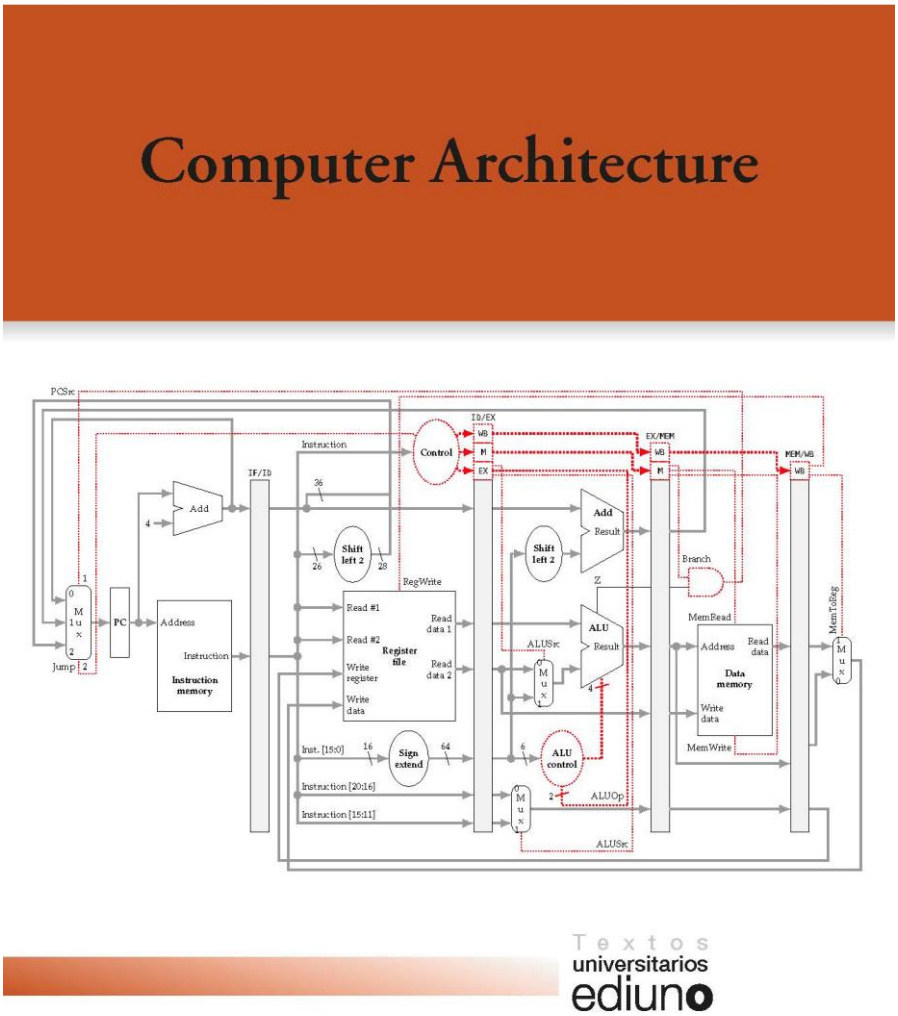
COMPUTER ARCHITECTURE TEAMWORK

Juan Carlos Granda Candás
José María López López
Manuel García Vázquez
Julio Molleda Meré
Rubén Usamentiaga Fernández
Joaquín Entrialgo Castaño
Francisco Javier de la Calle Herrero



Authors:

Mikel Fernández Esparta UO275688

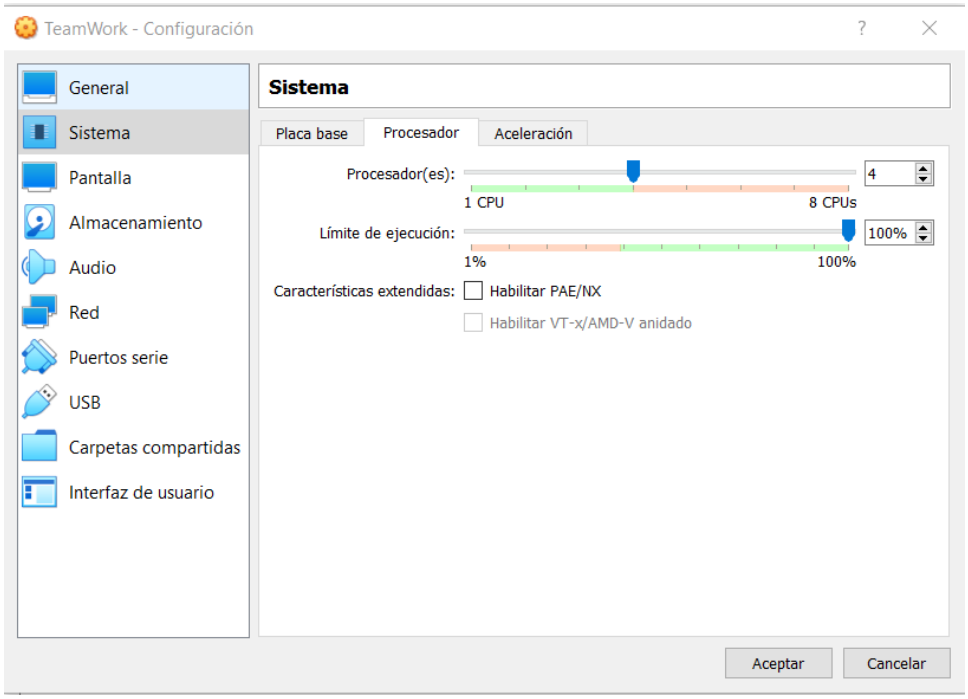
Dana Álvarez Murillo UO278249

Álvaro Díaz Saavedra UO278532

Group: PL-I-4 D

PHASE 1

The settings processor tab of our Virtual Machine for Teamwork.



CPU info

We execute the command “cat proc/cpuinfo”, in order to get more information of our CPU, we also saved a copy of it in the cpuinfo.txt file within the zip file.

```
student@2ac-teamwork:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 140
model name      : 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
stepping        : 1
cpu MHz         : 2803.200
cache size      : 12288 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht sys
call nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 cx16 pcid sse4
_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervisor lahf_lm abm 3dnowprefetch invpcid_single fsgsbase avx2 invpc
id rdseed clflushopt md_clear flush_l1d arch_capabilities
bugs            : spectre_v1 spectre_v2 spec_store_bypass swapgs
bogomips        : 5606.40
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

```
processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 140
model name      : 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
stepping        : 1
cpu MHz         : 2803.200
cache size      : 12288 KB
physical id     : 0
siblings        : 4
core id         : 1
cpu cores       : 4
apicid          : 1
initial apicid  : 1
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht sys
call nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 cx16 pcid sse4
_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervisor lahf_lm abm 3dnowprefetch invpcid_single fsgsbase avx2 invpc
id rdseed clflushopt md_clear flush_l1d arch_capabilities
bugs            : spectre_v1 spectre_v2 spec_store_bypass swapgs
bogomips        : 5606.40
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:

processor       : 2
vendor_id       : GenuineIntel
cpu family      : 6
model           : 140
model name      : 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
stepping        : 1
cpu MHz         : 2803.200
cache size      : 12288 KB
physical id     : 0
siblings        : 4
core id         : 2
cpu cores       : 4
apicid          : 2
initial apicid  : 2
fpu             : yes
```

```
id rdseed clflushopt md_clear flush_l1d arch_capabilities
bugs            : spectre_v1 spectre_v2 spec_store_bypass swapgs
bogomips        : 5606.40
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:

