INGENIERÍA DE LOS COMPUTADORES

OPENMP - LAB3

JOAN CLIMENT QUIÑONES
JAIME MORALES VAELLO
ADRIÁN RÍOS ARMERO
IVÁN LÓPEZ SANCHÍS

Tarea 0.1 Entrenamiento previo OpenMP:

0.1.1 ¿Para qué sirve la variable chunk?

Se utiliza en el contexto de los bucles for, cuando queremos paralelizar un código y hay un bucle for, la variable chunk se encarga de asignar el número de iteraciones que corresponden a cada hilo.

0.1.2 Explica completamente el pragma:

#pragma omp parallel shared(a,b,c,chunk) private(i)

- **¿Por qué y para qué se usa shared(a,b,c,chunk) en este programa?**

La variable shared se utiliza para indicar que variables deben de ser compartidas entre los hilos y cuales deben de ser privadas. En este caso de código, las variables a, b, c y chunk son compartidas entre los hilos y la variable i que en el código haría la función de contador es será independiente para cada hilo.

- **¿Por qué la variable i está etiquetada como private en el pragma?**

Para que cada hilo tenga su propia variable i. Esto nos proporciona solidez, pues si no fuese privada tendríamos problemas de concurrencia donde varios hilos, al tratarse de una variable de control, podrían intentar modificarla al mismo tiempo.

0.1.2 ¿Para qué sirve schedule? ¿Qué otras posibilidades hay?

La cláusula Schedule sirve para definir cómo se van a repartir las iteraciones entre los hilos de un equipo.

Las posibilidades que tenemos las encontramos a la hora de definir cómo se repartirán los módulos los distintos hilos, tenemos diferentes directivas.

Static: La asignación de iteraciones se realiza estáticamente. Cada hilo obtiene un bloque contiguo de iteraciones

Dynamic: La asignación de iteraciones se realiza de manera dinámica. Cada hilo obtiene un bloque de iteraciones cuando ha terminado con el bloque anterior.

Guided: Similar a dynamic, pero el tamaño del bloque disminuye de manera exponencial. Resulta útil cuando el número de iteraciones es desconocido.

0.1.3 ¿Qué tiempos y otras medidas de rendimiento podemos medir en secciones de código paralelizadas con OpenMP?

Con la función omp_get_wtime(), podemos sacar el tiempo que tarda en ejecutarse una fracción de código si ponemos una al inicio y otra al final. Si esto lo hacemos comparando el tiempo que tarda una fracción de código paralelizada y otra sin paralelizar podríamos sacar el speed up, además de la eficiencia que tiene el procesador al ejecutar dicho código paralelizado.

Tarea 0.2: Entrenamiento previo std::async

0.2.1 ¿Para qué sirve el parámetro std::launch::async?

Indica que la tarea se debe ejecutar de manera asíncrona en un hilo separado. La ejecución de la tarea, comienza inmediatamente después de llamar a std::async.

Por otra parte, está la alternativa al async, deferred, que contrariamente a la anterior instrucción, esta no ejecutará la tarea hasta que no se solicite el resultado usando, **std::future::get().**

0.2.2 Calcula el tiempo que tarda el programa con std::launch::async y std::launch::deferred. ¿A qué se debe la diferencia de tiempos?

Con std::launch::async el programa tarda un total de 3000ms y con std::launch::deferred tarda un total de 5000ms. Al tardar más tiempo con std::launch::deferrend nos indica que hay un cuello de botella en el hilo, por tanto, al ejecutar la tarea en un hilo separado con std::launch::async se deshace el cuello de botella y reduce el tiempo de ejecución del programa.

0.2.3 ¿Qué diferencia hay entre los métodos wait y get de std::future?

Wait se utiliza para esperar a que una tarea asíncrona termine sin necesidad de obtener su resultado inmediatamente, mientras que get se utiliza para esperar a que la tarea termine y obtener su resultado al mismo tiempo. Ambos métodos bloquean el hilo actual hasta que la tarea está completa.

0.2.4 ¿Qué ventajas ofrece std::async frente a std::thread?

Nos ofrece la posibilidad de ejecutar en paralelo o diferido, permite especificar las políticas de lanzamiento, tiene una gestión automática de los recursos, esto quiere decir que maneja automáticamente la creación y finalización del hilo. También nos ofrece la recuperación de resultados de forma asíncrona.

Tarea 0.3: Entrenamiento previo std::vector:

0.3.1: ¿Cuál de las dos formas de inicializar el vector y rellenarlo es más eficiente? ¿Por qué?

La segunda forma resulta más eficiente, esto se debe a que se reserva la memoria en una sola línea, a diferencia del primero que se hace más grande cada vez que se añade un elemento, es el push_back lo que lo hace más lento. También implica una mejora de rendimiento a la hora de acceder a posiciones de memoria, pues en el segundo podrás acceder directamente con el índice, lo que lo hace más rápido que el primero.

0.3.2: ¿Podría ocurrir algún problema al paralelizar los dos bucles for? ¿Por qué?

Si, podríamos tener varios problemas a la hora de paralelizar dos bucles for. Entre otras, podríamos tener problemas al compartir la memoria entre los hilos, también podría haber dependencia de datos entre los bucles. Esta dependencia podría causar conflictos entre las iteraciones si intentan acceder a los datos y estos están compartidos. Sin embargo, la biblioteca openMP aborda estos problemas con las cláusulas private y shared.

Tarea 2 Paralelización del revelado digital de fotografías

Tarea 2.1: Análisis del código y los distintos procesos

Gamma Curve

Para explicar la función Gamma Curve, hay que entender primero lo que es una curva gamma, los dispositivos tienen respuestas de luminosidad no lineales, para corregir esto, se utiliza la curva gamma que compensa las características no lineales que ayuda a que la imagen sea más natural a los ojos de las personas.

