

AC Teamwork

Group PLI2-B

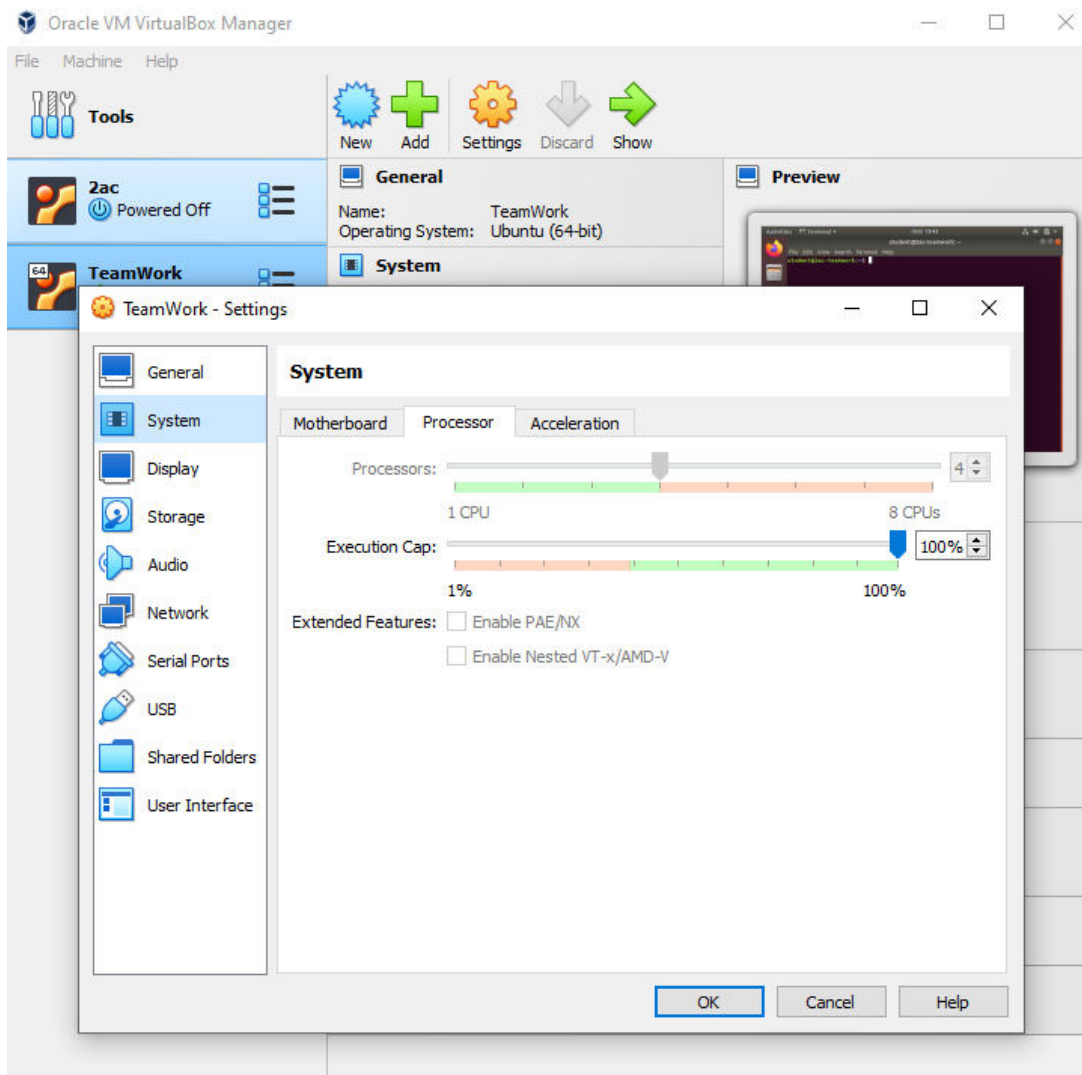
Eduardo Pérez Sánchez	UO294909
Jorge Carriles Ruiz	UO294721
Ana Pérez Bango	UO294100
Paula Díaz Álvarez	UO294067

Phase 1.....	2
Settings of the Virtual Machine.....	2
CPU information.....	2
Algorithm.....	5
Results.....	6
Measurements.....	8
Phase 2.....	9
SIMD.....	9
Algorithm.....	9
Results.....	11
Measurements.....	12
Multithread.....	13
Algorithm.....	13
Results.....	15
Measurements.....	16
Results.....	17
Work division.....	18

Phase 1

Link: [single-thread](#)

Settings of the Virtual Machine



CPU information

```
processor      : 0
vendor_id     : GenuineIntel
cpu family    : 6
model         : 158
model name    : Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
stepping      : 9
cpu MHz       : 2807.998
cache size    : 6144 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 syscall 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 pti fsgsbase bmi1 avx2 bmi2 invpcid
rdseed clflushopt md_clear flush_lld
bugs : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds
swapgs itlb_multihit srbds mmio_stale_data retbleed
bogomips : 5615.99
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 : 158
model name : Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
stepping : 9
cpu MHz : 2807.998
cache size : 6144 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 syscall 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 pti fsgsbase bmi1 avx2 bmi2 invpcid
rdseed clflushopt md_clear flush_lld
bugs : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds
swapgs itlb_multihit srbds mmio_stale_data retbleed
bogomips : 5615.99
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 : 158

