|  |
| --- |
| **PL4, Grupo A** |
| MEMORIA TRABAJO |
| FASE 1 - 2 |

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

Contenido

[FASE 1 2](#_Toc500751590)

[1. OBJETIVOS 2](#_Toc500751591)

[2. OPERACIONES 2](#_Toc500751592)

[3. ATRIBUTOS Y CONSTANTES 2](#_Toc500751593)

[4. FUNCIONES COMUNES 3](#_Toc500751594)

[1. createVector() 3](#_Toc500751595)

[2. removeVector() 4](#_Toc500751596)

[3. main() 4](#_Toc500751597)

[5. PRIMERA PARTE ----- PROYECTO SINGLETHREAD 5](#_Toc500751598)

[1. Operación Dif2 5](#_Toc500751599)

[2. Operación Count Positives 5](#_Toc500751600)

[3. Operación Sub 6](#_Toc500751601)

[6. SEGUNDA PARTE ----- PROYECTO SINGLETHREAD-SIMD 6](#_Toc500751602)

[1. SIMD 6](#_Toc500751603)

[2. Operación Dif2 9](#_Toc500751604)

[3. Operación Count Positives 11](#_Toc500751605)

[4. Operación Sub 13](#_Toc500751606)

[7. RESULTADOS OBTENIDOS 15](#_Toc500751607)

[8. DISTRIBUCIÓN DEL TRABAJO 15](#_Toc500751608)

[FASE 2 16](#_Toc500751609)

[1. OBJETIVOS 16](#_Toc500751610)

[2. ATRIBUTOS Y CONSTANTES 16](#_Toc500751611)

[3. FUNCIONES COMUNES 16](#_Toc500751612)

[1. createVector() 16](#_Toc500751613)

[2. removeVector() 16](#_Toc500751614)

[3. getNumbersProcessor() 16](#_Toc500751615)

[4. generateSizes() 17](#_Toc500751616)

[5. wait() 17](#_Toc500751617)

[6. Dif2() 17](#_Toc500751618)

[7. Sub() 18](#_Toc500751619)

[8. CountPositiveValues () 18](#_Toc500751620)

[9. main() 19](#_Toc500751621)

[9. PRIMERA PARTE Y SEGUNDA 20](#_Toc500751622)

[10. RESULTADOS OBTENIDOS 20](#_Toc500751623)

[11. DISTRIBUCIÓN DEL TRABAJO 20](#_Toc500751624)

**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.

1. **OPERACIONES**

A nuestro grupo se le asignaron 3 operaciones:

* Dif2
* Count Positives
* Sub

1. **ATRIBUTOS Y CONSTANTES**

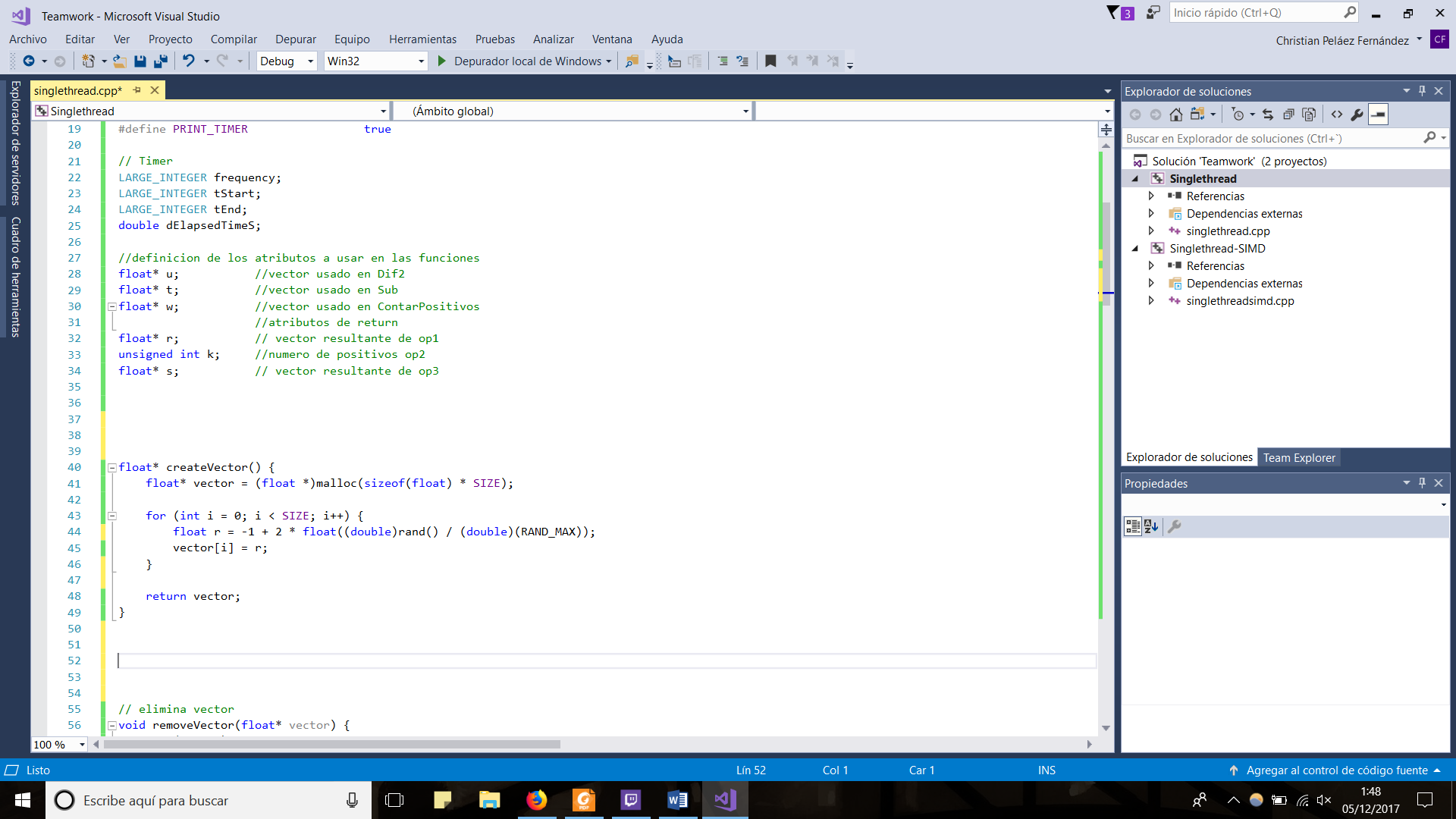
Se han declarado los siguientes atributos y constantes, las cuales son comunes para ambas partes:

* 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.

