

# Arquitecturas Avanzadas

Grupo AA-2-1

25/10/2019

Julio César Velasquez Cardenas 1397896  
Sergio Prada Maeso 1459122  
Juan Carlos Bermúdez Rodríguez 1455486

## A) Which loops have been vectorized and which have not? Check the source file to answer.

Se han vectorizado los bucles que no contienen en su interior instrucciones de impresión por pantalla.

Debido a las llamadas a la función "print", haciendo el report de nivel 5 obtenemos la siguiente información

```
LOOP BEGIN at saxpy.c(26,2)
  remark #15382: vectorization support: call to function printf(const char *__restrict__, ...) cannot be vectorized [ saxpy.c(27,4) ]
  remark #15344: loop was not vectorized: vector dependence prevents vectorization
LOOP END
```

Additionally, you can use the online compiler at <https://godbolt.org/> to check the machine code that has been generated. Compare the machine code generated when no optimization is applied (-O1) or when maximum optimization is applied (-O3). Regard that optimization reports cannot be read at <https://godbolt.org/> (that could be a nice extension to add, by the way).

The image displays two side-by-side screenshots of the Godbolt.org online compiler interface, comparing the assembly output for two optimization levels: -O1 (left) and -O3 (right).

**-O1 (Left):** The assembly code shows a loop starting at line 15, labeled `..B1.2:`. It contains instructions for setting up the stack frame, moving data, and a `printf` call at line 29. The loop body includes instructions for moving data from memory to registers and performing arithmetic operations. The loop ends at line 32, labeled `..B1.5:`.

**-O3 (Right):** The assembly code shows the same loop, but with significant vectorization. The instructions are more complex, involving `cvtdq2ps` (convert double to single precision), `padd` (packed add), and `movups` (move unaligned single-precision floating-point data). The loop body is more extensive, with many more instructions for vector operations. The loop ends at line 53, labeled `..B1.3:`.

Comparando lo que obtenemos en ambas opciones, observamos que para el primer bucle (instrucciones en **Amarillo**) corresponden a la línea del "for" mientras que las que están en **morado** son las instrucciones que ejecuta el código interno del bucle.

Observamos que al compilar con el -O3 lo que hace es ampliar el número de operaciones de igualación de los vectores que hace al **mismo tiempo** (vectorización) obligando así al compilador a hacer el menor número de saltos posible como se vé que hace en la versión de -O1.

**Besides vectorization, what other optimizations can you detect on the code? Try to compile with `-qopt-report=5` and see what additional information is generated by the compiler at the report file.**

```

LOOP BEGIN at saxpy.c(20,2)
  remark #15388: vectorization support: reference Y[i] has aligned access [ saxpy.c(21,11) ]
  remark #15388: vectorization support: reference X[i] has aligned access [ saxpy.c(21,4) ]
  remark #15305: vectorization support: vector length 4
  remark #15427: loop was completely unrolled
  remark #15399: vectorization support: unroll factor set to 8
  remark #15309: vectorization support: normalized vectorization overhead 0.025
  remark #15300: LOOP WAS VECTORIZED
  remark #15449: unmasked aligned unit stride stores: 2
  remark #15475: --- begin vector cost summary ---
  remark #15476: scalar cost: 10
  remark #15477: vector cost: 2.500
  remark #15478: estimated potential speedup: 3.900
  remark #15487: type converts: 1
  remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at saxpy.c(26,2)
  remark #15382: vectorization support: call to function printf(const char *__restrict__, ...) cannot be vectorized [ saxpy.c(27,4) ]
  remark #15344: loop was not vectorized: vector dependence prevents vectorization
LOOP END

LOOP BEGIN at saxpy.c(32,2)
  remark #15388: vectorization support: reference X[i] has aligned access [ saxpy.c(33,5) ]
  remark #15388: vectorization support: reference X[i] has aligned access [ saxpy.c(33,14) ]
  remark #15388: vectorization support: reference Y[i] has aligned access [ saxpy.c(33,21) ]
  remark #15305: vectorization support: vector length 4
  remark #15427: loop was completely unrolled
  remark #15399: vectorization support: unroll factor set to 8
  remark #15309: vectorization support: normalized vectorization overhead 0.047
  remark #15300: LOOP WAS VECTORIZED
  remark #15448: unmasked aligned unit stride loads: 2
  remark #15449: unmasked aligned unit stride stores: 1
  remark #15475: --- begin vector cost summary ---
  remark #15476: scalar cost: 8
  remark #15477: vector cost: 2.000
  remark #15478: estimated potential speedup: 3.820
  remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at saxpy.c(35,2)
  remark #15382: vectorization support: call to function printf(const char *__restrict__, ...) cannot be vectorized [ saxpy.c(36,4) ]
  remark #15344: loop was not vectorized: vector dependence prevents vectorization
LOOP END