```

model name : Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
stepping   : 9
cpu MHz    : 2807.998
cache size : 6144 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 syscall 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 pti fsgsbase bmi1 avx2 bmi2 invpcid
rdseed clflushopt md_clear flush_lld
bugs       : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds
swapgs itlb_multihit srbds mmio_stale_data retbleed
bogomips   : 5615.99
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       : 158
model name  : Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
stepping    : 9
cpu MHz     : 2807.998
cache size  : 6144 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 syscall 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 pti fsgsbase bmi1 avx2 bmi2 invpcid
rdseed clflushopt md_clear flush_lld

```

```
bugs      : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass lltf mds
swapgs itlb_multihit srbds mmio_stale_data retbleed
bogomips  : 5615.99
clflush size      : 64
cache_alignment  : 64
address sizes    : 39 bits physical, 48 bits virtual
power management:
```

Algorithm

Algorithms Blend #4

- ① Blend two images (X and Y) into image I
- ② It uses a blend level C in range [0, 1]

$$I(c)_i = C \times X(c) + (1 - C) \times Y(c), \forall c \in R, G, B$$



We developed an algorithm to blend 2 images (X and Y) into a destination image (D) according to a blend level c in $[0, 1]$. So for each pixel of the image we need to do:

$$D(color) = c \cdot X(color) + (1 - c) \cdot Y(color) \quad \forall color \in R, G, B$$

So, we need a function passing all the arguments with all the pointers for each color for each pixel for each image; and the blend level.

First, we need to check if the blend level is between 0 and 1. If it's not in that range we throw an exception.

Then, for each pixel in the images we apply the formula for each color (Red, Green and Blue).

After each, we need to check if the value for each color in the destination image is in a valid range between the MIN_PIXEL_COLOR (0) and MAX_PIXEL_COLOR (255).

- If the color is less than MIN_PIXEL_COLOR, we set it to the MIN_PIXEL_COLOR.
- If the color is greater than MAX_PIXEL_COLOR, we set it to the MAX_PIXEL_COLOR.

```
void filter (filter_args_t args, float c) {
    if (c<0 || c>1){
        perror("The c parameter is invalid.");
        exit(-2);
    }
}
```

```

for (uint i = 0; i < args.pixelCount; i++) {
    addColors((args.pRsrc + i),(args.pRsrc2 + i),(args.pRdst + i),c);
    addColors((args.pGsrc + i),(args.pGsrc2 + i),(args.pGdst + i),c);
    addColors((args.pBsrc + i),(args.pBsrc2 + i),(args.pBdst + i),c);
}

}

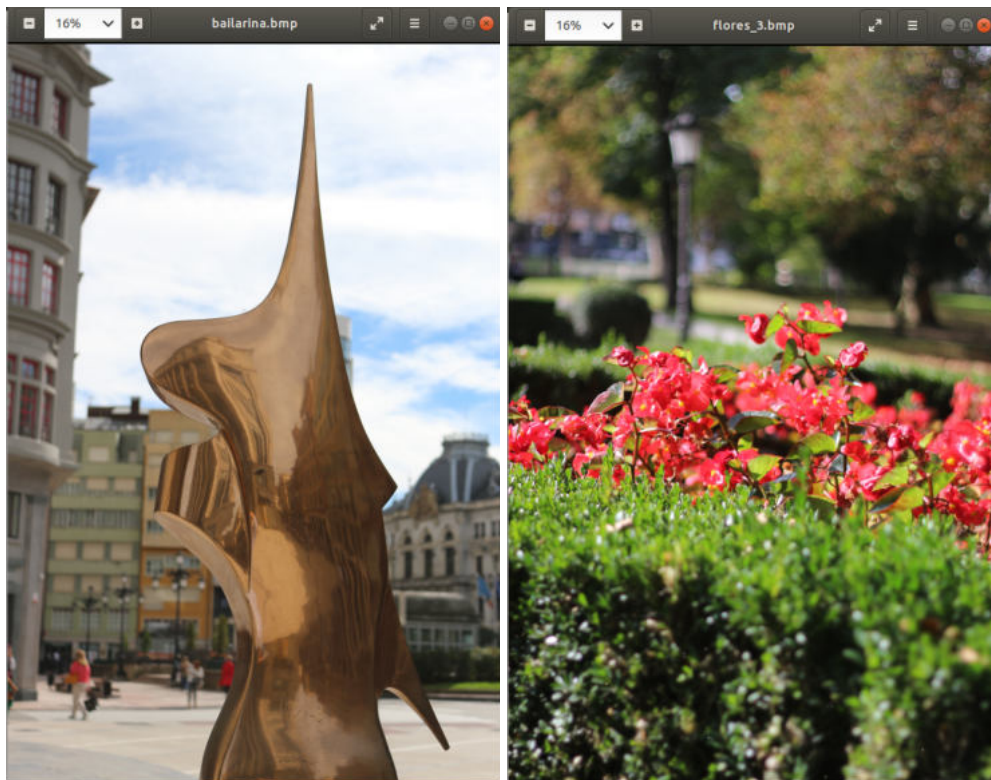
void addColors(data_t* pSrc, data_t* pSrc2, data_t* pDst, float c){
    *pDst = *(pSrc)*c + *(pSrc2)*(1-c);
    checkColorsInBound(*pDst);
}

void checkColorsInBound(data_t &pDstColor){
    if ((pDstColor)>MAX_PIXEL_COLOR){
        (pDstColor)=MAX_PIXEL_COLOR;
    }
    if ((pDstColor)<MIN_PIXEL_COLOR){
        (pDstColor)=MIN_PIXEL_COLOR;
    }
}


