Vectorization and SIMD extensions in OpenMP

1. Generating an Optimization Report with intel (icc) and GNU (gcc) compilers

To compile all the examples in this exercise you should setup first the right environment variables for each compiler (gcc or icc). Do it by issuing the corresponding command:

module load gcc/8.2.0 or *module load intel/19.0.0*

A – intel optimization reports

An optimization report shows what optimizations were applied by the compiler in your code and explains why some optimizations were not applied. In this set of exercises, we are mostly interested in code vectorization but the same compiler options can be used to check other optimizations.

To generate an optimization report in icc, use the `qopt-report-phase=[list]` compiler option together with `qopt-report[=n]`[1].

qopt-report-phase=[list] Specifies one or more optimizer phases for which optimization reports are generated. For instance, cg (the phase for code generation), loop (lfor loop nest optimization), vec (the phase for vectorization),...

qopt-report[=n] indicates the level of detail in the report. You can specify values 0 (no report is generated) through 5(greatest level of detail). Default level is 2, which produces a medium level of detail.

To generate a vectorization report, use the `qopt-report-phase=vec` compiler options together with `qopt-report=1` or `qopt-report=2`.

`qopt-report=2` generates a report with the loops in your code that were vectorized and explains why other loops were not vectorized.

Because some optimizations (such as vectorization) are turned off with the `O1` option, the compiler does not generate a report for them. Therefore, to generate a vectorization report, compile your project with the `O2` or `O3` options.

[1] For more information on the `qopt-report` and `qopt-report-phase` compiler options, see the *Compiler Options* section in the *Intel® C++ Compiler Developer Guide and Reference*.

Example of use

Our first example will be a simple SAXPY loop. The SAXPY loop performs a single precision multiplication on X and adds it to Y.

```
float X[SIZE], Y[SIZE], A;
for (int i=0; i<N; i++)
  X[i] = A*X[i] + Y[i];
```

You could find a complete program at the same directory that holds other examples used in this set of exercises. We can compile the program with this command (make sure you have already issued a *module load intel/19.0.0* command to set the appropriate environment variables):

```
icc -O3  -qopt-report=2 -qopt-report-phase=vec saxpy.c -o saxpy
```

And we get a saxpy.optrpt file with optimization information:

```
Begin optimization report for: main()

    Report from: Vector optimizations [vec]

LOOP BEGIN at saxpy.c(20,2)

   remark #15300: LOOP WAS VECTORIZED

LOOP END

LOOP BEGIN at saxpy.c(26,2)

  remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown below. Use level 5 report for details

LOOP END

LOOP BEGIN at saxpy.c(31,2)

   remark #15300: LOOP WAS VECTORIZED

LOOP END

LOOP BEGIN at saxpy.c(34,2)

   remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown below. Use level 5 report for details

LOOP END
```

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

- 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).

- 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.

B – GNU optimization reports

The GNU compiler has also a mechanism to report optimizations that have been applied to a program. The basic mechanism to generate this information is the following option that has to be added to the compilation command:

- *fopt-info-options = filename*

In this case, *options* is a list of '-' separated option keywords to select the dump details and optimizations. Information will be dumped to a file (*filename*). The options can be divided into three groups:

- options describing what kinds of messages should be emitted,
- options describing the verbosity of the dump, and
- options describing which optimizations should be included.

As we are primarily interested in vectorization optimization[2], some useful combinations are the following (try them with the saxpy.c program and check the output generated):

gcc -O3 -fopt-info (basic command, information is written to *stderr*)

*gcc -O3 -fopt-info-missed-vec=missed.txt (*information about missed optimizations)

*gcc -O3 -fopt-info-optimized-vec=optimized.txt (*information about missed optimizations)

*gcc -O3 -fopt-info-note-vec=note.txt (*verbose information about optimizations, transformations,…)

gcc -O3 -fopt-info-all-vec=all.txt (combines the three previous options)

Compile the *saxpy* example with gcc and the above mentioned options and look at the optimization information reported by the compiler.

What loops are vectorized when a -O2 option is used? And with the -O3 option?

Do icc and gcc vectorize the same loops?

[2] For more information on the fopt-info options, see the GCC Developer Options of the GNU gcc Compiler (https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html#Developer-Options)

2. Basic loops

In the following examples, first think from a theoretical point of view whether the loops are vectorizable or not and then analyze how they are processed by the compiler.