1. **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()
   1. Primera parte

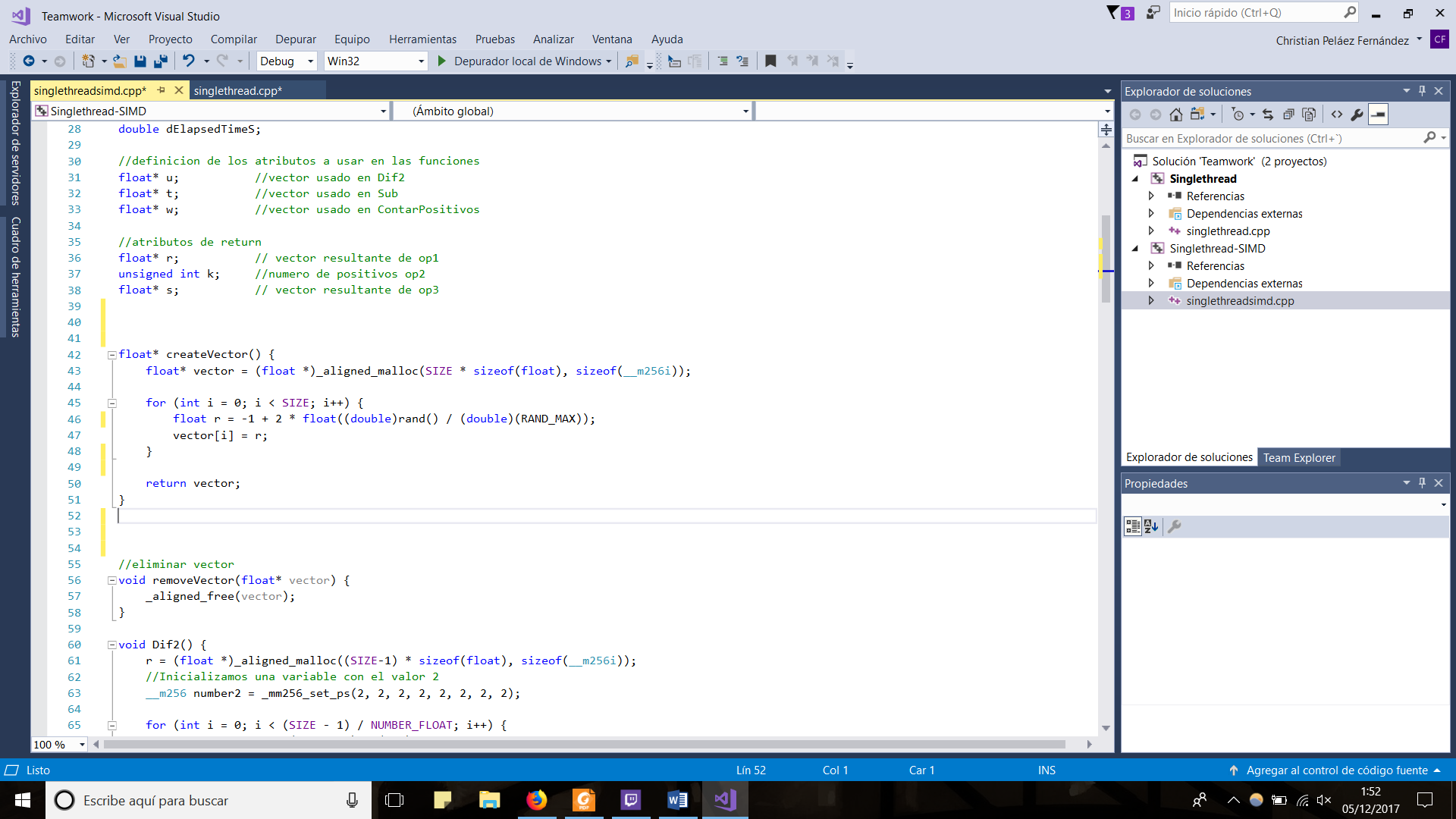


La finalidad de esta función es crear un vector de tamaño *SIZE*, inicializarlo con valores flotantes pertenecientes al intervalo [-1,1] y posteriormente devolverlo.

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

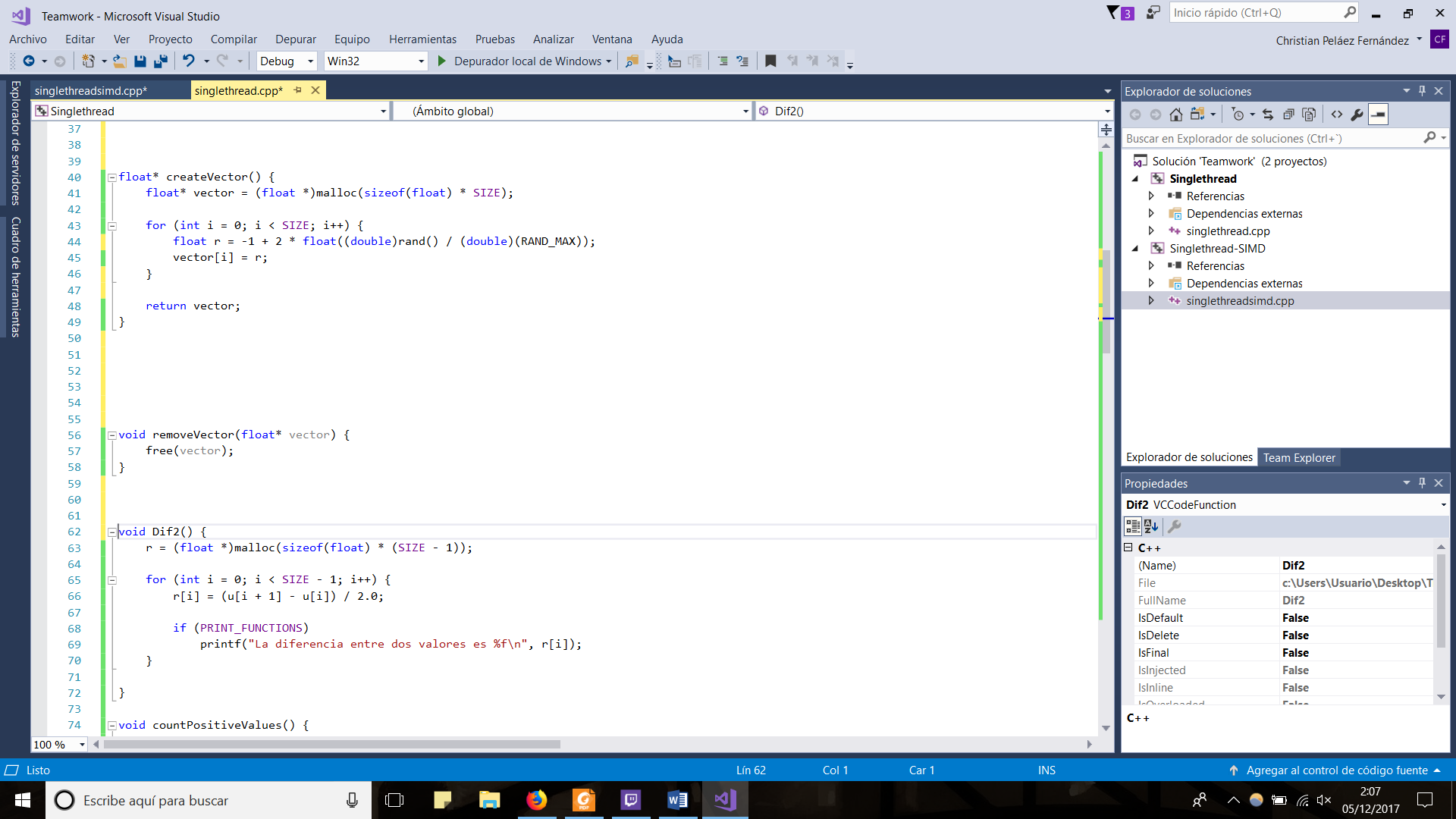
[-1,1], una vez inicializado el vector se devuelve.

