

PL4, Grupo A

# MEMORIA TRABAJO

FASE 1 - 2

Christian Peláez Fernández  
Sara García Rodríguez  
Lino Menéndez de Luarda Trabanco  
Luis Carlos Hurlé Fleitas  
Javier Martínez Álvarez  
12-12-2017

## Contenido

FASE 1.....	2
1. OBJETIVOS .....	2
2. OPERACIONES.....	2
3. ATRIBUTOS Y CONSTANTES .....	2
4. FUNCIONES COMUNES .....	3
1. <b>createVector()</b> .....	3
2. <b>removeVector()</b> .....	4
3. <b>main()</b> .....	4
5. PRIMERA PARTE ----- PROYECTO SINGLETHREAD .....	5
1. Operación Dif2.....	5
2. Operación Count Positives.....	5
3. Operación Sub .....	6
6. SEGUNDA PARTE ----- PROYECTO SINGLETHREAD-SIMD .....	6
1. SIMD.....	6
2. Operación Dif2.....	9
3. Operación Count Positives.....	11
4. Operación Sub .....	13
7. RESULTADOS OBTENIDOS.....	15
8. DISTRIBUCIÓN DEL TRABAJO .....	15
FASE 2.....	16
1. OBJETIVOS .....	16
2. ATRIBUTOS Y CONSTANTES .....	16
3. FUNCIONES COMUNES .....	16
1. <b>createVector()</b> .....	16
2. <b>removeVector()</b> .....	16
3. <b>getNumbersProcessor()</b> .....	16
4. <b>generateSizes()</b> .....	17
5. <b>wait()</b> .....	17
6. <b>Dif2()</b> .....	17
7. <b>Sub()</b> .....	18
8. <b>CountPositiveValues ()</b> .....	18
9. <b>main()</b> .....	19
9. PRIMERA PARTE Y SEGUNDA .....	20
10. RESULTADOS OBTENIDOS.....	20
11. DISTRIBUCIÓN DEL TRABAJO .....	20

# FASE 1

## 1. OBJETIVOS

- Desarrollar un programa que implemente operaciones dadas por los profesores.
- Realizar mediciones de tiempos sobre ese mismo programa y reflejarlas en una tabla.

## 2. OPERACIONES

A nuestro grupo se le asignaron 3 operaciones:

- Dif2  $R_i = (U_{i+1} - U_i)/2$
- Count Positives  $K = \text{number of positives in } \vec{W}$
- Sub  $S_i = V_i - U_i$

## 3. ATRIBUTOS Y CONSTANTES

Se han declarado los siguientes atributos y constantes, las cuales son comunes para ambas partes:

- **TIMES:** Constante de valor 10, representa el número de veces que se ejecutará un número de repeticiones del programa.
- **NTIMES:** Constante de valor 200, representa el número de repeticiones del programa.
- **SIZE:** Constante de valor 1024\*1024, representa el tamaño de algunos vectores.
- **PRINT\_FUNCTIONS:** Constante que activa o desactiva la muestra por pantalla de los valores devueltos por las operaciones Dif2, Count Positives y Sub.
- **PRINT\_TIMER\_FUNCTION:** Constante que activa o desactiva la muestra por pantalla de los tiempos de cada operación.
- **PRINT\_TIMER:** Constante que activa o desactiva la muestra por pantalla del tiempo total de las operaciones después de haberlas repetido **NTIMES**.
- **frecuencia:** atributo de tipo **LARGUE\_INTEGER** dado por el esqueleto del trabajo.
- **tStart:** atributo de tipo **LARGUE\_INTEGER** dado por el esqueleto del trabajo.
- **tEnd:** atributo de tipo **LARGUE\_INTEGER** dado por el esqueleto del trabajo.
- **dElapsedTime:** atributo de tipo **double** dado por el esqueleto del trabajo.
- **u:** vector de **floats** utilizado por la operación Dif2.
- **t:** vector de **floats** utilizado por la operación Sub.
- **w:** vector de **floats** utilizado por la operación Count Positives.
- **r:** vector de **floats**, es el resultado de Dif2.
- **k:** atributo de tipo entero el cual es el resultado de Count Positives.

- S: vector de `floats`, es el resultado de Sub.

## 4. FUNCIONES COMUNES

Para la realización de este trabajo hemos implementado unas funciones que ayudarán y simplificarán la tarea a cumplir. Estas funciones son comunes a ambas partes, aunque alguna cambie ligeramente dependiendo de la parte. Las funciones son:

### 1. `createVector()`

#### a. Primera parte

```
float* createVector() {
    float* vector = (float *)malloc(sizeof(float) * SIZE);

    for (int i = 0; i < SIZE; i++) {
        float r = -1 + 2 * float((double)rand() / (double)(RAND_MAX));
        vector[i] = r;
    }

    return vector;
}
```

La finalidad de esta función es crear un vector de tamaño `SIZE`, inicializarlo con valores flotantes pertenecientes al intervalo `[-1,1]` y posteriormente devolverlo.

Para ello hacemos uso de la función `malloc()`, la cual reserva un bloque de memoria dado un tamaño(en bytes).Puesto que nosotros queremos crear un vector de tamaño `SIZE` que albergue valores tipo `float`, necesitaremos un bloque de memoria de tamaño `SIZE * tamañoFloat`, el tamaño en bytes de `float` lo obtenemos de la función `sizeof(float)`. La función `malloc()` nos devuelve un puntero `*void` que mediante un casting lo transformamos en `*float`. De esta manera tendríamos creado el vector. Para inicializarlo lo recorremos mediante un bucle `for`, y haciendo uso de la función `rand()` le asignamos valores aleatorios de `[-1,1]`, una vez inicializado el vector se devuelve.

#### b. Segunda parte

```
float* createVector() {
    float* vector = (float *)_aligned_malloc(SIZE * sizeof(float), sizeof(__m256i));

    for (int i = 0; i < SIZE; i++) {
        float r = -1 + 2 * float((double)rand() / (double)(RAND_MAX));
        vector[i] = r;
    }

    return vector;
}
```

En la segunda parte, al trabajar con instrucciones SIMD debemos crear un vector con espacio en memoria alineado según el dato

m256 . Para ello a la hora de crear el vector se utiliza la función `_aligned_malloc()`, a la cual pasamos por parámetro el tamaño del bloque a reservar y el valor de la alineación (el tamaño del tipo de dato a guardar, en nuestro caso el tipo es de 256 bits). Una vez creado el vector, la inicialización es igual que en la primera parte.

## 2. `removeVector()`

### a. Primera parte

```
void removeVector(float* vector) {  
    free(vector);  
}
```

El objetivo de esta función es liberar el espacio utilizado por el vector pasado por parámetro, para ello hacemos uso de la función `free()`.

### b. Segunda parte

```
void removeVector(float* vector) {  
    _aligned_free(vector);  
}
```

A diferencia de la primera parte, debemos liberar un bloque de memoria alineado, debido a esto utilizamos la función `_aligned_free()`.