processor       : 2
vendor_id       : GenuineIntel
cpu family      : 6
model           : 140
model name      : 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
stepping        : 1
cpu MHz         : 2803.200
cache size      : 12288 KB
physical id     : 0
siblings        : 4
core id         : 2
cpu cores       : 4
apicid          : 2
initial apicid  : 2
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht sys
call nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 cx16 pcid sse4
_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervisor lahf_lm abm 3dnowprefetch invpcid_single fsgsbase avx2 invpc
id rdseed clflushopt md_clear flush_l1d arch_capabilities
bugs            : spectre_v1 spectre_v2 spec_store_bypass swapgs
bogomips        : 5606.40
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

```
processor       : 3
vendor_id       : GenuineIntel
cpu family      : 6
model           : 140
model name      : 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
stepping        : 1
cpu MHz         : 2803.200
cache size      : 12288 KB
physical id     : 0
siblings        : 4
core id         : 3
cpu cores       : 4
apicid          : 3
initial apicid  : 3
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht sys
call nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 cx16 pcid sse4
_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervisor lahf_lm abm 3dnowprefetch invpcid_single fsgsbase avx2 invpc
id rdseed clflushopt md_clear flush_l1d arch_capabilities
bugs            : spectre_v1 spectre_v2 spec_store_bypass swapgs
bogomips        : 5606.40
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

For the teamwork, we had to implement the “Blend: whiten mode#11”. For this case we use as input images bailarina.bmp and background_V.bmp, the output image will be saved in bailarina2.bmp.

We must use the base data type double.

```c
// Data type for image components
typedef double data_t;

const char* SOURCE_IMGX      = "bailarina.bmp";
const char* SOURCE_IMGY      = "background_V.bmp";
const char* DESTINATION_IMG = "bailarina2.bmp";
```

The information of the rgb arrays will be stored in the following struct:

```c
// Filter argument data type
typedef struct {
    data_t *pRsrcX; // Pointers to the R, G and B components
    data_t *pGsrcX;
    data_t *pBsrcX;
    data_t *pRsrcY; // Pointers to the R, G and B components
    data_t *pGsrcY;
    data_t *pBsrcY;
    data_t *pRdst;
    data_t *pGdst;
    data_t *pBdst;
    uint pixelCount; // Size of the image in pixels
} filter_args_t;
```

Single thread version

In the main.cpp file we completed all the TODO and FIXME labels, to later on calculate the average time it takes to run the program. Just like we learned in previous sessions of the laboratory classes. And finally save the destination image on the corresponding file we specified before on the constants.

```
/*****************************************************
 * TODO: Variables initialization.
 *    - Prepare variables for the algorithm
 *    - This is not included in the benchmark time
 */
struct timespec tStart, tEnd;
double dElapsedTimeS;
```

The algorithm that is needed to use to get both images to blend into one:

1. Blend of two images (X and Y) into image I

$$I(c)_i = \frac{256 \times Y(c)_i}{(255 - X(c)_i) + 1}, \quad \forall c \in R, G, B$$

This would be the code that we implemented in the filter method.

For all the destination pointers to the three colors (Red R, Green G and Blue B), we iterate until the number of pixels and apply the algorithm shown in the previous image. In the numerator the multiplication of the “Y” image (background image) and on the denominator the source image of the ballerina. And if it exceeds the maximum value 255, set the value to the maximum one.

```
void filter (filter_args_t args) {
    for (uint i = 0; i < args.pixelCount; i++) {
        *(args.pRdst + i) = (*(args.pRsrcY + i) * 256) / (( 255 - *(args.pRsrcX + i)) + 1);
        *(args.pGdst + i) = (*(args.pGsrcY + i) * 256) / (( 255 - *(args.pGsrcX + i)) + 1);
        *(args.pBdst + i) = (*(args.pBsrcY + i) * 256) / (( 255 - *(args.pBsrcX + i)) + 1);

        if (*(args.pRdst + i) > 255) {*(args.pRdst + i) = 255; }

        if (*(args.pGdst + i) > 255) {*(args.pGdst + i) = 255; }

        if (*(args.pBdst + i) > 255) {*(args.pBdst + i) = 255; }
    }
}
```

PHASE 2

SIMD single-threaded version:

In this version we used __m128d for the size of the vectors and define the size and items per packet.

```
#define VECTOR_SIZE       18 // Array size. Note: It is not a multiple of 8
#define ITEMS_PER_PACKET (sizeof(__m128)/sizeof(double))
typedef double data_t;
```

We also must create some other new variables of type __m128d for the arrays of colors of the source images and the resulting one for the output. For simplification and easier for us to manage some are just auxiliary variables that were used to compute the multiplications and additions. And initialize the constant value for 256.

