Arquitecturas Avanzadas

Grupo AA-2-1

25/10/2019

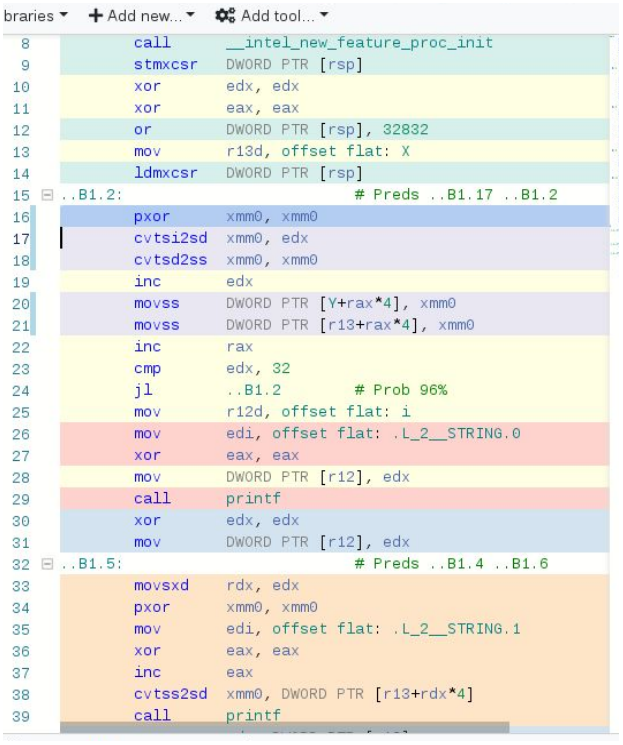
Julio César Velasquez Cardenas 1397896
Sergio Prada Maeso 1459122
Juan Carlos Bermúdez Rodríguez 1455486

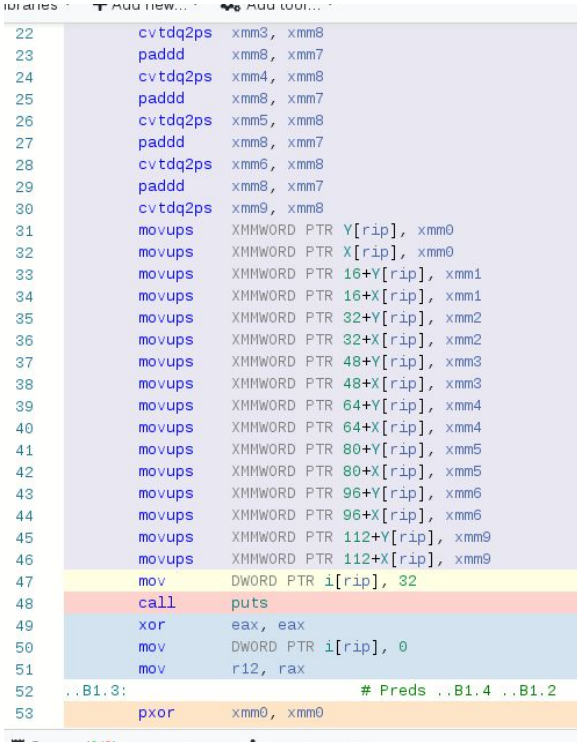
A) Which loops have been vectorized and which have not? Check the source file to answer.

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).





-O1 -O3

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 4rto 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 #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 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 #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: 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 #15305: vectorization support: vector length 4
   remark #15399: vectorization support: unroll factor set to 8
   remark #15309: vectorization support: normalized vectorization overhead 0.775
   remark #15300: 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

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

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

//Gathering

for (i=0; i<N; i++)
   Y[i] = X[index [i]];
```

```
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]
```

Scatter

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

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

//Scattering

for (i=0; i<N; i++)
   X[index [i]]= Y[i];
```