## 3. `main()`

```
117 int main() {  
118     time_t ti;  
119     srand((unsigned)time(&ti)); // Información sobre srand https://www.tutorialspoint.com/c\_standard\_library/c\_function\_srand.htm  
120  
121     double times[TIMES];  
122  
123     for (int j = 0; j < TIMES; j++) {  
124         times[j] = 0; //aseguramos valores reales  
125         for (int i = 0; i < NTIMES; i++) {  
126             u = createVector();  
127             w = createVector();  
128             t = createVector();  
129  
130             times[j] += timer(Dif2) + timer(countPositiveValues) + timer(Sub);  
131  
132             removeVector(u);  
133             removeVector(w);  
134             removeVector(t);  
135             removeVector(s);  
136             removeVector(r);  
137         }  
138  
139         if (PRINT_TIMER)  
140             printf("Elapsed total time in seconds: %f\n", times[j]);  
141     }  
142     free(times);  
143 }
```

Mediante la función `srand()` nos aseguramos de que cada vez que ejecutemos un `rand()` nos salgan distintos valores aleatorios. Creamos un array, `times`, de tamaño `TIMES` en el que guardaremos los tiempos de ejecución de las operaciones repetidas `NTIMES`.

En el bucle `for` que se ejecutará `NTIMES` creamos en cada iteración los vectores `U`, `W` y `T` mediante la función `createVectores()` y se obtienen los tiempos de las operaciones, que se suman y se guardan en el vector `times`, al final de dicho bucle se libera la memoria de los vectores `U`, `W`, `T`, `S` y `R`. Si la constante `PRINT_TIMES` estuviera activada se mostrarían los tiempos por pantalla. Por último, liberamos la memoria de los arrays usados (incluido el de `times` después de la toda la ejecución del programa).

## 5. PRIMERA PARTE ----- PROYECTO SINGLETHREAD

### 1. Operación Dif2

```
61 void Dif2() {
62     r = (float *)malloc(sizeof(float) * (SIZE - 1));
63
64     for (int i = 0; i < SIZE - 1; i++) {
65         r[i] = (u[i + 1] - u[i]) / 2.0;
66
67         if (PRINT_FUNCTIONS)
68             printf("La diferencia entre dos valores es %f\n", r[i]);
69     }
70 }
71 }
```

La operación `Dif2` calcula la media de la diferencia entre los valores con posiciones consecutivas del vector `U`. Para ello creamos un vector `R` que tiene un tamaño `SIZE-1`. Esto es debido a que al restar valores con posiciones consecutivas hacemos `SIZE-1` restas que se guardaran en `R`. Después de haberlo creado con la función `malloc()`, mediante un bucle, guardamos en cada posición el resultado de la operación. Finalmente dependiendo del valor de `PRINT_FUNCTIONS` se imprimirá o no por pantalla el resultado de cada operación.

### 2. Operación Count Positives

```
73 void countPositiveValues() {
74     k = 0;
75
76     for (int i = 0; i < SIZE; i++) {
77         if (w[i] >= 0.0)
78             k++;
79     }
80
81     if (PRINT_FUNCTIONS)
82         printf("El contador de numeros positivos es %d\n", k);
83 }
84 }
```

Esta operación cuenta el número de valores positivos del vector W, para ello inicializamos a cero el contador k, hacemos un bucle y si un valor del vector W es mayor o igual que 0 aumentamos el contador en una unidad. Al igual que en Dif2, dependiendo del valor de PRINT\_FUNCTIONS se imprimirá por pantalla o no, el número de positivos del vector W.

### 3. Operación Sub

```

85 void Sub() {
86     //inicializacion del vector V
87     float* v = (float *)malloc(sizeof(float) * (SIZE - 1));
88     for (int i = 0; i < SIZE - 1; i++) {
89         v[i] = k * r[i];
90     }
91
92     //codigo del programa
93     s = (float *)malloc(sizeof(float) * (SIZE - 1));
94     for (int i = 0; i < SIZE - 1; i++) {
95         s[i] = v[i] - u[i];
96         if (PRINT_FUNCTIONS)
97             printf("La resta es %f\n", s[i]);
98     }
99     //eliminar de memoria el vector V
100    removeVector(v);
101 }

```

Esta operación resta los valores de cada posición de los vectores V y U. Para realizar esta operación creamos el vector V reservando un espacio en memoria con la función malloc(), lo recorremos con un bucle y en cada posición guardamos el valor que resulta del producto de k por el valor de cada posición del vector R, inicializando de esta manera el vector V.

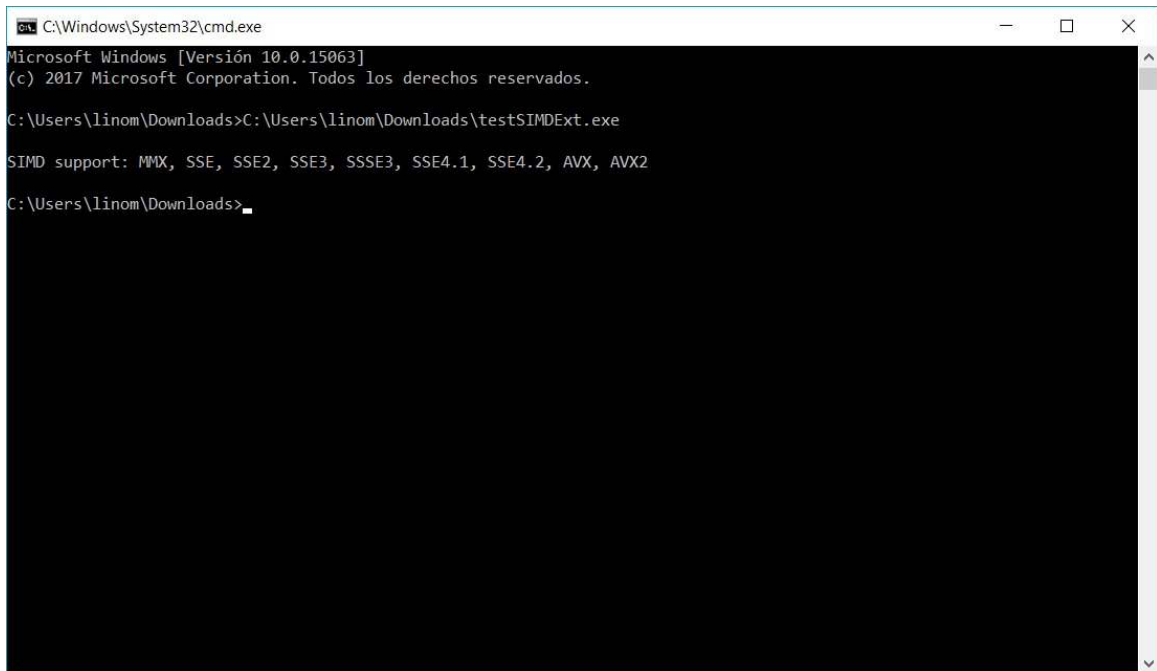
Asignamos espacio en memoria para el vector S, hacemos otro bucle donde vamos asignando a cada posición de S la resta de cada posición entre el vector V y el vector U. Finalmente eliminamos de memoria el vector V.

## 6. SEGUNDA PARTE ----- PROYECTO SINGLETHREAD-SIMD

### 1. SIMD

En el enunciado del trabajo se nos indicaba que el nivel máximo de extensiones SIMD era **AVX-512F**, ejecutando el programa dado por los profesores para mostrar el nivel máximo de extensiones SIMD que nuestros ordenadores soportaban obtuvimos que todos nuestros ordenadores soportaban como máximo hasta **AVX2**, así que hemos trabajado con esa extensión SIMD.

