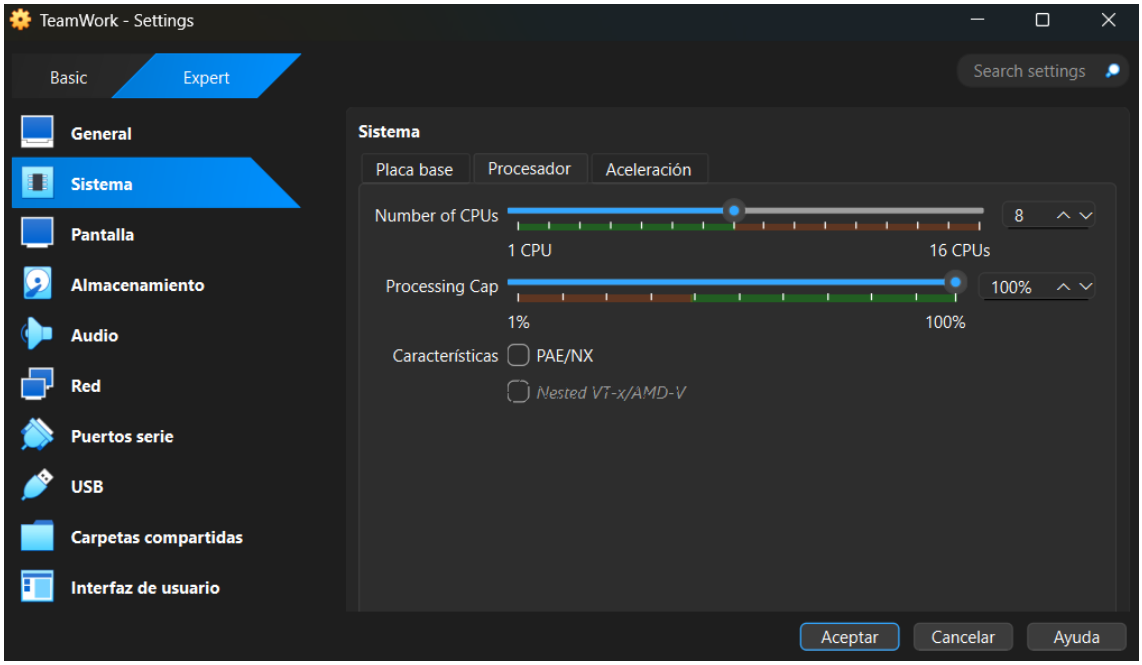
MEMORIA

Andrés Fernández García	UO306603
Yaiza Fernández Vega	UO301762
Pelayo García Díaz	UO306733
Daniel López Fernández	UO289510

