Memoria sobre el trabajo en grupo

Grupo A – PL1, Arquitectura de Computadores (21-22)

UO283319 – Juan Francisco Mier Montoto
UO270399 – Gaspar Campos-Ansó Fernández
UO284226 – Pablo del Dago Sordo
UO285381 – Jorge Loureiro Peña

Índice

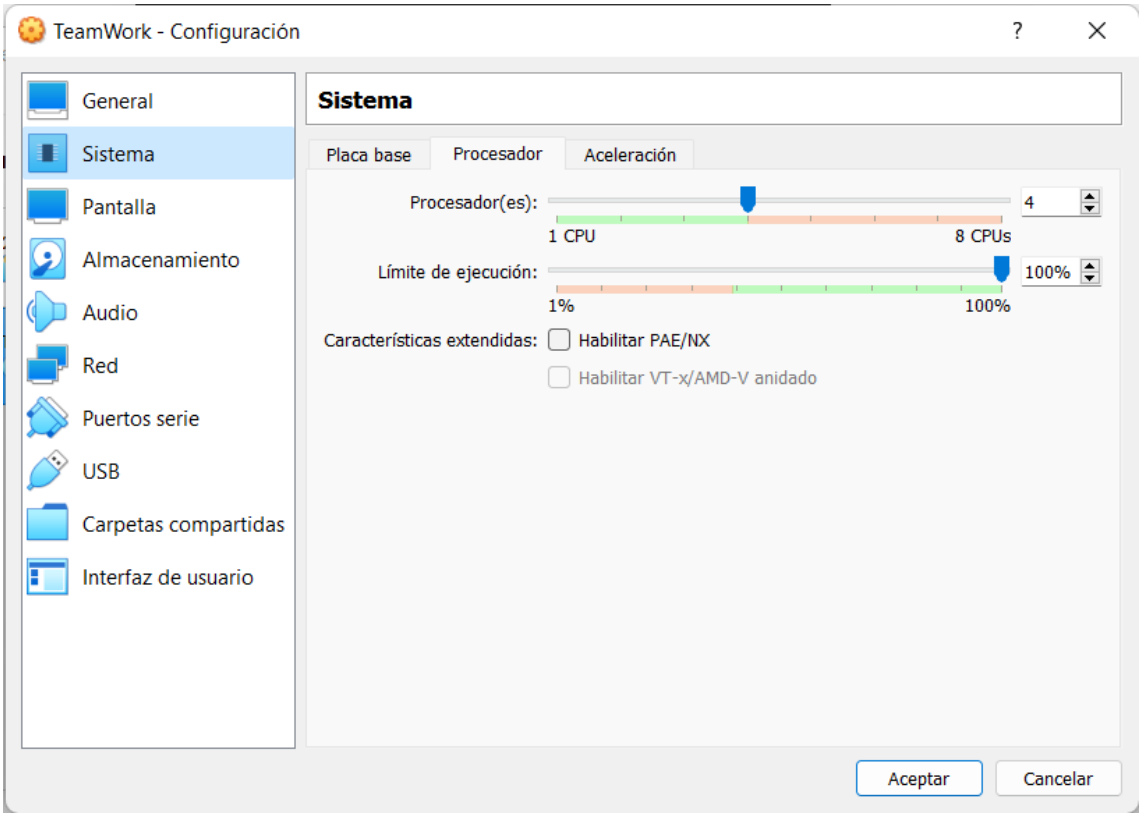
NOTA: se han corregido algunos errores presentes en la primera versión de la memoria.

Repositorio y configuración inicial

Para el desarrollo de esta fase se han seguido todos los pasos incluidos en el enunciado del trabajo:

- Se ha creado un fork en BitBucket del repositorio original para almacenar y compartir el código. Dicho fork se puede encontrar aquí.
- Se han utilizado máquinas virtuales con Ubuntu con las especificaciones indicadas. Para obtener los tiempos se han utilizado 4 cores a 1.2GHz con 8GB de RAM. Las especificaciones más concretas se pueden encontrar en el archivo "cpuinfo" del repositorio.
- Dentro de la propia máquina virtual se ha actualizado Ubuntu a la versión 20.04, se han actualizado todos los paquetes a través de dpkg y se han utilizado algunas extensiones extra dentro de VSCode.