- Captura de Lino:



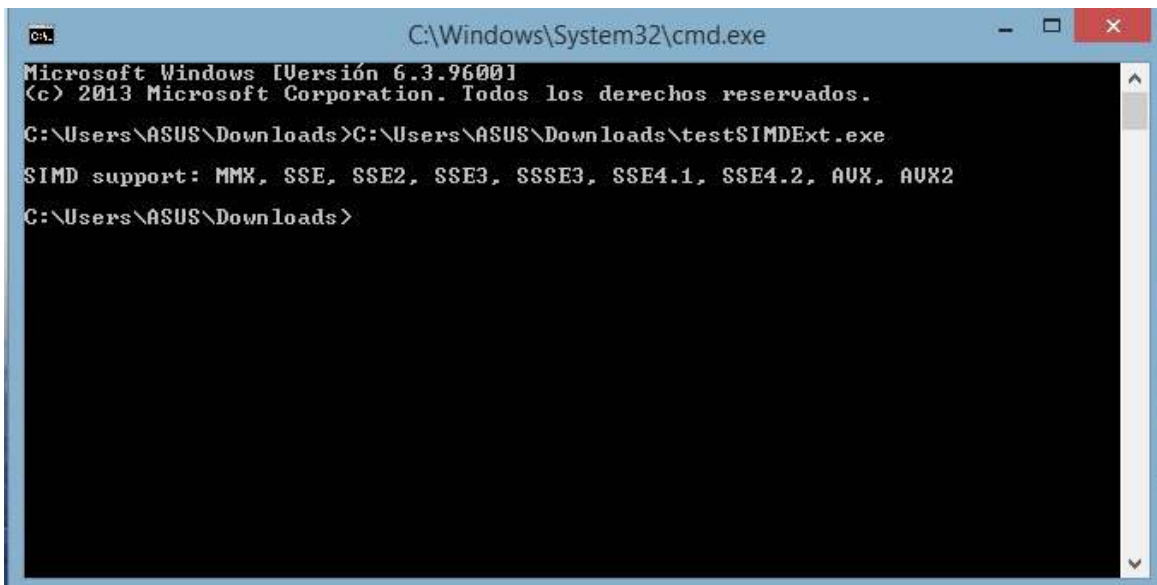
```
C:\Windows\System32\cmd.exe
Microsoft Windows [Versión 10.0.15063]
(c) 2017 Microsoft Corporation. Todos los derechos reservados.

C:\Users\linom\Downloads>C:\Users\linom\Downloads\testSIMDExt.exe

SIMD support: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2

C:\Users\linom\Downloads>
```

- Captura de Sara:



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Versión 6.3.9600]
(c) 2013 Microsoft Corporation. Todos los derechos reservados.

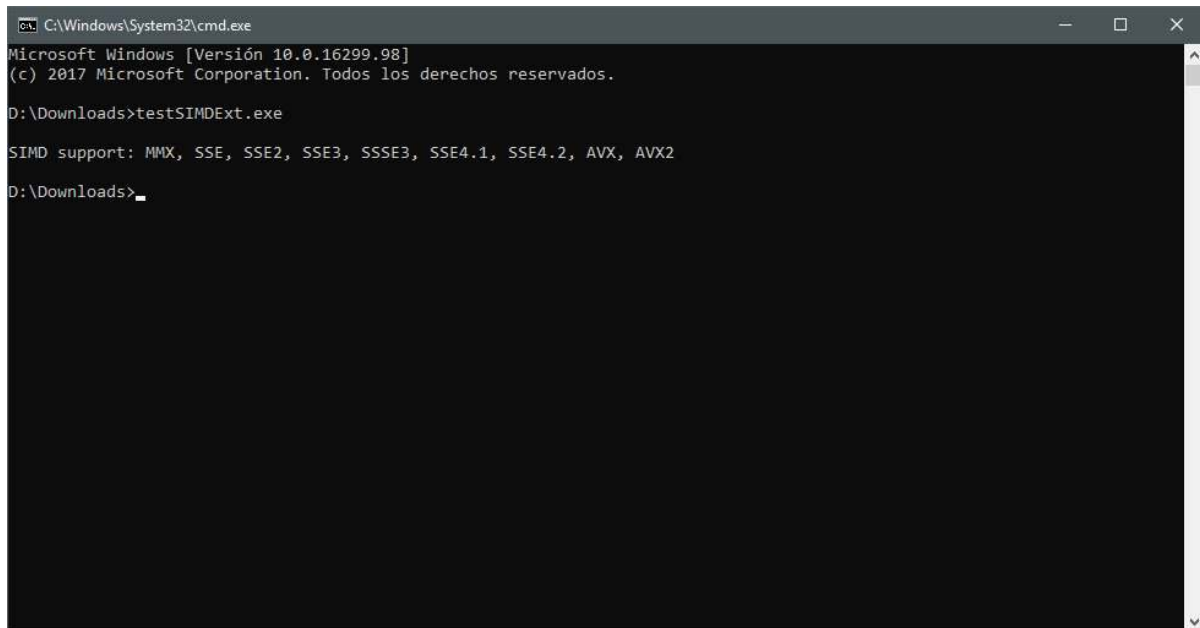
C:\Users\ASUS\Downloads>C:\Users\ASUS\Downloads\testSIMDExt.exe

SIMD support: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2

C:\Users\ASUS\Downloads>
```



- Captura de Javier:



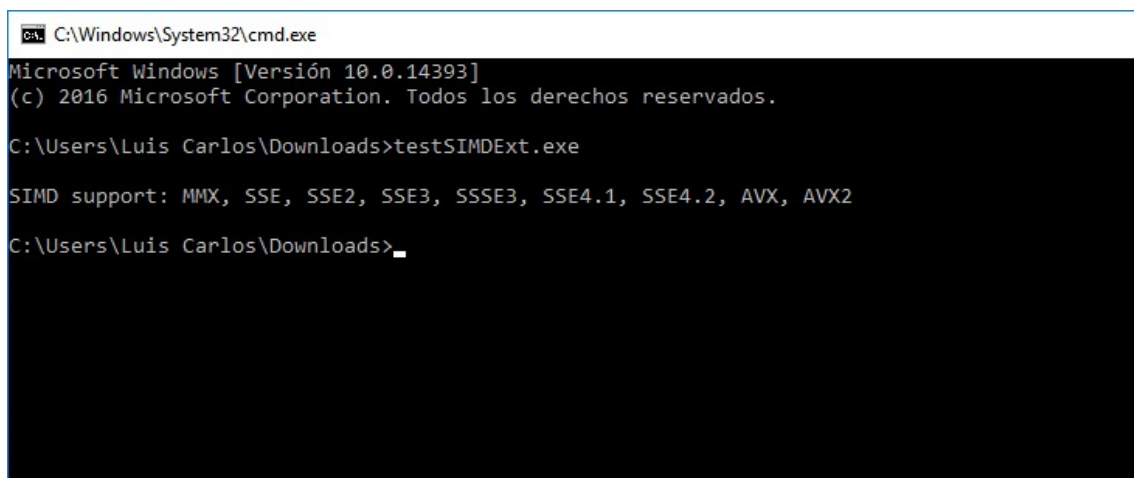
```
C:\Windows\System32\cmd.exe
Microsoft Windows [Versión 10.0.16299.98]
(c) 2017 Microsoft Corporation. Todos los derechos reservados.