La función gammaCurve calcula la curva gamma para ajustar los valores de los píxeles de la imagen que pasaremos por parámetro.

Color Balance

La función *colorBalance* ajusta el balance de color de la imágen proporcionada por parámetro.

```
float half_percent = percent / 200.0f;

std::vector<cv::Mat> tmpsplit;
cv::split(in,tmpsplit);
int max = (in.depth() == CV_8U ? 1<<8 : 1<<16) - 1;
```

Se calcula el half percent que nos servirá después para el cálculo de las variables lowval y highval. Con la función cv::split dividiremos la imagen de entrada en los tres canales de color (RGB)
Se calculan los valores de percentil bajo y alto, después se ajustan los valores que se salen por debajo o encima de *lowval* y *highval*, una vez se ha hecho esto, se normalizan los valores de los pixeles que pertenecen a cada canal entre 0 y la variable *max*, por último se combinan los canales con *cv:merge*.

```
    // saturate below the low percentile and above the high percentile
    tmpsplit[i].setTo(lowval,tmpsplit[i] < lowval);
    tmpsplit[i].setTo(highval,tmpsplit[i] > highval);

    // scale the channel
    cv::normalize(tmpsplit[i],tmpsplit[i],0,max,cv::NORM_MINMAX);
}
cv::merge(tmpsplit,out);
```

GammaCorrection

La función gammaCorrection toma la imágen que pasamos por parámetro, se llama a la función gammaCurve para generar una *LUT* cuya función es ayudar a la corrección gamma de los píxeles y a ahorrar tiempo.

```
cv::Mat tmp = cv::Mat::zeros(in.size(), in.type());
unsigned short curve[0x10000];
// create the gamma LUT
gammaCurve(curve, gamma);
```

Se recorren todos los píxeles de la imagen pasada por parámetro, obtenemos el valor corregido para cada uno, utilizando la tabla *LUT*, aplicamos la fórmula de correción gamma para todos los canales de color (RGB) y lo almacenamos en *tmp*.

```
for(int i = 0; i < in.rows; ++i)
{
    p = in.ptr<unsigned short>(i);
    tp = tmp.ptr<unsigned short>(i);
    for (int j = 0; j < in.cols; ++j)
    {
        tp[j*3] = a * curve[p[j*3]] + b;
        tp[j*3+1] = a * curve[p[j*3+1]] + b;
        tp[j*3+2] = a * curve[p[j*3+2]] + b;
    }
}
```

Sharpening

La función *sharpening* copia la imagen de manera que esta se encuentre desenfocada, y se combina con la imagen original para aplicar el enfoque.

```
cv::Mat blurry;
// create a blurred image
cv::GaussianBlur(in, blurry, cv::Size(), sigma);
out = in * (1 + amount) - blurry*amount;
```

Se crea la imagen borrosa con el *cv:GaussianBlur* en la variable *blurry* y se aplica el enfoque a la imagen de salida *out* la imagen borrosa con *amount*.

Enhance Details

La función *enhanceDetails* aumenta la diferencia entre los detalles finos y las areas suaves, consiguiendo resaltar los detalles de la imagen pasada por parámetro.

```
cv::Mat blur, inFloat;
// convert to float32
in.convertTo(inFloat, CV_32F, 1.0/65535);
// create a blurred image
cv::GaussianBlur(inFloat, blur, cv::Size(), sigma);
```

Se creará otra versión borrosa de la imagen de entrada, que se utilizará para afinar los detalles.

```
float* pIn, *pBlur;
for(int i = 0; i < in.rows; ++i)
{
    pIn = inFloat.ptr<float>(i);
    pBlur = blur.ptr<float>(i);
    for (int j = 0; j < in.cols; ++j)
    {
        for(int c = 0;c<3;c++)
        {
            float im = pIn[j*3+c];
            float b = pBlur[j*3+c];
            float d = im - b;
            pBlur[j*3+c] = b + d*amount;
        }
    }
}
```

En estos for anidados se utiliza la diferencia de los valores de los pixeles que hay en las imágenes (borrosa y en punto flotante) y se multiplica por *amount* para mejorar los detalles de la imagen, por último, se convierte la imagen resultante a 16 bits.

BLOOM

Antes de pasar a explicar el código de la función 'bloom', vamos a definir qué es el efecto Bloom. El efecto Bloom es una técnica visual que se utiliza para mejorar la calidad de los gráficos en videojuegos, películas y otros medios digitales. Este consiste en agregar un brillo intenso a las zonas más brillantes de una imagen, similar a lo que sucede cuando se mira directamente a una fuente de luz, haciendo que la imagen final sea más vibrante y realista.

A continuación, pasamos ya a analizar el código de la función:

- En primer lugar, se crean las variables las cuáles serán las encargadas de almacenar las imágenes intermedias. Seguidamente, convierte la imagen de entrada a tipo de dato 'Float32' y a espacio de color 'YCrCb'.

```
auto start = high_resolution_clock::now();

cv::Mat blur, mask, inFloat;
// convert to float32
in.convertTo(inFloat, CV_32F, 1.0/65536);
cv::Mat ycrcb;
// convert to YCrCb color space
cv::cvtColor(inFloat, ycrcb, cv::COLOR_BGR2YCrCb);
```

- A continuación, dividimos la imagen 'YCrCb' en 3 canales, y normalizamos el canal Y entre 0 y 1.

```
// split YCrCb into 3 channels
cv::Mat channels[3];
cv::split(ycrcb, channels);

// normalize Y channel between 0 and 1
cv::normalize(channels[0], mask, 0.0, 1.0, cv::NORM_MINMAX);
```