```

Detecta las alineaciones de acceso a los vectores para prepararse para la vectorización. Desenrolla el vector y le asigna un factor para desenrollar el bucle y recorrerlo más rápidamente.

## **B)What loops are vectorized when a `-O2` option is used? And with the `-O3` option?**

Con la opción `-O2`:

missed: Sale un archivo vacío indicando que el compilador no tiene suficiente permiso como para observar ninguna optimización que no se haya hecho.

optimized: Nos sale un archivo vacío indicando que no se ha hecho ninguna optimización.

note: Nos sale un archivo vacío.

Con la opción `-O3`:

missed: Nos salen muchas instrucciones que no han podido ser ejecutadas en el 2o y 4to bucles for, debido a que el compilador no está seguro de si podrá llevar a cabo dichas optimizaciones manteniendo el resultado secuencial de la ejecución.

Optimized:

---

```

saxpy.c:32:2: note: loop vectorized
saxpy.c:20:2: note: loop vectorized

```

Nos salen las líneas de los vectores que han sido optimizados.

Note:

## Do icc and gcc vectorize the same loops?

Si, según los comandos observados y ejecutados con ambas versiones icc y gcc, ambos optimizan los bucles 1 y 3 y dejan sin vectorizar los bucles 2 y 4.

## Is this kind of loop vectorized? How is the operation performed according to the machine code generated?

### Linear loop: Si/Haciendo las iteraciones de 4 en 4.

```
[Intel(R) Advisor can now assist with vectorization and show optimization
report messages with your source code.
See "https://software.intel.com/en-us/intel-advisor-xe" for details.]

Intel(R) C Intel(R) 64 Compiler for applications running on Intel(R) 64, Version 19.0.5.281 Build 20190815

Compiler options: -O3 -qopt-report=5 -qopt-report-phase=vec -o linear

Begin optimization report for: main()

    Report from: Vector optimizations [vec]

LOOP BEGIN at linear.c(13,2)
    remark #15388: vectorization support: reference X[i] has aligned access    [ linear.c(14,4) ]
    remark #15385: vectorization support: vector length 4
    remark #15399: vectorization support: unroll factor set to 4
    remark #15380: LOOP WAS VECTORIZED
    remark #15449: unmasked aligned unit stride stores: 1
    remark #15475: --- begin vector cost summary ---
    remark #15476: scalar cost: 4
    remark #15477: vector cost: 1.000
    remark #15478: estimated potential speedup: 4.000
    remark #15487: type converts: 1
    remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at linear.c(18,2)
    remark #15382: vectorization support: call to function printf(const char * __restrict__, ...) cannot be vectorized    [ linear.c(19,4) ]
    remark #15344: loop was not vectorized: vector dependence prevents vectorization
LOOP END
=====
```