CAPTURA DEL PROCESADOR



FASE 1

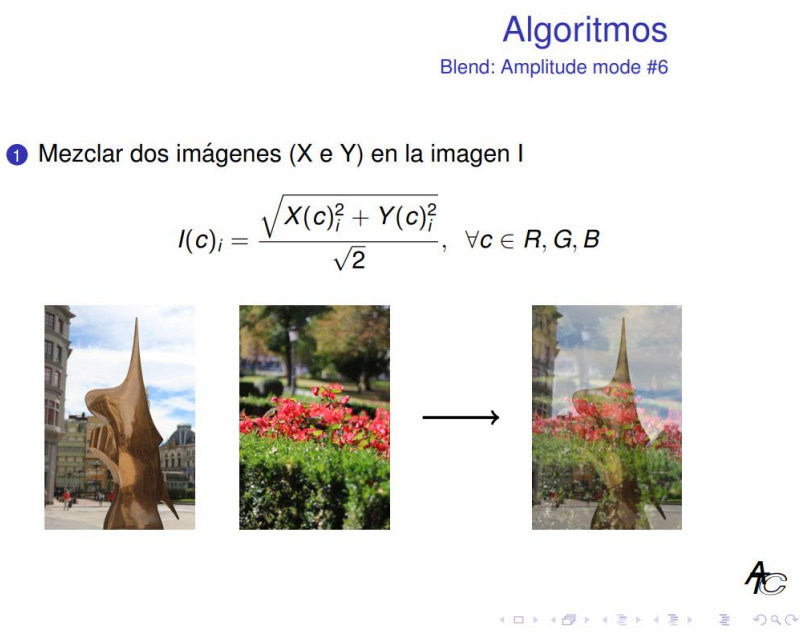
Programa Monohilo

Para realizar el ejercicio lo primero que debemos hacer es añadir la segunda imagen que usaremos para mezclarla con la primera. Además, tendremos que añadir los punteros a los componentes RGB de esta, así como su alto y ancho. Quedando así:

```c
const char *SOURCE_IMG = "bailarina.bmp";
const char *SOURCE_IMG2 = "flores_3.bmp";
const char *DESTINATION_IMG = "bailarina2.bmp";

// Filter argument data type
typedef struct
{
    data_t *pRsrc; // Pointers to the R, G and B components
    data_t *pGsrc;
    data_t *pBsrc;
    data_t *pRsrc2;
    data_t *pGsrc2;
    data_t *pBsrc2;
    data_t *pRdst;
    data_t *pGdst;
    data_t *pBdst;
    uint pixelCount; // Size of the image in pixels
} filter_args_t;
```

Posteriormente debemos implementar el filtro pedido, que en nuestro caso es el siguiente:



Para ello realizamos las siguientes modificaciones en la función filter del esqueleto:

```c
void filter(filter_args_t args)
{
    /*********************************************
     * Algorithm.
     */

    for (uint i = 0; i < args.pixelCount; i++)
    {
        *(args.pRdst + i) = sqrt((*(args.pRsrc + i) * *(args.pRsrc + i)) + (*(args.pRsrc2 + i) * *(args.pRsrc2 + i))) / sqrt(2);
        *(args.pGdst + i) = sqrt((*(args.pGsrc + i) * *(args.pGsrc + i)) + (*(args.pGsrc2 + i) * *(args.pGsrc2 + i))) / sqrt(2);
        *(args.pBdst + i) = sqrt((*(args.pBsrc + i) * *(args.pBsrc + i)) + (*(args.pBsrc2 + i) * *(args.pBsrc2 + i))) / sqrt(2);
    }
}
```

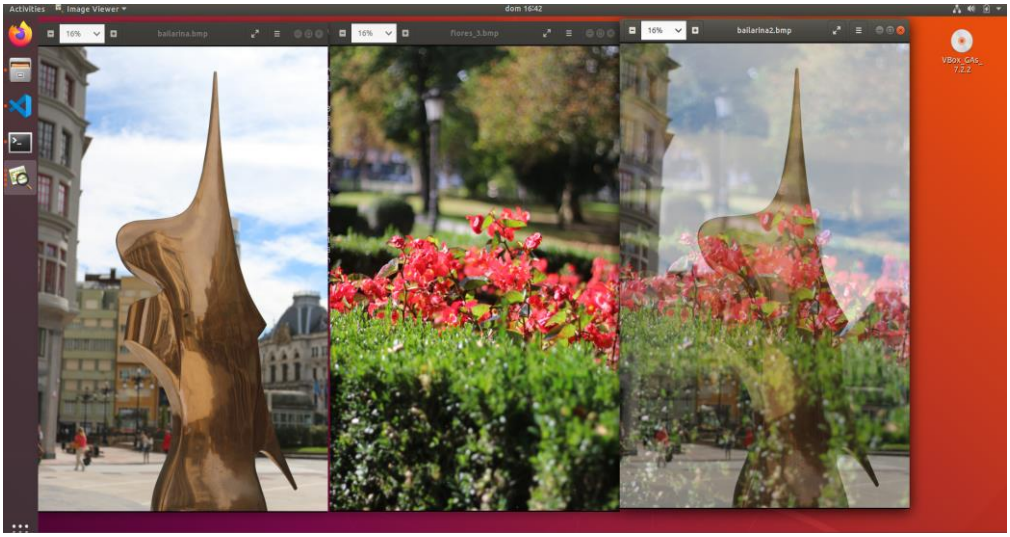
La función recorre todos los píxeles de las imágenes y calcula el nuevo valor de cada componente RGB aplicando el algoritmo exigido. En cada iteración del bucle accede a los valores mediante punteros, realiza la operación de la fórmula y guarda el resultado en los

punteros de destino. Después dentro del main realizamos una llamada a la función dentro de un bucle para aumentar el tiempo de ejecución y así realizar las mediciones necesarias. Todo esto midiendo el tiempo de ejecución del programa con la función clock_gettime.

Por último, añadimos unas comprobaciones previas para que las imágenes fuente existan, que sean del mismo tamaño, y que la llamada a clock_gettime se realice correctamente.

En nuestro caso no es necesario aplicar un balance de color porque la fórmula ya los mantiene equilibrados, sin ningún tipo de saturación.