Como especificado, se ha utilizado la siguiente configuración de la máquina virtual:



Fase 1: single-thread

Desarrollo del algoritmo

Explicación del código

La estructura del código viene proporcionada por el repositorio original, de modo que solo hay que hacer unas pequeñas modificaciones para adaptar el código al algoritmo en cuestión, en nuestro caso el nº12.

Como para nuestro algoritmo se requieren mezclar dos imágenes, hay que añadir al código un objeto que almacene la imagen en la que apoyarse. Dicha imagen, proporcionada por el paquete de imágenes de prueba, es “background_V.bmp”, por lo que por defecto el programa buscará esta imagen en el directorio raíz.

```
// Se inicializan los objetos principales y se abren las imágenes.
CImg<data_t> srcImage(SOURCE_IMG);
CImg<data_t> aidImage(HELP_IMG);
```

También se tienen que crear variables que almacenen los componentes RGB de la imagen auxiliar.

```
data_t *pRaid, *pGaid, *pBaid;
```

Con el objetivo de comprobar que el algoritmo en sí funciona, se ha utilizado la misma imagen que la proporcionada en el ejemplo del algoritmo, “bailarina.bmp”. Además, la imagen resultante de la mezcla se guarda en “result.bmp”.

```
const char* SOURCE_IMG      = "bailarina.bmp"; //
const char* HELP_IMG        = "background_V.bmp";
const char* DESTINATION_IMG = "result.bmp"; // nor
```

Antes de comenzar el algoritmo, se comprueban que ambas imágenes tengan el mismo tamaño, puesto que no tendría sentido sumar dos imágenes con diferentes dimensiones.

Se inicializan las componentes:

```
// Punteros a la imagen original
pRsrc = srcImage.data();          // componente roja
pGsrc = pRsrc + height * width; // componente verde
pBsrc = pGsrc + height * width; // componente azul

// Punteros a la imagen de apoyo
pRaid = aidImage.data();          // componente roja
pGaid = pRaid + height * width; // componente verde
pBaid = pGaid + height * width; // componente azul

// Punteros a la imagen resultante
pRdest = pDstImage;                // componente roja
pGdest = pRdest + height * width; // componente verde
pBdest = pGdest + height * width; // componente azul
```

se guarda el tiempo inicial;

```
// Tiempo inicial
if (clock_gettime(CLOCK_REALTIME, &tStart)==-1) {
    printf("Error al obtener el tiempo inicial.");
    exit(1);
}
```

y se comienza con el tratamiento de imágenes. El algoritmo se repite 17 veces para que esté dentro del objetivo puesto en el enunciado de estar entre 5 y 10 segundos.

El algoritmo en cuestión se divide en tres fases:

1. Se aplica la fórmula y se guardan los valores obtenidos.
2. Se comprueban que los valores estén entre 0 y 255.
3. Se aplican los valores al píxel de la imagen resultante.

Traducido a código:

1.
```
// Algoritmo real:      You, a day ago • añadidas repeticio
// Blend: blacken mode #12
red = 255 - ((256 * (255 - pRaid[i])) / (pRsrc[i] + 1));
green = 255 - ((256 * (255 - pGaid[i])) / (pGsrc[i] + 1));
blue = 255 - ((256 * (255 - pBaid[i])) / (pBsrc[i] + 1));
```

2.
```
red = max(min(red, 255), 0);
green = max(min(green, 255), 0);
blue = max(min(blue, 255), 0);
```
3.
```
pRdest[i] = red;
pGdest[i] = green;
pBdest[i] = blue;
```

Resultados de tiempos

Se registran los tiempos con la menor carga de CPU posible para obtener resultados reales:

T1	T2	T3	T4	T5	
6,265039	6,600691	5,797651	6,169624	6,856766	
T6	T7	T8	T9	T10	
5,716565	6,087448	5,568398	6,283128	5,704669	
Media	Desviación T	Interv. Conf-	Interv. Conf+		
6,1049979	0,41716502	5,44382845	6,76616735		

Como esperado, los tiempos están entre cinco y diez segundos. Se utilizarán estos tiempos en la siguiente fase para comparar el rendimiento de los programas y obtener aceleraciones.