- Del mismo modo, establecemos a 1.0 los píxeles que se encuentren por encima del umbral especificado y aplicamos un desenfoque gaussiano a los píxeles binarizados. Este desenfoque se define como un efecto de suavizado para mapas de bits.

```
// set to 1.0 only pixels above the threshold
cv::threshold(mask, mask, threshold, 1.0, cv::THRESH_BINARY);
// apply gaussian blur to thresholded pixels
cv::GaussianBlur(mask, mask, cv::Size(), sigma);
```

- Finalmente, se convierte la máscara calculada a una imagen de 3 canales y 16 bits y se imprime el tiempo que se lleva a cabo para el efecto bloom.

```
// convert the computed mask to 3 channel image and 16 bit
cv::cvtColor(mask, mask, cv::COLOR_GRAY2BGR);
mask.convertTo(out, CV_16U, 65535);

auto end = high_resolution_clock::now();
auto elapsed_ms = duration_cast<milliseconds>(end - start);

cout << "Bloom: " << elapsed_ms.count()<<"ms"<<endl;
```

DENOISE

En primer lugar, definiremos el efecto ‘denoise’, el efecto para la eliminación del ruido de una imagen. Antes de nada, explicaremos que es el ruido en una imagen. El ruido en una imagen es la alteración arbitraria de brillo y color en una imagen. Este proceso de reducción de ruido es comúnmente utilizado para suavizar las componentes de crominancia y mejorar la calidad visual de la imagen al eliminar detalles de alta frecuencia que pueden deberse al ruido.

A continuación, pasamos a analizar el código de esta función:

- En primer lugar, se crea una variable que almacenará la imagen. Seguidamente, convierte la imagen de entrada a tipo de dato ‘Float32’ y a espacio de color ‘YCrCb’.

```
auto start = high_resolution_clock::now();

cv::Mat ycrcb;
// convert to float32
in.convertTo(ycrcb, CV_32F, 1.0/65535.0);
// convert to YCrCb color space
cv::cvtColor(ycrcb, ycrcb, cv::COLOR_BGR2YCrCb);
```

- A continuación, como en el efecto anterior, dividimos la imagen ‘YCrCb’ en 3 canales. Además, aplicamos el filtro de mediana para reducir el ruido en los canales de crominancia.

```
// split ycrcb into 3 channels
std::vector<cv::Mat> channels;
cv::split(ycrcb, channels);

// remove noise from chrominance channels
cv::medianBlur(channels[1], channels[1], windowSize);
cv::medianBlur(channels[2], channels[2], windowSize);
```

- Finalmente, convertimos la imagen de nuevo a tipo ‘RGB’ y 16 bits, e imprimimos el tiempo que ha llevado a cabo el proceso de reducción de ruido.

```
// convert back to RGB and 16 bits
cv::merge(channels, out);
cv::cvtColor(out, out, cv::COLOR_YCrCb2BGR);
out.convertTo(out, CV_16U, 65535);

auto end = high_resolution_clock::now();
auto elapsed_ms = duration_cast<milliseconds>(end - start);

cout<<"Denoise: "<<elapsed_ms.count()<<"ms"<<endl;
```

EQUALIZATION

En primer lugar, como en los anteriores procesos, vamos a definir de que trata el efecto 'equalization'. Esta función realiza la ecualización de histograma en una imagen en el espacio de color HSV. La ecualización del histograma de una imagen es una transformación que pretende obtener para una imagen un histograma con una distribución uniforme. Es decir, que exista el mismo número de píxeles para cada nivel de gris del histograma de una imagen monocroma. El resultado de la ecualización maximiza el contraste de una imagen sin perder información de tipo estructural, esto es, conservando su entropía.

A continuación, pasamos a analizar el código de la función:

- En primer lugar, se crea una variable que almacenará la imagen. Seguidamente, convierte la imagen de entrada a tipo de dato 'Float32' y a espacio de color 'HSV'.

```
auto start = high_resolution_clock::now();
cv::Mat hsv;
// convert to float32
in.convertTo(hsv, CV_32F, 1.0/65535.0);
// convert to HSV color space
cv::cvtColor(hsv, hsv, cv::COLOR_BGR2HSV);
```

- A continuación, dividimos la imagen 'HSV' en tres canales y normalizamos los valores entre el nivel mínimo de negro y el nivel máximo de blanco. Además, aumentamos la saturación multiplicando el canal de saturación por (1 + saturación).

```
// split HSV into 3 channels
std::vector<cv::Mat> channels;
cv::split(hsv, channels);
// normalize values between minimum black level and maximum white level
cv::normalize(channels[2], channels[2], black, white, cv::NORM_MINMAX);
// increase saturation
channels[1] *= (1 + saturation);
```

- Finalmente, convertimos la imagen de nuevo a tipo 'RGB' y 16 bits, e imprimimos el tiempo que ha llevado a cabo el proceso de ecualización.

```
// convert back to RGB and 16 bit
cv::merge(channels, out);
cv::cvtColor(out, out, cv::COLOR_HSV2BGR);
out.convertTo(out, CV_16U, 65535);

auto end = high_resolution_clock::now();
auto elapsed_ms = duration_cast<milliseconds>(end - start);

cout<<"Equalization: "<<elapsed_ms.count()<<"ms"<<endl;
```

DEBAYER

Antes de nada, vamos a explicar de qué trata el Mosaico de Bayer (de lo que trata esta función). El mosaico de Bayer es un tipo de matriz de filtros, rojos, verdes y azules, que se sitúa sobre un sensor digital de imagen para hacer llegar a cada fotodiodo la información de luminosidad correspondiente a una sección de los distintos colores primarios. El mosaico de Bayer está formado por un 50 % de filtros verdes, un 25 % de rojos y un 25 % de azules. Interpolando dos muestras verdes, una roja y una azul, se obtiene un pixel de color.

