Universidad de Oviedo

Escuela de Ingeniería Informática
Universidad de Oviedo

Arquitectura de Computadores

Trabajo en grupo

PL1-A

Jorge Gota Ortín UO301023

Javier Ortín Rodenas UO299855

Andrés Fernández-Junquera Fernández UO302086

Alejandro Jarillo Pineda UO302394

Índice

1. Algoritmo a aplicar a la imagen

1.1. Algoritmo matemático

En nuestro caso, el algoritmo a implementar es el número 3. Tiene como entrada una única imagen y genera como salida una versión de la misma en blanco y negro. Sean (R, G, B) las componentes de un píxel de la entrada, las componentes (R', G', B') del píxel equivalente en la imagen de salida vienen dadas por:

$$R' = G' = B' = 255 - (0.3 \cdot R + 0.59 \cdot G + 0.11 \cdot B)$$

Se pasa a escala de grises con la siguiente media ponderada:

$$L = 0.3 \cdot R + 0.59 \cdot G + 0.11 \cdot B$$

A continuación, se invierte al hacer $R' = G' = B' = 255 - L$, de ahí el nombre de "inversión en blanco y negro".

Matemáticamente, los valores de la imagen destino deberían estar comprendidos siempre entre 0 y 255. Debido a la sencillez de este algoritmo, no será sencillo saturar, pues el resultado de las operaciones tendrá un valor adecuado siempre.

1.2. Funcionamiento general e implementación monohilo

Comencemos analizando la función `main` y el funcionamiento general del programa:

```
int main() {
    // 1 -- Load source image
    CImg<data_t> srcImage;
    if(loadSourceImage(srcImage)) {
        return IMG_ERROR;
    }

    // 2 -- Allocate memory and prepare filter structures
    filter_args_t filter_args;
    data_t *pDstImage = NULL;
    unsigned int width = 0, height = 0, nComp = 0;

    if(prepareFilter(srcImage, filter_args, pDstImage, width, height, nComp)) {
        return MEMORY_ERROR;
    }

    // 3 -- Measure execution time over REPS iterations
    double elapsedTimeS = 0.0;
    if(measureExecutionTime(filter_args, elapsedTimeS)) {
        free(pDstImage); // cleanup
        return TIMING_ERROR;
    }
    printf("\nElapsed time: %f s\n", elapsedTimeS);

    // 4 -- Display and save output image
    showAndSaveResult(pDstImage, width, height, nComp);

    // 5 -- Free memory
    free(pDstImage);

    return 0;
}
```

El funcionamiento general del programa es el mismo para las tres implementaciones:

En primer lugar, cargamos la imagen teniendo en cuenta la posibilidad de que esta pueda no existir y manejando el error en tal caso. A continuación, iniciamos las variables necesarias, deteniendo la ejecución si no fuese posible asignar memoria adecuadamente. Una vez hecho esto, medimos el tiempo que tarda en ejecutarse el algoritmo 100 veces (para que tarde el tiempo mínimo deseado), manejando los errores de cronometrado en caso de haberlos. Finalmente, se muestra el tiempo transcurrido así como la imagen destino.

Veamos ahora cómo se estructura el algoritmo del filtro:

```
// Filter function: converts RGB to grayscale and applies the inversion
void filter (filter_args_t args) {
    for (uint i = 0; i < args.pixelCount; i++) {
        // Compute grayscale using weighted sum of RGB components
        data_t value = MAX_VAL - (R_COEF * args.pRsrc[i] + G_COEF* args.pGsrc[i] + B_COEF * args.pBsrc[i]);
        // Write the same grayscale value to each output channel
        args.pRdst[i] = value;
        args.pGdst[i] = value;
        args.pBdst[i] = value;
    }

}
```

El algoritmo recorre los píxeles de la imagen, aplicando para cada uno de ellos el algoritmo matemático visto.

1.3. Implementación multihilo

El funcionamiento general del programa es el mismo. Por tanto, nos centraremos únicamente en las modificaciones necesarias para implementar la funcionalidad multihilo.