Resultado:



Medidas:

	t_1 (s)	t_2 (s)	t_3 (s)	t_4 (s)	t_5 (s)	t_6 (s)	t_7 (s)	t_8 (s)	t_9 (s)	t_10 (s)	Media t_1 a t_10 (s)	Desv. Típ. t_1 a t_10 (s)	Intervalo de Confianza de la media (95%) Inf. (s)	Sup. (s)
release	8,6076	8,8232	8,5821	8,7618	8,6215	8,5486	8,6294	8,6017	8,6336	8,7650	8,6575	0,0870	8,4834	8,8315

FASE 2

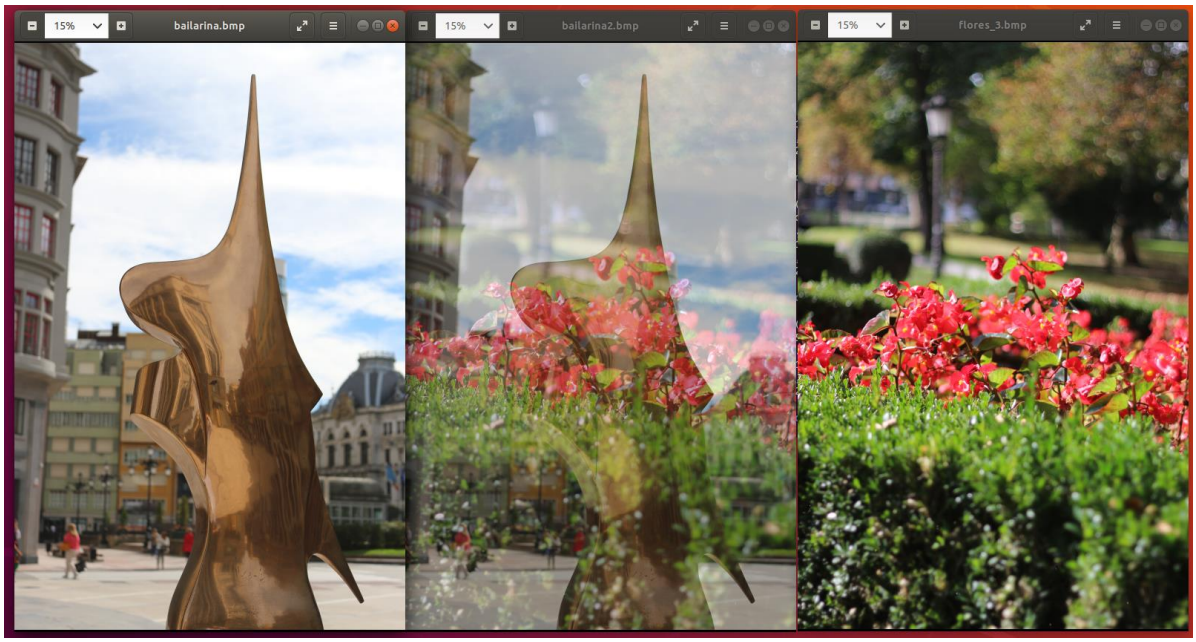
Programa SIMD

	t_1 (s)	t_2 (s)	t_3 (s)	t_4 (s)	t_5 (s)	t_6 (s)	t_7 (s)	t_8 (s)	t_9 (s)	t_10 (s)	Media t_1 a t_10 (s)	Desv. Típica t_1 a t_10 (s)	Intervalo de confianza de la media (95%) Inf. (s)	Sup. (s)	Aceleración (s)
single_thread	8,6076	8,8232	8,5821	8,7618	8,6215	8,5486	8,6294	8,6017	8,6336	8,765	8,65745	0,087003474	8,549420919	8,765479081	
simd	2,578516	2,265321	2,54763	2,301989	2,449401	2,38358	2,386713	2,344787	2,341178	2,318405	2,391752	0,098376694	2,2696012	2,5139028	3,619710572