1.- Linear loop

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

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

2.- Reduction loop

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

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

3.- Conditional loop

```
for (i=0; i<N; i++)
    if (X[i] > Y[i])
        Z[i] = X[i];
   else
        Z[i] = 0;
```

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

4.- Gathering and scattering operations

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

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)?

> **Note:** in modern processors, the AVX2 instruction set introduced a gather instruction to solve the performance problems that inhibits vectorization (*vpgatherdd / vpgatherqd*) for performance reasons.

3. OpenMP SIMD

As seen in the previous examples, compilers have auto-vectorization optimization passes that attempt to make use of the SIMD instructions but will often fail to vectorize code or cannot safely vectorize the code without additional information from the programmer. OpenMP is a directive-based programming interface for shared memory and accelerator based systems. In OpenMP 4.0, SIMD directives were added to help compilers generate efficient vector code. The SIMD directives explicitly enable vectorization in the compiler and act as hints or instructions sent to the compiler's vectorization pass to improve its analysis and the quality of the vector code being generated. These directives also help to scope which SIMD statements in the code the compiler should attempt to vectorize. OpenMP's SIMD directives can be placed before a for loop or a function declaration.

a) SIMD Constructs for Loops

The basis of OpenMP 4.0 SIMD extensions is the *simd* construct. This new construct instructs the compiler on vectorizing the loop. The SIMD construct can be applied to a loop to indicate that the iterations of the loop can be divided into contiguous chunks of a specific length and, for each chunk, multiple iterations can be executed with multiple SIMD lanes concurrently within the chunk while preserving all data dependencies of the original serial program and its execution. The syntax of the simd construct is as follows:

```
#pragma omp simd [clause[[,] clause] ...] new-line
        for-loops
```

Clauses are optional and provide additional information to the compiler. Existing clauses are: *safelen(length), simdlen(length), linear(list[: linear-step]), aligned(list[: alignment]), private(list), lastprivate(list), reduction(reduction-identifier : list),*

collapse(n). Below you will find some examples that illustrate the usage of some of these clauses. For the rest, refer to the OpenMP complete specification.

a.1) omp simd safelen()

Loop carried data dependencies hinder parallel programming. If an iteration of the loop requires data that is only calculated in a previous iteration of the loop, it has a data dependency. Because vectorized loops perform multiple iterations of the loop concurrently, the value will change if a data dependency is present in the vectorized loop. The safelen clause guarantees that no iterations of the loop will execute simultaneously inside the SIMD loop if the distance between the iterations is smaller than the value specified in the safelen clause. The actual number of loop iterations that will be performed in a single iteration of the SIMD loop is implementation defined, but it will not exceed the value specified in the safelen clause.

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

If the pragma is present as shown in the code below, what values are valid for the variable z (assume that z is only known at execution time)?

```
#pragma omp simd safelen ( 4 )
float X[N], Y [N] ;
  for ( int i = z ; i < N ; i ++) {
     X[ i ] = X[ i – z ] +Y [ i ] ;
  }
```

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.

a.2) omp simd private

Data sharing clauses can be used to break false data dependencies that may hinder SIMD execution. The private clause guarantees that x is private to each SIMD chunk of the loop. This indicates that the values in x will not cross between the partitioned chunks of the loop. Otherwise, the compiler may think that the variable x contains a write-write or write-read conflict. The private clause is used to remove this false dependency.

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

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.

b) Function vectorization

Beyond the *simd* construct for loops, OpenMP 4.0 introduces the *declare simd* construct, which can be applied to a function to enable the creation of one or more versions that can process multiple instances of each argument usinrm g SIMD instructions from a single invocation from a SIMD loop. The syntax of the declare simd construct is as follows:

#pragma omp declare simd [clause[[,] clause] ...] new-line

[#pragma omp declare simd [clause[[,] clause] ...] new-line]

function definition or declaration

where clause is one of the following: *simdlen(length), linear(linear-list[: linear-step]), aligned(argument-list[: alignment]), uniform(argument-list), inbranch, notinbranch*.

We will illustrate the usage of some function vectorization clauses with the following example that shows a SAXPY function that is called from a main loop. Compile the application and check if the loop that calls the SAXPY function is vectorized or not.

```
void saxpy (float *X, float *Y, int i, float A){

  X[i] = A * X[i] + Y[i];

}

int main () {
   float  X[SIZE], Y[SIZE], A;
   for (int i=0; i<N; i++)
     saxpy( X, Y, I, A);
 }
```

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*.

The answer to the previous question is inlining; the compiler has first inlined the function saxpy and afterwards it has vectorized the loop.

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.