D:\Downloads>testSIMDExt.exe

SIMD support: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2

D:\Downloads>_
```

- Captura de Luis:



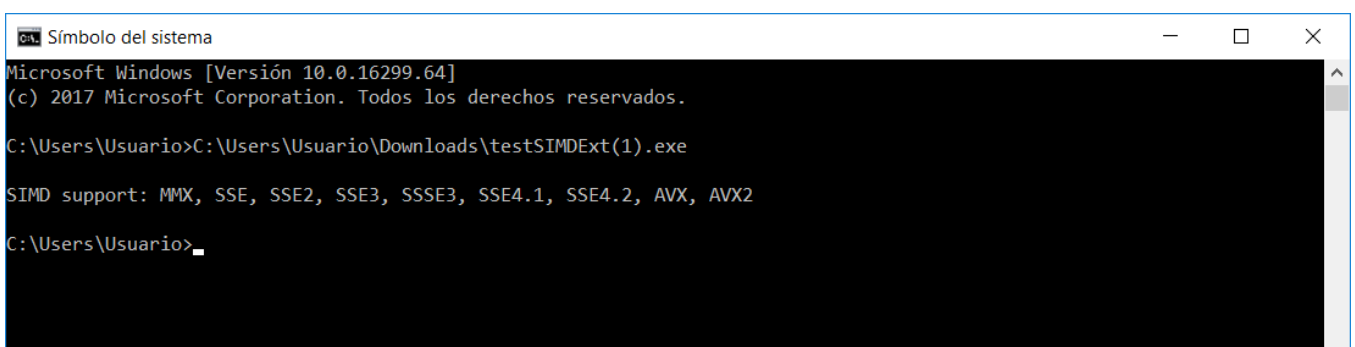
```
C:\Windows\System32\cmd.exe
Microsoft Windows [Versión 10.0.14393]
(c) 2016 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Luis Carlos\Downloads>testSIMDExt.exe

SIMD support: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2

C:\Users\Luis Carlos\Downloads>_
```

- Captura Christian:



```
Símbolo del sistema
Microsoft Windows [Versión 10.0.16299.64]
(c) 2017 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Usuario>C:\Users\Usuario\Downloads\testSIMDExt(1).exe

SIMD support: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, AVX2

C:\Users\Usuario>_
```

## 2. Operación Dif2

```
55 void Dif2() {
56     r = (float *)_aligned_malloc((SIZE-1) * sizeof(float), sizeof(__m256i));
57     //Inicializamos una variable con el valor 2
58     __m256 number2 = _mm256_set_ps(2, 2, 2, 2, 2, 2, 2, 2);
59
60     for (int i = 0; i < (SIZE - 1) / NUMBER_FLOAT; i++) {
61         __m256 valuei = (__m256 *)&u[(i+1) * NUMBER_FLOAT];
62         __m256 valuei_minus_1 = (__m256 *)&u[i * NUMBER_FLOAT];
63         __m256 value = _mm256_sub_ps(valuei, valuei_minus_1);
64         value = _mm256_div_ps(value, number2);
65
66         float *p = (float*)&value;
67         for (int j = 0; j < NUMBER_FLOAT; j++) {
68             r[i*NUMBER_FLOAT + j] = *p;
69             p++;
70             if (PRINT_FUNCTIONS)
71                 printf("La diferencia entre dos valores es %f\n", r[i+j]);
72         }
73     }
74 }
75
```

Mediante la función `_aligned_malloc()` creamos un vector `R` de `floats` de tamaño `SIZE-1`. A continuación inicializamos a 2 las 8 partes de 32 bits de la variable `number2` de tipo `m256`, ya que posteriormente tendremos que dividir entre 2 cada elemento del vector de tipo `m256` que contenga los resultados de la resta entre elementos del vector `U`.

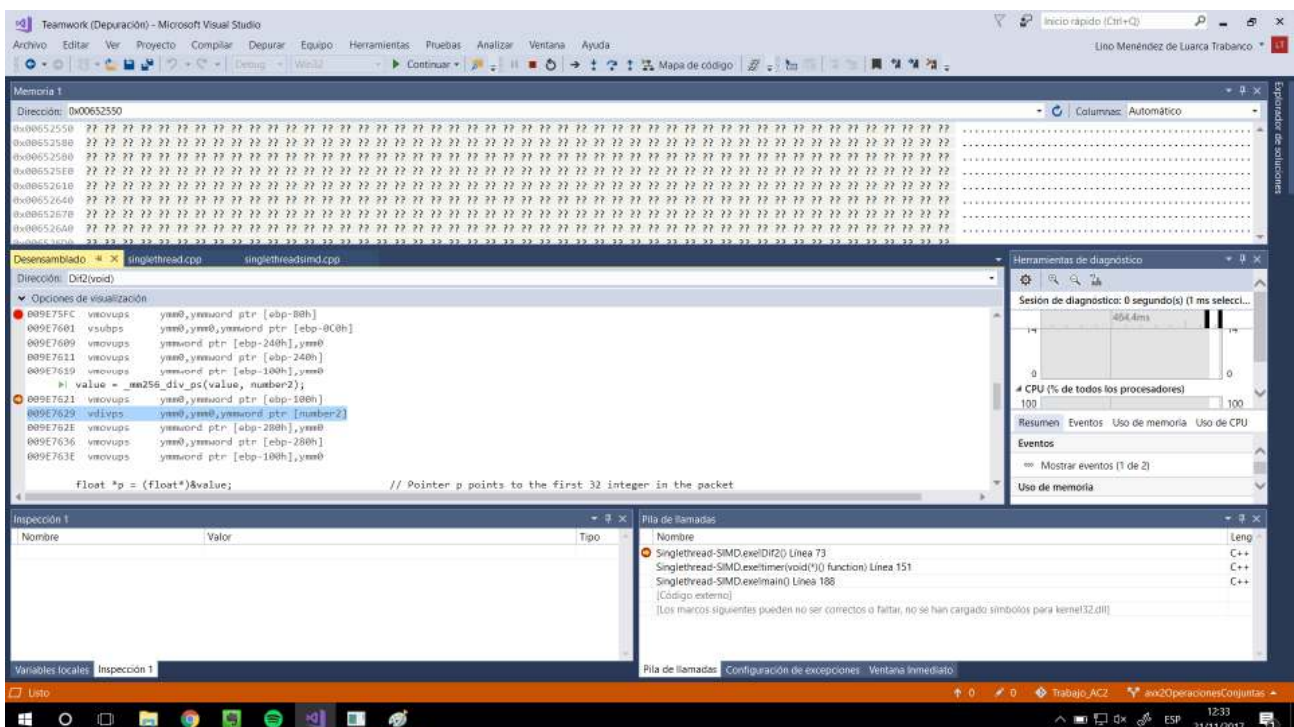
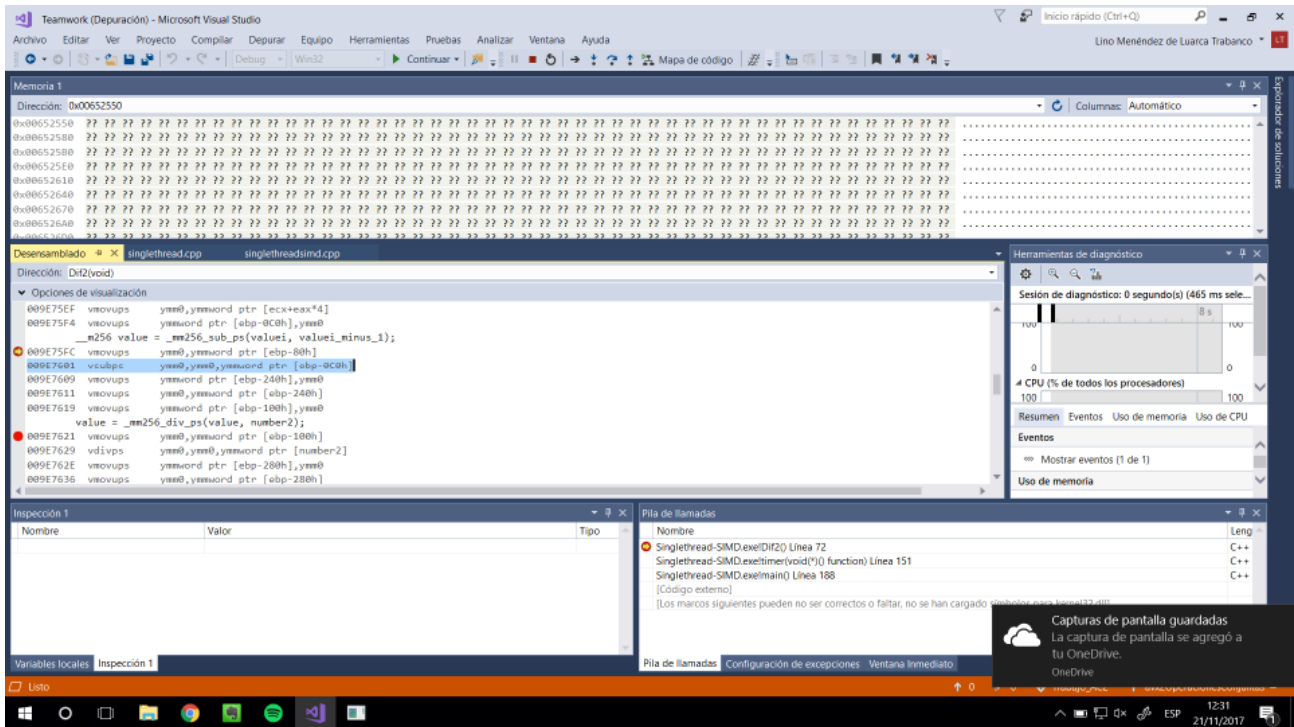
A través un bucle `for` vamos asignando en cada interacción a la variable `valuei` y `valuei_minus_1` partes del vector de tipo `float`, cogiendo de ocho en ocho los floats (o en su defecto 256 bits). La variable `valuei` se le asigna un vector de una posición más adelantada que `valuei_minus_1`. Mediante la función `_mm256_sub_ps()` restamos `valuei - value_minus_1` y guardamos el resultado en la variable de tipo `m256 value`. Esta variable se divide entre `number2` usando la función `_mm256_div_ps()`.

Al final del bucle hacemos un casting a la variable `value`, transformándola en un vector de `floats`. Para ello creamos un puntero `float(p)` el cual apuntará a la dirección de inicio de la variable `value` (cuyo valor es un tipo `m256`, el cual alberga

los bits de 8 floats). Mediante un bucle **for** anidado, recorreremos el vector R y le asignamos los valores a los que apunte el puntero **p(floats)**.

Dependiendo del valor de **PRINT\_FUNCTIONS** se imprimirá por pantalla, o no, el valor de la operación **Dif2** de cada elemento.

Finalmente se ejecuta poniendo un punto de interrupción en la instrucción **\_mm256\_sub\_ps** y **\_mm256\_div\_ps** para obtener su equivalencia en desensamblado tal como indica el enunciado del trabajo.



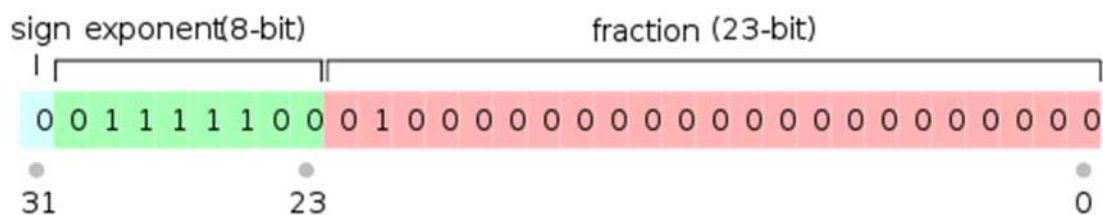
### 3. Operación Count Positives