```
__m128d RsrcY, GsrcY, BsrcY, RsrcX, GsrcX, BsrcX;
__m128d num256 = _mm_set1_pd(256);
__m128d resRsrc256,resGsrc256,resBsrc256;
__m128d res255MinusRsrc,res255MinusGsrc,res255MinusBsrc;
__m128d resFinR,resFinG,resFinB; //Res ffinal result after division
```

We will call the function in the main this way, iterating 25 times to get better results to for the average time:

```
if(clock_gettime(CLOCK_REALTIME, &tStart)){
    perror("clock_gettime");
    exit(EXIT_FAILURE);
}

for (uint j = 0; j < 25; j++) {
    filter(filter_args);
}

if(clock_gettime(CLOCK_REALTIME, &tEnd)){
    perror("clock_gettime");
    exit(EXIT_FAILURE);

}
```

In this algorithm to process the images, we had to iterate over the number of pixels of the image (the width * height) by the items per packet that are needed for each.

First, we load the values of the pointers for the source colors of the images and add 1 so that it increases with the necessary intrinsic for __m128, _mm_loadu_pd.

Then to multiply the value we use _mm_mul_pd passing as parameters the variables for the source color and the value 256 in the variable num256.

After that, the corresponding subtraction on the denominator with _mm_sub_pd and lastly store the value of the division in the resFinX variables by using _mm_div_pd and then _mm_storeu_pd.

Just like in the single-thread version previously, the condition in case some of the values exceed the maximum one, 255.

```
void filter (filter_args_t args) {
    for (uint i = 0; i < args.pixelCount; i+=ITEMS_PER_PACKET) {
        RsrcY = _mm_loadu_pd(args.pRsrcY + i);
        GsrcY = _mm_loadu_pd(args.pGsrcY + i);
        BsrcY = _mm_loadu_pd(args.pBsrcY + i);

        RsrcX = _mm_loadu_pd(args.pRsrcX + i);
        GsrcX = _mm_loadu_pd(args.pGsrcX + i);
        BsrcX = _mm_loadu_pd(args.pBsrcX + i);

        resRsrc256 = _mm_mul_pd(RsrcY,num256);
        resGsrc256 = _mm_mul_pd(GsrcY,num256);
        resBsrc256 = _mm_mul_pd(BsrcY,num256);

        res255MinusRsrc = _mm_sub_pd(num256,RsrcX);
        res255MinusGsrc = _mm_sub_pd(num256,GsrcX);
        res255MinusBsrc = _mm_sub_pd(num256,BsrcX);

        resFinR = _mm_div_pd(resRsrc256,res255MinusRsrc);
        resFinG = _mm_div_pd(resGsrc256,res255MinusGsrc);
        resFinB = _mm_div_pd(resBsrc256,res255MinusBsrc);

        _mm_storeu_pd(args.pRdst + i,resFinR);
        _mm_storeu_pd(args.pGdst + i,resFinG);
        _mm_storeu_pd(args.pBdst + i,resFinB);
```

```
        for(uint j = 0; j < ITEMS_PER_PACKET;j++) {
            if ((args.pRdst + i)[j] > 255) {(args.pRdst + i)[j] = 255; }
            if ((args.pGdst + i)[j] > 255) {(args.pGdst + i)[j] = 255; }
            if ((args.pBdst + i)[j] > 255) {(args.pBdst + i)[j] = 255; }
        }
    }
}
```

Multi-thread version:

First, we must calculate the number of threads that are needed depending on the hardware.

This number will be 4 as the number of cores is 4 an each parallel thread needs 1 core to run on parallel.

```
const uint NUM_THREADS = std::thread::hardware_concurrency();
```

Then, we define a struct that will be used by all threads. Each thread has the information of the rows that it will process while executing the filter as well as the struct containing the rgb pointers of the source images and the destination image. The filter method is the same as in the mono-thread version but now it returns void* so it can be added to the threads in their creation.