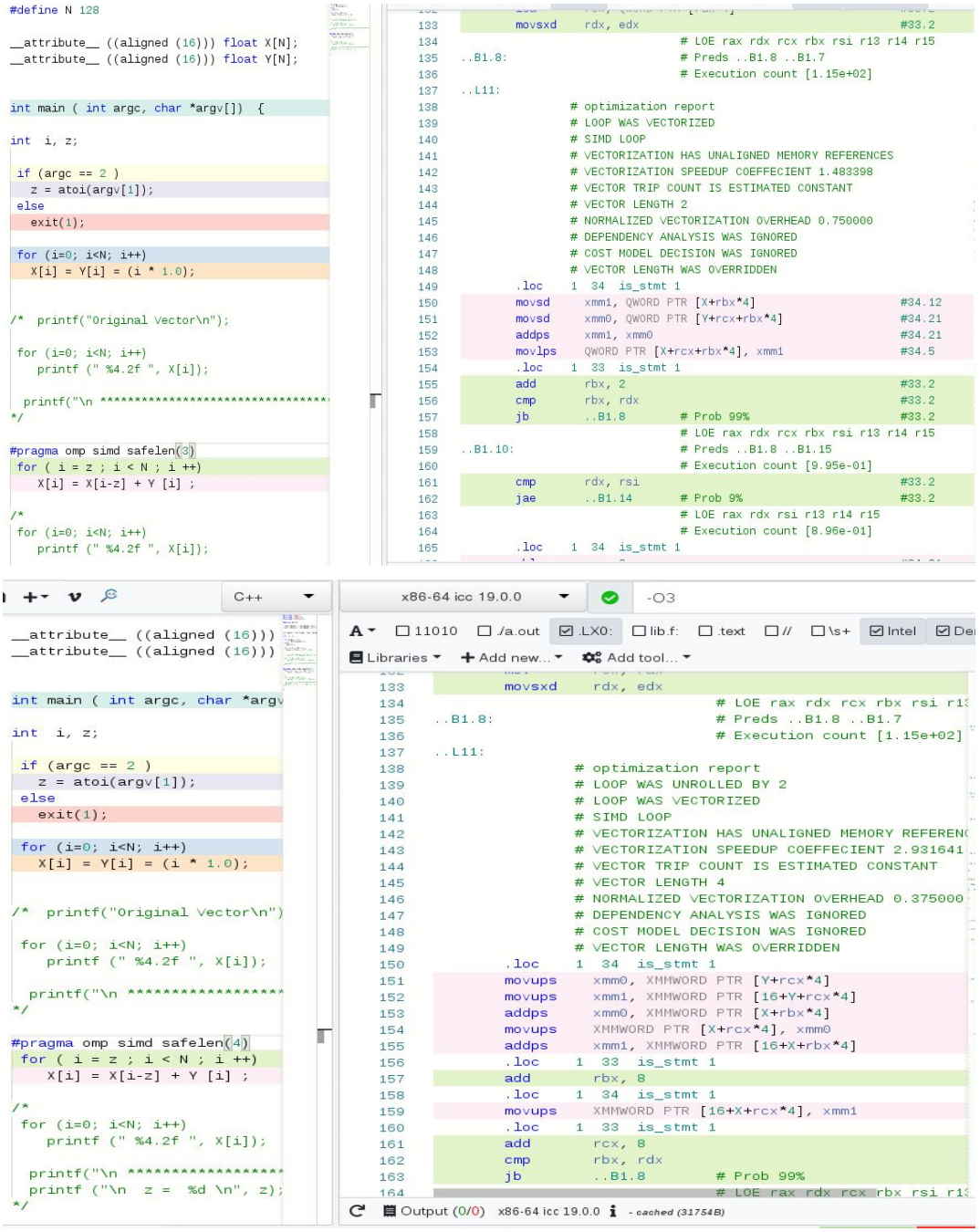
```
114   mov     ecx, DWORD PTR i[rip]
115   mov     eax, DWORD PTR i[rip]
116   cdqe
117   mov     edx, DWORD PTR index[0+rax*4]
118   movsx   rax, ecx
119   movss   xmm0, DWORD PTR Y[0+rax*4]
120   movsx   rax, edx
121   movss   DWORD PTR X[0+rax*4], xmm0
122   mov     eax, DWORD PTR i[rip]
```

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?

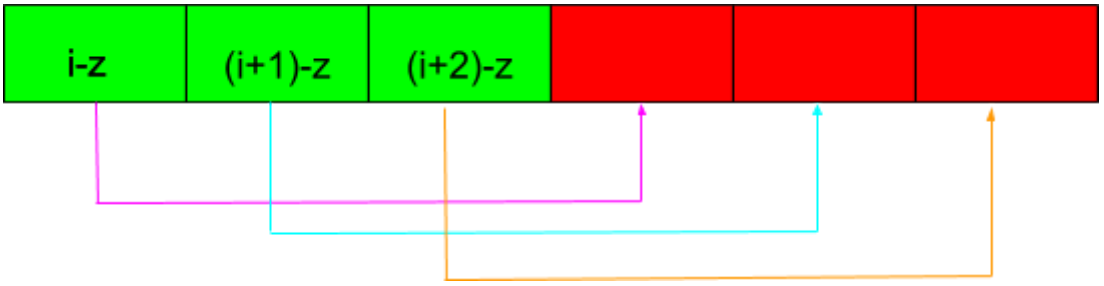
Cuando compilamos con “safelen()”, aseguramos al compilador poder hacer las ejecuciones de los “X” datos que esteen consecutivos en memoria para ahorrar tiempo. Ya que como programadores conocemos mejor cómo funciona el código.
Sin este pragma, el compilador no se atreve a hacer múltiples ejecuciones 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)