* 1. Segunda parte



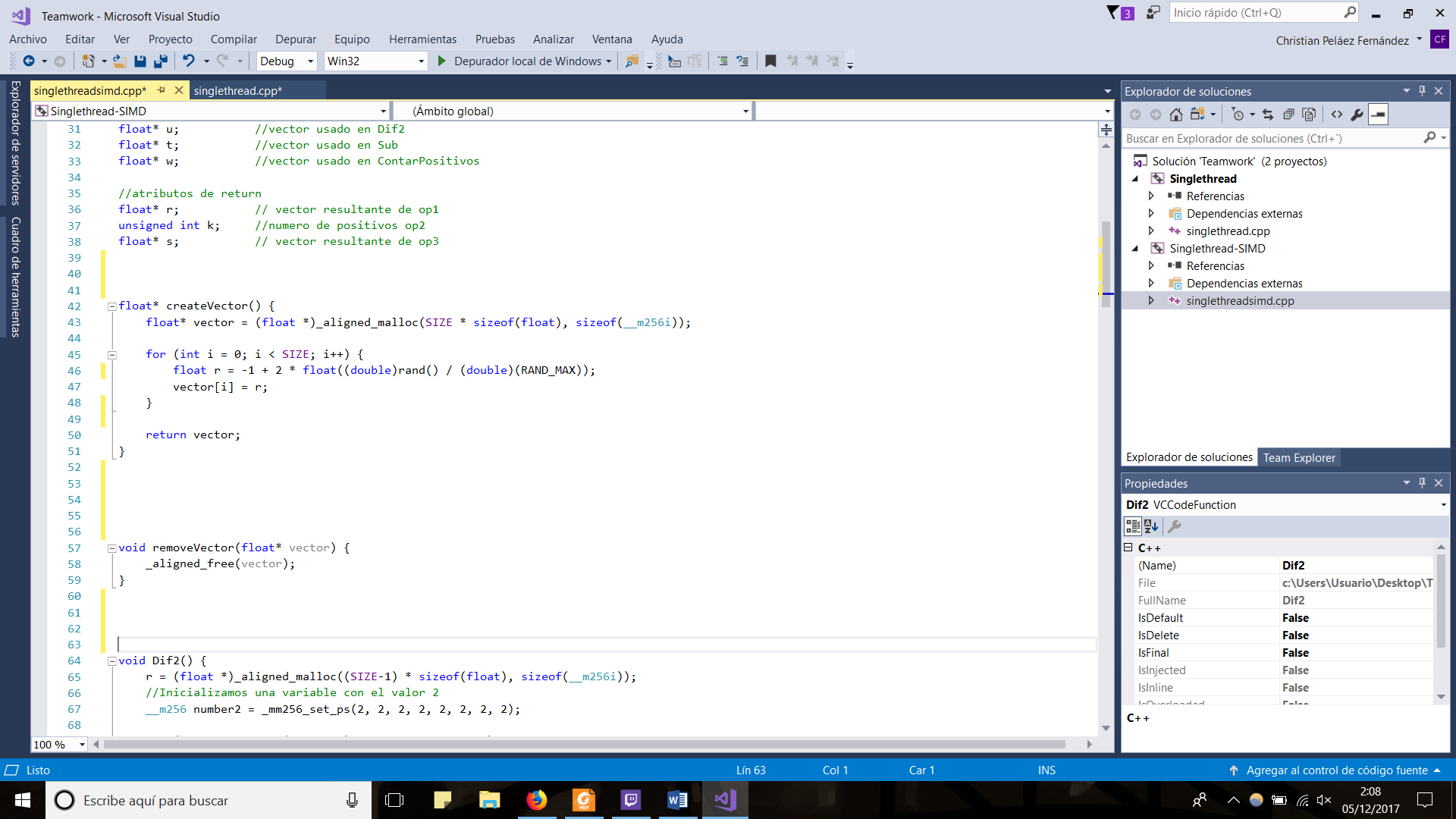
En la segunda parte, al trabajar con instrucciones SIMD debemos crear un vector con espacio en memoria alineado según el dato m256 . Para ello a la hora de crear el vector se utiliza la función \_aligned\_malloc(), a la cual pasamos por parámetro el tamaño del bloque a reservar y el valor de la alineación(el 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.

1. removeVector()
   1. Primera parte



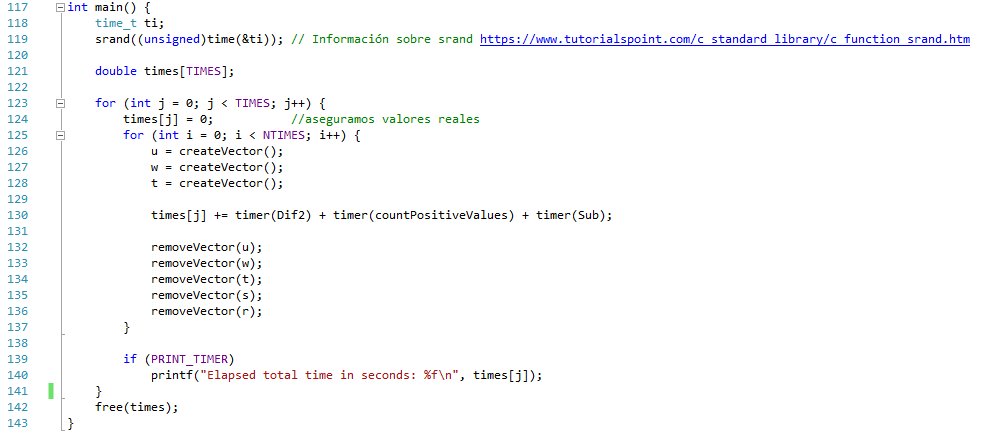
El objetivo de esta función es liberar el espacio utilizado por el vector pasado por parámetro, para ello hacemos uso de la función free().

* 1. Segunda parte



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

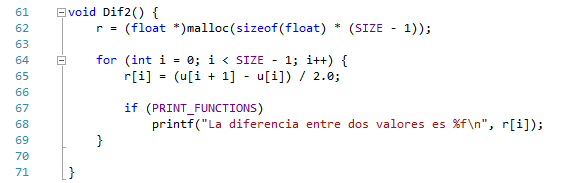
1. main()



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.

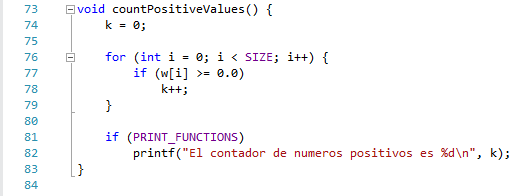
En el bucle for que se ejecutará NTIMES creamos en cada iteración los vectores U, W y T mediante la función createVectores() y se obtienen los tiempos de las operaciones, que se suman y se guardan en el vector times, al final de dicho bucle se libera la memoria de los vectores U, W, T, S y R. Si la constante 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).

1. **PRIMERA PARTE ----- PROYECTO SINGLETHREAD**
2. ***Operación Dif2***



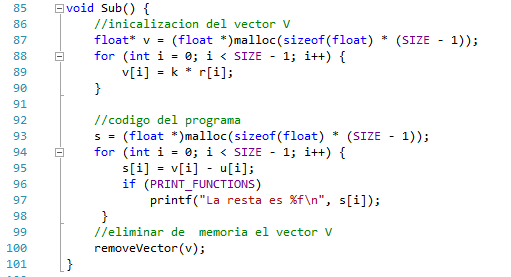
La operación Dif2 calcula la media de la diferencia entre los valores con posiciones consecutivas del vector U. Para ello creamos un vector R que tiene un tamaño SIZE-1. Esto es debido a que al restar valores con posiciones consecutivas hacemos SIZE-1 restas que se guardaran en R. Después de haberlo 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.

1. **Operación Count Positives**



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.

1. **Operación Sub**



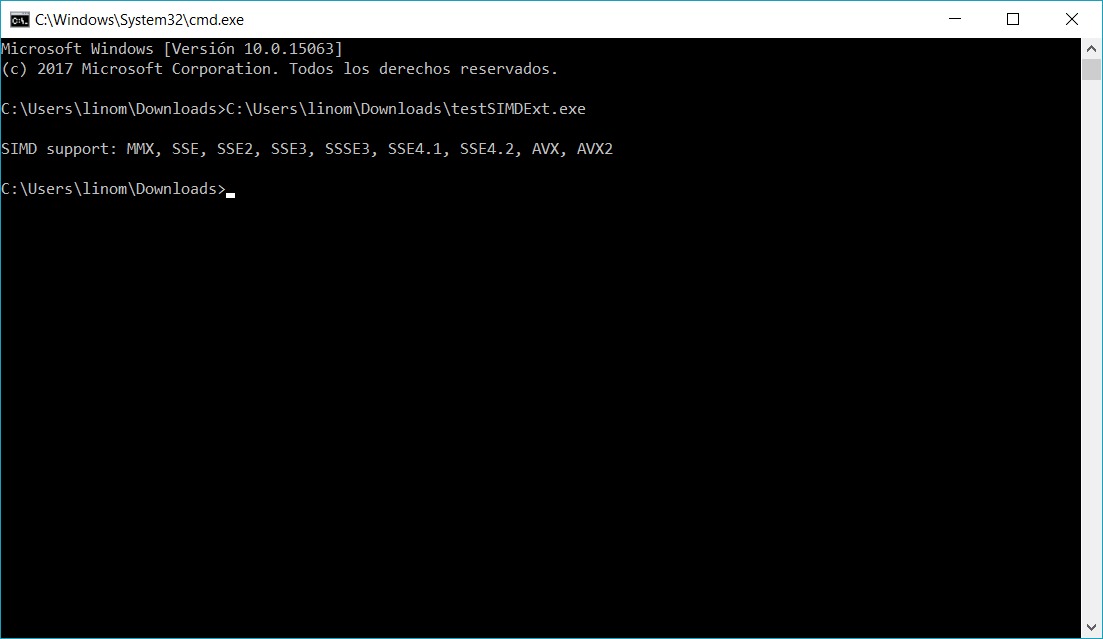
Esta operación resta los valores de cada posición de los vectores V y U. Para realizar esta operación creamos el vector V reservando un espacio en memoria con la función malloc(), lo recorremos con un bucle y en cada posición guardamos el valor que resulta del producto de k por el valor de cada posición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

