



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	2
1.1. Algoritmo matemático	2
1.2. Funcionamiento general e implementación monohilo	2
1.2.1. Implementación multihilo	3
2. Imágenes de entrada y salida	5
2.1. Implementación Monohilo	5
2.2. Implementación SIMD	6
2.3. Implementación Multihilo	7
3. Comparativa mediante diffImages	8
3.1. SIMD vs Monohilo	8
3.2. Multihilo vs Monohilo	8
4. Resultados experimentales	10
4.1. Fase 1 – SingleThread: DEF	10
4.2. Fase 2 – SIMD	10
4.3. Fase 2 – Multihilo	10
4.4. Cálculo de aceleraciones	10
5. Análisis de los resultados	11
6. División del trabajo	11

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.

1.2.1. 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.

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

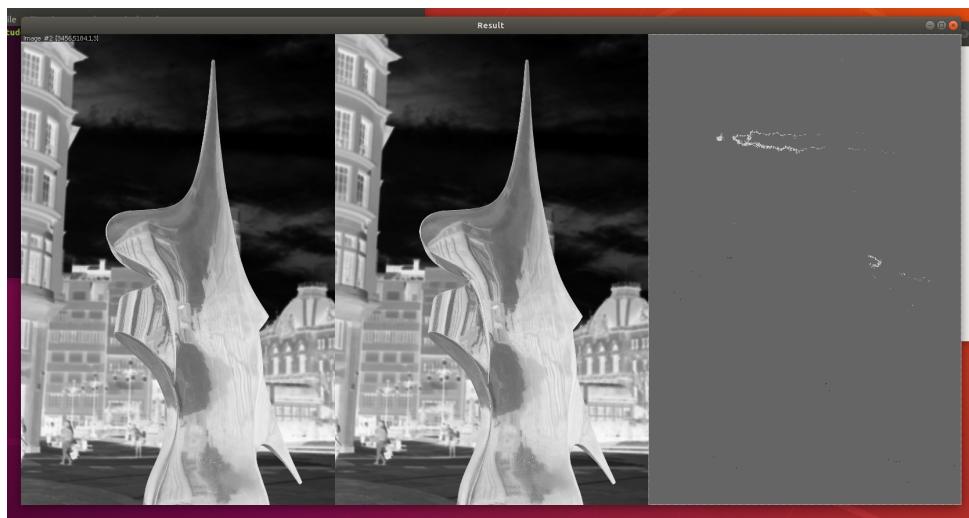
Comparación visual para la implementación multihilo

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.

3.1. SIMD vs Monohilo

En esta figura se incluye la ejecución de la aplicación `diffImages` comparando la imagen resultante del programa SIMD con la generada por la versión monohilo. Idealmente, al tratarse del mismo algoritmo matemático, la diferencia debe ser completamente nula.



Ejecución de `diffImages` comparando SIMD con Monohilo

```
student@zac-teamwork: ~/Desktop/repo
File Edit View Search Terminal Help
student@zac-teamwork:~/Desktop/repo$ ./diffImages 2-mono.bmp 2-simd.bmp

Mean: 0.001997
Variance: 0.002083

student@zac-teamwork:~/Desktop/repo$
```

Salida por consola

3.2. Multihilo vs Monohilo

A continuación, se muestra la comparación entre la salida del programa multihilo y la del programa monohilo. Nuevamente, al implementar exactamente la misma operación sobre cada píxel, el resultado debería ser idéntico.



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

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