A continuación, pasamos a analizar el código de la función:

- En primer lugar, convertimos la imagen RAW a imagen RGB.

```
auto start = high_resolution_clock::now();
processor->raw2image();
int width = processor->imgdata.sizes.iwidth;
int height = processor->imgdata.sizes.iheight;
int orientation = processor->imgdata.sizes.flip;
```

- A continuación, se crea un buffer de unshorts que contiene el patrón Bayer de un solo canal. Dentro de este bucle, se obtiene el índice del píxel, donde cada pixel es un array de 4 shorts 'RGBG'. Además, en la condición, se comprueba si la fila es par, significa que se obtendrá el color rojo si x es par y si x es impar, el color verde. En caso contrario, la fila sea impar, se obtendrá el color verde si x es par y si x es impar, el color azul.

```
// create a buffer of ushorts containing the single channel bayer pattern
std::vector<ushort> bayerData;
for ( int y = 0; y < height; y++ )
{
    for ( int x = 0; x < width; x++ )
    {
        // get pixel idx
        int idx = y * width + x;

        // each pixel is an array of 4 shorts rgbg
        ushort *rgbg = processor->imgdata.image[idx];

        // even rows are RGRGRG..., odds are GBGBGB...
        // even rows are RGRGRG..., get red if x is even or green if odd
        if (y % 2 == 0)
        {
            bool red = x % 2 == 0;
            bayerData.push_back(rgbg[red ? 0 : 1]);
        }
        // odd rows are GBGBGB..., get green if x is even or blue if odd
        else
        {
            bool green = x % 2 == 0;
            bayerData.push_back(rgbg[green ? 3 : 2]);
        }
    }
}
```

- Del mismo modo, se crea una matriz de OpenCV con el patrón Bayer. Seguidamente, se aplica el algoritmo de debayerización.

```
// create an OpenCV matrix with the bayer pattern
cv::Mat imgBayer(height, width, CV_16UC1, bayerData.data());
cv::Mat imgDeBayer;
// apply the debayering algorithm
cv::cvtColor(imgBayer, imgDeBayer, cv::COLOR_BayerBG2BGR);
out = imgDeBayer;
```

- Finalmente, se aplica la orientación adecuada a la imagen debayerizada y se imprime el tiempo que lleva a cabo realizar esta función.

```
switch(orientation)
{
    case 2:
        cv::flip(out, out, 0);
        break;
    case 3:
        cv::rotate(out, out, cv::ROTATE_180);
        break;
    case 4:
        cv::flip(out, out, 1);
        break;
    case 5:
        cv::rotate(out, out, cv::ROTATE_90_COUNTERCLOCKWISE);
        break;
    case 6:
        cv::rotate(out, out, cv::ROTATE_90_CLOCKWISE);
        break;
    case 7:
        cv::flip(out, out, 0);
        cv::rotate(out, out, cv::ROTATE_90_CLOCKWISE);
        break;
    case 8:
        cv::flip(out, out, 0);
        cv::rotate(out, out, cv::ROTATE_90_COUNTERCLOCKWISE);
        break;
}

auto end = high_resolution_clock::now();
auto elapsed_ms = duration_cast<milliseconds>(end - start);

cout<<"De Bayer: "<<elapsed_ms.count()<<"ms"<<endl;
```

SCREENMERGE

Antes de nada, como en los anteriores procesos, vamos a explicar un poco lo que hace esta función a la imagen. El modo "screen" produce un efecto de fusión suave y es especialmente útil cuando se desea superponer dos imágenes de manera que la más clara tenga un impacto significativo en el resultado final.
A continuación, pasamos a analizar el código de la función:

- En primer lugar, convertimos las imágenes de entrada a tipo 'Float32' para evitar desbordamiento y creamos una matriz temporal inicializada con ceros.

```
auto start = high_resolution_clock::now();

cv::Mat inFloat1, inFloat2;
// convert to float32 to avoid overflow
in1.convertTo(inFloat1, CV_32F, 1.0/65535);
in2.convertTo(inFloat2, CV_32F, 1.0/65535);
cv::Mat tmp = cv::Mat::zeros(inFloat1.size(), inFloat1.type());
float* pIn1, *pIn2, *pTmp;
```

- Seguidamente, aplicamos el modo de fusión "screen".

```
// apply the screen mode merge
for(int i = 0; i < in1.rows; ++i)
{
    pIn1 = inFloat1.ptr<float>(i);
    pIn2 = inFloat2.ptr<float>(i);
    pTmp = tmp.ptr<float>(i);
    for (int j = 0; j < in1.cols; ++j)
    {
        for(int c = 0; c < 3; c++)
        {
            float im = pIn1[j * 3 + c];
            float m = pIn2[j * 3 + c];
            pTmp[j * 3 + c] = 1.0 - (1.0 - m) * (1.0 - im);
        }
    }
}
```

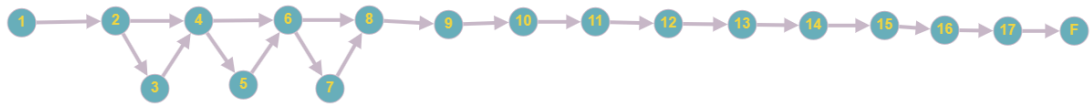
- Finalmente, convertimos la matriz temporal al tipo de dato de salida (CV_16U) y escalamos los valores, e imprimimos el tiempo tomado para el proceso de fusión en modo "screen".

```
tmp.convertTo(out, CV_16U, 65535);

auto end = high_resolution_clock::now();
auto elapsed_ms = duration_cast<milliseconds>(end - start);

cout<<"Screen mode merge: "<< elapsed_ms.count()<<"ms"<<endl;
```

