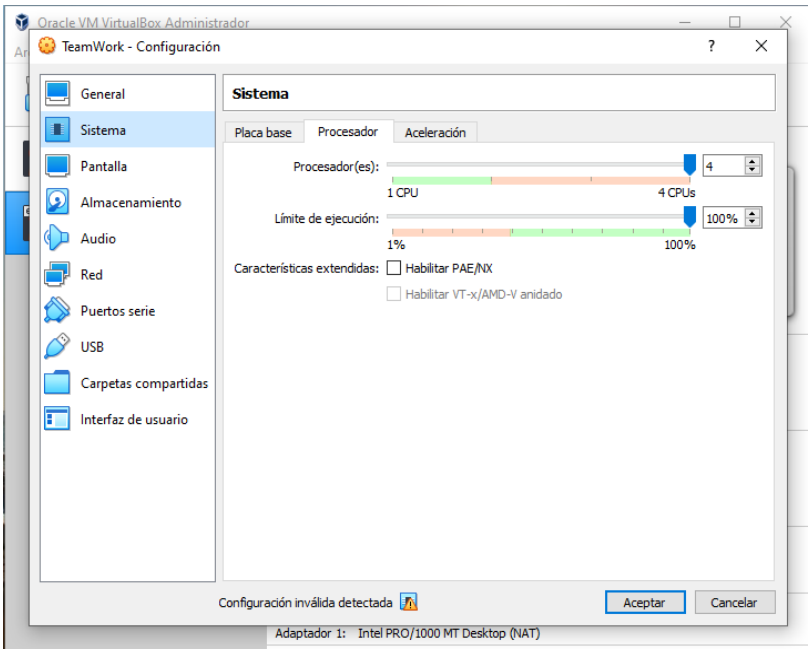
Fase 2

Parte 1: Se desarrolla un programa monohilo con extensiones SIMD.

Configuración inicial de la máquina virtual para la realización del trabajo:



Una vez configurada, comenzamos las modificaciones en el fichero *main.cpp*:

Primero, incluimos las librerías necesarias para el funcionamiento del programa, entre ellas la librería de las funciones intrínsecas.

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <CImg.h>
#include <time.h>
#include <immintrin.h> // Required to use intrinsic functions
```

Almacenamos las componentes de los píxeles de la imagen usando el tipo flotante**.**

```
// Data type for image components
typedef float data_t;
```

Posteriormente, creamos los punteros a la imagen a modificar y a la imagen modificada. Estas fueron, la imagen *uniovi_2.bmp* y, a partir de ella, la imagen *uniovi_2_final.bmp*. También definimos el número de elementos por paquete, usando paquetes __m128 de flotantes, y declaramos el número de repeticiones a usar en el bucle de la medición del tiempo que se verá más adelante.

```
// Pointers to the source image and the destination image
const char *SOURCE_IMG      = "/home/student/Pictures/normal/uniovi_2.bmp";
const char *DESTINATION_IMG = "/home/student/Pictures/uniovi_2_final.bmp";

// Number of items per packet
#define ITEMS_PER_PACKET (sizeof(__m128) / sizeof(data_t))

// Number os repetitions for the loop of the elapsed time
int nREPS = 30;
```

Realizado por: Alejandro Álvarez Solís, Carlos Concheso Cubillas, María Teresa Fernández Coro y Pablo Alonso Alonso.

Método *main()* de la clase:

Intentamos cargar la imagen original y capturamos la excepción en caso de no ser posible, además de mostrar el mensaje pertinente. Creación e inicialización de las variables necesarias posteriormente, obteniendo así la información de las componentes de la imagen.

```
// Try to load the source image
// If it is not possible, catch the exception
cimg::exception_mode(0);
try {

    // Open file and object initialization
    CImg<data_t> srcImage(SOURCE_IMG);

    // Creating the variables
    data_t *pRsrc, *pGsrc, *pBsrc;  // Pointers to the R, G and B components
    data_t *pDstImage;  // Pointer to the new image pixels
    uint width, height; // Width and height of the image
    uint nComp; // Number of image components

    // Variables initialization
    srcImage.display(); // Displays the source image
    width = srcImage.width();   // Getting information from the source image
    height = srcImage.height();
    nComp = srcImage.spectrum();
```

Definimos el tamaño del array según las dimensiones de la imagen con la que trabajamos. A continuación, pedimos memoria dinámicamente con la función _mm_malloc() para poder almacenar las componentes de la imagen resultante del algoritmo.