## Is this kind of loop vectorized? How is the operation performed according to the machine code generated?

### Reduction loop: Si/Haciendo las iteraciones de 8 en 8

```
[Intel(R) Advisor can now assist with vectorization and show optimization
report messages with your source code.
See "https://software.intel.com/en-us/intel-advisor-xe" for details.]

Intel(R) C Intel(R) 64 Compiler for applications running on Intel(R) 64, Version 19.0.5.281 Build 20190815

Compiler options: -O3 -qopt-report=5 -qopt-report-phase=vec -o reduction

Begin optimization report for: main()

    Report from: Vector optimizations [vec]

LOOP BEGIN at reduction.c(18,2)
    remark #15388: vectorization support: reference X[i] has aligned access    [ reduction.c(19,4) ]
    remark #15385: vectorization support: vector length 4
    remark #15399: vectorization support: unroll factor set to 4
    remark #15380: LOOP WAS VECTORIZED
    remark #15449: unmasked aligned unit stride stores: 1
    remark #15475: --- begin vector cost summary ---
    remark #15476: scalar cost: 7
    remark #15477: vector cost: 1.750
    remark #15478: estimated potential speedup: 4.000
    remark #15487: type converts: 2
    remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at reduction.c(23,2)
    remark #15382: vectorization support: call to function printf(const char * __restrict__, ...) cannot be vectorized    [ reduction.c(24,4) ]
    remark #15344: loop was not vectorized: vector dependence prevents vectorization
LOOP END

LOOP BEGIN at reduction.c(30,2)
    remark #15388: vectorization support: reference X[i] has aligned access    [ reduction.c(31,12) ]
    remark #15385: vectorization support: vector length 4
    remark #15399: vectorization support: unroll factor set to 8
    remark #15389: vectorization support: normalized vectorization overhead 0.775
    remark #15380: LOOP WAS VECTORIZED
    remark #15448: unmasked aligned unit stride loads: 1
    remark #15475: --- begin vector cost summary ---
    remark #15476: scalar cost: 5
    remark #15477: vector cost: 1.250
    remark #15478: estimated potential speedup: 3.810
    remark #15488: --- end vector cost summary ---
LOOP END
```

## Is this kind of loop vectorized? How is the operation performed according to the machine code generated?

Conditional loop: Si/Hace las iteraciones de 4 en 4.

```
Intel(R) Advisor can now assist with vectorization and show optimization
report messages with your source code.
See "https://software.intel.com/en-us/intel-advisor-xe" for details.

Intel(R) C Intel(R) 64 Compiler for applications running on Intel(R) 64, Version 19.0.5.281 Build 20190815

Compiler options: -O3 -qopt-report=5 -qopt-report-phase=vec -o conditional

Begin optimization report for: main()

    Report from: Vector optimizations [vec]

LOOP BEGIN at conditional.c(15,2)
  remark #15388: vectorization support: reference X[i] has aligned access    [ conditional.c(16,4) ]
  remark #15305: vectorization support: vector length 4
  remark #15399: vectorization support: unroll factor set to 4
  remark #15300: LOOP WAS VECTORIZED
  remark #15449: unmasked aligned unit stride stores: 1
  remark #15475: --- begin vector cost summary ---
  remark #15476: scalar cost: 4
  remark #15477: vector cost: 1.000
  remark #15478: estimated potential speedup: 4.000
  remark #15487: type converts: 1
  remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at conditional.c(17,2)
  remark #15527: loop was not vectorized: function call to rand(void) cannot be vectorized    [ conditional.c(18,11) ]
LOOP END

LOOP BEGIN at conditional.c(20,2)
  remark #15388: vectorization support: reference Z[i] has aligned access    [ conditional.c(22,3) ]
  remark #15388: vectorization support: reference X[i] has aligned access    [ conditional.c(22,10) ]
  remark #15305: vectorization support: vector length 4
  remark #15399: vectorization support: unroll factor set to 4
  remark #15300: LOOP WAS VECTORIZED
  remark #15448: unmasked aligned unit stride loads: 2
  remark #15449: unmasked aligned unit stride stores: 1
  remark #15475: --- begin vector cost summary ---
  remark #15476: scalar cost: 24
  remark #15477: vector cost: 2.500
  remark #15478: estimated potential speedup: 9.600
  remark #15488: --- end vector cost summary ---
LOOP END

LOOP BEGIN at conditional.c(26,2)
  remark #15382: vectorization support: call to function printf(const char *__restrict__, ...) cannot be vectorized    [ conditional.c
(27,4) ]
```