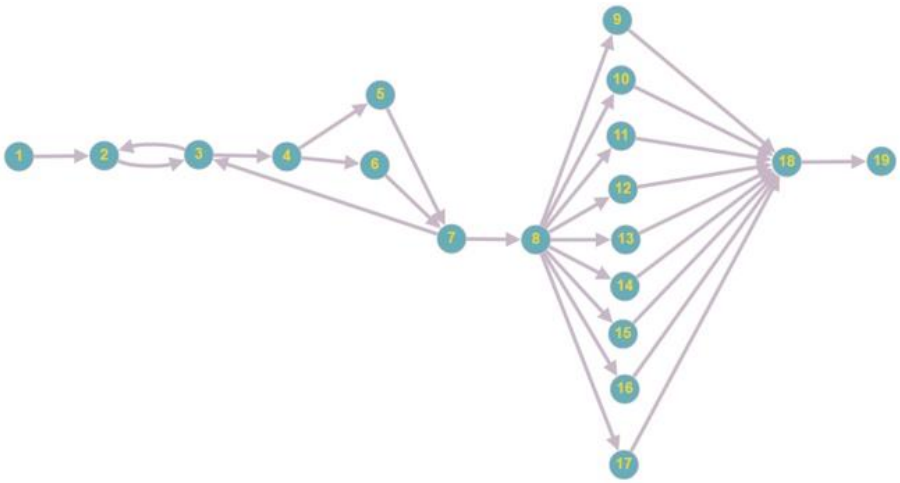
Tarea 2.2 Analiza el código de cada proceso

Grafos de Control de Flujo

main: Grafo que representa los caminos lógicos que puede recorrer el programa al realizar el procesar una imagen.

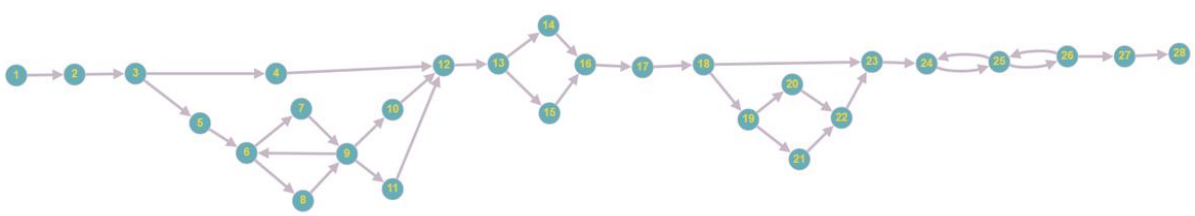


debayer (nodo 9 del main): Grafo que representa los caminos lógicos que puede recorrer el programa al aplicar el algoritmo de Debayer para separar los píxeles de la imagen en 3 canales diferentes.



denoise (nodo 10 del main): Grafo que representa los caminos lógicos que puede recorrer el programa al transformar la imagen en una imagen RGB.

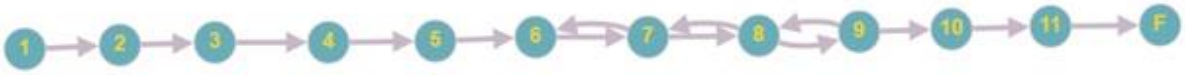
gammaCorrection y gammaCurve (nodo 11 del main): Grafo que representa los caminos lógicos que puede recorrer el programa al hacer la corrección gamma a la imagen.



colorBalance (nodo 12 del main): Grafo que representa los caminos lógicos que puede recorrer el programa al distribuir equitativamente los canales de color de la imagen.

equalization (nodo 13 del main): Grafo que realiza la ecualización del histograma en una imagen de entrada.

enhanceDetails (nodo 14 del main): Grafo que representa los caminos lógicos que puede recorrer el programa al realzar los detalles de la imagen de entrada.



bloom (nodo 15 del main): Grafo que representa los caminos lógicos que puede recorrer el programa al aplicar un efecto de bloom a la imagen de entrada.

screenMerge (nodo 16 del main): Grafo que representa los caminos lógicos que puede recorrer el programa al fusionar 2 imágenes.

sharpening: Grafo que representa los caminos lógicos que puede recorrer el programa al aumentar la nitidez de una imagen de entrada.

Tarea 2.3 Paralelización del programa

Para paralelizar el programa nos hemos fijado, sobre todo, en el flujo del programa que antes hemos detallado. Dicho flujo presenta ciertos patrones que podremos aprovechar para paralelizar la ejecución del mismo.

Respondiendo a las preguntas del ejercicio 2.3, el programa sí que obtiene un mejor rendimiento si lo paralelizamos, es una mejora sustancial si de las funciones más largas se trata. El problema que hay es que, si paralelizamos las funciones que ya lo están de base, se produce una sobre paralelización y una degradación del rendimiento del programa. También nos han ocurrido errores, los fallos por condición de carrera, que se deben a que, al ejecutar el programa paralelizado, ambos subprocesos acceden a la misma variable.

	sin paralelizar
debayer	828
denoise	1361
gammaCUrve creation	2
gammaCorrection	316
colorBalance	2821
equalization	781
enhanceDetails	6120
Bloom	6841
screen mode merge	637

Como vemos en la imagen, las funciones tardan una cantidad considerable de tiempo. Las más lentas son “enhanceDetails” y “Bloom”. A continuación, analizaremos función por función qué podemos hacer para mejorar el rendimiento de las mismas, en todos los casos las soluciones son la media de varias ejecuciones para dar un valor más exacto de la eficiencia al realizar la paralelización.