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:

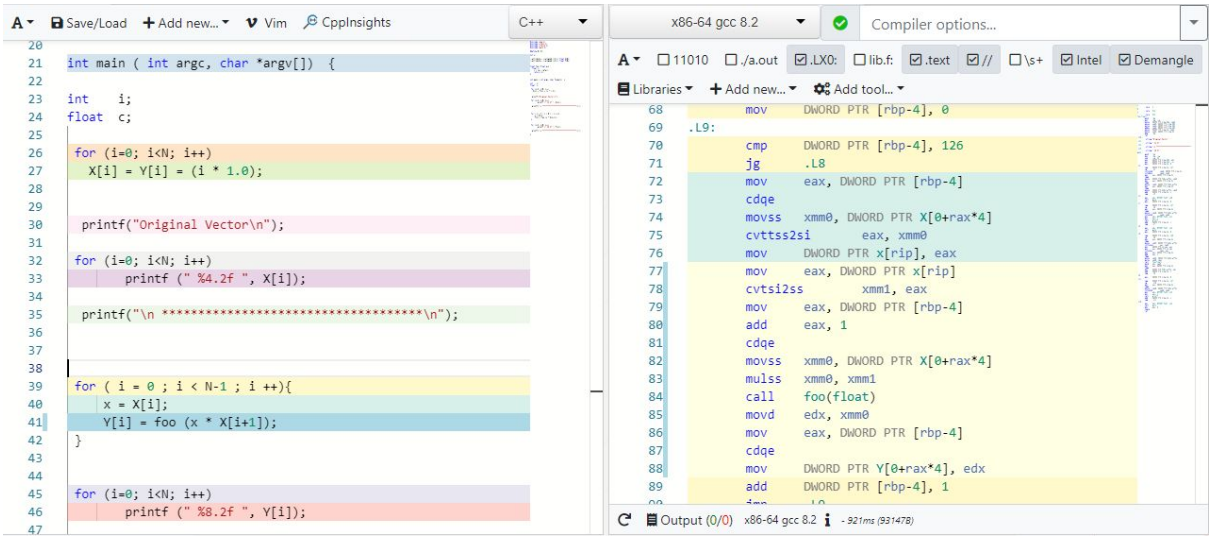


En este dibujo, expresamos que los datos que calculamos (rojos) se calculan en función de los datos ya calculados por iteraciones anteriores (verdes). De manera que al poner “safelen(4)” estaríamos intentando calcular un 4rto valor en base a una posición del array que aun no está calculada, y por ello,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