Which loop/s is vectorized? If a loop is not vectorized, what's the reason? How can you use the `#pragma vector` to override such vectorization inhibition (check the use of this pragma at the icc compiler documentation)?

Gather

<pre>for (i=0; i&lt;N; i++)     printf (" %4.2f ", X[i]);  printf("\n *****\n");  //Gathering for (i=0; i&lt;N; i++)     Y[i] = X[index [i]];</pre>	<pre>75     mov     eax, DWORD PTR i[rip] 76     cdqe 77     mov     eax, DWORD PTR index[0+rax*4] 78     mov     edx, DWORD PTR i[rip] 79     cdqe 80     movss   xmm0, DWORD PTR X[0+rax*4] 81     movsx   rax, edx 82     movss   DWORD PTR Y[0+rax*4], xmm0 83     mov     eax, DWORD PTR i[rip]</pre>
---	--

Scatter

<pre>for (i=0; i&lt;N; i++)     printf (" %4.2f ", Y[i]);  printf("\n *****\n");  //Scattering   for (i=0; i&lt;N; i++)     X[index [i]] = Y[i];</pre>	<pre>114     mov     ecx, DWORD PTR i[rip] 115     mov     eax, DWORD PTR i[rip] 116     cdqe 117     mov     edx, DWORD PTR index[0+rax*4] 118     movsx   rax, ecx 119     movss   xmm0, DWORD PTR Y[0+rax*4] 120     movsx   rax, edx 121     movss   DWORD PTR X[0+rax*4], xmm0 122     mov     eax, DWORD PTR i[rip] ---</pre>
--	---

Ambos están vectorizados.

Given the following loop and its corresponding OpenMP pragma, what are the compilation differences if the `safelen()` pragma is or is not present?



Cuando compilamos con “safelen()”, aseguramos al compilador poder hacer las ejecuciones de los “X” datos que esteen consecutivos en memoria para ahorrar tiempo. Ya que como programadores conocemos mejor cómo funciona el código.

Sin este pragma, el compilador no se atreve a hacer múltiples ejecuciones debido a la posibilidad de tardar más o cometer errores.

**Check how icc and gcc deal with this example if safelen() value is either 3 or 4. Look at the differences in the machine code generated by each compiler in each case. (explicacion)**

```

#define N 128

__attribute__((aligned(16))) float X[N];
__attribute__((aligned(16))) float Y[N];

int main ( int argc, char *argv[] ) {
    int i, z;

    if (argc == 2 )
        z = atoi(argv[1]);
    else
        exit(1);

    for (i=0; i<N; i++)
        X[i] = Y[i] = (i * 1.0);

    /* printf("Original Vector\n");

    for (i=0; i<N; i++)
        printf (" %4.2f ", X[i]);

    printf("\n *****");
*/

#pragma omp simd safelen(3)
    for ( i = z ; i < N ; i ++ )
        X[i] = X[i-z] + Y [i] ;

    /*
    for (i=0; i<N; i++)
        printf (" %4.2f ", X[i]);

    printf("\n *****");
    printf("\n z = %d \n", z);
*/
}