```
struct threadStruct{
    uint startRow, numberOfRows;
    filter_args_t args;
};


void* filter (void* t) {
    threadStruct* thread = (threadStruct*) t;
    threadStruct threadS = *thread;

    for (uint i =  threadS.startRow; i < threadS.args.pixelCount; i++) {
        *(threadS.args.pRdst + i) = (*(threadS.args.pRsrcY + i) * 256) / (( 255 - *(threadS.args.pRsrcX + i)) + 1);
        *(threadS.args.pGdst + i) = (*(threadS.args.pGsrcY + i) * 256) / (( 255 - *(threadS.args.pGsrcX + i)) + 1);
        *(threadS.args.pBdst + i) = (*(threadS.args.pBsrcY + i) * 256) / (( 255 - *(threadS.args.pBsrcX + i)) + 1);

        if (*(threadS.args.pRdst + i) > 255) {*(threadS.args.pRdst + i) = 255; }

        if (*(threadS.args.pGdst + i) > 255) {*(threadS.args.pGdst + i) = 255; }

        if (*(threadS.args.pBdst + i) > 255) {*(threadS.args.pBdst + i) = 255; }
    }
    return NULL;
}
```

In the main method, we declare the thread arrays before starting to measure the time and define the sections of rows that will be given to each thread depending on the number of threads.

```
uint sections = filter_args.pixelCount / NUM_THREADS; //Sections in which the image is divided according to the number of threads
pthread_t threads[NUM_THREADS]; //Array of threads to be used to execute the algorithm in parallel
struct threadStruct threadStructs[NUM_THREADS]; //Array of our own structs used to give the needed details to the threads to be used
```

Finally, the changed algorithm (repeated 25 times, as in the previous cases) creates the threads and passes them to the filter_args containing the information on the images and the method filter that will be executed.

```
for(uint j = 0; j < 25; j++) {
    for(uint i = 0; i < NUM_THREADS; i++){ //Repeating the process for each thread to be created
        threadStructs[i] = {}; //Making all the values of the struct 0 to make it easier for us to give the values of the attributes
        if(i != 0){  //Placing the corresponding start row to the threads
            threadStructs[i].startRow = threadStructs[i-1].startRow + threadStructs[i-1].numberOfRows + 1;
        }

        threadStructs[i].numberOfRows = sections; //Places the correct number of rows in the threads

        if( i == NUM_THREADS-1){ //Fixes the number of rows corresponding to the last thread by assigning to it the remaining pixels
            threadStructs[i].numberOfRows = filter_args.pixelCount - threadStructs[i].startRow;
        }

        threadStructs[i].args = filter_args;

        if(pthread_create(&threads[i], NULL, filter, &threadStructs[i])!=0){ //Creates the thread with the given values
            perror("creating thread");
            exit(EXIT_FAILURE);
        }
    }
}
```

After the execution the threads are joined and the image is stored.