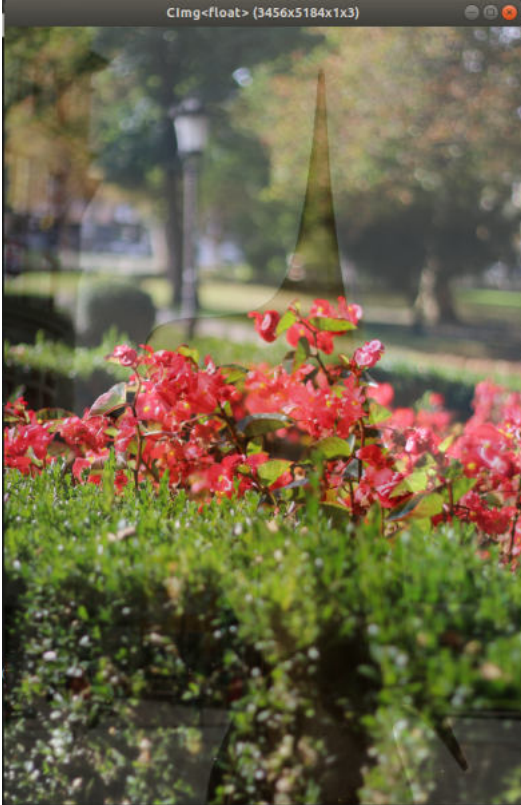

```

Results

Source Images



Result Images

Solution		
	0	0.25
Solution		
Blend level	0.5	

Solution		
Blend level	0.75	1

Measurements

Response time	6.862018	6.589900	6.542546	6.553357	6.531901
	6.612769	6.637149	6.693308	6.574751	6.657206

The mean:

$$\begin{aligned}\bar{x} &= \frac{\Sigma Time}{10} = \frac{6.862018+6.612769+6.589900+6.637149+6.542546+6.693308+6.553357+6.574751+6.531901+6.657206}{10} = \\ &= \frac{66.254905}{10} = 6.625491\end{aligned}$$

The standard deviation:

$$\begin{aligned}s &= \sqrt{\frac{\sum_{i=1}^{10} (x_i - \bar{x})^2}{10-1}} = \sqrt{\frac{0.055945+0.000162+0.001267+0.000136+0.006880+0.004599+0.005203+0.002575+0.008759+0.001006}{9}} = \\ &= \sqrt{\frac{0.086532}{9}} = 0.098954\end{aligned}$$

The confidence interval (95%):

$$\begin{aligned}[\bar{x} - 1.96 \cdot \frac{s}{\sqrt{10}}, \bar{x} + 1.96 \cdot \frac{s}{\sqrt{10}}] &= [6.625491 - 1.96 \cdot \frac{0.098954}{\sqrt{10}}, 6.625491 + 1.96 \cdot \frac{0.098954}{\sqrt{10}}] = \\ &= [6.564159, 6.686823]\end{aligned}$$

Phase 2

SIMD

Link: [simd](#)

Algorithm

The SIMD algorithm consists in dividing the images into packets with a defined number of items (in this case the size is 256 divided by the size of the data_t, that is, $256/32 = 8$ floats)

So we add to the destination image:

- vSrc (extracts 8 elements from the source image array) multiplied by the blend level
- vSrc2 (extracts 8 elements from the source image 2 array) multiplied by one minus the blend level

In each iteration, it will take the next 8 items until the number of packets is reached (number of bits of the Images divided by 256 bits, that is, the number of pixels (width*height*numberOfComponents) divided by 256 bits).

As the number of pixels might be not multiple of 8, then some pixels won't be in the last packet. So we have to process them separately with the filterDataInExcess method.

This method gets the number of pixels that weren't processed (dataInExcess) and applies the filter formula to each one of them

For example if the images were of 19 pixels, then there would be 2 packets of 8 elements ($19/8$) and 3 elements ($19\%8 = \text{dataInExcess}$) that weren't processed

```
for (int i=0; i<nPackets;i++){
    vSrc = _mm256_loadu_ps(src + ITEMS_PER_PACKET*i)
    vSrc2 = _mm256_loadu_ps(src2 + ITEMS_PER_PACKET*i);
    *(__m256 *) (dst + ITEMS_PER_PACKET*i) = _mm256_add_ps(vSrc*c, vSrc2*(1-c));
}
filterDataInExcess(src, src2, dst, c);
checkColorsInBounds(dst);
```