```

**Assembly for safelen(3):**

```

133 movsxd    rdx, edx                #33.2
134
135 ..B1.8:                                # LOE rax rdx rcx rbx rsi r13 r14 r15
136                                     # Preds ..B1.8 ..B1.7
137                                     # Execution count [1.15e+02]
138
139 ..L11:                                # optimization report
140                                     # LOOP WAS VECTORIZED
141                                     # SIMD LOOP
142                                     # VECTORIZATION HAS UNALIGNED MEMORY REFERENCES
143                                     # VECTORIZATION SPEEDUP COEFFICIENT 1.483398
144                                     # VECTOR TRIP COUNT IS ESTIMATED CONSTANT
145                                     # VECTOR LENGTH 2
146                                     # NORMALIZED VECTORIZATION OVERHEAD 0.750000
147                                     # DEPENDENCY ANALYSIS WAS IGNORED
148                                     # COST MODEL DECISION WAS IGNORED
149                                     # VECTOR LENGTH WAS OVERRIDDEN
150 .loc 1 34 is_stmt 1
151 movsd     xmm1, QWORD PTR [X+rbx*4]    #34.12
152 movsd     xmm0, QWORD PTR [Y+rcx+rbx*4] #34.21
153 addps     xmm1, xmm0                  #34.21
154 movlps    QWORD PTR [X+rcx+rbx*4], xmm1 #34.5
155 .loc 1 33 is_stmt 1
156 add       rbx, 2                      #33.2
157 cmp       rbx, rdx                    #33.2
158 jnb       ..B1.8                      #33.2
159
160 ..B1.10:                               # LOE rax rdx rcx rbx rsi r13 r14 r15
161                                     # Preds ..B1.8 ..B1.15
162                                     # Execution count [9.95e-01]
163 cmp       rdx, rsi                    #33.2
164 jae       ..B1.14                      #33.2
165
166 ..B1.14:                               # LOE rax rdx rsi r13 r14 r15
167                                     # Execution count [8.96e-01]
168 .loc 1 34 is_stmt 1

```

**Assembly for safelen(4):**

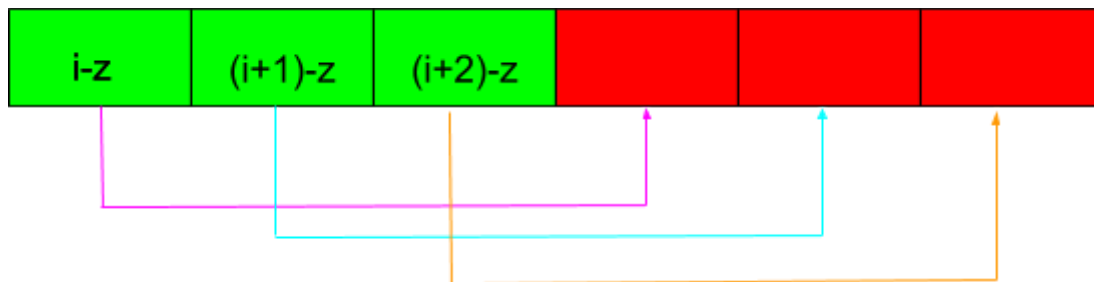
```

133 movsxd    rdx, edx                #33.2
134
135 ..B1.8:                                # LOE rax rdx rcx rbx rsi r13 r14 r15
136                                     # Preds ..B1.8 ..B1.7
137                                     # Execution count [1.15e+02]
138
139 ..L11:                                # optimization report
140                                     # LOOP WAS UNROLLED BY 2
141                                     # LOOP WAS VECTORIZED
142                                     # SIMD LOOP
143                                     # VECTORIZATION HAS UNALIGNED MEMORY REFERENCES
144                                     # VECTORIZATION SPEEDUP COEFFICIENT 2.931641
145                                     # VECTOR TRIP COUNT IS ESTIMATED CONSTANT
146                                     # VECTOR LENGTH 4
147                                     # NORMALIZED VECTORIZATION OVERHEAD 0.375000
148                                     # DEPENDENCY ANALYSIS WAS IGNORED
149                                     # COST MODEL DECISION WAS IGNORED
150                                     # VECTOR LENGTH WAS OVERRIDDEN
151 .loc 1 34 is_stmt 1
152 movups     xmm0, XMMWORD PTR [Y+rcx*4]
153 movups     xmm1, XMMWORD PTR [16+Y+rcx*4]
154 addps     xmm0, xmm1
155 movups     xmm0, XMMWORD PTR [X+rcx*4], xmm0
156 addps     xmm1, XMMWORD PTR [16+X+rbx*4]
157 .loc 1 33 is_stmt 1
158 add       rbx, 8
159 .loc 1 34 is_stmt 1
160 movups     xmm0, XMMWORD PTR [16+X+rcx*4], xmm0
161 .loc 1 33 is_stmt 1
162 add       rcx, 8
163 cmp       rbx, rdx
164 jnb       ..B1.8

