|  |  |
| --- | --- |
|  |  |

The Ministry of Education and Science of the Russian Federation

Lobachevsky State University of Nizhni Novgorod

Computing Mathematics and Cybernetics faculty

The competitiveness enhancement program   
of the Lobachevsky State University of Nizhni Novgorod   
among the world's research and education centers

Strategic initiative  
“Achieving leading positions in the field of supercomputer technology   
and high-performance computing”

**PROGRAMMING AND OPTIMIZATION   
FOR INTEL XEON PHI**

*Practice 5. Optimization of applications for Intel Xeon Phi.   
Using Intel C/C++ Compiler. Vectorization.*

Nizhni Novgorod

2014

# Objectives

The objective of this practice is to study basic techniques of code vectorization on Intel Xeon Phi. This objective includes the following activities:

1. To study different techniques of code vectorization on Intel Xeon Phi.
2. To master diagnostic tools and mechanisms of vectorization on a simple example.
3. To study ways of vectorization of mathematical routines.
4. To consider more sophisticated cases arising in applications.

# Abstract

This practice is devoted to mastering vectorization in C code using Intel C/C++ Compiler. We consider basic topics of vectorization and illustrate the main ways to use vectorization on Intel Xeon Phi coprocessor. We use a simple example that seems to naturally allow vectorization, which is, however, not performed by the compiler because of potential vector dependencies. Verious vectorization techniques are demonstrated on this example: restrict keyword and #pragma ivdep to guarantee there are no potential dependencies, explicit vectorization using #pragma simd, Array notation and Elemental functions. We consider vectorization of loops that include mathematical routines. Finally more sophisticated cases are discussed. We do not aim to fully investigate all or some vectorization techniques, nor particularly hard examples. On the contrary, we provide an overview of various vectorization techniques illustrated on simple examples and concentrate on general ideas of vectorization. The theoretical material is given in the previous lecture, some intricate cases are considered in the next lecture.

# BRIEF OVERVIEW

In the introductory segment we briefly review the material of the previous lecture. The review includes general concept of vectorization, historical development of vector extensions on CPU, main features of vector extensions on Xeon Phi. We stress that vectorization is equally important for all coprocessor programming modes (offload, symmetric, coprocessor-only); in all examples of this lecture we use native mode only. It is important to use two level of parallelism on Xeon Phi: core level and vector level. There two levels are somewhat independent, which allows us to separately consider vectorization in this practice while using single-threaded examples. However, all techniques and ideas are equally relevant for multithreaded programs.

One of the main ways of vectorization in C/C++ and Fortran programs on both CPUs and Intel Xeon Phi coprocessors is loop vectorization performed by the compiler. Loop vectorization is vectorized execution of several loop iterations in parallel. Naturally, not all loops can be vectorized. The compiler checks if vectorization is possible and seems efficient and, in case both conditions are met, generates vectorized code. A software developer can provide additional information or give additional guarantees that may influence compiler decision about vectorization of the loop.

We start from a simple example that performs some arithmetic operations with 4 arrays in a loop.

void vectorization\_simple(float\* a, float\* b, float\* c,  
 float\* d, int n)

{

for (int i = 0; i < n; i++)

{

a[i] = b[i] \* c[i];

c[i] = a[i] + b[i] - d[i];

}

}

Loop iterations are independent so the loop can be vectorized in case arrays **a**, **b**, **c**, **d** are not overlapping. First we check whether Intel compiler performs vectorization of the loop. We write a function main that calls the given function and build the application with **-vec-report3** compiler flag that returns a report on vectorization (report level 3 corresponds to medium level of details in the report; the latest version of the compiler support levels 6 and 7 that additionally give recommendations on vectorization). The report shows that the loop was not vectorized because of data dependencies. We investigate the reason of reported data dependencies, whether they are real, and the source of the dependencies. We explain that the compiler is forbidden to perform an incorrect vectorization of the loop that might be not equivalent to the non-vectorized version. Thus, the compiler has to be very conservative and any potential data dependency prevents vectorization, unless additional information is provided to the compiler. The function in our example does not provide information about memory location of addresses pointed by **a**, **b**, **c**, **d**. The compiler has to take into account all pointers can overlap in memory. However, in case a software developer knows that there can be no overlapping or any other data dependency, there are ways to point in out to the compiler.