```

76 void countPositiveValues() {
77     k = 0;
78
79     __m256 mask = _mm256_set_ps(0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000, 0x80000000);
80     //Calculate count
81     for (int j = 0; j < SIZE / NUMBER_FLOAT; j++) {
82         __m256 value = *(__m256*)&w[j * NUMBER_FLOAT];
83         __m256 and = _mm256_and_ps(value, mask);           // mascara para mirar el bit mas significativo
84
85         float *p = (float*)&and;
86         for (int i = 0; i < NUMBER_FLOAT; i++) {
87             if (*(p+i) != 0x80000000) {
88                 k++;
89             }
90         }
91     }
92     if (PRINT_FUNCTIONS)
93         printf("El contador de numeros positivos es %d\n", k);
94
95 }
96

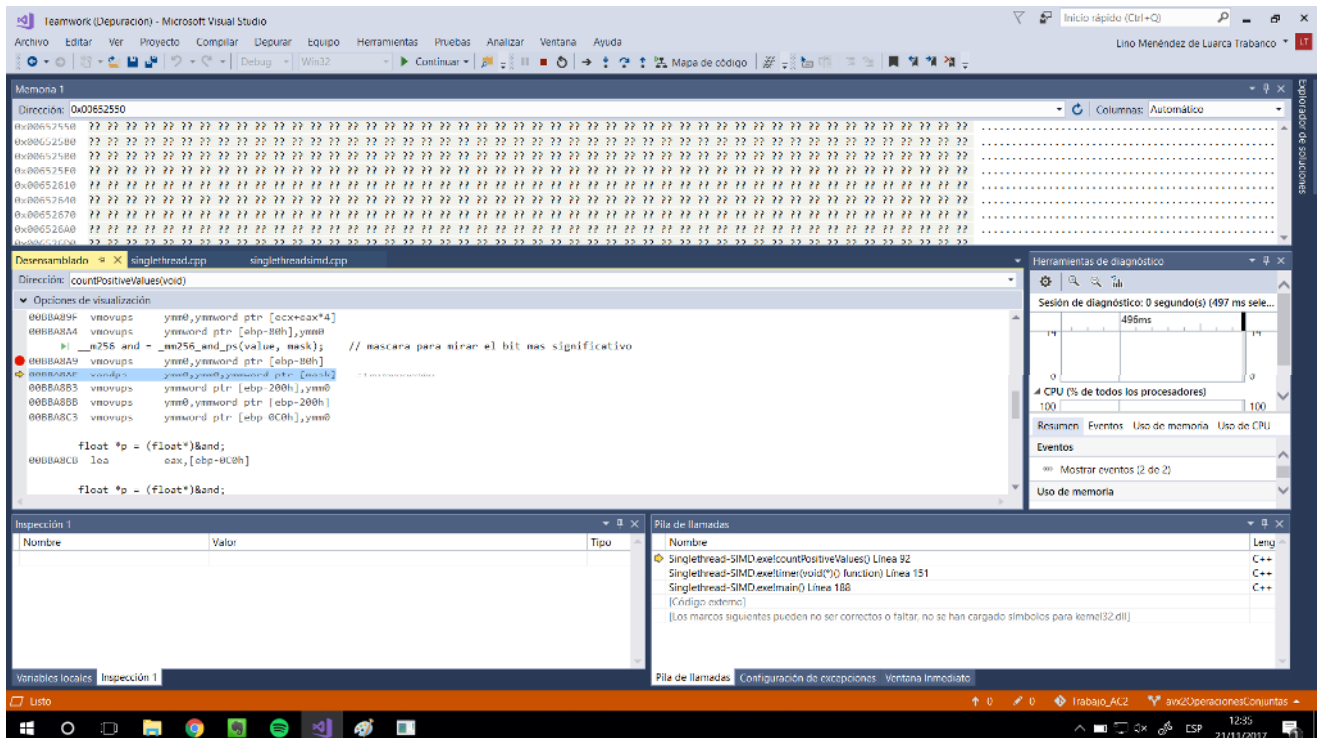
```

Para comprobar el número de positivos en el vector W creamos una máscara de tipo `m256` cuyo valor en cada elemento es `0x80000000` (el bit más significativo está a 1). Creamos un bucle for y en cada iteración de este vamos asignando elementos `m256` del vector W a la variable `value`, realizando una operación **AND** bit a bit entre la máscara y la variable `value` y comprobamos si el resultado de la **and** en cada uno de los 8 elementos floats es igual a `0x80000000`, si no fuese así aumentamos el contador de positivos en una unidad puesto que según la definición de la **IEEE-754**, el bit más significativo de un valor tipo `float` es el bit de signo, si este está a 1 significa que es negativo y si está a cero es positivo, como se puede ver a continuación.



Al igual que en las anteriores operaciones dependiendo del valor de `PRINT_FUNCTIONS` se imprimirá, o no, por pantalla el número de positivos en el vector W.

Finalmente se ejecuta poniendo un punto de interrupción en la instrucción `_mm256_and_ps` para obtener su equivalencia en desensamblado tal como indica el enunciado del trabajo.



## 4. Operación Sub