Fase 2: multi-thread y SIMD

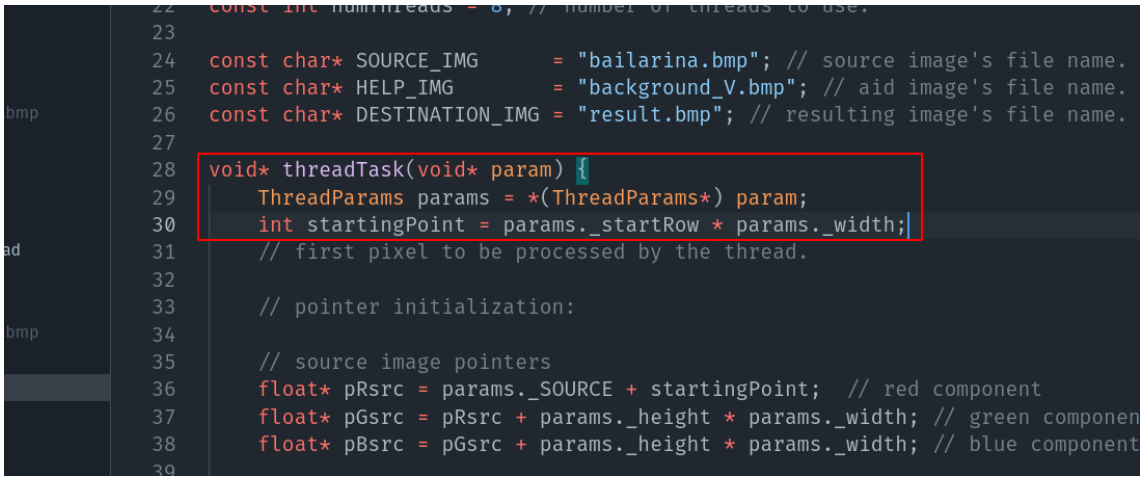
Multi-thread

Desarrollo del algoritmo

Explicación del código

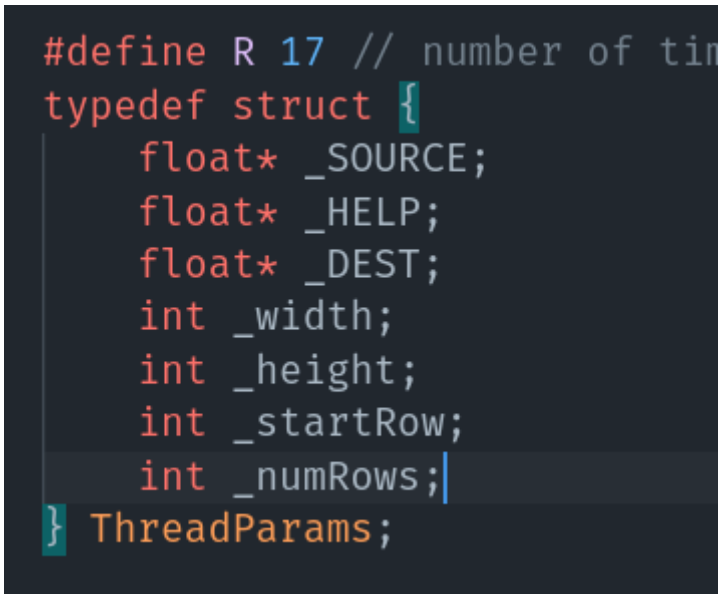
El código está inspirado en la versión single-thread, al igual que la versión SIMD, ya que el objetivo del algoritmo sigue siendo el de juntar dos imágenes en una. La diferencia, obviamente, es que esta vez se utilizan varios hilos, que procesan diferentes partes de la imagen simultáneamente, con lo que el tiempo total de ejecución debería ser mucho inferior.