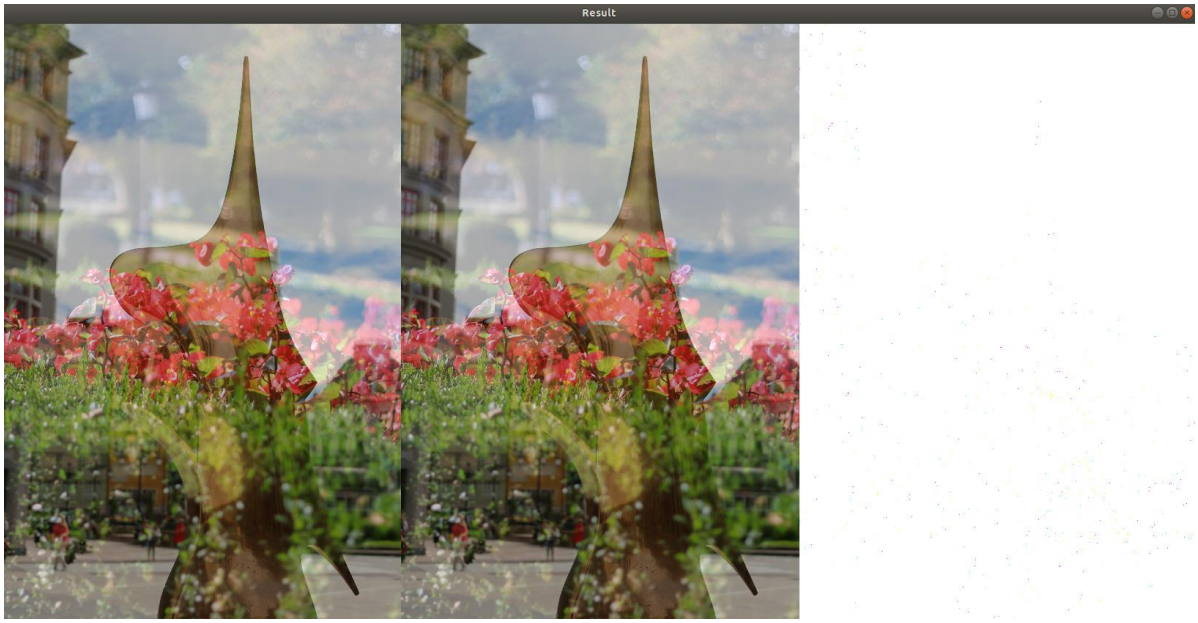
colores de las imágenes a procesar por el programa. El filtro se aplica mediante el uso de instrucciones AVX (__m256), procesando los pixeles en bloques y mejorando el rendimiento del programa respecto a la ejecución secuencial. Se realizaron 10 medidas de forma análoga a como se hizo en la ejecución secuencial para obtener los siguientes resultados:

Como se puede ver, los tiempos de ejecución del programa que utiliza las funciones SIMD son considerablemente menores a los obtenidos con la versión secuencial del programa, resultando en una aceleración de aproximadamente 3.62 respecto a la versión original.

Resultado:



Utilizando la aplicación de diffImages obtenemos una pequeña diferencia respecto a la original, con una media de 0.000392.



Programa Multihilo

El programa multihilo busca paralelizar el filtro de imágenes mediante multiples hilos de forma que el procesamiento se realice de manera concurrente. Cada imagen ocupa una serie de posiciones en la memoria y para paralelizarlo dividimos ese rango de posiciones en el número de hilos a utilizar que en este caso es 4.

Cada hilo ejecuta el filtro pedido y por cada iteración se crean los hilos usando la funcion:

pthread_create(&threads[t], NULL, filter_segment, &tdata[t]);

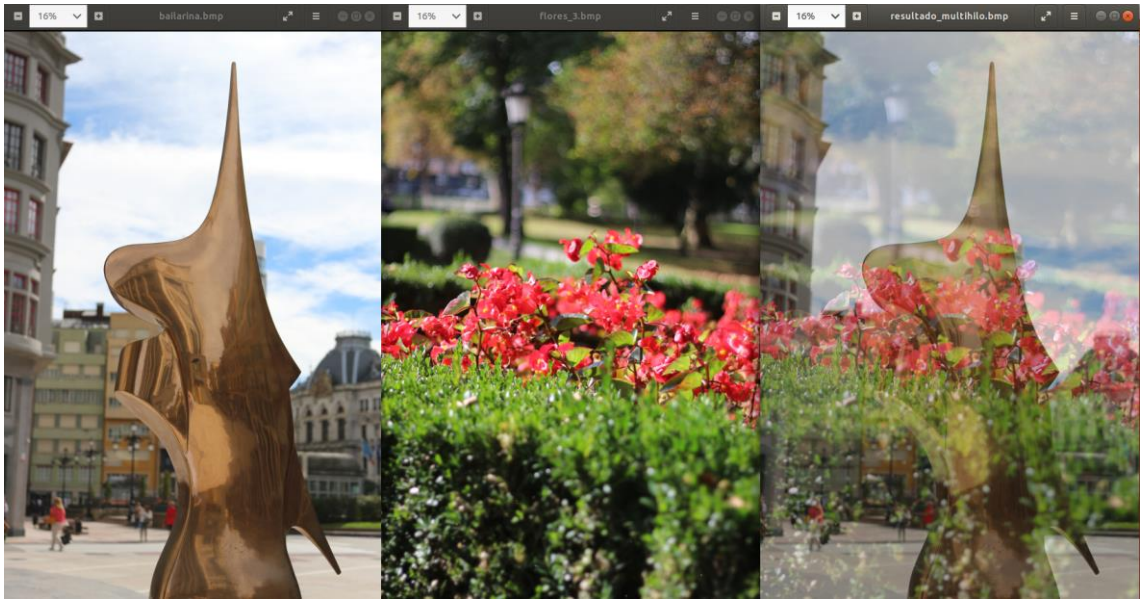
Al finalizar se espera a cada hilo con la función:

pthread_join(threads[t], NULL);