En esta versión, se asignará un `filter_args_t` a cada hilo para que itere sobre los píxeles que le sean asignados. Por tanto, hemos modificado el `struct` para que almacene también los píxeles inicial y final que ha de recorrer:

```
//Structure containing arguments for each thread/filter
typedef struct {
    data_t *pRsrc; // Pointer to the Red source channel
    data_t *pGsrc; // Pointer to the Green source channel
    data_t *pBsrc; // Pointer to the Blue source channel
    data_t *pRdst; // Pointer to the Red destination channel
    data_t *pGdst; // Pointer to the Green destination channel
    data_t *pBdst; // Pointer to the Blue destination channel
    uint pixelCount; // Size of the image in pixels

    // Work distribution for multithreading
    unsigned int start;  //First pixel index processed by this thread
    unsigned int end;   //One-past-the-end index processed by this thread
} filter_args_t;
```

Veamos cómo se distribuye el trabajo entre los hilos:

```
//WORK DISTRIBUTION -- Splits pixel workload evenly among threads
void setupThreadWork(filter_args_t& sharedArgs, filter_args_t* threadArgs,
                     unsigned int numThreads) {

    unsigned int baseWorkPerThread = sharedArgs.pixelCount / numThreads;
    unsigned int remaining = sharedArgs.pixelCount % numThreads;

    unsigned int startIndex = 0;

    for (unsigned int i = 0; i < numThreads; i++) {

        //Give one extra pixel to the first 'remaining' threads
        unsigned int extra = (i < remaining ? 1 : 0);
        unsigned int endIndex = startIndex + baseWorkPerThread + extra;

        //All threads share the same source/destination pointers
        threadArgs[i].pRsrc = sharedArgs.pRsrc;
        threadArgs[i].pGsrc = sharedArgs.pGsrc;
        threadArgs[i].pBsrc = sharedArgs.pBsrc;

        threadArgs[i].pRdst = sharedArgs.pRdst;
        threadArgs[i].pGdst = sharedArgs.pGdst;
        threadArgs[i].pBdst = sharedArgs.pBdst;

        //Thread's processing interval
        threadArgs[i].start = startIndex;
        threadArgs[i].end   = endIndex;

        startIndex = endIndex;
    }
}
```

El parámetro `shared_args` contiene la información común a todos los hilos, que ha sido agrupada en forma de `filter_args_t` para facilitar su manejo. A partir de él se inician los campos del resto de `structs` de este tipo.

Más aún, en este bucle se asignan los píxeles de inicio y final. Para ello, se calculan primero los píxeles base (`baseWorkPerThread`) de cada hilo, a los que se puede sumar un píxel extra en función del resto de la división. De este modo, nos aseguramos de que se procesen todos los píxeles de la imagen aunque el número total de ellos no sea múltiplo del número de hilos. Además, este sistema distribuye la carga de manera uniforme entre los hilos, en lugar de asignar todos los píxeles restantes (`remaining`) a un solo hilo.

El programa itera `REPS` número de veces, creando y destruyendo en cada una de ellas los hilos, y asignándoles la carga de trabajo apropiada a cada uno. La implementación del paso a blanco y negro se lleva a cabo del mismo modo que en la versión monohilo, pero cada hilo solo itera por los píxeles que le corresponden.

1.4. Implementación SIMD

Como las instrucciones que aplicamos a cada píxel son idénticas, podemos trabajar con instrucciones SIMD que realicen las mismas operaciones sobre una agrupación de píxeles. En nuestro caso, el enunciado nos asignó el tipo de píxel `double` y el tamaño de paquete de `128bits`, lo que se traduce en un tipo SIMD `__m128d`.

En estas condiciones, nuestro ancho de paquete viene dado por:

$$\texttt{width} = \frac{\texttt{sizeof(\_\_m128)}}{\texttt{sizeof(double)}} = \frac{\texttt{128bits}}{\texttt{64bits}} = 2$$

Cada paquete albergará dos píxeles. Analicemos la implementación en código:

```
uint simd_width =  sizeof(__m128d) / sizeof(data_t);
uint i = 0; // Current pixel index

// We load auxiliary packages with the weights
__m128d coefR = _mm_set1_pd(R_COEF);
__m128d coefG = _mm_set1_pd(G_COEF);
__m128d coefB = _mm_set1_pd(B_COEF);
__m128d maxVal = _mm_set1_pd(MAX_VAL);
```