Para la función “debayer”, no hemos podido encontrar una solución viable por la cual paralelizar el código de forma que reduzca de forma significativa el tiempo de ejecución.

Para la función “denoise”, hemos usado la directiva parallel y sections. La función realiza llamadas a otras funciones para quitar el ruido de la imagen y luego le vuelve a poner color. Dichas funciones son casi independientes entre sí, es decir, no comparten variables por lo que son perfectamente paralelizables con la directiva sections. Por otra parte, agrupamos las funciones de dos en dos y no las mandamos

cada una a un hilo por su dependencia, pues no se podría calcular, por ejemplo, la segunda función sin haber calculado antes la primera.

Para la función “gammaCurve creation”, hemos utilizado la directiva for para realizar la paralelización. Esta consiste en mandar cada iteración del bucle que se encarga de tratar la curva gamma, a un hilo diferente para cada nivel de intensidad ‘i’.

Para la función “gammaCorrection”, paralelizamos el bucle principal que aplica la corrección gamma a cada píxel de la imagen de entrada. Es imprescindible que cada hilo tenga variables privadas como se indica en la declaración, pues cada subproceso trabaja en diferentes partes de la imagen y necesita variables privadas. Dentro del bucle principal se realizan las operaciones de corrección gamma a cada color RGB por separado.

Para la función “colorBalance”, realizamos la paralelizaición del bucle principal permitiendo así que varios hilos puedan operar en diferentes canales de color al mismo tiempo. En este caso, no haría falta declarar ninguna variable private, pues la ejecución del programa se realiza sin problemas al paralelizar el bucle for.

Para la función “equalization”, no hemos podido encontrar una solución viable por la cual paralelizar el código de forma que reduzca de forma significativa el tiempo de ejecución, ya que en esta función se realizan cálculos que dependen del anterior.

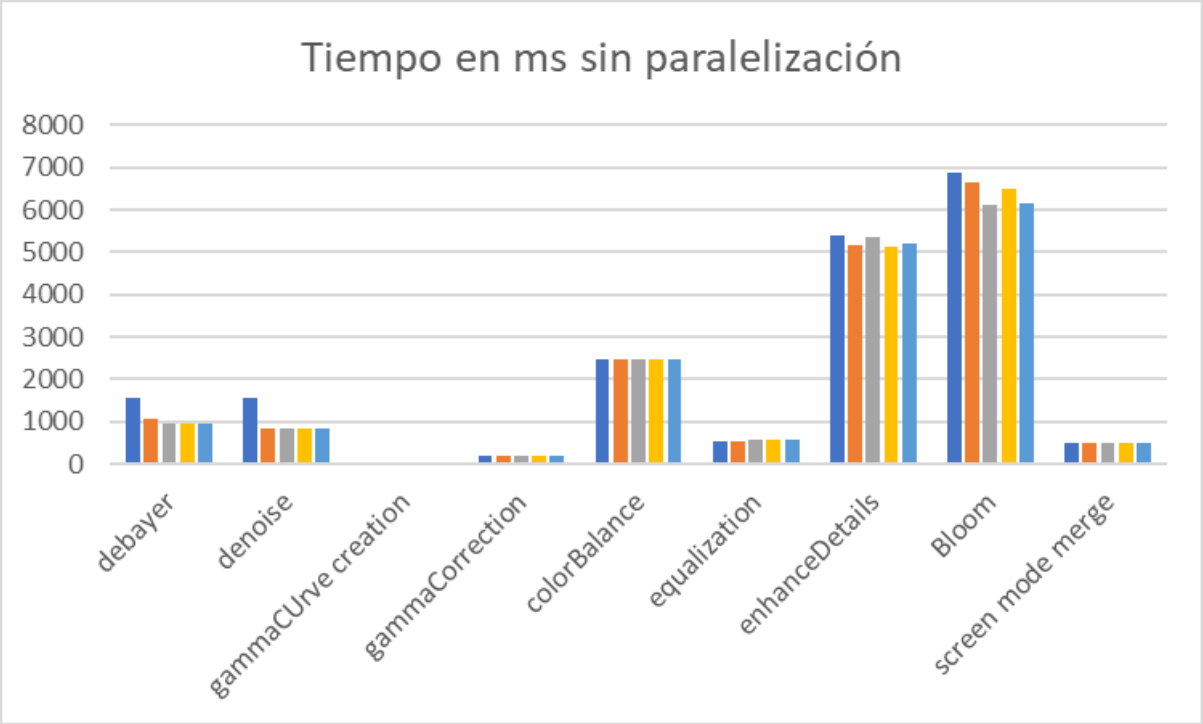
Para la función “enhanceDetails”, hemos realizado dos bloques diferentes. El primero, en el que agrupamos en sections las funciones in.convertTo() y cv::GaussianBlur(), cada una se ejecutará en un hilo diferente. El segundo bloque, que consiste en reconstruir el desenfoque de la imagen, se hace por 3 for anidados. Este trabaja con los punteros float que se declaran al inicio, pIn y pBlur. Para la paralelización necesitaremos que cada hilo trabaje con variables privadas y no se compartan las variables entre ellos pudiendo, ocurrir condiciones de carrera. Por esto, se pasan en el pragma como privadas. Sin embargo, al paralelizar el bucle aumenta significativamente el tiempo de ejecución debido a una sobreparalelización del mismo, por tanto, hemos decidido no paralelizarlo.