```

Cuando el safelen es de 3, le expresamos al compilador que es seguro para el programa ejecutar 3 iteraciones del bucle de forma seguida ya que los datos que coja serán correctos.

En este caso, la diferencia entre  $i$  y  $z$  en el programa es de 3, así que nos aseguramos que los datos que cogemos anteriores, están correctamente calculados:



En este dibujo, expresamos que los datos que calculamos (**rojos**) se calculan en función de los datos ya calculados por iteraciones anteriores (**verdes**). De manera que al poner “safelen(4)” estaríamos intentando calcular un 4rto valor en base a una posición del array que aun no está calculada, y por ello, se generan más instrucciones cuando se ejecuta con “safelen(4)”.

## Omp simd private

Check the code generated by the compiler when this clause is used and compare it when no clause is present. Are there any differences? Look at the machine code and try to understand how the loop is computed.

## Clause open mp not used

## Clause open mp used

```

}

int main ( int argc, char *argv[]) {
    int i;
    float c;

    for (i=0; i<N; i++)
        X[i] = Y[i] = (i * 1.0);

    printf("Original Vector\n");

    for (i=0; i<N; i++)
        printf (" %4.2f ", X[i]);

    printf("\n *****\n");

#pragma omp simd private (x)
    for ( i = 0 ; i < N-1 ; i ++){
        x = X[i];
        Y[i] = foo (x * X[i+1]);
    }

    for (i=0; i<N; i++)
        printf (" %8.2f ", Y[i]);

    printf("\n *****\n");
}

```

```

A- 11010 /a.out .LX0: lib.f: .text // \s+ Intel Demangle
70 jmp .L8
71
72 .L9:
73 mov eax, DWORD PTR [rbp-8]
74 cdqe
75 movss xmm0, DWORD PTR X[0+rax*4]
76 cvttss2si eax, xmm0
77 mov eax, DWORD PTR [rbp-12], eax
78 cvtsi2ss xmm1, eax
79 mov eax, DWORD PTR [rbp-8]
80 add eax, 1
81 cdqe
82 movss xmm0, DWORD PTR X[0+rax*4]
83 mulss xmm0, xmm1
84 call foo(float)
85 movd edx, xmm0
86 mov eax, DWORD PTR [rbp-8]
87 cdqe
88 mov DWORD PTR Y[0+rax*4], edx
89 add DWORD PTR [rbp-8], 1
90
91 .L8:
92 cmp DWORD PTR [rbp-8], 127
93 jl .L9
94 cmp DWORD PTR [rbp-8], 127
95 jne .L10
96 mov eax, DWORD PTR [rbp-8]
97 mov DWORD PTR [rbp-4], eax
98
99 .L10:
100 mov DWORD PTR [rbp-4], 0

```

La primera versión (sin pragma), para una ejecución del programa, la variable X es **global**, lo que comporta que para varias ejecuciones del bucle simultáneas, la X tome valores incorrectos ya que todas las iteraciones escriben su valor a leer en la misma variable. Esto dará lugar a resultados incorrectos en la ejecución.