```
 // Array size. Note: It is not a multiple of 8
#define VECTOR_SIZE (width * height)

// Allocate memory space for destination image components
pDstImage = (data_t *) _mm_malloc(width * height * nComp * sizeof(data_t), sizeof(data_t));
if (pDstImage == NULL) {
    perror("Allocating destination image");
    exit(-2);
}
```

Los punteros de las componentes RGB de la imagen original apuntan ahora al componente del array correspondiente.

```
// Pointers to the component arrays of the source image
pRsrc = srcImage.data();        // pRcomp points to the R component array
pGsrc = pRsrc + height * width; // pGcomp points to the G component array
pBsrc = pGsrc + height * width; // pBcomp points to the B component array
```

Creamos las variables para la medición del tiempo transcurrido en la ejecución del programa y otra donde almacenar el número de paquetes del array resultante.

```
// Create the variables for the time measurement
struct timespec tStart, tEnd;
double dElapsedTimeS = 0;

// Calculate the number of packets of the resulting array
int nPackets = (VECTOR_SIZE * sizeof(data_t) / sizeof(__m128));
```

Comprobamos si es necesario incrementar el número de paquetes y creamos los paquetes de 128 bits.

Como se explica en el código, los vectores de datos creados antes no tienen por qué estar alojados en posiciones de memoria alineadas al tamaño de paquete usado

(__m128), para acceder a su contenido es necesario utilizar una instrucción vectorial de carga desalineada. Utilizamos como destino una variable temporal por cada vector.

Definimos e inicializamos los paquetes del tipo __m128.

```
// If it is not an exact number we need to add one more packet
if (((VECTOR_SIZE * sizeof(data_t)) % sizeof(__m128)) != 0) {
    nPackets++;
}

// 32 bytes (128 bits) packets. Used to stored aligned memory data
__m128 vred, vgreen, vblue, vrg, vrgb;

// Define the packets that multiply the components
__m128 paquete03, paquete059, paquete011, paquete255;

// Initialize those packets
paquete03 = _mm_set1_ps(0.3);
paquete059 = _mm_set1_ps(0.59);
paquete011 = _mm_set1_ps(0.11);
paquete255 = _mm_set1_ps(255);
```

Tomamos el tiempo de comienzo de la ejecución del algoritmo.

```
// Start time
if (clock_gettime(CLOCK_REALTIME, &tStart) == -1) {
    printf("ERROR: clock_gettime: %d, \n", errno);
    exit(EXIT_FAILURE);
}
```

A continuación, comenzamos el bucle de la medición de tiempo para nuestro algoritmo. Realizamos 30 repeticiones como en el programa monohilo de la fase 1 del trabajo, la variable del número de repeticiones se encuentra declarada antes de comenzar el método main().

En este bucle se realizarán las operaciones con funciones SIMD necesarias para obtener el resultado deseado por nuestro algoritmo, es decir, la modificación de una imagen a color a una en escala de grises.

En la suma de dos vectores alineados necesitamos hacer el casting adecuado para el tipo __m128.

Tras procesar los paquetes necesitaremos comprobar si el número de elementos del vector es múltiplo del número de elementos por paquete, en caso de no ser así, es necesario calcular el número de elementos en exceso e iterar secuencialmente dichos elementos.

```
// Loop with the same number of repetitions as the single-thread version
for (int i = 0; i < nREPS; i++) {
    // Data arrays vred, rgreen and vblue must not be memory aligned to __m128 data (32 bytes)
    // so we use intermediate variables to avoid execution errors
    // We make an unaligned load of vred, rgreen and vblue
    vred = _mm_loadu_ps(pRsrc);
    vgreen = _mm_loadu_ps(pGsrc);
    vblue = _mm_loadu_ps(pBsrc);

    // Begin the treatment of the first packet group
    // Multiply the components and the constants
    vred = _mm_mul_ps(vred, paquete03);
    vgreen = _mm_mul_ps(vgreen, paquete059);
    vblue = _mm_mul_ps(vblue, paquete011);

    // Performs the addition of two aligned vectors
    // each vector containing 128 bits / 32 bytes = 4 floats
    // Intermediate step to put together all the components
    vrg = _mm_add_ps(vred, vgreen);

    // Performs the addition of two aligned vectors
    // each vector containing 128 bits / 32 bytes = 4 floats
    // All the components are in dst img array with this operation
    *(__m128 *)pDstImage = _mm_add_ps(vrg, vblue);
```

Realizado por: Alejandro Álvarez Solís, Carlos Concheso Cubillas, María Teresa Fernández Coro y Pablo Alonso Alonso.