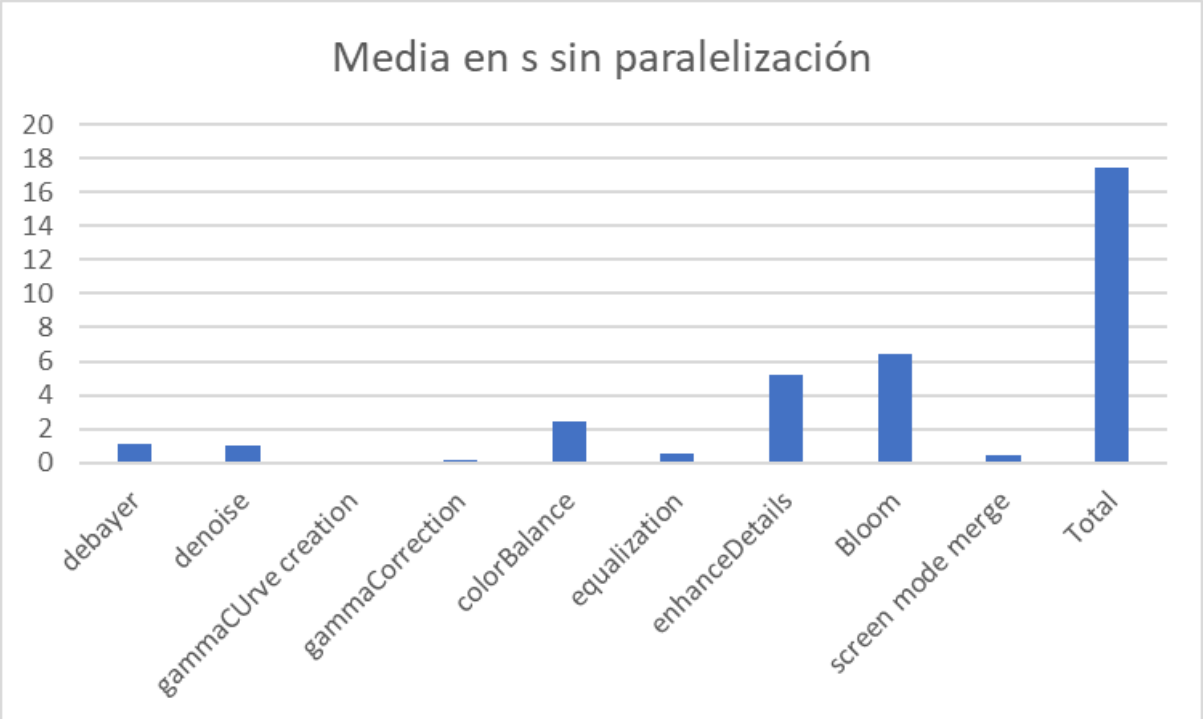
Para la función “Bloom”, no hemos podido encontrar una solución viable por la cual paralelizar el código de forma que reduzca de forma significativa el tiempo de ejecución, por lo cual hemos decidido ejecutar de manera paralela esta función y la función “enhanceDetails”, ya que realizar cálculos independientes la una de la otra.

Para la función “screen mode merge”, en la cual se realizan tres bucles for anidados, hemos optado por paralelizar el segundo bucle for ya que es este el que se encarga de dividir en los tres canales de color RGB para luego interpolarlos. Con la directiva schedule(static) dividimos las iteraciones en bloques y cada hilo se encarga de un

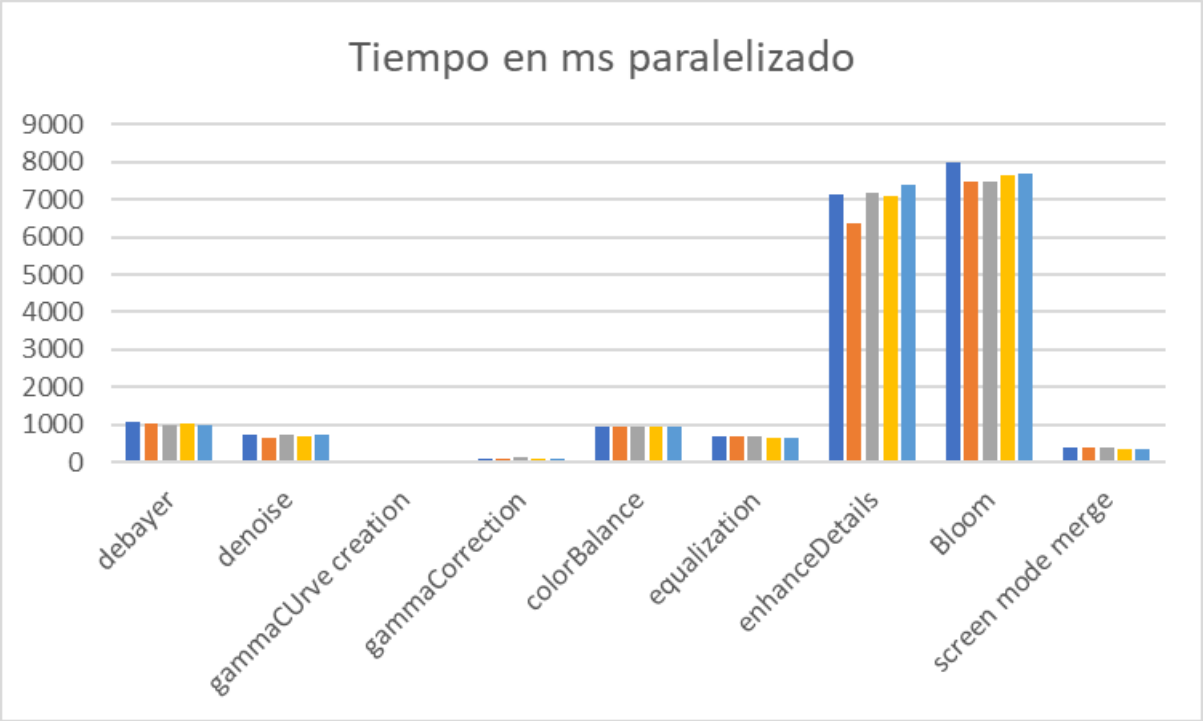
bloque. Al ser la asignación estática ningún hilo cambia de bloque durante la ejecución.