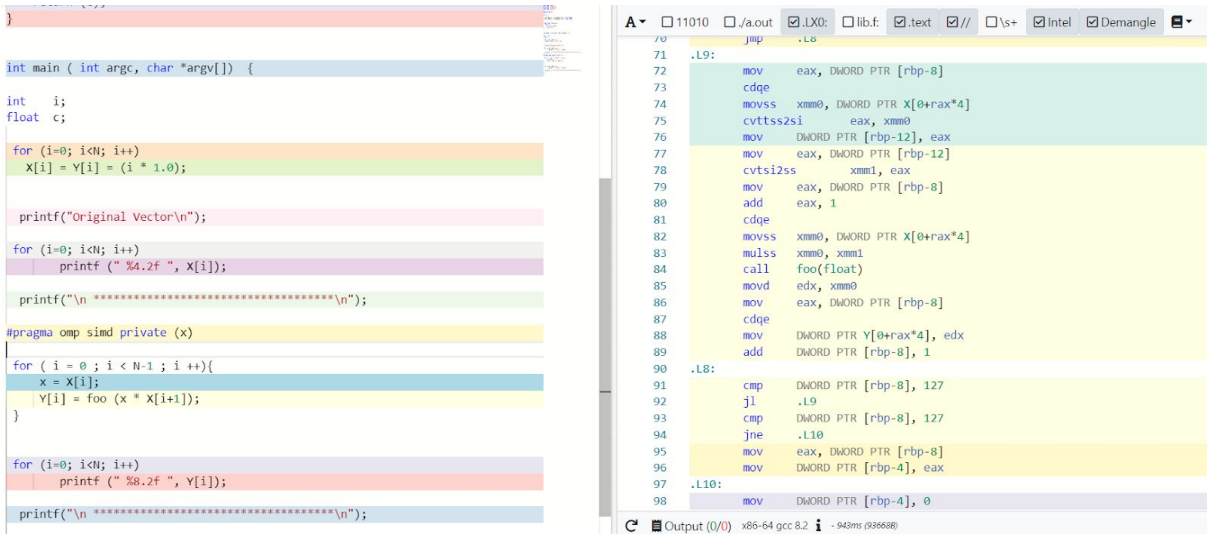
Omp simd private

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



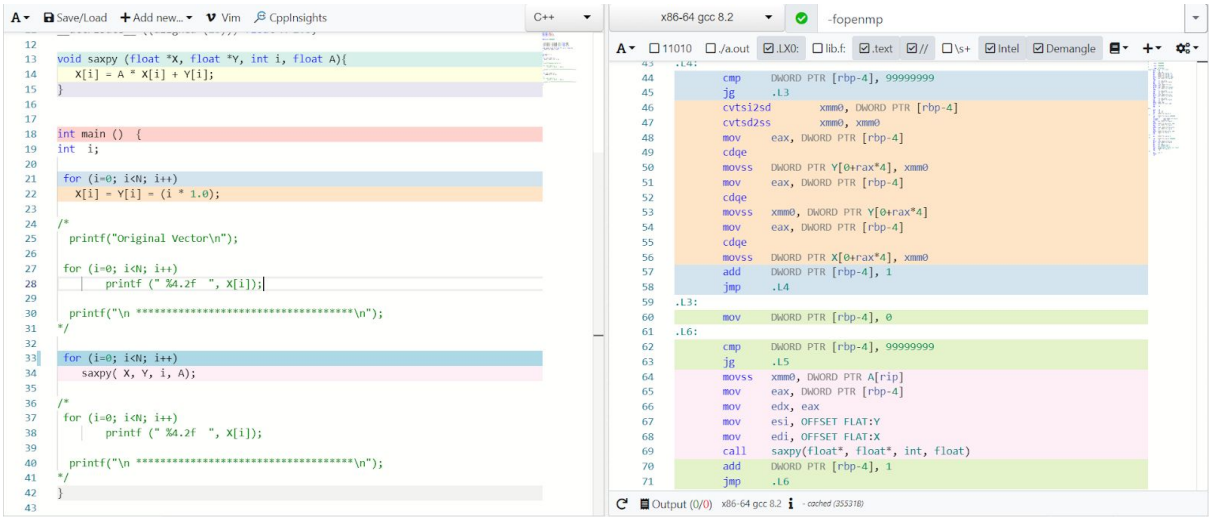
Clause open mp used

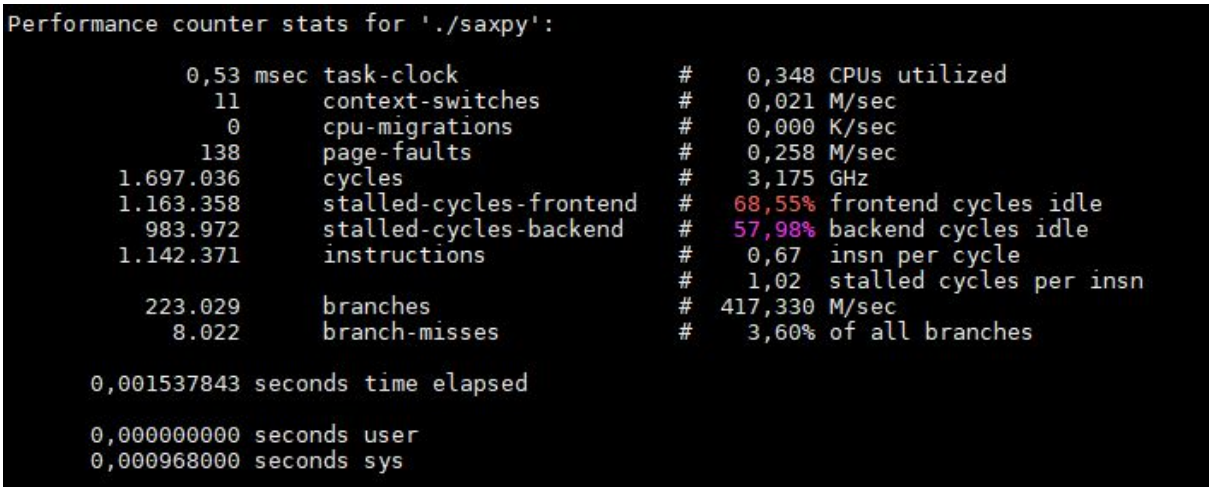


La primera versión (sin pragma), para una ejecución del programa, la variable X es **global,**lo que comporta que para varias ejecuciones del bucle simultáneas, la X tome valores incorrectos ya que todas las iteraciones escriben su valor a leer en la misma variable. Esto dará lugar a resultados incorrectos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





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.