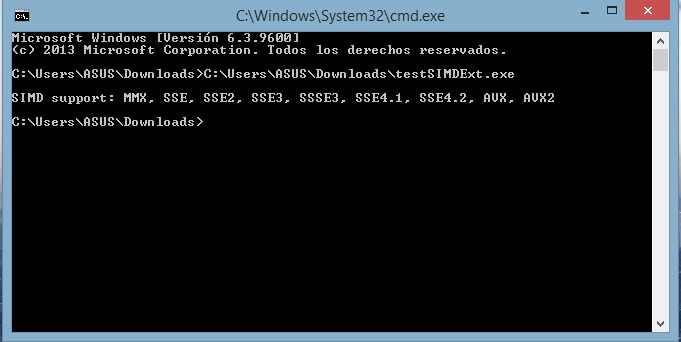
Asignamos espacio en memoria para el vector S, hacemos otro bucle donde vamos asignando a cada posición de S la resta de cada posición entre el vector V y el vector U. Finalmente eliminamos de memoria el vector V.

1. **SEGUNDA PARTE ----- PROYECTO SINGLETHREAD-SIMD**
2. **SIMD**

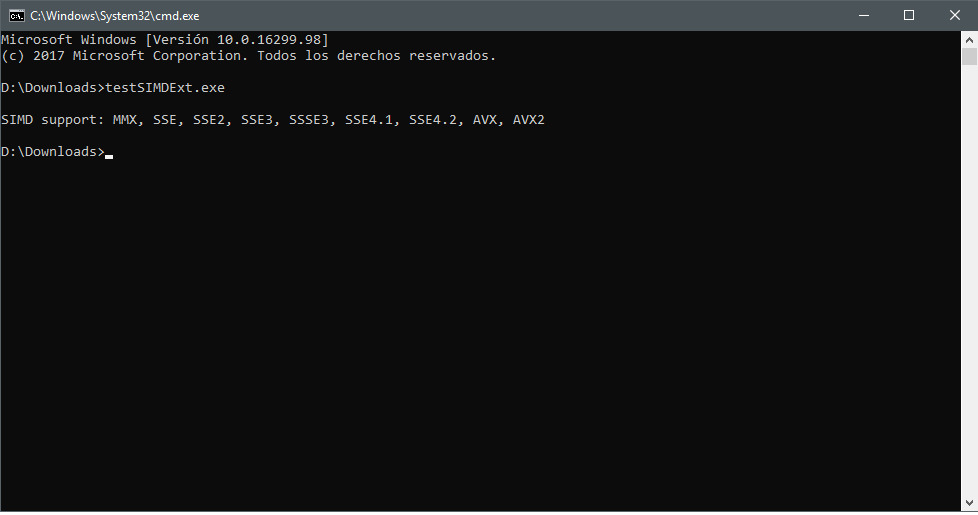
En el enunciado del trabajo se nos indicaba que el nivel máximo de extensiones SIMD era **AVX-512F,** ejecutando el programa dado por los profesores para mostrar el nivel máximo de extensiones SIMD que nuestros ordenadores soportaban obtuvimos que todos nuestros ordenadores soportaban como máximo hasta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

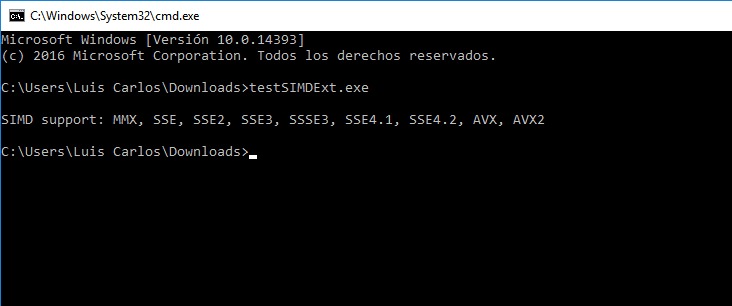
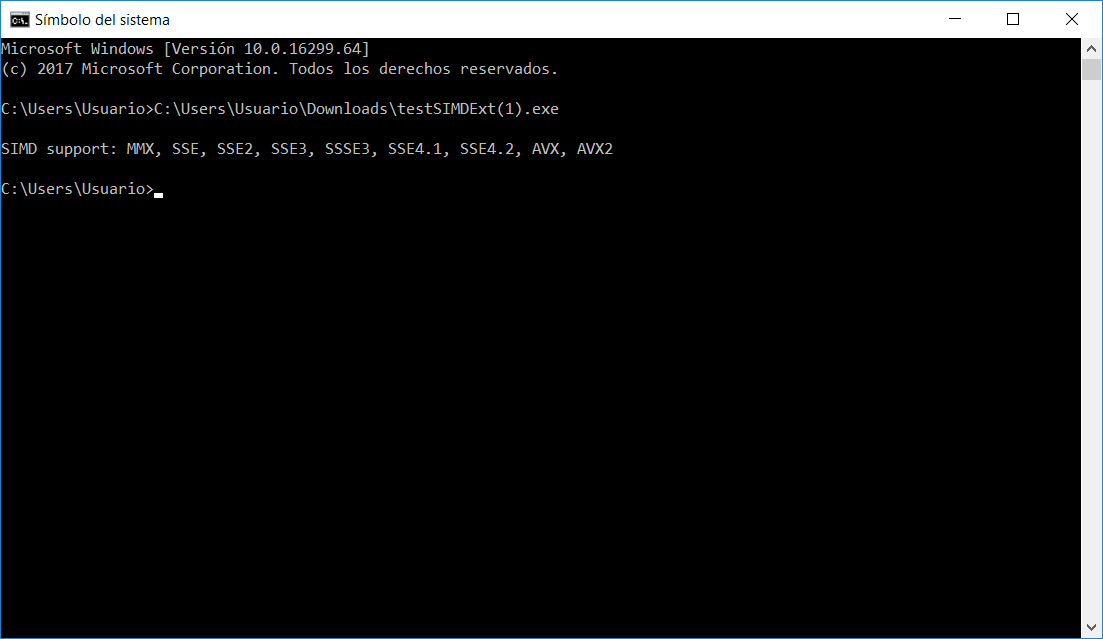
* Captura de Lino:



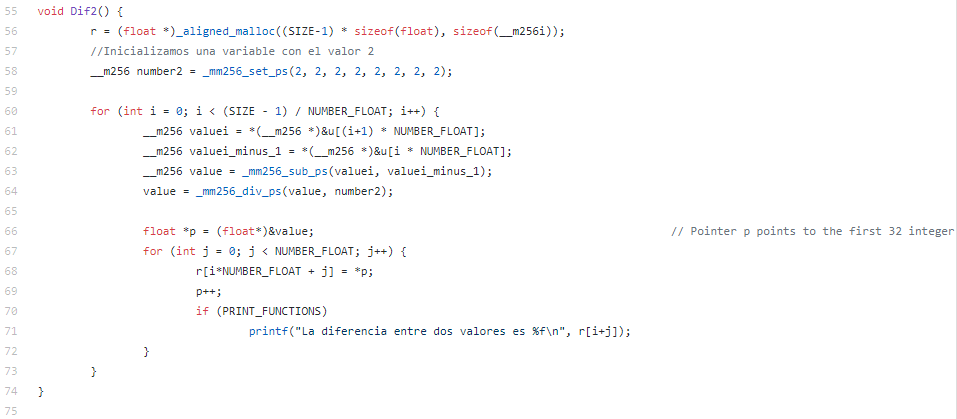
* Captura de Sara:

* Captura de Javier:



* Captura de Luis:
* Captura Christian:

1. **Operación Dif2**

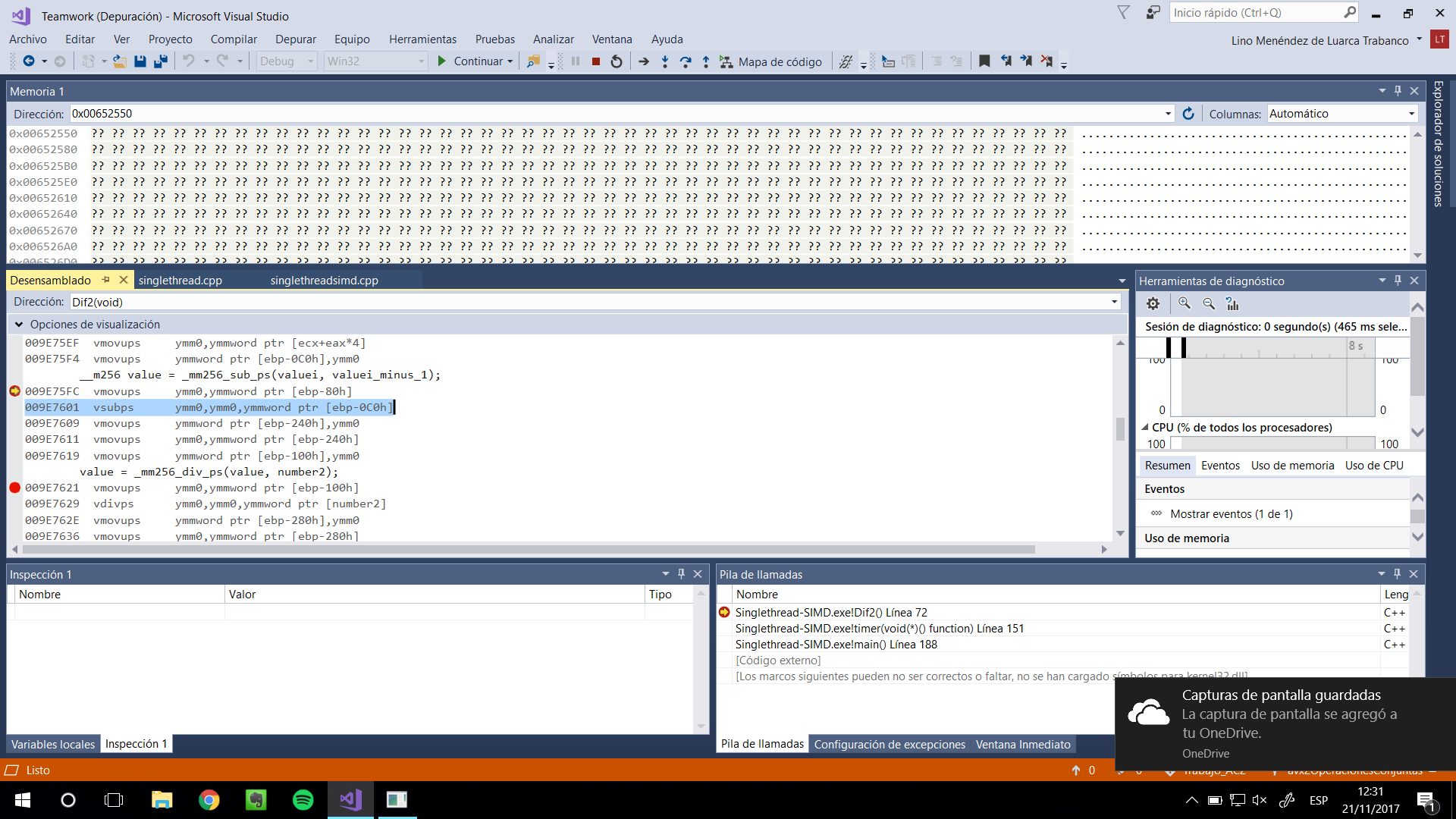


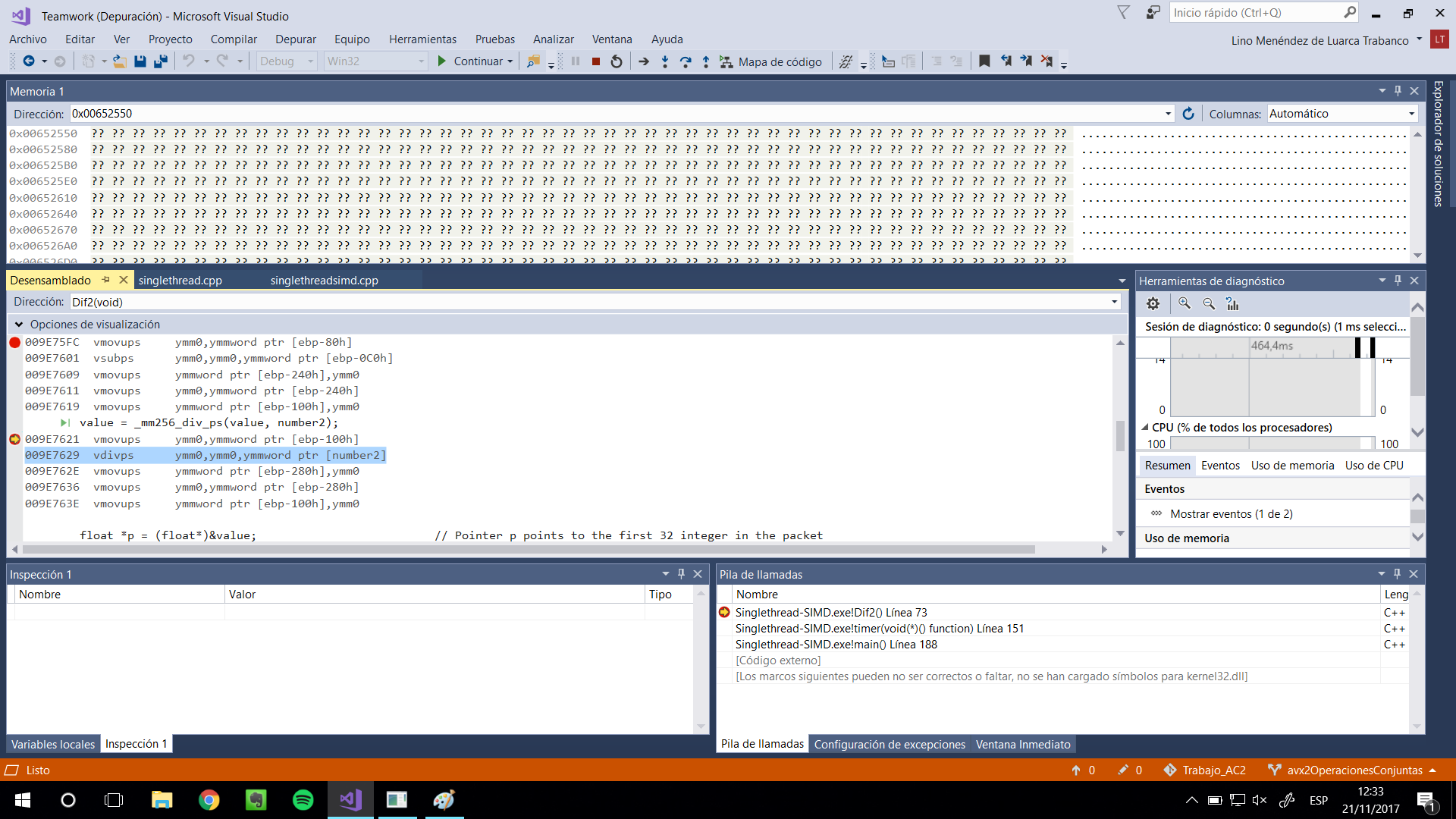
Mediante la función \_aligned\_malloc() creamos un vector R de floats de tamaño SIZE-1. A continuación inicializamos a 2 las 8 partes de 32 bits de la variable number2 de tipo m256, ya que posteriormente tendremos que dividir entre 2 cada elemento del vector de tipo m256 que contenga los resultados de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

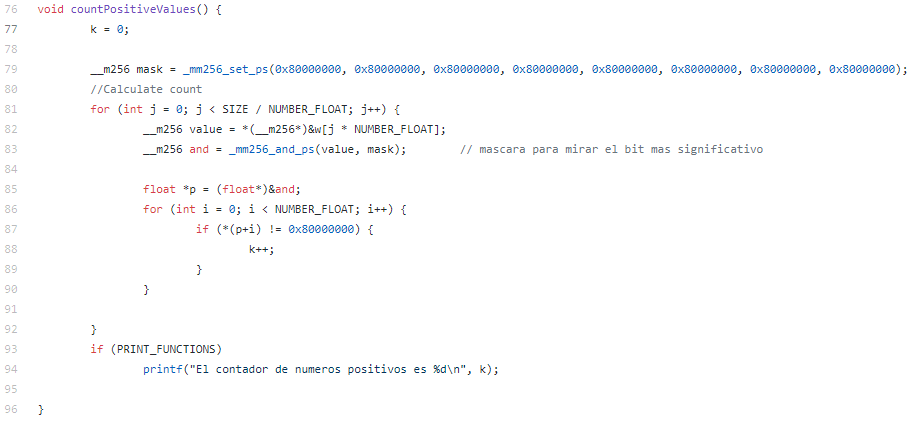
Al final del bucle hacemos un casting a la variable value, transformándola en un vector de floats. Para ello creamos un puntero float(p) el cual apuntará a la dirección de inicio de la variable value (cuyo valor es un tipo m256, el cual alberga los bits de 8 floats). Mediante un bucle for anidado, recorremos 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.