El filtro comienza cargando paquetes auxiliares con los coeficientes que intervienen en el algoritmo matemático. Este paso se realiza solo una vez en el filtro entero, no para cada píxel. Una vez hecho esto, se recorre la imagen por paquetes trabajando con las tres componentes a la vez (no se avanza componente a componente). Para cada paquete, se cargan los píxeles de la imagen fuente y se opera siguiendo el algoritmo matemático. En este caso, hemos optado por utilizar instrucciones combinadas SIMD que multiplican los dos últimos parámetros, sumando el resultado de esta operación al tercero:

```
// Iterate by packages
for (; i <= args.pixelCount - simd_width; i += simd_width) {
    __m128d imgR = _mm_loadu_pd(args.pRsrc + i);
    __m128d imgG = _mm_loadu_pd(args.pGsrc + i);
    __m128d imgB = _mm_loadu_pd(args.pBsrc + i);

    //  (R * coefR) + (G * coefG)
    __m128d sum = _mm_fmadd_pd(imgR, coefR, _mm_mul_pd(imgG, coefG));
    // Adding the blue component: sum + (B * coefB)
    sum = _mm_fmadd_pd(imgB, coefB, sum);
    __m128d result = _mm_sub_pd(maxVal, sum);

    // Save the results
    *(__m128d *) (args.pRdst + i) = result;
    *(__m128d *) (args.pGdst + i) = result;
    *(__m128d *) (args.pBdst + i) = result;
}

// Remaining pixels
```

De manera similar al caso multihilo, el número de píxeles podría no ser múltiplo del ancho de paquete. Por tanto, tratamos los píxeles sobrantes con el algoritmo monohilo (tras el comentario "*Remaining pixels*" se tratan como en la versión monohilo, se ha omitido de la imagen al haberse explicado ya).

2. Imágenes de entrada y salida

En esta sección se muestran las imágenes originales y las imágenes resultantes tras aplicar el algoritmo de inversión en blanco y negro para cada una de las implementaciones desarrolladas: monohilo, SIMD y multihilo.

2.1. Implementación Monohilo

Imagen original

Resultado tras aplicar el algoritmo

Comparación visual para la implementación monohilo

2.2. Implementación SIMD

Imagen original

Resultado tras aplicar el algoritmo

Comparación visual para la implementación SIMD

2.3. Implementación Multihilo

Imagen original

Resultado tras aplicar el algoritmo

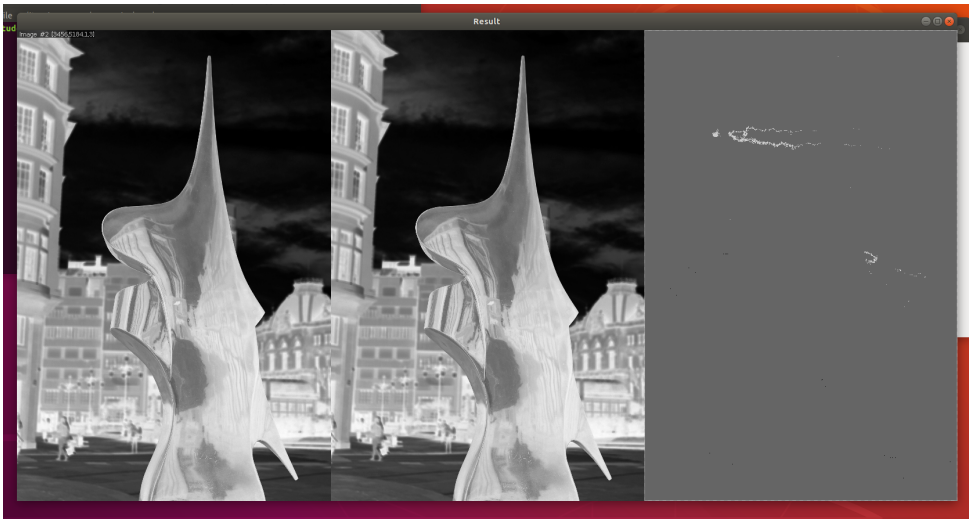
Comparación visual para la implementación multihilo

