Arquitectura de Computadores. Laboratorio 2. Grupo B. Curso 2019-2020.

Documentación del trabajo grupal.

Realizado por: Iván Álvarez López (UO264862) e Ignacio Gómez Gasch (UO271548).

- **Descripción del problema:**

Nos ha sido planteado en clase de laboratorio el problema de diseñar un programa en C que aplicase un algoritmo sobre una imagen dada, habiendo tres versiones del programa a implementar: monohilo simple, monohilo con instrucciones intrínsecas y multihilo.

El algoritmo difiere entre los grupos, y al nuestro nos ha sido asignado el algoritmo de **binarización de imágenes**, que convierte una imagen en otra que sólo usa los colores blanco y negro; previamente debiéremos de haber hecho una **conversión a escala de grises** de la imagen original para posteriormente binarizarla.

- Conversión a escala de grises de un pixel:
$$L[i] = 0.3 \cdot R[i] + 0.59 \cdot G[i] + 0.11 \cdot B[i]$$
Siendo R, G y B las capas cromáticas de la imagen original, convirtiéndose a una sola capa monocromática L.
- Binarización de un pixel:
$$L[i] = (L[i] < Threshold)\ ?\ 0\ :\ 255;$$
Siendo *Threshold* un valor umbral elegido, que nosotros establecimos como 127.

Además, debemos de tener en cuenta que el **tipo base** para nuestras imágenes ha de ser ***int*** y que para la **versión monohilo con instrucciones intrínsecas** debemos de utilizar **paquetes de 128 bits**.

- **Primera fase:**

La primera fase nos resultó relativamente simple, ya que el dominio del problema se extendía a poco más de entender el uso de punteros que emplea el lenguaje C y comprender el funcionamiento de la librería utilizada, *CImg.h*.

Utilizamos un constructor de dicha librería, mediante el cual pudimos inicializar un objeto *CImg<int>* directamente desde la imagen *bailarina.bmp*, desde el cual podemos obtener la anchura, la altura y el espectro de la imagen (siendo 1 para imágenes monocromáticas, 3 para imágenes a color y 4 para imágenes con transparencias).

No obstante, la foto resultado ha de ser inicializada en un espacio de memoria nuevo, el cual reservamos con la función *malloc*. Los punteros de cada espectro de la imagen original son consecutivos, es decir, una vez obtenido el puntero al primer espectro de la imagen (rojo), el siguiente (verde) se encontrará justamente después de " altura * anchura " posiciones de memoria del primero,

y lo mismo desde el puntero al espectro verde con respecto a su consecutivo, que es el azul. La imagen final solamente tiene un espectro, así que solamente necesita un puntero.

Empleamos la librería *errno.h* empleada en las prácticas de clase para la medición de tiempos.

Recorriendo consecutivamente las 4 capas espectrales de las que hemos hablado (rojo de la 1ª imagen, verde de la 1ª imagen, azul de la 1ª imagen y monocromo de la 2ª imagen), podemos aplicar directamente el algoritmo requerido: se realiza la suma de los píxeles análogos de cada una de las capas de la 1ª imagen y en base a la comparación de este resultado con el *Threshold*, se establece el píxel análogo de la capa monocromo como 0 o como 255.

Una vez construidos los elementos de la imagen resultado, podemos utilizar otro constructor de la librería para guardar la foto.

Para poder compilar el programa, debimos haber descargado la librería *CImg.h* de su página web y haber descargado también la librería *Xlib.h*, de la cual depende la anterior. Creamos un archivo Makefile para generar el ejecutable, además de utilizar la instrucción de compilación recomendada por la propia librería de manipulación de fotografías:

g++ -o Main.exe Main.cpp -O2 -L/usr/X11R6/lib -lm -lpthread -lX11

Fuente: http://cimg.eu/reference/group__cimg__overview.html

Tras la ejecución del ejecutable generado, obtuvimos la imagen resultado a partir de la original sin mayor complicación.

Para una mejor resolución de los tiempos medidos, el algoritmo se repite 50 veces mediante un bucle.

- **Segunda fase. Primera parte:**

El traslado de nuestro algoritmo monohilo a un análogo con instrucciones intrínsecas nos resultó bastante más tedioso, ya que este tipo de instrucciones tienen una dificultad superior a la que la fase anterior nos expuso.