Additionally, a **#pragma omp declare simd uniform () linear ()** might be add before the function definition.

> The ***uniform([argument],…)*** clause indicates that the given function argument will not change between any of the concurrent function calls in a SIMD loop (i.e. the value is shared between the SIMD lanes of the loop).
>
> The ***linear([argument]:linearstep, …)*** clause indicates what the value of the function argument will be if the function is called multiple times concurrently. The argument placed in the ***linear*** clause will be increased by the linear step value between each successive function call.

Add the two *omp simd* directives, with the appropriate arguments and verify that now the loop and the function are properly vectorized. Try it out with both *icc* and *gcc*.

Run the application with *perf stat* and measure the execution time and the number of instructions executed. Compare these values with the ones obtained when the function was inlined and vectorized. Are all the results coherent? If not, which cases provide substantially different results than expected? Why?

c) Memory alignment

Memory alignment is an important consideration for writing high performance code. Memory alignment refers to the divisibility of the memory location of a value stored in memory. If a variable's memory address is divisible by n, it is said to be aligned by n or on a n byte boundary. So, for example, if a variable x was stored at the memory address 0x37FD10, it would be aligned on a 1, 2, 4, 8, and 16 byte boundary.

Each data type should ideally be stored at a memory location aligned by the smallest power of two that can contain the size of the data type. These values are given in Table 1.

Data Type	Alignment
char	1
short	2
int	4
long	8
float	4
double	8
long double	16
XMM register	16
YMM register	32
ZMM register	64

Table 1. Memory alignment for each data type

Accessing unaligned memory is slower because of how memory is read from the cache. A cache is organized as a set of cache lines that store a set of continuous memory. The x86 processor can read or write to unaligned memory addresses without any penalty if it is within the same cache line. However, if the memory access

crosses this cache line boundary, the processor is actually accessing the memory from two cache lines along with a shifting operation to combine the two accesses. When a cache line is split by a memory access it is usually about three times slower than a standard memory access.

Memory alignment is especially important when accessing memory with SIMD instructions. SIMD instructions can read or write a large amount of data in a single instruction. If the base memory address is unaligned, then a greater number of memory accesses will split a cache line. When the vector register becomes larger, the ratio of standard reads to cache line splits decreases and the penalty becomes more severe.

For example, if data was read sequentially from a misaligned memory address with a 64 byte cache line size, specify how many reads there would be for a cache line split in the case of

- A XMM register.
- A YMM register,
- A ZMM register

If a cache line split is three times slower than an access contained in the cache line, what average penalty will be observed in each of the above cases.

The compiler will automatically align most data known at compile time on a 32 or 16 byte boundary. Dynamically allocated memory is not guaranteed to return aligned addresses so they should be aligned by the memory allocator or the programmer. Some compilers offer the ability to specify memory alignment as well as dynamic memory allocation that returns aligned pointers.

The compiler often cannot determine the alignment properties of data that is linked from other files or when they are dynamically allocated. The aligned clause asserts to the compiler that a variable is aligned. Each pointer in the aligned clause can have a positive integer alignment applied to it. If no alignment value is given to the compiler, an implementation defined default value is assumed. Using this clause allows the compiler to safely use SIMD instructions that have strict alignment requirements. If this clause is used, the programmer is responsible for ensuring that the data is in fact aligned

#pragma omp simd aligned([ptr] : [alignment], . . .)

In the following example, could X and Y start at the memory address 0x37FD10? If not, which is the smallest address they could start at.

```
float X[ SIZE ] , Y[ SIZE ] , A;
#pragma omp simd aligned (X : 32 , Y : 32 )
for ( int i = 0 ; i < SIZE ; i ++){
   X[ i ] = A∗X[ i ] + Y[ i ] ;
 }
```

d) Split compilation

The last example illustrates how to apply vectorization with OpenMP directives when the application is splitted in different files. We have a library file (saxpy_lib.c and saxpy_lib.h) that contains the *saxpy* function and a main program (saxpy_main.c) that performs array initializations and calls function *saxpy*.

Add the necessary directives to generate an executable where saxpy operations are vectorized.

Basic steps to compile and link both files (applies to gcc and icc):

gcc -c saxpy_lib.c (this command generates a saxpy_lib.o object)

gcc -c sacpy_main.c (this command generates a saxpy_main.o object)

gcc -o saxpy_executable saxpy_lib.o saxpy_main.o (both objects a linked and saxpy_executable is generated)

You should add all the extra options for compilation (optimization level, openmp usage, vectorization reporting, …), as needed.