```
void filterDataInExcess(data_t* src, data_t* src2, data_t* dst, float c) {
    //Number of pixels in excess (not multiple of ITEMS_PER_PACKET)
    int dataInExcess = (PIXEL_COUNT)%(sizeof(__m256)/sizeof(data_t));

    // Calculate the addition for each pixel in excess
    for (int i =0; i< dataInExcess; i++){
        *(dst + nPackets * ITEMS_PER_PACKET + i) = *(src + nPackets *
ITEMS_PER_PACKET + i)*c + *(src2 + nPackets * ITEMS_PER_PACKET + i)*(1-c);
    }
}
```

To use the diffImages program first we have to give it the permissions to be executed with:

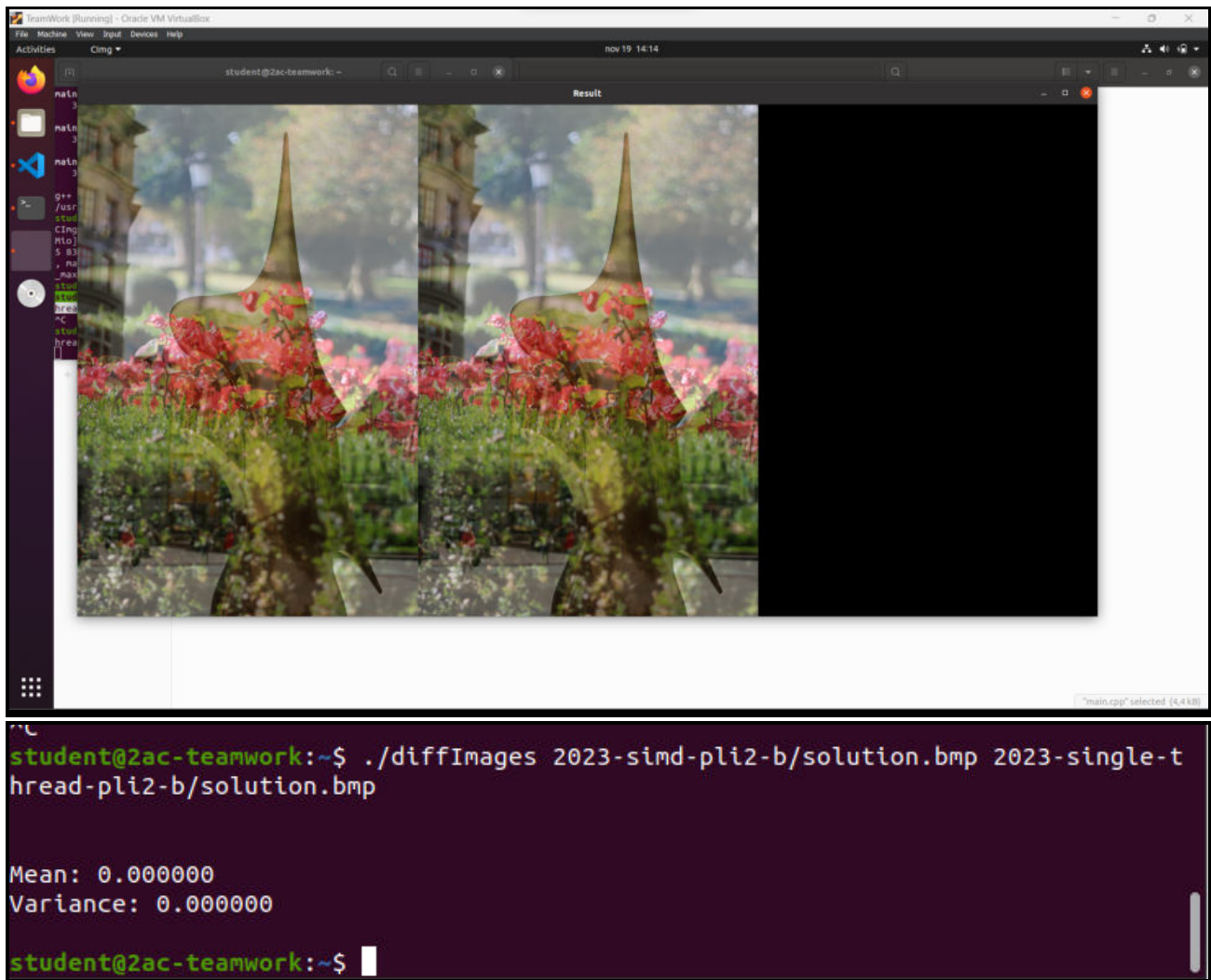
```
> chmod +x diffImages
```

With the following command we can check its permissions:

```
> ll diff*  
-rwxrwxr-x 1 student student 1394968 nov 19 13:43 diffImages
```

Then we use the following command to compare the solution with the single thread version and the SIMD version:

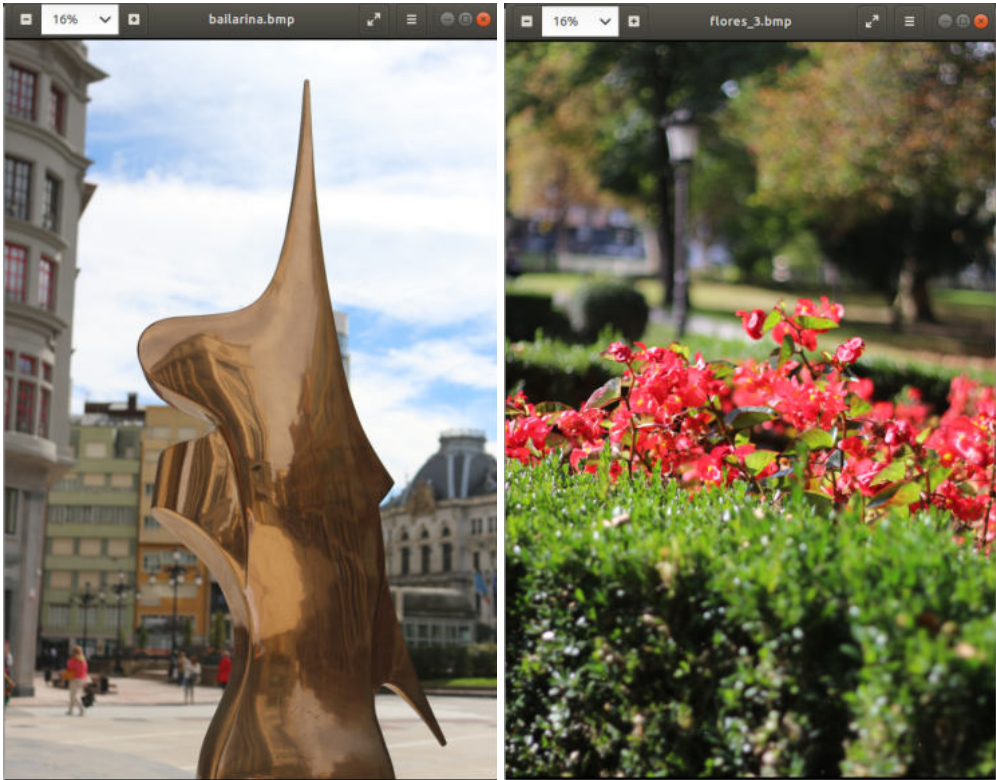
```
> ./diffImages 2023-simd-pli2-b/solution.bmp  
2023-single-thread-pli2-b/solution.bmp
```

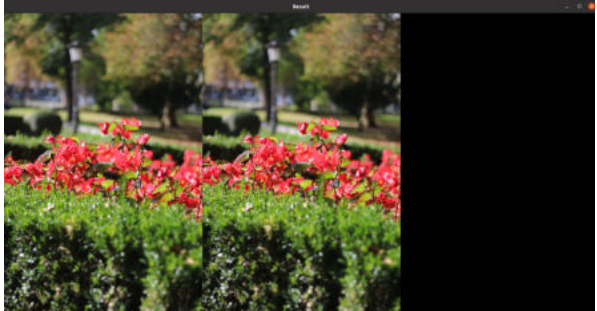
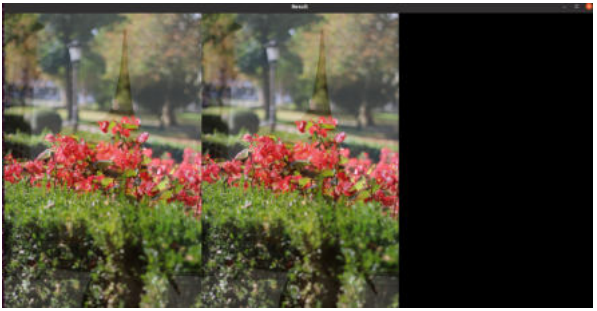



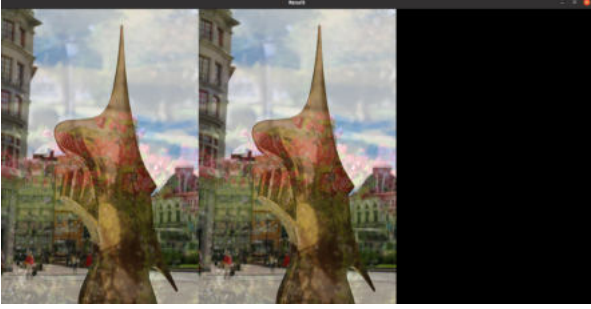

We can see the images are equal as the result is black and the mean and variance is 0.

Results

Source Images



Solution		
Blend level	0	0.25
Solution		

Blend level	0.5	
Solution		
Blend level	0.75	1

Measurements

Response time	2.959145	2.968701	2.827799	2.849870	3.007849
	2.910932	2.961203	2.937147	2.789309	2.871863

The mean:

$$\bar{x} = \frac{\sum Time}{10} = \frac{2.959145+2.968701+2.827799+2.849870+3.007849+2.910932+2.961203+2.937147+2.789309+2.871863}{10} = \frac{29.083818}{10} = 2.9083818$$

The standard deviation:

$$s = \sqrt{\frac{\sum_{i=1}^{10} (x_i - \bar{x})^2}{10-1}} = \sqrt{\frac{0.002577+0.003638+0.006494+0.003424+0.009894+0.000007+0.002790+0.000827+0.014178+0.001334}{9}} = \sqrt{\frac{0.045162}{9}} = 0.070838019$$

The confidence interval (95%):

$$[\bar{x} - 1.96 \cdot \frac{s}{\sqrt{10}}, \bar{x} + 1.96 \cdot \frac{s}{\sqrt{10}}] = [2.9083818 - 1.96 \cdot \frac{0.070838019}{\sqrt{10}}, 2.9083818 + 1.96 \cdot \frac{0.070838019}{\sqrt{10}}] = [2.864475941, 2.952287659]$$

The speedup:

$$S = \frac{\bar{x}_{single\ thread}}{\bar{x}_{sim}} = \frac{6.625491}{2.9083818} = 2.278068$$

Multithread

Link: [multithread](#)

Algorithm

We must create a number of threads equal to the property siblings obtained in the cpu info (4 siblings). We store that value in NUM_THREADS. Also, we will store the threads in the thread array.

We create the thread by assigning to it their task (ThreadJob method) and passing to this method a number to identify each thread (the value of the variable i of the loop)