3. Comparativa mediante `diffImages`

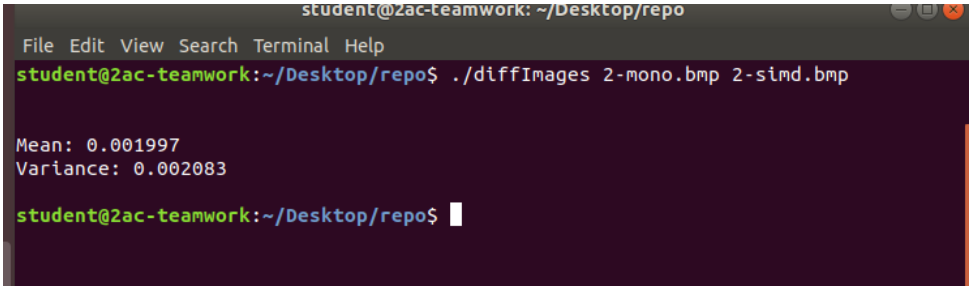
En esta sección se muestran las capturas de la herramienta `diffImages`, utilizada para comparar píxel a píxel las imágenes generadas por las distintas implementaciones del algoritmo. Esta utilidad permite comprobar visualmente si existen diferencias entre los resultados producidos por cada variante del programa.

Como el algoritmo matemático es el mismo para todas ellas, no debería de haber ninguna diferencia (idealmente).

3.1. SIMD vs Monohilo



Ejecución de `diffImages` comparando SIMD con Monohilo



Salida por consola

La diferencia, aunque existente, es mínima. Consideramos también una versión SIMD sin instrucciones combinadas `fmadd`, que sí producía exactamente la misma imagen. No obstante, esta otra versión tenía menor rendimiento. Al ser tan escasa la diferencia visual, terminamos por decantarnos por la versión con instrucciones `fmadd`. Las discrepancias se deben a la naturaleza de las propias instrucciones, pues la semántica del programa es la misma.

3.2. Multihilo vs Monohilo

Ejecución de `diffImages` comparando Monohilo con Multihilo

```
student@2ac-teamwork:~/Desktop/diffImages$ ./diffImages bailarinaMonohilo.bmp ba
ilarinaMultihilo.bmp


Mean: 0.000000
Variance: 0.000000
```

Salida por consola

La imagen generada es exactamente la misma.

4. Resultados experimentales

4.1. Fase 1 – SingleThread: DEF

Fase 1 SingleThread: DEF

1	2	3	4	5	6	7	8	9	10	Media	Desv. Típ.	Inf. 95 %	Sup. 95 %
8.33	7.62	8.39	8.84	9.09	9.21	9.00	8.96	9.42	10.86	8.972	0.845442685	8.447989195	9.496011

4.2. Fase 2 – SIMD

Fase 2 SIMD

1	2	3	4	5	6	7	8	9	10	Media	Desv. Típ.	Inf. 95 %	Sup. 95 %
10.31	9.10	9.34	9.31	9.38	9.50	10.10	9.27	9.17	9.40	9.488	0.397570399	9.241583314	9.734417

4.3. Fase 2 – Multihilo

Fase 2 Multihilo 10 hilos (2×5 procesadores)

1	2	3	4	5	6	7	8	9	10	Media	Desv. Típ.	Inf. 95 %	Sup. 95 %
6.10	5.25	5.20	5.13	5.78	5.90	6.12	5.48	5.96	5.23	5.615	0.39789027	5.368385056	5.861615

4.4. Cálculo de aceleraciones

Los tres programas realizan la misma tarea (tienen la misma carga de trabajo), solamente cambia cómo la llevan a cabo. Por tanto, podemos calcular la aceleración a partir de los tiempos medios obtenidos para cada implementación. Calculamos la aceleración relativa frente a la versión monohilo, definida como:

$$S_{\text{implementación}} = \frac{T_{\text{monohilo}}}{T_{\text{implementación}}}$$

- **Aceleración del multihilo frente al monohilo**:

$$S_{\text{multihilo}} = \frac{8.972}{5.615} \approx \mathbf{1.598}$$

  La versión multihilo es aproximadamente un 59.8 % más rápida que la versión secuencial.

- **Aceleración del SIMD frente al monohilo**:

$$S_{\text{SIMD}} = \frac{8.972}{9.488} \approx \mathbf{0.95}$$

  La versión SIMD es más lenta que la versión monohilo los tipos de datos utilizados.

5. Análisis de los resultados

6. División del trabajo