Con el pragma, conseguimos asegurar que cada iteración tenga su propia variable x y solo pueda ser modificada por la misma iteración. Lo que dará lugar a valores correctos.

## Function vectorization

The compiler has applied another optimization to enable vectorization. Which one? Look at the machine code generated by the compiler to see the optimization that has been applied (you might run the application using perf record and perf report to see the machine code). Measure the execution time and the number of executed instructions with perf stat.

```

12
13 void saxpy (float *X, float *Y, int i, float A){
14     X[i] = A * X[i] + Y[i];
15 }
16
17
18 int main () {
19     int i;
20
21     for (i=0; i<N; i++)
22         X[i] = Y[i] = (i * 1.0);
23
24     /*
25     printf("Original Vector\n");
26
27     for (i=0; i<N; i++)
28         printf (" %4.2f ", X[i]);
29
30     printf("\n *****\n");
31     */
32
33     for (i=0; i<N; i++)
34         saxpy( X, Y, i, A);
35
36     /*
37     for (i=0; i<N; i++)
38         printf (" %4.2f ", X[i]);
39
40     printf("\n *****\n");
41     */
42 }
43

```

```

x86-64 gcc 8.2 -fopenmp
A- 11010 /a.out .LX0: lib.f: .text // \s+ Intel Demangle
44 .L4:
45 cmp DWORD PTR [rbp-4], 99999999
46 jg .L3
47 cvtsi2sd xmm0, DWORD PTR [rbp-4]
48 cvtsd2ss xmm0, xmm0
49 mov eax, DWORD PTR [rbp-4]
50 cdqe
51 movss DWORD PTR Y[0+rax*4], xmm0
52 mov eax, DWORD PTR [rbp-4]
53 cdqe
54 movss xmm0, DWORD PTR Y[0+rax*4]
55 mov eax, DWORD PTR [rbp-4]
56 cdqe
57 movss DWORD PTR X[0+rax*4], xmm0
58 add DWORD PTR [rbp-4], 1
59 jmp .L4
60
61 .L3:
62 mov DWORD PTR [rbp-4], 0
63
64 .L6:
65 cmp DWORD PTR [rbp-4], 99999999
66 jg .L5
67 movss xmm0, DWORD PTR A[rip]
68 mov eax, DWORD PTR [rbp-4]
69 mov edx, eax
70 mov esi, OFFSET FLAT:Y
71 mov edi, OFFSET FLAT:X
72 call saxpy(float*, float*, int, float)
73 add DWORD PTR [rbp-4], 1
74 jmp .L6

```



```

Performance counter stats for './saxpy':

    0,53 msec task-clock                #    0,348 CPUs utilized
         11 context-switches           #    0,021 M/sec
          0 cpu-migrations              #    0,000 K/sec
        138 page-faults                #    0,258 M/sec
    1.697.036 cycles                    #    3,175 GHz
    1.163.358 stalled-cycles-frontend  #   68,55% frontend cycles idle
    983.972 stalled-cycles-backend     #   57,98% backend cycles idle
    1.142.371 instructions              #    0,67 insn per cycle
                                           #    1,02 stalled cycles per insn
        223.029 branches                #   417,330 M/sec
          8.022 branch-misses           #    3,60% of all branches

0,001537843 seconds time elapsed

0,000000000 seconds user
0,000968000 seconds sys

```

El código ha efectuado las operaciones de 4 en 4 (tal y como se ve en los registros cuando suma/resta 4 a los valores de *i* con los que itera el *for*), esto se debe a que el compilador ha hecho la función *saxpy* en la misma línea (*inlining*).

**Modify the code to avoid such optimization (intel and GNU compilers use a different mechanism; look for both and add the appropriate changes in the code to prevent function *saxpy* from being vectorized).**

Añadiendo la directiva `#pragma omp declare simd uniform()` `linear()` antes de la realización del bucle *for* de la función *main*, conseguimos que el programa haga la vectorización adecuadamente.