```
for (int i = 0; i < NUM_THREADS; i++ ) {
    int *numberThread = new int(i);
    if(pthread_create(&thread[i], NULL, ThreadJob, (void*)numberThread) != 0)
    {
        fprintf(stderr, "ERROR: Creating thread %d\n", i);
        sleep(1);
    }
}
```

After finishing all the creation, we need to join the threads so they wait to end at the same time.

```
for ( int i = 0; i < NUM_THREADS; i++ ) {
    pthread_join(thread[i], NULL );
}
```

Now, we will see the ThreadJob function. This function receives the number of the thread as a parameter so it can pass it to the filter function.

```
void * ThreadJob(void * i) {
    int numberThread = *((int *) i);
    filter(filter_args, blendLevel, numberThread);

    return NULL;
}
```

The filter function will work as in the single thread but each thread will process the number of pixels between [minPixel, maxPixel) depending on the numberThread.

- The minPixel is the numberThread multiplied by the numberOfPackets
- The maxPixel will be numberOfPackets multiplied by the numberThread, and you add the numberOfPackets

The numberOfPackets is calculated as the total number of pixels divided by the total number of threads. So, we must be careful when the number of pixels is not multiple of the total number of threads; because if it's not then a number of pixels won't be processed.

So we determine the maxPixel as the total number of pixels if it's the last thread, so the last thread will also process the data in excess.

```

void filter (filter_args_t args, float c, int numberThread) {
    //Checks that c is correct
    if (c<0 || c>1){
        perror("The c parameter is invalid.");
        exit(-2);
    }

    //Calculates the maximum position that the thread will process
    uint maxPixel = (uint)numberOfPackets*numberThread +
(uint)numberOfPackets;
    if (numberThread== NUM_THREADS-1){
        maxPixel= filter_args.pixelCount;
    }

    //Applies the filter for the minimum position and maximum position
that the thread will process
    for (uint i = numberThread*(int)numberOfPackets; i < maxPixel; i++) {
        addColors((args.pRsrc + i), (args.pRsrc2 + i), (args.pRdst + i), c);
        addColors((args.pGsrc + i), (args.pGsrc2 + i), (args.pGdst + i), c);
        addColors((args.pBsrc + i), (args.pBsrc2 + i), (args.pBdst + i), c);
    }
}

```

To determine calculate the time it takes we need to sum:

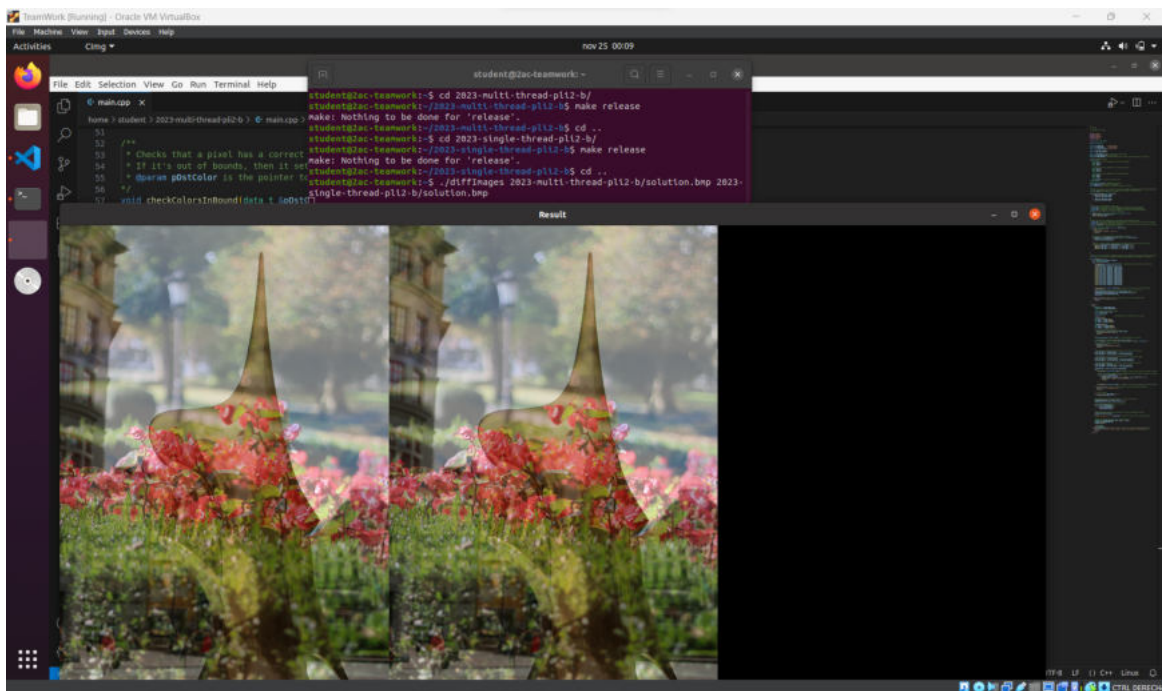
- The time it takes to create the threads
- The largest time it took to a thread to do the task. So we store after doing the task, the time it took in the array times. Then, we calculate the maximum in that array.

We use the following command to compare the solution with the single thread version and the SIMD version:

```

> ./diffImages 2023-multi-thread-pli2-b/solution.bmp
2023-single-thread-pli2-b/solution.bmp

```

```
student@zac-teamwork:~/2023-single-thread-pli2-b$ cd ..
student@zac-teamwork:~$ ./diffImages 2023-multi-thread-pli2-b/solution.bmp 2023-
single-thread-pli2-b/solution.bmp

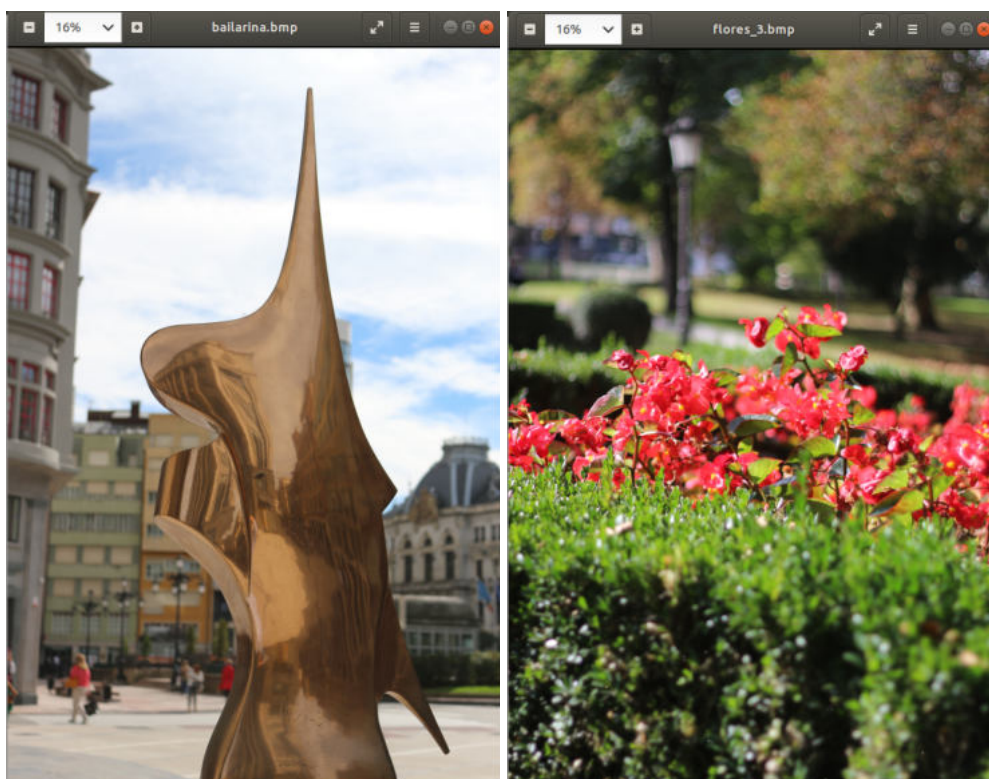
Mean: 0.000000
Variance: 0.000000

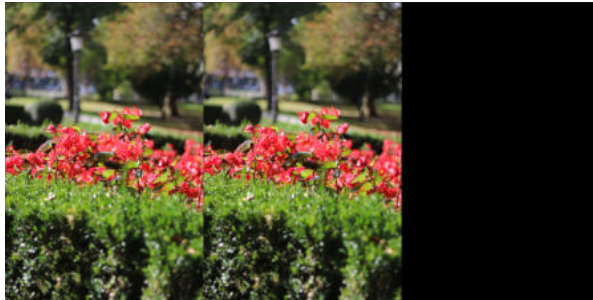
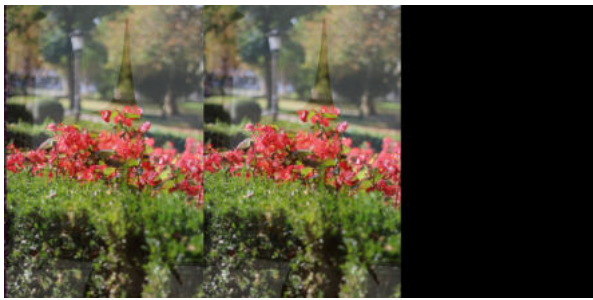
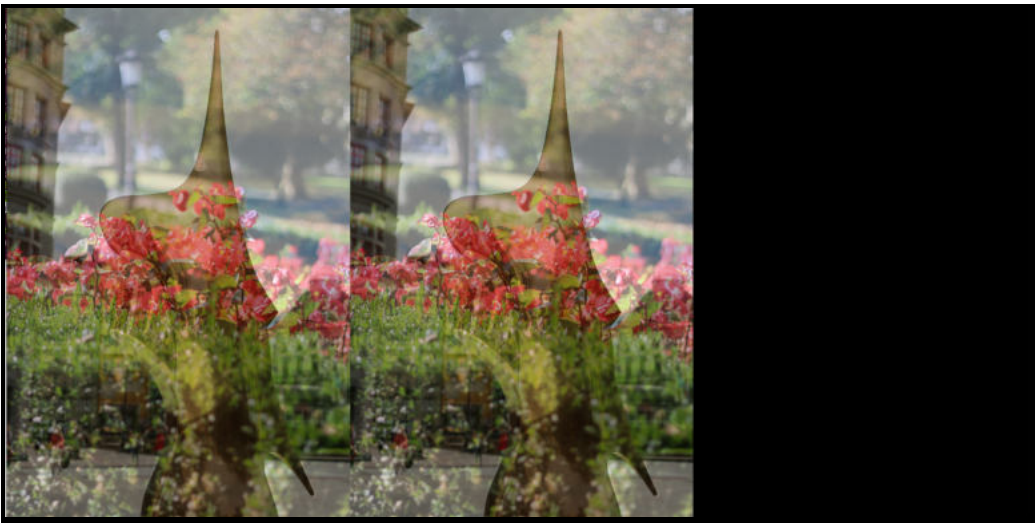


student@zac-teamwork:~$
```

We can see the images are equal as the result is black and the mean and variance is 0.

Results

Source Images



Solution				
Blend level	0		0.25	
Solution				
Blend level	0.5			
Solution				
Blend level	0.75		1	

Measurements

Response time	3.301774	3.007537	2.853043	2.891380	2.815010
	2.878906	2.549789	2.562748	2.924252	3.071305

The mean:

$$\bar{x} = \frac{\Sigma Time}{10} = \frac{3.301774+3.007537+2.853043+2.891380 +2.815010+2.878906+2.549789+2.562748+2.924252+3.071305}{10} =$$

$$= \frac{28.855744}{10} = 2.885574$$

The standard deviation:

$$s = \sqrt{\frac{\sum_{i=1}^{10} (x_i - \bar{x})^2}{10-1}} = \sqrt{\frac{0.173222+0.014874+0.001058+0.0000337+0.004979+0.00004447+0.112752+0.10422+0.00149596+0.034496}{9}} =$$

$$= \sqrt{\frac{0.44717513}{9}} = 0.2229038482$$

$$[\bar{x} - 1.96 \cdot \frac{s}{\sqrt{10}}, \bar{x} + 1.96 \cdot \frac{s}{\sqrt{10}}] = [2.885574 - 1.96 \cdot \frac{0.2229038482}{\sqrt{10}}, 2.885574 + 1.96 \cdot \frac{0.2229038482}{\sqrt{10}}] =$$

$$= [2.747417, 3.023731]$$

The speedup:

$$S = \frac{\bar{x}_{single\ thread}}{\bar{x}_{multi\ thread}} = \frac{6.625491}{2.885574} = 2.296074$$