El componente principal de esta nueva parte del código es la función “threadTask()”:



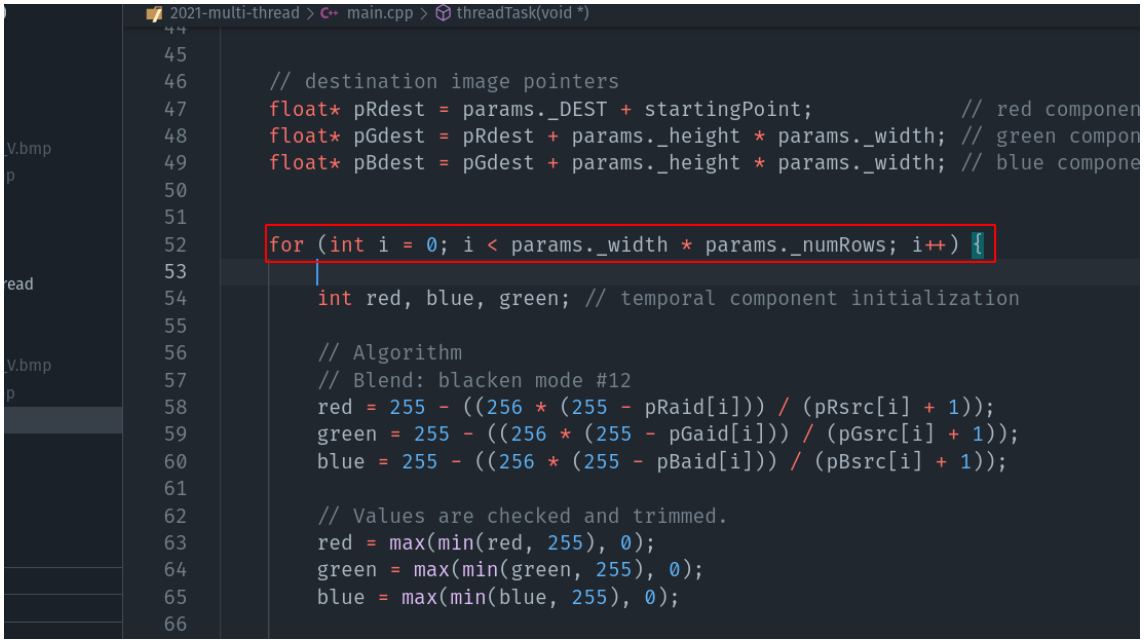
```
23
24  const char* SOURCE_IMG      = "bailarina.bmp"; // source image's file name.
25  const char* HELP_IMG        = "background_V.bmp"; // aid image's file name.
26  const char* DESTINATION_IMG = "result.bmp"; // resulting image's file name.
27
28  void* threadTask(void* param) {
29      ThreadParams params = *(ThreadParams*) param;
30      int startingPoint = params._startRow * params._width;
31      // first pixel to be processed by the thread.
32
33      // pointer initialization:
34
35      // source image pointers
36      float* pRsrc = params._SOURCE + startingPoint;  // red component
37      float* pGsrc = pRsrc + params._height * params._width; // green componen
38      float* pBsrc = pGsrc + params._height * params._width; // blue component
```

Este método recibe un vector de parámetros con estructura “ThreadParams”, que almacena en su interior diferentes atributos sobre las imágenes a procesar y sobre el hilo actual:



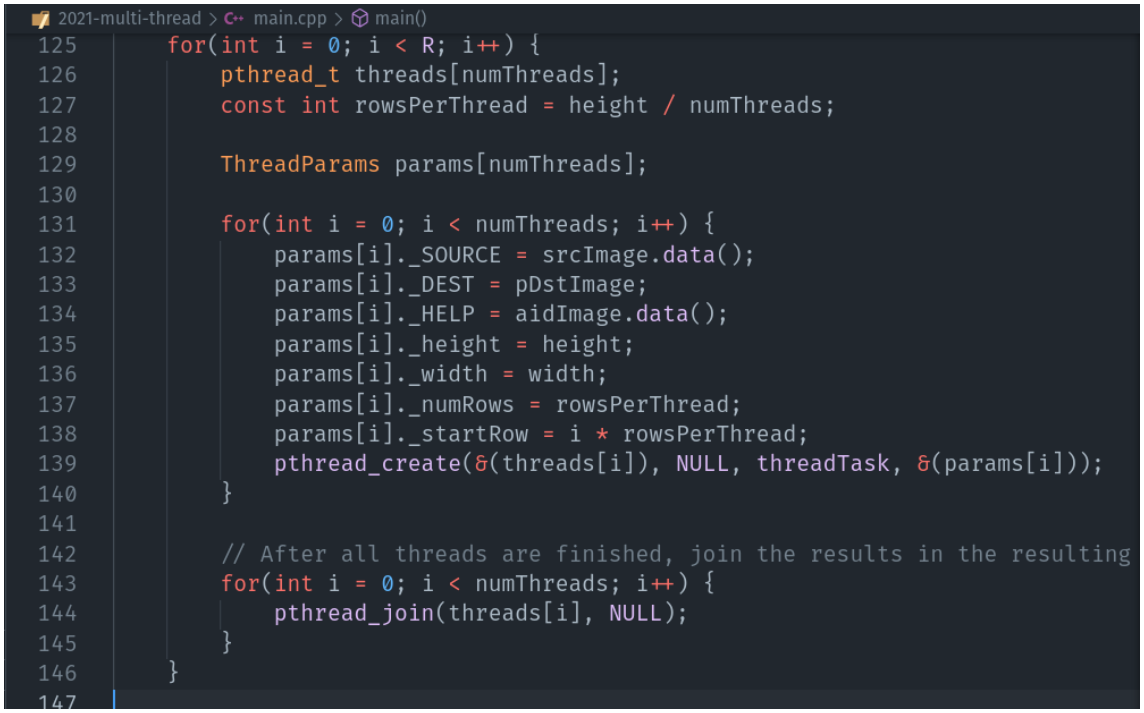
```
#define R 17 // number of tim
typedef struct {
    float* _SOURCE;
    float* _HELP;
    float* _DEST;
    int _width;
    int _height;
    int _startRow;
    int _numRows;
} ThreadParams;
```