Tiempos del programa





Hemos añadido una nueva medida que es el tiempo total que tarda la ejecución del programa, vemos que suele tardar menos de 19 segundos.



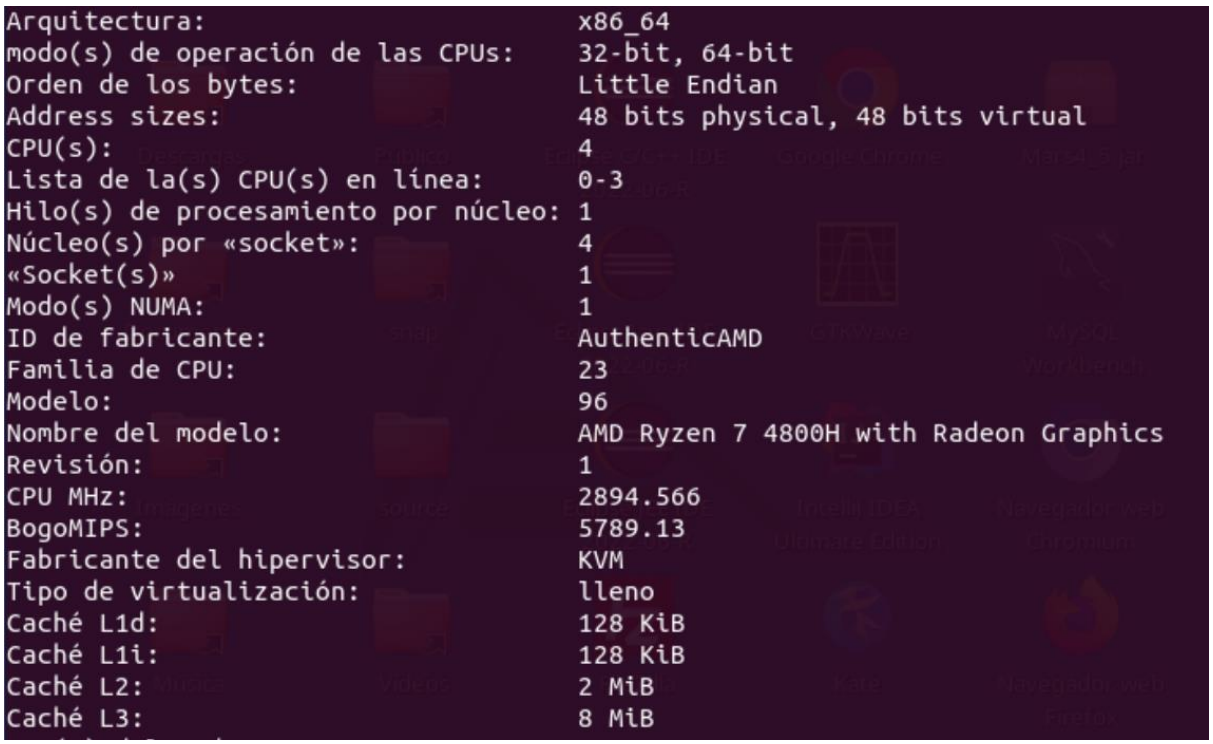
Se han reducido los tiempos considerablemente de la función de *colorBalance* y de *denoise*, aunque los tiempos de *enhanceDetails* y *Bloom* han aumentado.

Aunque los tiempos de *enhanceDetails* y *Bloom* sean más altos, al ejecutarse en paralelo conseguimos reducir significativamente el tiempo total de la ejecución.

● **Caracterización de la máquina paralela en la que se ejecuta el programa. (p.ej. Número de nodos de cómputo, sistema de caché, tipo de memoria, etc.).**

En Linux use la orden: cat /proc/cpuinfo para acceder a esta información o lscpu.

El programa ha sido ejecutado en una máquina virtual con las siguientes características:



```
Arquitectura:                         x86_64
modo(s) de operación de las CPUs:     32-bit, 64-bit
Orden de los bytes:                   Little Endian
Address sizes:                        48 bits physical, 48 bits virtual
CPU(s):                               4
Lista de la(s) CPU(s) en línea:       0-3
Hilo(s) de procesamiento por núcleo:  1
Núcleo(s) por «socket»:               4
«Socket(s)»                           1
Modo(s) NUMA:                         1
ID de fabricante:                     AuthenticAMD
Familia de CPU:                       23
Modelo:                               96
Nombre del modelo:                    AMD Ryzen 7 4800H with Radeon Graphics
Revisión:                             1
CPU MHz:                              2894.566
BogoMIPS:                             5789.13
Fabricante del hipervisor:            KVM
Tipo de virtualización:               lleno
Caché L1d:                            128 KiB
Caché L1i:                            128 KiB
Caché L2:                             2 MiB
Caché L3:                             8 MiB
```

● **¿Qué significa la palabra ht en la salida de la invocación de la orden anterior?**

La palabra ht en el campo flags se refiere a que nuestra CPU tiene Hyper-threading lo cual, permite a los núcleos de la CPU ejecutar dos o más hilos de manera paralela.

Github

El repositorio donde hemos trabajado los miembros del grupo es el siguiente: https://github.com/ara130-ua/IC-Grupo5-3