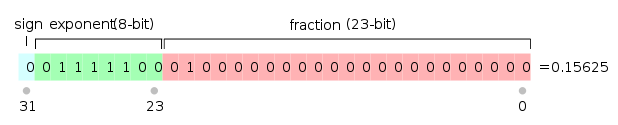


1. **Operación Count Positives**

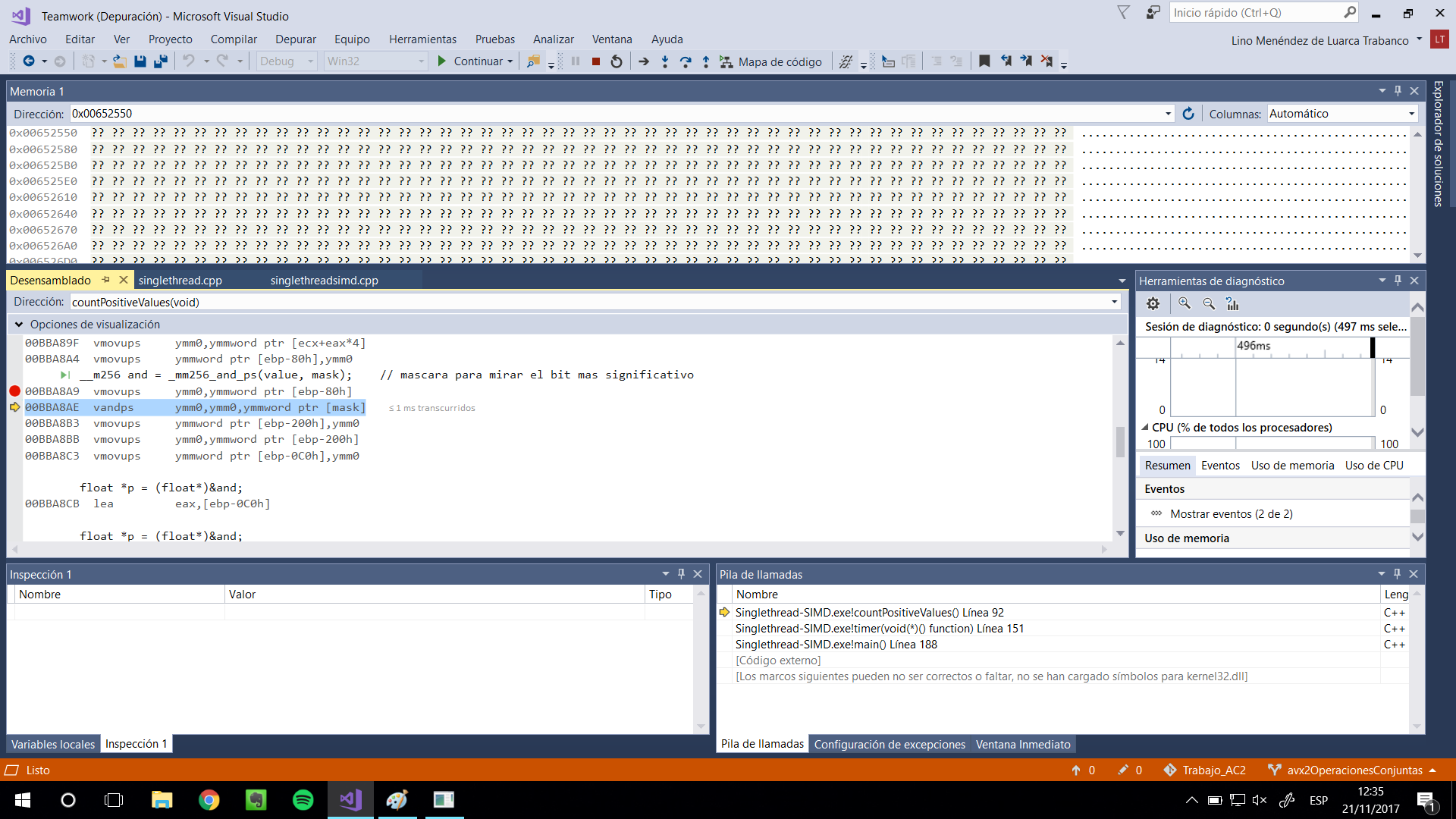


Para comprobar el número de positivos en el vector W creamos una máscara de tipo m256 cuyo valor en cada elemento es 0x80000000(el bit más significativo está a 1).

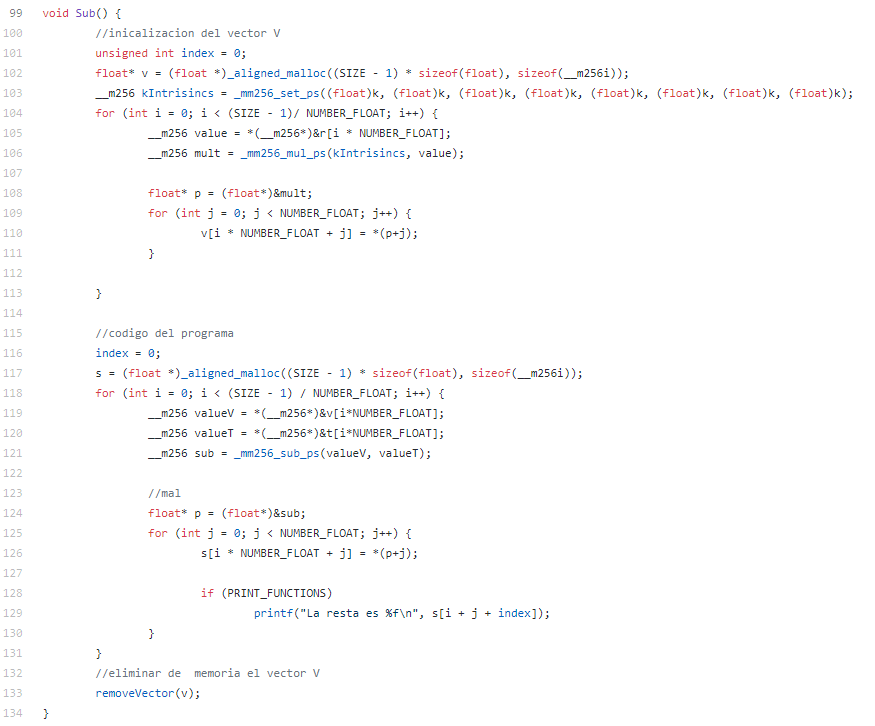
Creamos un bucle for y en cada iteración de este vamos asignando elementos m256 del vector W a la variable value, realizando una operación AND bit a bit entre la máscara y la variable value y comprobamos si el resultado de la and en cada uno de los 8 elementos floats es igual a 0x80000000, si no fuese 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.