```
99 void Sub() {
100     //inicializacion del vector V
101     unsigned int index = 0;
102     float* v = (float *)_aligned_malloc((SIZE - 1) * sizeof(float), sizeof(__m256i));
103     __m256 kIntrinsics = _mm256_set_ps((float)k, (float)k, (float)k, (float)k, (float)k, (float)k, (float)k, (float)k);
104     for (int i = 0; i < (SIZE - 1) / NUMBER_FLOAT; i++) {
105         __m256 value = (__m256*)&r[i * NUMBER_FLOAT];
106         __m256 mult = _mm256_mul_ps(kIntrinsics, value);
107
108         float* p = (float*)&mult;
109         for (int j = 0; j < NUMBER_FLOAT; j++) {
110             v[i * NUMBER_FLOAT + j] = *(p+j);
111         }
112     }
113
114     //codigo del programa
115     index = 0;
116     s = (float *)_aligned_malloc((SIZE - 1) * sizeof(float), sizeof(__m256i));
117     for (int i = 0; i < (SIZE - 1) / NUMBER_FLOAT; i++) {
118         __m256 valueV = (__m256*)&v[i*NUMBER_FLOAT];
119         __m256 valueT = (__m256*)&t[i*NUMBER_FLOAT];
120         __m256 sub = _mm256_sub_ps(valueV, valueT);
121
122         //mal
123         float* p = (float*)&sub;
124         for (int j = 0; j < NUMBER_FLOAT; j++) {
125             s[i * NUMBER_FLOAT + j] = *(p+j);
126         }
127
128         if (PRINT_FUNCTIONS)
129             printf("La resta es %f\n", s[i + j + index]);
130     }
131 }
132 //eliminar de memoria el vector V
133 removeVector(v);
134 }
```

Para realizar esta operación primero debemos inicializar el vector V, que según la definición, es un vector dado por la multiplicación de cada valor de R y k. Tiene un tamaño idéntico a R, debido a que es el mismo vector multiplicado por un escalar. Acto seguido se crea la variable `kIntrinsics` de tipo `m256`, con el valor de k, convertido a `float`, en cada una de sus 8 posiciones.

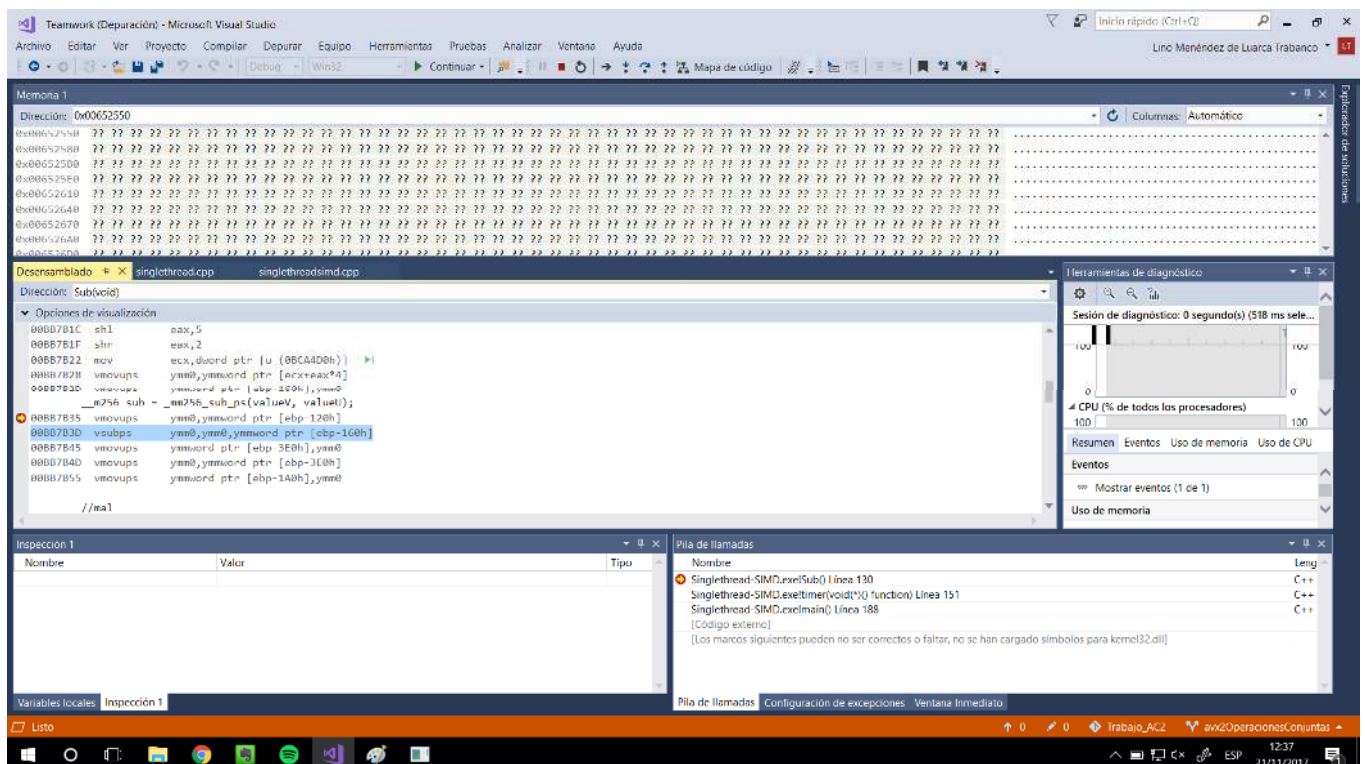
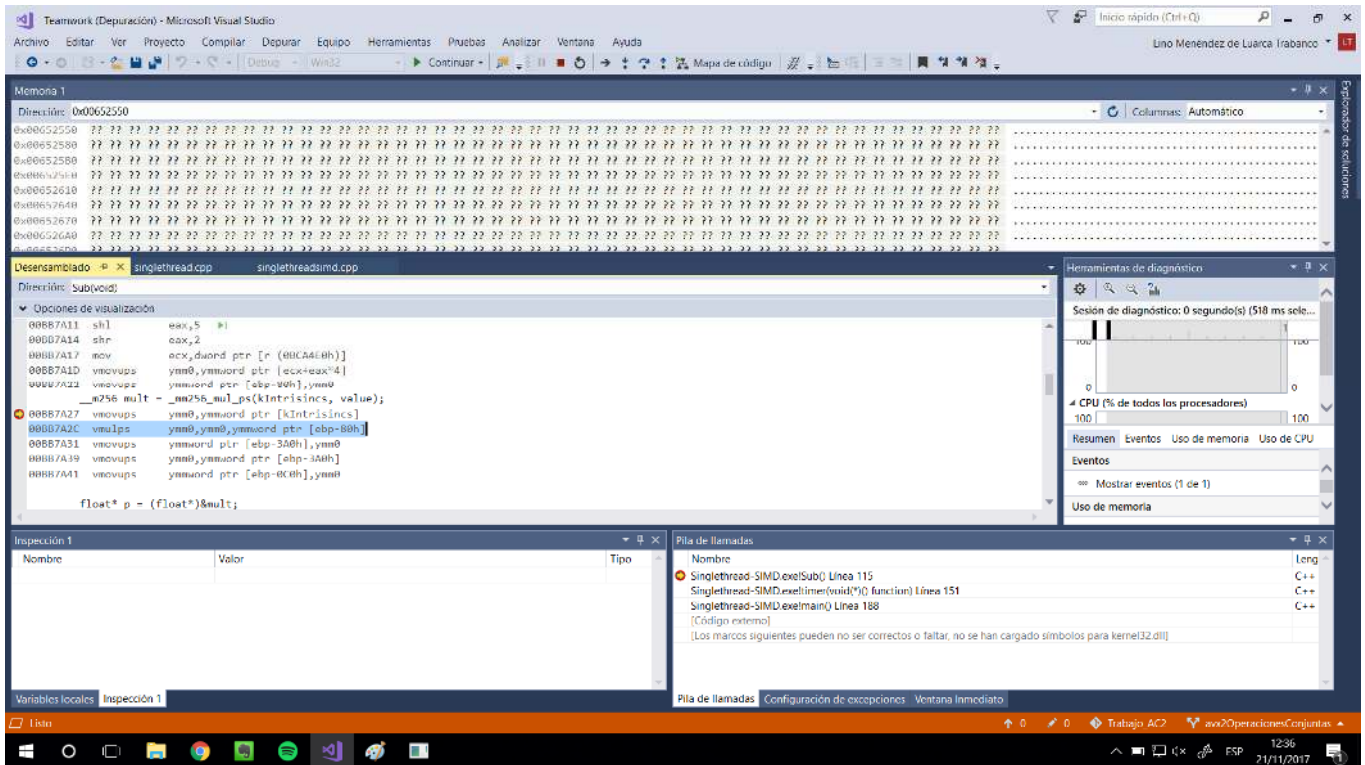
Creamos un bucle `for`, para hacer la operación indicada anteriormente (`_mm256_mul_ps()` donde se le pasa como parámetros `value` y `kIntrinsics`). A continuación, se hace un casting del valor de la multiplicación a un vector de floats, y mediante un bucle `for` se va guardando en el vector V cada resultado de la multiplicación en un vector de `floats`.

Inicializado ya el vector V, creamos el vector S con la función `_aligned_malloc`. Mediante un bucle ejecutado `SIZE-1/NUMBER_FLOAT`, convertimos ocho floats del vector V y del T en un valor de `m256`, que se restaran mediante la función `__mm256_sub_ps()`, guardando este valor en la variable `sub`. Al final del bucle se hace un casting de `sub` a vector `float`, y se va guardando en S los resultados de las restas.