```
// Treatment of the rest of the packets
for (int i = 0; i < nPackets; i++){
    // Load next packet
    vred = _mm_loadu_ps((pRsrc + ITEMS_PER_PACKET * i));
    vgreen = _mm_loadu_ps((pGsrc + ITEMS_PER_PACKET * i));
    vblue = _mm_loadu_ps((pBsrc + ITEMS_PER_PACKET * i));

    // Multiply the components and the constants
    vred = _mm_mul_ps(vred, paquete03);
    vgreen = _mm_mul_ps(vgreen, paquete059);
    vblue = _mm_mul_ps(vblue, paquete011);

    // Intermediate step to put all the components together
    vrg = _mm_add_ps(vred, vgreen);

    // Last step to put the components together
    vrgb = _mm_add_ps(vrg, vblue);

    // Save the info into the dst img
    *(__m128 *)(pDstImage + ITEMS_PER_PACKET * i) = _mm_sub_ps(paquete255, vrgb);
}
```

```
    // Calculate the elements in excess
    int dataInExcess = (VECTOR_SIZE) % (sizeof(__m128) / sizeof(data_t));

    // Surplus data can be processed sequentially
    for (int i = 0; i < dataInExcess; i++) {
        *(pDstImage + 2 * ITEMS_PER_PACKET + i) = *(pRsrc + 2 * ITEMS_PER_PACKET + i) +
                                                  *(pGsrc + 2 * ITEMS_PER_PACKET + i) +
                                                  *(pBsrc + 2 * ITEMS_PER_PACKET + i);
    }
}
```

Tras finalizar la ejecución del algoritmo, tomamos el valor del tiempo y calculamos el tiempo transcurrido.

Como se comentó anteriormente, el resultado deseado por nuestro algoritmo es la modificación de una imagen a color a una en escala de grises, es por ello que la imagen final debería tener solo el componente de la luminosidad (L). Es necesario establecer el número de componente de la imagen como 1, el usado en imágenes en blanco y negro.

Por último, creamos y guardamos la imagen resultante.

```
// End time
if (clock_gettime(CLOCK_REALTIME, &tEnd) == -1) {
    printf("ERROR: clock_gettime: %d, \n", errno);
    exit(EXIT_FAILURE);
}

// Calculate the elapsed time in the execution of the algorithm
dElapsedTimeS = (tEnd.tv_sec - tStart.tv_sec);
dElapsedTimeS += (tEnd.tv_nsec - tStart.tv_nsec) / 1e+9;

// Using nComp = 1 for B/W images
nComp = 1;

// Create a new image object with the calculated pixels
CImg<data_t> dstImage(pDstImage, width, height, 1, nComp);
dstImage.save(DESTINATION_IMG);
```

Sin olvidarnos de liberar el espacio de memoria proporcionado por la función _mm_malloc() al comienzo del programa mediante el uso de la función _mm_free().

```
// Free the memory allocated at the beginning
_mm_free(pDstImage);
```

Realizado por: Alejandro Álvarez Solís, Carlos Concheso Cubillas, María Teresa Fernández Coro y Pablo Alonso Alonso.

Parte 2: Se desarrolla un programa multihilo.

Con la misma configuración de la máquina virtual, comenzamos las modificaciones en el fichero *main.cpp*:

Primero, incluimos las librerías necesarias para el funcionamiento del programa.

```
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <CImg.h>
#include <time.h>
#include <iostream>
#include <stdlib.h>
#include <pthread.h>
```

Almacenamos las componentes de los píxeles de la imagen usando el tipo flotante**.**

Posteriormente, creamos los punteros a la imagen a modificar y a la imagen modificada. Estas fueron, la imagen *uniovi_2.bmp* y, a partir de ella, la imagen *uniovi_2_final.bmp*.

El número de hilos empleados coincide con el nivel de concurrencia del sistema, es decir, el sistema dispone de un procesador de 4 núcleos, usamos entonces 4 hilos.