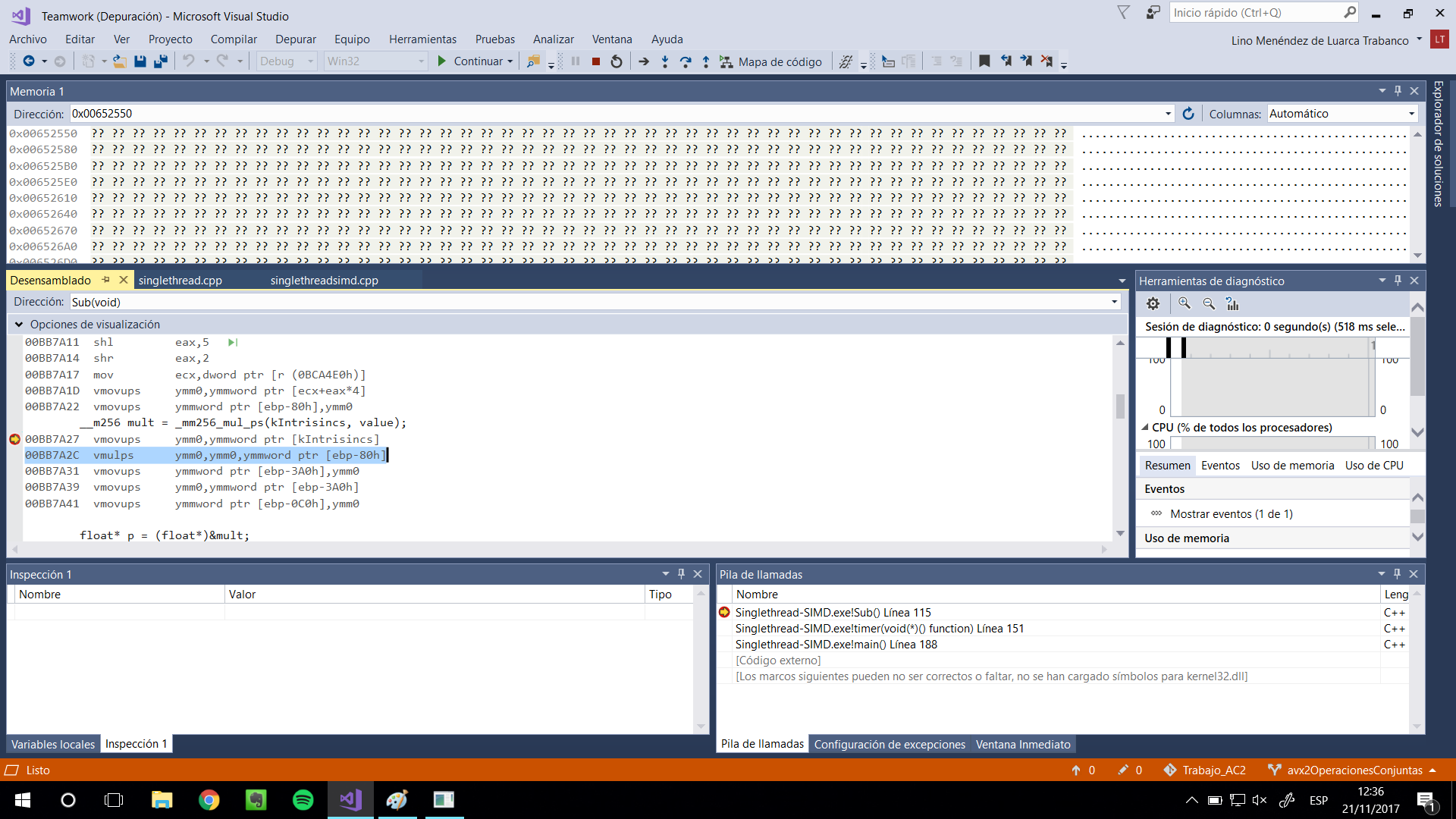
1. **Operación Sub**

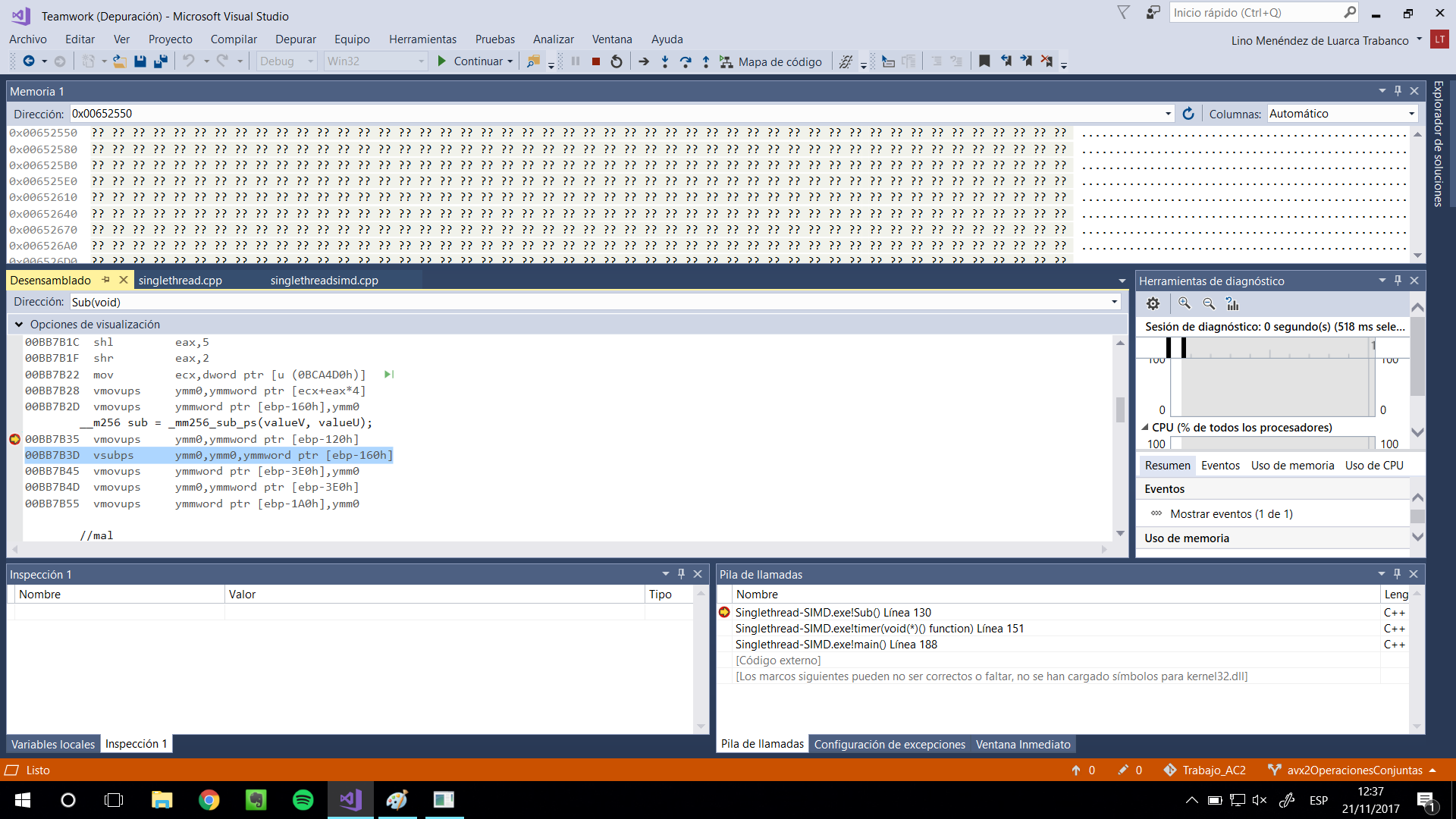


Para realizar esta operación primero debemos inicializar el vector V, que según la definición, es un vector dado por la multiplicación de cada valor de R y k. Tiene un tamaño idéntico a R, debido a que es el mismo vector multiplicado por un escalar. Acto seguido se crea la variable kIntrisincs de tipo 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 kIntrisincs). 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.



1. **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.



1. **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.

1. **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

1. **FUNCIONES COMUNES**

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

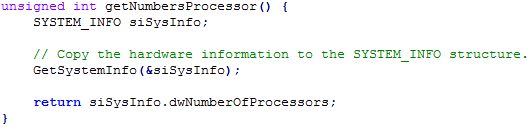
1. createVector()

Explicada ya en la Fase 1.

1. removeVector()

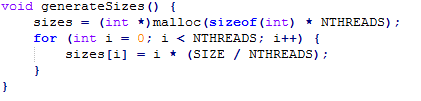
Explicada ya en la Fase 1.