Dependiendo del valor de PRINT\_FUNCTIONS se mostrará o no por pantalla el resultado de las restas. Por último, liberaremos la memoria del vector V.

Finalmente se ejecuta poniendo un punto de interrupción en la instrucción `_mm256_and_ps` para obtener su equivalencia en desensamblado tal como indica el enunciado del trabajo.



## 7. RESULTADOS OBTENIDOS

Después de ejecutar ambos programas y haber obtenido los tiempos concluimos que el programa que utiliza las extensiones SIMD tiene una mayor productividad que el que no las utiliza, en concreto el programa que utiliza las SIMD es 5,4 veces más productivo. Todo esto se refleja en el archivo **PL4-GrupoA.xlsx**. Se han guardado los datos obtenidos, se han calculado las medias, desviaciones típicas y la aceleración de SIMD respecto al otro programa.

	MEDIA	DESV. TIPICA	ACELER.
SINGLETHREAD	5,80486310	0,09120173	-
SINGLETHREAD-SIMD	1,08036590	0,02094909	5,37305287

## 8. DISTRIBUCIÓN DEL TRABAJO

El trabajo en general se ha hecho de manera grupal, aunque hay partes en las cuales hay integrantes que han trabajado más. Este es el caso de Lino y Javier, la segunda parte ha sido casi completada por ellos, a excepción de ciertas ideas propuestas por el resto de integrantes del grupo.



# FASE 2

## 1. OBJETIVOS

- Realizar versiones multihilo de los programas creados anteriormente para obtener una ganancia extra de rendimiento.

## 2. ATRIBUTOS Y CONSTANTES

A parte de los atributos y constantes declarados en la Fase 1, para la creación de las versiones multihilo se han declarado los siguientes atributos y constantes, los cuales son comunes para ambas partes:

- **NTHREADS**: Constante con el valor del número de hilos soportados por el procesador. Este número se obtiene con la función `getNumbersProcessor()`.
- **hThreadArray**: Vector de HANDLE, utilizado para almacenar los hilos
- **sizes**: Vector de enteros. En cada posición almacenará la posición de inicio a partir de la cual trabajará cada hilo sobre un vector de tamaño **SIZE**.
- **arrayK**: vector de enteros. En cada posición cada hilo guardará el número de positivos de la función CountPositive

## 3. FUNCIONES COMUNES

Para la realización multihilo de los programas hemos mantenido algunas funciones de la Fase 1 y hemos añadido otras:

1. **createVector()**  
Explicada ya en la Fase 1.
2. **removeVector()**  
Explicada ya en la Fase 1.
3. **getNumbersProcessor()**