```
for(uint i = 0; i< NUM_THREADS; i++){ //Iterates over the array of threads joining them
    if(0!=pthread_join(threads[i],NULL)){
        perror("join");
        exit(EXIT_FAILURE);
    }
}
```

Conclusion:

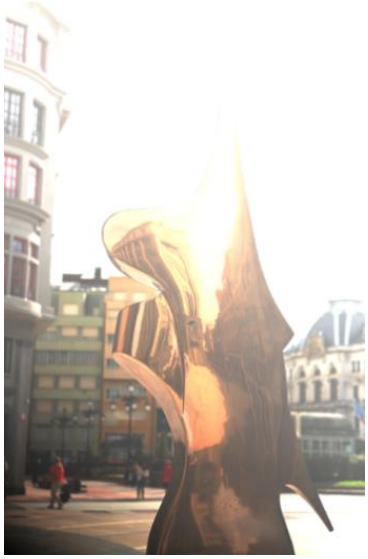
As we see in the spreadsheet, the average time in multi-thread is better than single thread. On the other hand, the performance of the single thread version with SIMD extensions is worse as executing such a simple program is more complicated and slower using intrinsic.

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	Average	Std. Dev.	Confidence lower 95%	Confidence higher 95%	Speedup
Single thread	7,52	7,2	7,21	6,62	6,69	7,53	6,66	6,64	6,92	7,37	7,036	0,373	6,709	7,363	x
Multi thread	6,74	6,82	7,11	6,94	6,79	6,83	6,96	6,38	6,65	6,85	6,807	0,196	6,635	6,979	1,034
SIMD	9,17	9,31	9,31	9,54	9,14	9,68	9,43	9,15	9,2	9,5	9,343	0,187	9,179	9,507	0,753

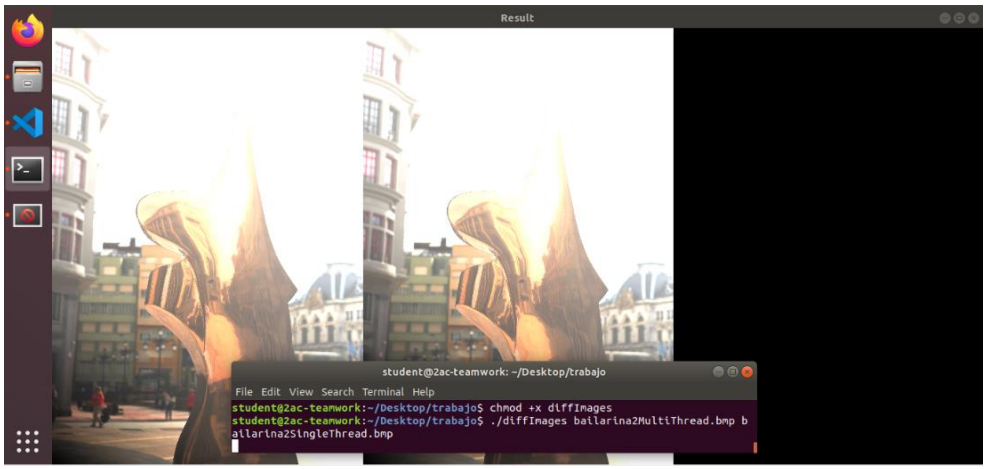
Image comparison:

These are the original images used in the programs before blending the first one to the second one.

The resulting image is this one. As you can see in the comparisons below, the image is the same after the execution of the three different programs.



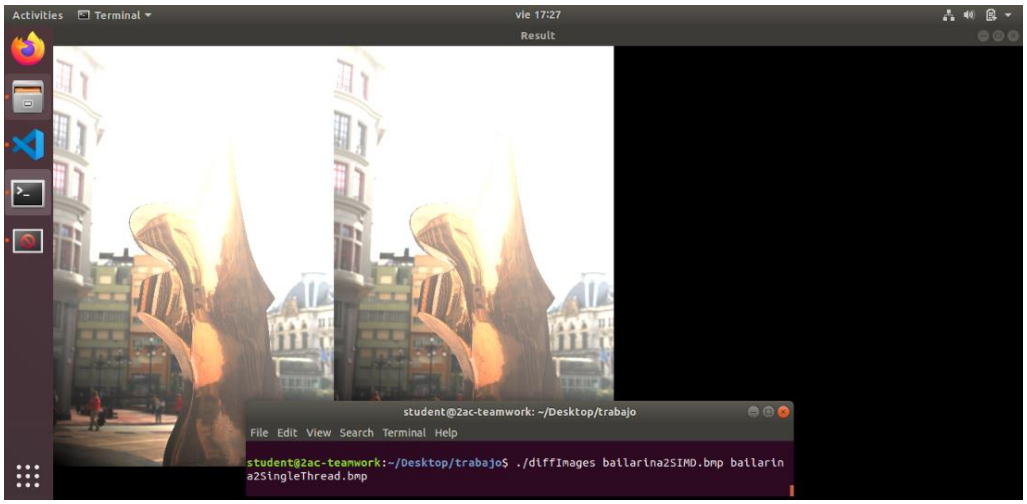
-Comparison between the monothread program and the multithread program:




```
student@2ac-teamwork:~/Desktop/trabajo$ ./diffImages bailarina2MultiThread.bmp bailarina2SingleThread.bmp


Mean: 0.000000
Variance: 0.000000
```

-Comparison between the monothread program and the SIMD program:




```
student@2ac-teamwork:~/Desktop/trabajo$ ./diffImages bailarina2SIMD.bmp bailarina2SingleThread.bmp


Mean: 0.000000
Variance: 0.000000
```

Work distribution

We decided to work the three of us together through meetings on Discord for a couple of days to implement the algorithms for the image processing filter. We helped one another by searching for information online, sharing our ideas for the code, making sure they were the correct way to implement it etc. The report and measures times were also done all together.