Como apunte curioso, durante la implementación del algoritmo omitimos que el paquete tuviese que ser de 128 bits, por lo que estuvimos empleando erróneamente paquetes de 256 bits, y para la manipulación de paquetes de esta naturaleza (sumas, multiplicaciones y comparaciones) era necesario que el procesador del equipo tuviese, además de la extensión SSE (usada para *_mm_malloc*), la extensión AVX, la cual en el equipo en el que estábamos trabajando, en el de Iván, no se encontraba disponible (procesador Intel Celeron J1900, de arquitectura Silvermont Bay Trail).

Tras readaptarlo todo a paquetes de 128 bits, con el empleo de las extensiones SSE y SSE2 nos bastaba, las cuales sí que estaban disponibles.

Nos vimos obligados a trabajar con números en coma flotante en esta parte, ya que la propia operación de conversión a escala de grises nos obliga a realizar sumas con datos decimales.

Cada paquete *__m128* va a guardar cierta cantidad de números en coma flotante; para hallar dicha cantidad, dividimos el tamaño que ocupa uno de esos paquetes entre el tamaño que ocupa un número en coma flotante (*ELEMENTSPERPACKET*). Con un razonamiento similar, tomando el tamaño de

cada capa de las imágenes como el ancho por la altura, podemos hallar cuántos paquetes va a utilizar cada capa.

En el algoritmo propiamente dicho, iteramos cada paquete de cada capa de la fotografía. Cargamos en variables auxiliares (*vRcomp*, *vGcomp*, *vBcomp*) los valores del paquete iterado mediante la función *_mm_load_ps*. Habiendo establecido previamente tres paquetes auxiliares (*vRgrayscale*, *vGgrayscale*, *vBgrayscale*) con los factores por los cuales hay que multiplicar cada píxel para convertir la foto a escala de grises con la función *_mm_set1_ps*, realizamos dicha multiplicación mediante la función *_mm_mul_ps*. La suma mediante la cual obtenemos la capa única en escala de grises la realizamos mediante la función *_mm_add_ps*.

La segunda parte del algoritmo, la binarización, ya no podemos hacerla mediante funciones intrínsecas, por lo que tenemos que iterar los elementos del paquete para realizar la comparación, volcando el resultado final en un nuevo paquete auxiliar (*auxPackage*), el cual es alineado finalmente con su posición correspondiente en el vector de paquetes de datos en coma flotante (*pLnew*).

La declaración de la imagen resultado es análoga al proceso realizado en la primera fase, asimismo lo es la compilación y ejecución del programa.

- **Segunda fase. Segunda parte:**

Para realizar el algoritmo multihilo, creamos dos funciones, *ThreadProcGrayScalise* y *ThreadProcBinarise*, las cuales son las ejecutadas por los distintos hilos para llevar a cabo la conversión a escala de grises de la foto y su posterior binarización respectivamente.

Las componentes de la foto original pasan a tener que ser atributos globales, para que los distintos hilos puedan acceder a ellas desde los procedimientos anteriormente mencionados.

El concepto de nuestro algoritmo es relativamente simple en cuanto a su abstracción: basándose en la constante THREADS, invoca tantos hilos como se le indique, a cada uno de los cuales se le asigna una porción de cada capa de la imagen para que trabajen con ella.

Primeramente se invocan los hilos que convierten la imagen a escala de grises mediante la función *pthread_create*, luego se espera que se terminen de ejecutar todos los hilos implicados en dicha conversión mediante la función *pthread_join*. Para binarizar, se emplea la misma mecánica, pero se utilizan hilos de binarización en este caso.

Principalmente habíamos pensado en realizar la multiplicación de cada capa por su factor de conversión a escala de grises antes de sumar los resultados de esto, pero esto complicaba el algoritmo teniendo que usar vectores adicionales que daban una carga extra innecesaria al algoritmo.

En el documento Excel de mediciones adjunto se detalla la variación de la duración del algoritmo en función de la cantidad de hilos que se invoquen. Demostramos así que la mejora óptima del algoritmo mediante el uso de múltiples hilos se da en el caso en el que se utilicen tantos hilos como núcleos de ejecución haya disponibles en el equipo.

Realizado por: Iván Álvarez López (UO264862) e Ignacio Gómez Gasch (UO271548).