Las únicas diferencias a partir de aquí es el nombre de los atributos, que utilizan dicho vector, y la cantidad de veces que se repite el algoritmo, definida por el bucle for:



Esto es, la cantidad de filas a procesar por el hilo multiplicado por la anchura de cada fila.

Dentro del main(), se utiliza la función previamente mencionada para procesar la imagen, y antes de esto, se pasan al vector los parámetros de tipo “ThreadParams”. Una vez que todos los hilos terminan, se unen con el último bucle for de la captura y la imagen se completa.



Resultados de tiempos

Como era de esperar, con las mismas repeticiones que el algoritmo single-thread, el tiempo de ejecución final es bastante inferior a la versión anterior:

	T1	T2	T3	T4	T5
	1,916997	2,116	1,808359	2,186763	2,225248
	T6	T7	T8	T9	T10
	2,192794	1,903139	2,126243	2,041272	1,89332
	Media	Desviación T	Interv. Conf-	Interv. Conf+	Aceleración
	2,0410135	0,14960714	1,8038995	2,2781275	2,99115998

En conclusión, es casi 3 veces más rápido que la versión single-thread. Esto es debido obviamente al procesamiento simultáneo de toda la imagen en comparación a su contraparte mono núcleo.

Versión SIMD

Desarrollo del algoritmo

Explicación del código

Esta versión es estructuralmente igual a la de single-thread. Las pequeñas diferencias se encuentran dentro del propio algoritmo y en algunas variables nuevas.

- Respecto a constantes, tenemos “ITEMSPERPACKET”, que se refiere al número de floats que caben en un paquete __m256. Se utiliza posteriormente para delimitar los bucles. En variables, “nPixels” y “nPackets” son las únicas diferentes, ambas en relación con la constante previamente descrita.
- Dentro del algoritmo, existe un nuevo cauce que sigue la información de las imágenes iniciales hasta llegar a la imagen destino, utilizando instrucciones SIMD exclusivamente y tratando por supuesto con paquetes __m256:
    1. Primero se inicializan los paquetes con los que se va a operar, de tipo “__m256”:

```
// Packets for each image are initialized.
__m256 kRsrc, kGsrc, kBsrc;
__m256 kRaid, kGaid, kBaid;
__m256 kRdest, kGdest, kBdest;
```

    2. Luego, se leen los valores de la imagen y se introducen en los paquetes:

```
// Packets are read and translated from float*
kRsrc = _mm256_loadu_ps(pRsrc);
kGsrc = _mm256_loadu_ps(pGsrc);
kBsrc = _mm256_loadu_ps(pBsrc);

kRaid = _mm256_loadu_ps(pRaid);
kGaid = _mm256_loadu_ps(pGaid);
kBaid = _mm256_loadu_ps(pBaid);
```