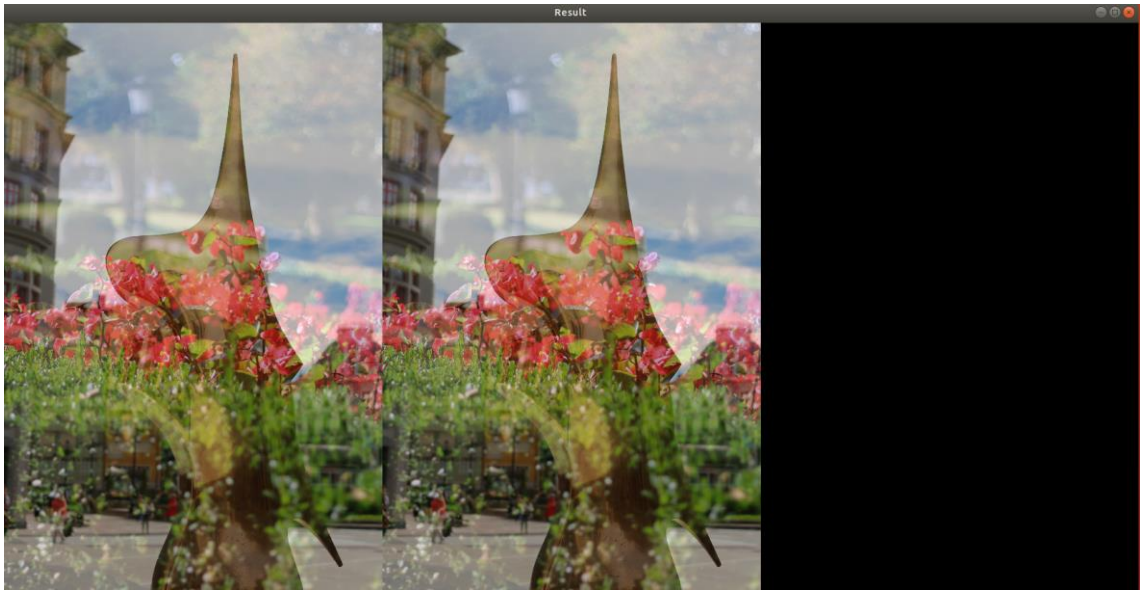
Se ejecuto el programa 10 veces para obtener los tiempos de respuesta mediante la función clock_gettime, para comparar los resultados con el programa monohilo.

	t_1 (s)	t_2 (s)	t_3 (s)	t_4 (s)	t_5 (s)	t_6 (s)	t_7 (s)	t_8 (s)	t_9 (s)	t_10 (s)	Media t_1 a t_10 (s)	Desv. Tipica t_1 a t_10 (s)	Intervalo de confianza de la media (95%)		Aceleración (s)
													Inf. (s)	Sup. (s)	
single_thread	8,6076	8,8232	8,5821	8,7618	8,6215	8,5486	8,6294	8,6017	8,6336	8,765	8,65745	0,087003474	8,483443052	8,831456948	...
multi_thread	0,5830	0,5377	0,5263	0,5629	0,5204	0,5419	0,5510	0,5212	0,5661	0,5304	0,5440802	0,020077735	0,503924729	0,584235671	15,9120843

Como se puede observar los tiempos de respuesta son bastante menores en el programa multihilo, y cuenta con una aceleracion de 15,912 respecto al monohilo. Resultado:



Como podemos observar con el programa diffImages no existe diferencia entre las imágenes, ya que devuelve una imagen completamente en negro.


Result

Cómo se repartió el trabajo:

Andrés Fernández García	Programa monohilo y multihilo
Horas	6 horas, 25%
Yaiza Fernández Vega	Programa monohilo y simd
Horas	6 horas, 25%
Pelayo García Díaz	Programa monohilo y multihilo
Horas	6 horas, 25%
Daniel López Fernández	Programa monohilo y simd
Horas	6 horas, 25%