Then we study three ways of providing additional information to the compiler: restrict keyword in function parameters (**a, b, c, d**), compiler directives **#pragma ivdep** and **#pragma simd** before the loop. In the first case we guarantee that the pointers declared with restrict keyword do not overlap, thus no restricted region of memory can be accessed via another pointer. In the second case we guarantee there are no potential data dependencies in the following loop. This option basically orders the compiler to ignore potential data dependencies; however, if the compiler finds a proven dependency, the loop will not be vectorized anyway. The combination of **#pragma ivdep** and **#pragma vector always** is a fairly standard way to assist vectorization performed by the compiler. Although quite powerful, this is still an implicit way to vectorize a loop. A principally different alternative is to use **#pragma simd** compiler directive which is an explicit order to vectorize the loop. It gives a software developer much higher level of control of vectorization compared to **#pragma ivdep** that can be useful, for instance, to vectorize structurally complex code (including internal loops or intricate object-oriented constructions), but also requires much deeper understanding of vectorization.

Compiler directive **#pragma simd** has several optional parameters that control vectorization. For example, **reduction** parameter is used to perform a reduction operation (such as sum or maximum) in a fashion similar to the corresponding option in OpenMP. We demonstrate another important parameter **vectorlength** that defines the amount of loop iterations that can be performed in parallel. This parameter is very convenient to deal with loops with limited vectorization potential: several adjacent iterations are independent, but “further” iterations have data dependencies. We consider such example and vectorize it on maximum possible vector length.

We continue with **Array notation** for code vectorization. Some algorithms can be directly expressed as a sequence of operations on vectors. In this case an Intel Cilk Plus extension called **Array notation** can be used to achieve good performance and auto vectorization. It allows to perform vector operations without explicitly writing loops via special operation **:** that applies the given operation to every element of the given interval. We consider a simple example of vector addition using **Array notation** and demonstrate usage of **Array notation** for the previous example. Further details of **Array notation** are given in the next lecture.

We discuss **Elemental functions**. Loop auto vectorization requires no internal function calls except functions that are inlined. In some cases it is hard to provide function inlining because of the complexity of the code. Additionally, manual inlining is often not recommended for code quality and maintainability reasons. **Elemental functions** in Intel Cilk Plus might help vectorization in this case. An elemental function is declared with **\_\_declspec(vector)**, its body performs an operation with one element, and the function allows parallel processing during loop vectorization. We show a simple example, further information is given in the next lecture.

The next segment of the practice is devoted to vectorization of loops with calls to mathematical routines. We consider the following example:

void exp\_loop(float\* a, float\* b, float\* c,  
 float\* d, int n)

{

#pragma ivdep

for (int i = 0; i < n; i++)

a[i] = b[i] + c[i] + expf(d[i]);

}

By using **#pragma ivdep** we prevent the problem with potential data dependencies similar to the previously considered example. Vectorization report confirms that the loop was vectorized. We discuss mechanisms of vectorization of such loops in case there are no hardware SIMD operations for mathematical routines being used. A short introduction to **SVML** (short vector math library) is given. The compiler inserts calls to SVML routines in case the loop is vectorized, and calls to scalar **LibM** routines otherwise. For loops with huge amount of iterations it is recommended to precompute values of mathematical routines for all iterations if possible by using **VML** (vector math library), part of Intel MKL (math kernel library); we give a short example.

In the final segment we demonstrate several more sophisticated examples that arise from applications and showcase some tips and tricks for performance boosting that might be not obvious at the first glance.

# FOR STUDENTS

For further information on vectorization we recommend [2].

# References

1. Rahman R. Intel® Xeon Phi™ Coprocessor Vector Microarchitecture.   
   URL: [<http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-vector-microarchitecture>]
2. Green R. Vectorization Essentials.   
   URL: [<http://software.intel.com/en-us/articles/vectorization-essential>]
3. Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual.
4. Jeffers J., Reinders J. Intel Xeon Phi Coprocessor High Performance Programming. Morgan Kaufmann, 2013.

# Individual work

1. Implement scalar and all vectorized versions of the first example considered for Intel Xeon Phi. Compare performance of all versions, explain the results.
2. Implement dense matrix-vector product. Find out details of vectorization using vectorization report, assist vectorization if necessary. Does vectorization depend on row-major or column-major matrix storage?
3. Perform a study similar to the previous assignment for dense matrix-matrix product computed by definition.
4. Study the considered example on loops with calls to mathematical routines. Find the minimum number of iterations which allows VML to outperform SVML.