Results

As we can see the SIMD version and multi threaded version are faster than the single thread version. We obtained a speedup of 2.278068 with the SIMD version and 2.296074 with the multi thread version.

We can see that they are more or less the same, but the multi threaded version would be better when the number of threads created and siblings is higher than the number of items in the SIMD version. As with the SIMD we can process 8 items at the same time, and with the multi-thread we could have more threads processing more rows of the image at the same time. Obviously, we cannot increase the number of threads indefinitely because the performance could get even worse.

Work division

Member	Work	%
Eduardo Pérez Sánchez	<ul style="list-style-type: none">• phase 1 (specially time measurements and images)• phase 2 SIMD: initialization of variables and algorithm• phase 2 multi: threadJob and error handling	25
Jorge Carriles Ruiz	<ul style="list-style-type: none">• phase 1 (specially time measurements and algorithm)• phase 2 SIMD: dataInExcess management and error handling• phase 2 multi: algorithm and creation of threads	25
Ana Pérez Bango	<ul style="list-style-type: none">• phase 1 (specially algorithm and error handling), settings and cpu info• phase 2 SIMD: dataInExcess, time measurement and report calculations• phase 2 multi: error handling, time measurement and report calculations	25
Paula Díaz Álvarez	<ul style="list-style-type: none">• phase 1 (specially algorithm and error handling) and report calculations• phase 2 SIMD: algorithm and time measurement• phase 2 multi: algorithm and threadJob	25