```
unsigned int getNumbersProcessor() {  
    SYSTEM_INFO siSysInfo;  
  
    // Copy the hardware information to the SYSTEM_INFO structure.  
    GetSystemInfo(&siSysInfo);  
  
    return siSysInfo.dwNumberOfProcessors;  
}
```

Con esta función se consigue el número de hilos soportados por el procesador. Para ello definimos una variable(`siSysInfo`), de tipo `SYSTEM_INFO`, y gracias a la función `GetSystemInfo()` guardaremos los

datos relativos al sistema. Por último, se coge el valor del atributo `dwNumberOfProcessors`.

#### 4. `generateSizes()`

```
void generateSizes() {
    sizes = (int *)malloc(sizeof(int) * NTHREADS);
    for (int i = 0; i < NTHREADS; i++) {
        sizes[i] = i * (SIZE / NTHREADS);
    }
}
```

Mediante esta función obtenemos un vector con las posiciones iniciales de un vector de tamaño `SIZE` que le corresponden a cada hilo. Para ello inicializamos `size` con un tamaño de `NTHREADS` usando la función `malloc()`, posteriormente mediante un `for` que recorre el vector `size`, le vamos asignando la posición inicial que le corresponde a cada hilo.

#### 5. `wait()`

```
void wait() {
    WaitForMultipleObjects(NTHREADS, hThreadArray, true, INFINITE);
}
```

Función que, mediante la utilización de otra función llamada `WaitForMultipleObjects()`, espera a que todos los hilos hayan finalizado su tarea.

#### 6. `Dif2()`

```
void Dif2() {
    for (unsigned int i = 0; i < NTHREADS; i++) {
        hThreadArray[i] = CreateThread(NULL, 0, Dif2Proc, &sizes[i], 0, NULL);
    }
    wait();
}
```

Función equivalente a la operación `Dif2()`. En esta función mediante un bucle, creamos `NTHREAD` hilos haciendo uso de la función `CreateThread()`, para almacenarlos en el array de hilos `hThreadArray`. Cada uno de estos hilos ejecutará la función `Dif2Proc()`, donde se encuentra el procedimiento. Finalmente mediante la función `wait()` esperamos a que todos los hilos finalicen su ejecución y, por tanto, sus valores.

## 7. Sub()

```
void Sub() {
    //codigo del programa
    for (unsigned int i = 0; i < NTHREADS; i++) {
        hThreadArray[i] = CreateThread(NULL, 0, SubProc, &sizes[i], 0, NULL);
    }
    wait();
}
```

Función equivalente a la operación Sub(). En esta función mediante un bucle, creamos NTHREAD hilos haciendo uso de la función CreateThread(), para almacenarlos en el array de hilos hThreadArray. Cada uno de estos hilos ejecutará la función SubProc(), donde se encuentra el procedimiento. Finalmente mediante la función wait() esperamos a que todos los hilos finalicen su ejecución y, por tanto, sus valores.

## 8. CountPositiveValues ()

```
void CountPositiveValues() {
    for (unsigned int i = 0; i < NTHREADS; i++) {
        arrayK[i] = 0;
        hThreadArray[i] = CreateThread(NULL, 0, CountPositiveValuesProc, &sizes[i], 0, NULL);
    }

    wait();
    k = 0;

    for (unsigned int i = 0; i < NTHREADS; i++) {
        k += arrayK[i];
    }

    if (PRINT_FUNCTIONS)
        printf("El contador de numeros positivos es %d\n", k);
}
```

Función equivalente a la operación Count Positives. Mediante un bucle, inicializamos a 0 cada posición del vector arrayK, se crea cada hilo y se lo asignamos al vector hThreadArray. Cada uno de estos hilos ejecutará la función CountPositiveValuesProc(), donde se encuentra el procedimiento. Al final del bucle mediante la función wait() esperamos a que todos los hilos acaben su ejecución. Cuando esto se produzca recorreremos el arrayK, sumamos todos los valores que ha proporcionado cada hilo y almacenamos en la variable k la suma.

## 9. main()

```
int main() {
    time_t ti;
    srand((unsigned)time(&ti)); // Informacion sobre srand https://www.tutorialspoint.com/cplusplus/article/srand/

    double times[TIMES];
    generateSizes(); //generar la lista de indices

    for (int j = 0; j < TIMES; j++) {
        times[j] = 0; //aseguramos valores reales
        for (int i = 0; i < NTIMES; i++) {
            //vectores aleatorios
            u = createVector();
            w = createVector();
            t = createVector();

            //vectores aux
            arrayK = (unsigned int *)malloc(sizeof(unsigned int) * (NTHREADS));

            //vectores resultantes
            r = (float *)malloc(sizeof(float) * (SIZE - 1));
            s = (float *)malloc(sizeof(float) * (SIZE - 1));

            times[j] += timer(Dif2);
            times[j] += timer(CountPositiveValues);
            times[j] += timer(Sub);

            std::free(arrayK);
            removeVector(u);
            removeVector(w);
            removeVector(t);
            removeVector(s);
            removeVector(r);
        }

        if (PRINT_TIMER)
            printf("Elapsed total time in seconds: %f\n", times[j]);
    }
}
```

El funcionamiento de esta función es el mismo que en la Fase 1, la diferencia más notable es la inclusión de la inicialización del vector **arrayK** usada para almacenar el cálculo del escalar de cada hilo.

## 9. PRIMERA PARTE Y SEGUNDA

El algoritmo utilizado en la Fase 2 es exactamente el mismo que hemos utilizado en la Fase1, tanto para la parte sin instrucciones SIMD como para la parte que sí las utiliza. La única diferencia es el uso del `multiThread`. A diferencia de la versión `SingleThread`, `Dif2()`, `Sub()` y `CountPositiveValues()`, llaman a los hilos. Las funciones multihilo son: `Dif2Proc()`, `SubProc()` y `CountPositiveValuesProc()`. En cada una de estas funciones convertimos el parámetro(`index`), el cual es la posición inicial desde la que trabajará el hilo, a `int`. Primeramente, se hace un casteo de `LPVOID` a `int*` usando `reinterpret_cast` para coger el valor a donde apunta. Mediante la variable `tamaño` obtenemos el número de posiciones asignadas al hilo. El bucle se iniciará en la `posición inicial` asignada y finalizará en `posición inicial + tamaño - 1`. El resto del cuerpo de las funciones multihilo es el algoritmo utilizado para resolver cada operación en Fase 1.

## 10. RESULTADOS OBTENIDOS

Después de ejecutar ambos programas, sobre un ordenador con 4 hilos, y haber obtenido los tiempos concluimos que el programa que utiliza las extensiones SIMD tiene una mayor productividad que el que no las utiliza, en concreto el programa que utiliza las SIMD es **4.12** veces más productivo. Todo esto se refleja en el archivo **PL4-GrupoA.xlsx** (Hoja "Fase 2"). Se han guardado los datos obtenidos, se han calculado las medias, desviaciones típicas y la aceleración de SIMD respecto al otro programa. Además, se hace una comparación de tiempos con el SingleThread dándonos que el *MultiThread* es **3.81** veces más productivo (muy cerca de la teórica). Sin embargo, en el *MultiThread-SIMD* es aún más abismal (aun estando muy alejado de la teórica) dándonos una aceleración de **15.75**.

	MEDIA	DESV. TIPICA	ACCELER. RESPECTO SINGLETHREAD	ACCELER. RESPECTO MULTITHREAD
SINGLETHREAD	5,80486310	0,09120173	-	-
SINGLETHREAD-SIMD	1,08036590	0,02094909	5,37305287	-
MULTITHREAD	1,52137570	0,09128496	3,81553557	-
MULTITHREAD-SIMD	0,36850290	0,00638393	15,75255744	4,12853115

## 11. DISTRIBUCIÓN DEL TRABAJO

La Fase2 ha sido esencialmente realizada por Javier y Lino.