```
// Data type for image components
typedef float data_t;

// Pointers to the source image and the destination image
const char* SOURCE_IMG      = "/home/student/Pictures/normal/uniovi_2.bmp";
const char* DESTINATION_IMG = "/home/student/Pictures/uniovi_2_final.bmp";

// Number of threads
#define NUM_HILOS  4
```

Declaramos la función de conversión de la foto para cada hilo y el número de repeticiones a usar en el bucle de la medición del tiempo transcurrido en la ejecución del algoritmo, son 30 repeticiones como en el programa monohilo y el de extensiones SIMD.

Creamos la estructura para el procesamiento de los hilos y de la foto. También, declaramos las variables para los componentes de la imagen que serán posteriormente inicializados.

```
// Function declaration
void *photoConversor(void *v);

// Number os repetitions for the loop of the elapsed time
int nREPS = 30;

// Structure of piels to process
typedef struct {
    int pixel_init;
    int pixel_fin;
} thread_structure;

typedef struct {
    // Creating the variables for the image components
    uint width, height; // Width and height of the image
    data_t *pRsrc, *pGsrc, *pBsrc; // Pointers to the R, G and B components
    data_t *pLdest; // Pointer to the L component
    data_t L;   // Luminosity of the image
} photoStructure;

photoStructure photo;
```

Método *main()* de la clase:

Al igual que en el programa monohilo con extensiones SIMD intentamos cargar la imagen original y capturamos la excepción en caso de no ser posible, además de

mostrar el mensaje pertinente. A continuación, creamos e inicializamos las variables necesarias para obtener la información de las componentes de la imagen.

Realizamos una petición de memoria dinámicamente con la función malloc() para poder almacenar las componentes de la imagen resultante del algoritmo.

```
// Creating the variables for the image components
data_t *pDstImage; // Pointer to the new image pixels
uint nComp; // Number of image components

// Variables initialization
srcImage.display(); // Displays the source image
photo.width  = srcImage.width(); // Getting information from the source image
photo.height = srcImage.height();
nComp  = srcImage.spectrum();

// Allocate memory space for destination image components
pDstImage = (data_t *) malloc (photo.width * photo.height * nComp * sizeof(data_t));
if (pDstImage == NULL) {
    perror("Allocating destination image");
    exit(-2);
}
```

Los punteros de las componentes RGB y L de la imagen original apuntan ahora al componente del array correspondiente.

```
// Pointers to the componet arrays of the source image
photo.pRsrc = srcImage.data();                          // pRcomp points to the R component array
photo.pGsrc = photo.pRsrc + photo.height * photo.width; // pGcomp points to the G component array
photo.pBsrc = photo.pGsrc + photo.height * photo.width; // pBcomp points to the B component array
photo.pLdest = pDstImage;                               // pLdest points to the L component array
```

Al igual que en el programa monohilo con extensiones SIMD, explicado anteriormente, realizamos la medición del tiempo transcurrido en la ejecución del programa. Para ello, comenzamos creando las variables necesarias y tomando el tiempo de comienzo de la ejecución.

El resultado deseado por nuestro algoritmo es la modificación de una imagen a color a una en escala de grises. La imagen final, por tanto, debería tener solo el componente de la luminosidad (L). Es necesario establecer el número de componente de la imagen como 1, el usado en imágenes en blanco y negro.

```
// Create the variables for the time measurement
struct timespec tStart, tEnd;
double dElapsedTimeS = 0;

// Start time
if (clock_gettime(CLOCK_REALTIME, &tStart) == -1) {
    printf("ERROR: clock_gettime: %d, \n", errno);
    exit(EXIT_FAILURE);
}

// Using nComp=1 for B/W images
nComp = 1;
```

Creamos vectores del tamaño del número de hilos establecido previamente. También, damos tamaño a los hilos de la imagen.

```
// Arrays of threads
pthread_t threads[NUM_HILOS];
thread_structure thread[NUM_HILOS];

// Size of the image threads
int tamImageThread = photo.width * photo.height / NUM_HILOS;
```

Realizamos el algoritmo de la medición de tiempo con la ejecución de los hilos. Se requiere realizar un bucle para poder obtener resultados significativos.

En este bucle se realiza la llamada al método que realiza la conversión de la foto para cada hilo, mostrado a continuación.