3. Se opera con los paquetes como se hace en la versión original:

```
// The algorithm itself using SIMD instructions only.

// (255 - pRaid[i]))
kRdest = _mm256_sub_ps(_mm256_set1_ps(255), kRaid);
kGdest = _mm256_sub_ps(_mm256_set1_ps(255), kGaid);
kBdest = _mm256_sub_ps(_mm256_set1_ps(255), kBaid);

// (256 * (255 - pRaid[i])
kRdest = _mm256_mul_ps(_mm256_set1_ps(256), kRdest);
kGdest = _mm256_mul_ps(_mm256_set1_ps(256), kGdest);
kBdest = _mm256_mul_ps(_mm256_set1_ps(256), kBdest);

// (256 * (255 - pRaid[i])) / (pRsrc[i] + 1)
kRdest = _mm256_div_ps(kRdest, _mm256_add_ps(kRsrc, _mm256_set1_ps(1)));
kGdest = _mm256_div_ps(kGdest, _mm256_add_ps(kGsrc, _mm256_set1_ps(1)));
kBdest = _mm256_div_ps(kBdest, _mm256_add_ps(kBsrc, _mm256_set1_ps(1)));

// 255 - ((256 * (255 - pRaid[i])) / (pRsrc[i] + 1))
kRdest = _mm256_sub_ps(_mm256_set1_ps(255), kRdest);
kGdest = _mm256_sub_ps(_mm256_set1_ps(255), kGdest);
kBdest = _mm256_sub_ps(_mm256_set1_ps(255), kBdest);
```

4. Se comprueba que los valores estén dentro de los valores 0 y 255:

```
// Trim offscale values (<0, >255)
kRdest = _mm256_max_ps(_mm256_set1_ps(0), _mm256_min_ps(_mm256_set1_ps(255), kRdest));
kGdest = _mm256_max_ps(_mm256_set1_ps(0), _mm256_min_ps(_mm256_set1_ps(255), kGdest));
kBdest = _mm256_max_ps(_mm256_set1_ps(0), _mm256_min_ps(_mm256_set1_ps(255), kBdest));
```

5. Por último, se crean vectores que apunten a los paquetes que se encargan de guardar la información de estos directamente a la imagen de destino:

```
// Float pointers to final packets. These allow to select each pixel.
float *prd = (float *) &kRdest;
float *pgd = (float *) &kGdest;
float *pbd = (float *) &kBdest;
```

```
// Convert packets into floats.
for(long unsigned int j = 0; j < ITEMSPERPACKET; j++) {

    *pRdest = *prd;
    *pGdest = *pgd;
    *pBdest = *pbd;
    prd++ ; pgd++ ; pbd++ ;

    pRdest++ ; pBdest++ ; pGdest++ ;
}
```

Al final del algoritmo, se devuelven los punteros a la posición original para que la siguiente repetición repita el algoritmo con los mismos datos sin salirse de la imagen.

```
pRsrc += ITEMSPERPACKET ; pGsrc += ITEMSPERPACKET ; pBsrc += ITEMSPERPACKET ;
pRaid += ITEMSPERPACKET ; pGaid += ITEMSPERPACKET ; pBaid += ITEMSPERPACKET ;
```

El resto del código es exactamente igual a la versión original en single-thread.

Resultados de tiempos

	T1	T2	T3	T4	T5
	2,830456	2,872345	2,976634	2,988612	2,953014
	T6	T7	T8	T9	T10
	2,983439	2,836259	2,909732	2,942618	2,947491
	Media	Desviación T	Interv. Conf-	Interv. Conf+	Aceleración
	2,92406	0,05919746	2,83023729	3,01788271	2,08784974

Como resulta aparente a simple vista, el algoritmo SIMD es más rápido que su contraparte en mono núcleo, pues se opera con paquetes enteros al mismo tiempo en vez de operar pixel a pixel, pero la diferencia de tiempo no es tan grande como la que hay con multi núcleo.

Reparto del trabajo

El trabajo se ha repartido de esta manera:

- Gaspar Campos-Ansó: control de tiempos. (15%)
- Juan Mier: memoria, multi-thread, algoritmo simd, estructura single-thread. (50%)
- Pablo del Dago: algoritmo single-thread, control calidad SIMD. (15%)
- Jorge Loureiro: control de calidad single-thread, estructura SIMD. (20%)