con píxeles blancos ó negros, en donde un píxel negro representa un punto de un borde, y uno blanco cualquier otro punto de la imagen (también se puede hacer al revés, el resultado final sería equivalente, con la única diferencia de que obtendríamos bordes blancos sobre fondo negro).

Su idea central es muy simple: Si concebimos la imagen cuyos bordes queremos obtener no como una representación gráfica bidimensional, como sería lo habitual, sino como la representación gráfica de una función matemática de dos variables enteras, x e y, siendo estas variables las coordenadas cartesianas de cada píxel y $f(x, y)$ su correspondiente valor de gris, es muy sencillo extraer los bordes si nos limitamos a marcar sobre el lienzo aquellos píxeles, (x, y), para los que el módulo del gradiente de $f(x, y)$ supera una determinada cota mínima [15].

A continuación describimos los pasos básicos del *Detector de Bordes de Canny* [16]:

1. Para cada píxel de la imagen,

   - Calcular la *Derivada parcial respecto a X por Sobel* 3×3, aplicando la siguiente máscara de convolución:

-1	0	1
-2	0	2
-1	0	1

Cuadro 2: Filtro de Sobel 3×3, derivada X

     La anterior matriz debe interpretarse como sigue: El píxel central es el punto al que se le calcula la derivada. Ésta es el resultado de sumar los productos parciales obtenidos al multiplicar el valor de gris de cada vecino por su factor correspondiente de la matriz de convolución (en total 8 productos, o 6 si excluímos los dos productos afectados por el factor 0).

   - Calcular la *Derivada parcial respecto a Y por Sobel* 3×3, aplicando la siguiente máscara de convolución:

-1	-2	-1
0	0	0
1	2	1

Cuadro 3: Filtro de Sobel 3×3, derivada Y

[13] Detecta bordes correctos y no marca como bordes límites inexistentes.
[14] La gama de valores de gris va, por tanto, del 0 (negro) al 255 (blanco).
[15] El gradiente de una función bidimensional (aunque esta idea se puede generalizar a cualquier número de dimensiones) en un punto de su dominio es un vector cuya dirección indica hacia dónde la función experimenta una mayor variación de su valor (en este caso, ese *valor* es un *nivel de gris*). El módulo del gradiente, por otra parte, mide la intensidad, o "pendiente", de dicha variación
[16] El método aquí descrito no se corresponde exactamente con el algoritmo original, sino más bien con una versión simplificada del mismo. Los bordes obtenidos, no obstante, siguen teniendo, en general, una calidad muy aceptable, y además esta versión "sencilla" sirve mucho mejor a los objetivos didácticos del presente ejercicio

2. Para cada píxel de la imagen,

   - Calcular el módulo del vector gradiente (sus componentes x e y son, respectivamente, las derivadas parciales X e Y calculadas en el paso anterior [17]).
   - Si el módulo obtenido supera cierto umbral (parametrizable), marcar como integrante del borde el píxel procesado.

3. **Opcionalmente**, para cada píxel de la imagen que haya sido marcado como borde:

   - Calcular el argumento del vector gradiente (la tangente de este ángulo viene dada por el cociente, en ese punto, entre la derivada parcial respecto a Y y la derivada parcial respecto a X [18]).
   - Desmarcar como borde el píxel si el módulo del gradiente para ese píxel no es un máximo local en la dirección indicada por el gradiente. Esto último debe hacerse comparando el módulo del gradiente en el píxel procesado (el candidato a borde) con el módulo del gradiente de dos de sus ocho vecinos. Determinar qué dos vecinos son los que hay que considerar en esta comparación es algo que depende del argumento del gradiente. Por ejemplo, si el argumento del gradiente está próximo a los 90^o, o si ronda los -90^o (es indiferente), los dos vecinos que hay que tomar para comparar gradientes son el píxel de arriba y el de abajo; si, poniendo otro ejemplo, el módulo del gradiente está en torno a 0^o, o si es aproximadamente 180^o, los dos vecinos que hay que considerar son el de la derecha y el de la izquierda.

     La siguiente figura ilustra todos los casos posibles. Teniendo en cuenta que el píxel procesado tiene 8 vecinos, se divide la circunferencia en 8 sectores circulares, y se eligen los dos vecinos en función de en qué sector caiga el argumento del gradiente.

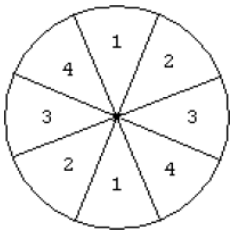

Figura 3: Intervalos de variación del argumento del gradiente

Argumento	**Píxeles vecinos a considerar**
Sectores 1: Entre $\frac{3\pi}{8}$ y $\frac{5\pi}{8}$, o entre $\frac{-5\pi}{8}$ y $\frac{-3\pi}{8}$	Superior e inferior
Sectores 2: Entre $\frac{pi}{8}$ y $\frac{3\pi}{8}$, o entre $\frac{-7\pi}{8}$ y $\frac{-5\pi}{8}$	Superior derecho e inferior izquierdo
Sectores 3: Entre $\frac{pi}{8}$ y $\frac{-\pi}{8}$, o entre $\frac{7\pi}{8}$ y $\frac{-7\pi}{8}$	Derecho e izquierdo
Sectores 4: Entre $\frac{7pi}{8}$ y $\frac{5\pi}{8}$, o entre $\frac{-3\pi}{8}$ y $\frac{-\pi}{8}$	Inferior derecho y superior izquierdo

Cuadro 4: Elección de los dos vecinos con los que comparar módulos de gradientes

[17] El módulo del vector (x, y) es $\sqrt{x^2 + y^2}$.
[18] Argumento del Gradiente $= arctan\frac{derivadaY}{derivadaX}$.