1. **getNumbersProcessor()**



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.

1. generateSizes()



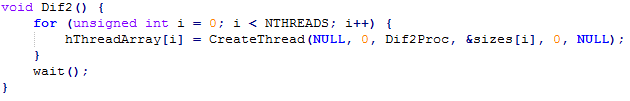
Mediante esta función obtenemos un vector con las posiciones iniciales de un vector de tamaño SIZE que le corresponden a cada hilo. Para ello inicializamos size con un tamaño de NTHREADS usando la función malloc(), posteriormente mediante un for que recorre el vector size, le vamos asignando la posición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

1. wait()

https://cdn.discordapp.com/attachments/244425335364517888/389511757162348545/unknown.png

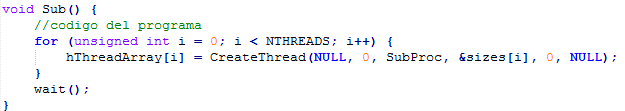
Función que, mediante la utilización de otra función llamada WaitForMultipleObjects(), espera a que todos los hilos hayan finalizado su tarea.

1. Dif2()



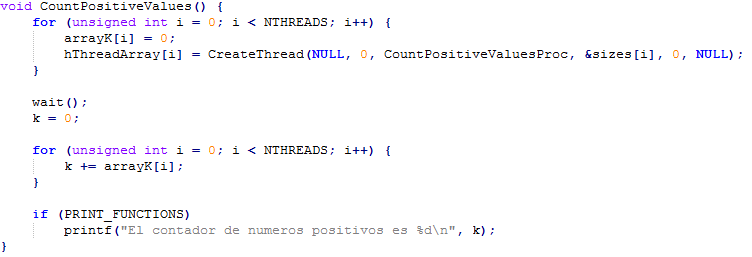
Función equivalente a la operación Dif2(). En esta función mediante un bucle, creamos NTHREAD hilos haciendo uso de la función CreateThread(), para almacenarlos en el array de hilos hThreadArray. Cada uno de estos hilos ejecutará la función Dif2Proc(), donde se encuentra el procedimiento. Finalmente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

1. Sub()



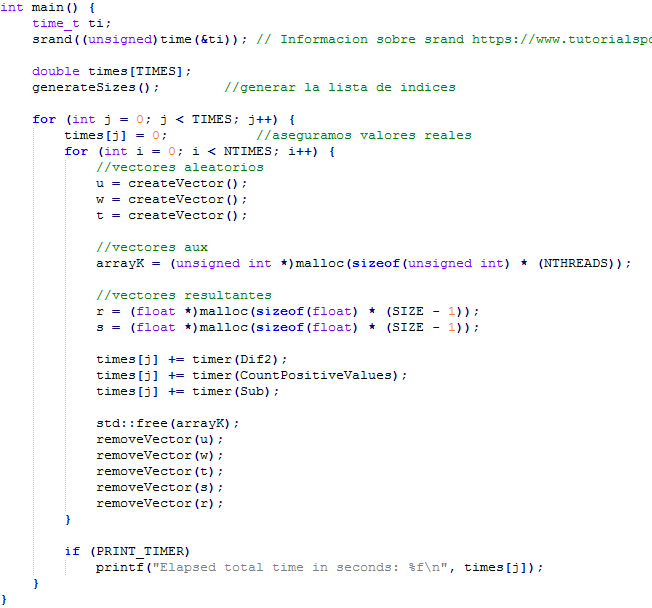
Función equivalente a la operación Sub(). En esta función mediante un bucle, creamos NTHREAD hilos haciendo uso de la función CreateThread(), para almacenarlos en el array de hilos hThreadArray. Cada uno de estos hilos ejecutará la función SubProc(), donde se encuentra el procedimiento. Finalmente mediante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

1. CountPositiveValues ()



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 recorremos el arrayK, sumamos todos los valores que ha proporcionado cada hilo y almacenamos en la variable k la suma.

1. main()



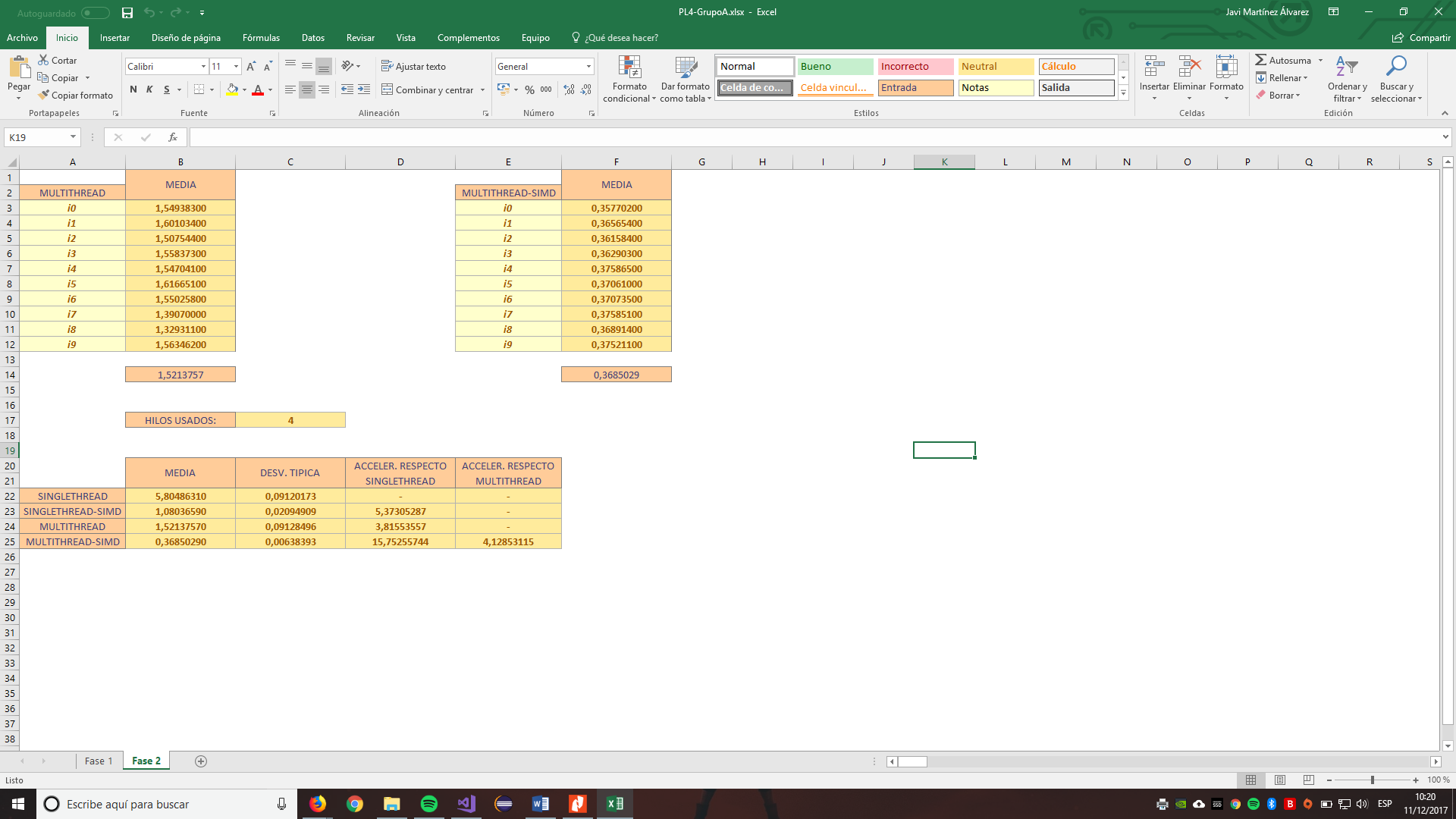
El funcionamiento de esta función es el mismo que en la Fase 1, la diferencia más notable es la inclusión de la inicialización del vector arrayK usada para almacenar el cálculo del escalar de cada hilo.

1. **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 eluso 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.

1. **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.**



1. **DISTRIBUCIÓN DEL TRABAJO**

La Fase2 ha sido esencialmente realizada por Javier y Lino.