```
// Time measurement and threads execution
// Making a loop so we obtain significant results
// We considerate the differences between last thread and the others
for (int j = 0; j < nREPS; j++) {
    for (int h = 0; h < NUM_HILOS; h++) {
        thread[h].pixel_init = tamImageThread * h;
        thread[h].pixel_fin = thread[h].pixel_init + tamImageThread;

        if (h == NUM_HILOS-1)
            thread[h].pixel_fin = photo.width * photo.height;

        int pthread_ret = pthread_create(
            &threads[h],
            NULL,
            photoConversor,
            &(thread[h])
        );

        if(pthread_ret) {
            printf("ERROR: phtread_create error code: %d \n", pthread_ret);
            exit(EXIT_FAILURE);
        }

        pthread_join(threads[h], NULL);
    }
}
```

```
// Conversor for the photo in every single thread
void *photoConversor(void *v) {
    thread_structure *thread = (thread_structure*)v;

    // Initial and final pixels
    int pixel_init = thread->pixel_init;
    int pixel_fin = thread->pixel_fin;

    for (long i = pixel_init; i < pixel_fin; i++) {
        photo.L = 0.3 * photo.pRsrc[i] + 0.59 * photo.pGsrc[i] + 0.11 * photo.pBsrc[i];
        photo.L = 255 - photo.L;
        photo.pLdest[i] = photo.L;
    }

    return NULL;
}
```

Por último, tomamos el valor del tiempo al finalizar la ejecución del algoritmo. Calculamos el tiempo transcurrido y, creamos y guardamos la imagen resultante.

```
    // End time
    if (clock_gettime(CLOCK_REALTIME, &tEnd) == -1) {
        printf("ERROR: clock_gettime: %d, \n", errno);
        exit(EXIT_FAILURE);
    }

    // Calculate the elapsed time in the execution of the algorithm
    dElapsedTimeS = (tEnd.tv_sec - tStart.tv_sec);
    dElapsedTimeS += (tEnd.tv_nsec - tStart.tv_nsec) / 1e+9;

    // Create a new image object with the calculated pixels
    CImg<data_t> dstImage(pDstImage, photo.width, photo.height, 1, nComp);
    dstImage.save(DESTINATION_IMG);
```

Realizado por: Alejandro Álvarez Solís, Carlos Concheso Cubillas, María Teresa Fernández Coro y Pablo Alonso Alonso.

Tabla con mediciones y estadísticas del programa monohilo con extensiones SIMD:

Elapsed time												Media Single-thread
0,478947	0,461650	0,498520	0,486062	0,490180	0,501941	0,464259	0,463829	0,460994	0,458250	0,427448	0,421749	6,339908388
Media		Desviación típica		Intervalo de confianza (95%)				Productividad		Aceleración		
0,466533962		0,025263296		Inferior		Superior		128,6080006		20,28546671		
				0,41600737		0,517060553						

Tabla con mediciones y estadísticas del programa multihilo:

Se calcula la productividad multiplicándola por el número de hilos.

Elapsed time												Media Single-thread
0,870961	0,726462	0,727851	0,758827	0,649513	0,654086	0,660732	0,665903	0,667966	0,663197	0,674482	0,660026	6,339908388
Media		Desviación típica		Intervalo de confianza (95%)				Productividad		Aceleración		
0,693532475		0,064752833		Inferior		Superior		346,0544512		54,58350974		
				0,56402681		0,82303814						

En ambas tablas se encuentra la media del tiempo transcurrido en la ejecución del programa monohilo de la fase 1, y respecto a dicho valor se calcula la aceleración.

En conclusión, se puede apreciar como la aceleración del programa multihilo es mayor a la del monohilo con extensiones SIMD, y por tanto posee un mayor rendimiento.

Estimación del porcentaje del proyecto que representa el trabajo realizado por cada miembro:

Parte 1, programa monohilo con extensiones SIMD: Alejandro Álvarez Solís 25% y María Teresa Fernández Coro 25%.

Parte 2, programa multihilo: Carlos Concheso Cubillas 25% y Pablo Alonso Alonso 25%.

Realizado por: Alejandro Álvarez Solís, Carlos Concheso Cubillas, María Teresa Fernández Coro y Pablo Alonso Alonso.