# How to Improve Code Vectorization

The Haswell Galileo  CPUs in CINECA compute nodes and Broadwell CPUs in the Marconi nodes provide 256-bit vector registers and AVX/AVX2 (Advanced Vector Extensions) instruction sets. The Xeon Phi CPUs in the KNL nodes provide 512-bit vector registers and AVX-512 instruction sets. To get the most performance out of these processors, users need to take advantage of these strengths and try to improve the usage of vectorization instructions in their code. This document provides a guideline on how to get vectorization information and improve code vectorization.

## Get Vectorization Information

There are various ways to get information regarding how a code is vectorized. The following information is for Intel compilers. For GCC compilers please refer to the corresponding man page or documentation.

**1.The compiler option '-S' can be used to generate assembly code instead of binary. In the assembly code, SSE vector instructions generally operate on xmm registers, AVX and AVX2 on ymm registers, and AVX512 on zmm registers. AVX, AVX2, and AVX512 instructions are prefixed with 'v'.**

Example of compiling a C++ program to generate assembly code instead of binary code:

```
icpc -S -o vec_add.s vec_add.cc
```

Example of generated vectorized assembly code:

```
vldmxcsr  64(%
rsp)
vaddpd    (%rsp), %zmm0, %
zmm1
vmovupd   %zmm1, (%rsp)
```

**2.Compiler option '-qopt-report=5' can be used to generate an optimization report, which contains vectorization information. To generate report for vectorization only, use "-qopt-report -qopt-report-phase=vec".**

Compile and link Fortran/C/C++ program using corresponding Intel compiler with the vectorization option:

Example:

```
module load intel
icc -g -qopenmp -O2 example.c –o example
```

Some Intel compiler options are listed below:

| | |
|---|---|
| -g | Build application with debug information to allow binary-to-source correlation in the reports. |
| -qopenmp | Enable generation of multi-threaded code if OpenMP directives/pragmas exist. |
| -O2 (or higher) | Request compiler optimization. |
| -vec | Enable vectorization if option O2 or higher is in effect (enabled by default). |
| -simd | Enable SIMD directives/pragmas (enabled by default). |

For details of these options refer to man page or documentations of Intel compilers.

# Explicit Vectorization

### Compiler SIMD directives/pragmas

Users can add compiler SIMD directives/pragmas to the source code to tell the compiler that dependency does not exist, so that the compiler can vectorize the loop when the user re-compiles the modified source code. Such SIMD directives/pragmas include:

```
#pragma vector always: instruct to vectorize a loop if it is safe to do so
#pragma vector align: assert that data within the loop is aligned on 16B boundary
#pragma ivdep: instruct the compiler to ignore potential data dependencies
#pragma simd: enforce vectorization of a loop
```

### OpenMP directives/pragmas

Users can use OpenMP 4.0 new directives/pragmas for explicit vectorization:

```
#pragma omp simd: enforce vectorization of a loop
#pragma omp declare simd: instruct the compiler to vectorize a function
#pragma omp parallel for simd: target same loop for threading and SIMD, with each thread executing SIMD
instructions
```

### SIMD enabled functions

Users can also declare and use SIMD enabled functions. In the example below, function foo is declared as a SIMD enabled function (vector function), so it is vectorized. So is the for loop in which it is called.

```
__attribute(vector)
float foo(float);
void vfoo(float *restrict a, float *restrict b, int n)
{
    int i;
    for (i=0; i<n; i++) { a[i] = foo(b[i]); }
}
float foo(float x) { ... }
```

# Programming Guidelines for Writing Vectorizable Code

- Use simple loops, avoid variant upper iteration limit and data-dependent loop exit conditions
- Write straight-line code: avoid branches, most function calls or *if* constructs
- Use array notations instead of pointers
- Use unit stride (increment 1 for each iteration) in inner loops
- Use aligned data layout (memory addresses)
- Use structure of arrays instead of arrays of structures
- Use only assignment statements in the innermost loops
- Avoid data dependencies between loop iterations, such as read-after-write, write-after-read, write-after-write
- Avoid indirect addressing
- Avoid mixing vectorizable types in the same loop
- Avoid functions calls in innermost loop, except math library calls