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

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**

*Lecture 6. Optimization of applications for Intel Xeon Phi.   
Intel C/C++ compiler.*

Nizhni Novgorod

2014

# Objectives

The objective of this lecture is to study principles of performance optimization for Intel Xeon Phi*.* This objective includes the following activities:

1. To study programming tools that enable using Intel Xeon Phi, particularly **offload** mode.
2. To continue exploring vectorization.
3. To learn capabilities of the compiler and loop profiler.
4. To discuss load balancing in parallel programming for Xeon Phi.

# Abstract

This lecture is devoted to studying optimization techniques for Intel Xeon Phi. It is based on hardware details, programming models and vectorization capabilities discussed in previous lectures. The first segment introduces new elements of the **offload** mode that handle memory. We consider and compare explicit and implicit memory management. The second segment focuses on aspects of vectorization that were not properly considered previously. We also revise auto vectorization, compiler directives, Array notation and Elemental functions. In the third segment we introduce loop profiler and optimization reports, and discuss load balancing in parallel applications. The material is accompanied by examples.

# BRIEF OVERVIEW

The first segment of the lecture is devoted to offload mode of Xeon Phi programming. Previously we have given a brief description and considered some simple examples. In this lecture we discuss offload mode in detail.

By default the compiler generates code for offload pieces of program that is suitable for both coprocessor and CPU. This allows programs using offload mode run without Xeon Phi coprocessors. The code is copied to the coprocessor automatically. A software developer should specify data to copy to the coprocessor and back.

Memory of coprocessor and RAM is not shared, thus memory exchanges are needed. There are two modes of data transfers: explicit and implicit. In Fortran only explicit model can be used. In explicit model a software developer specifies data to be copied between coprocessor memory and RAM using special directives. In implicit model one specifies variables that are accessible from both CPU and coprocessor. We discuss and compare these models.

The second segment of the lecture is devoted to vectorization using Intel C/C++ Compiler. These techniques are equally applicable for all Xeon Phi programming modes, and CPU as well. We briefly overview material on vectorization from previous lectures and practices. Our main focus is #pragma simd compiler directive and Array notation in Intel Cilk Plus that have not been discussed in detail previously. We describe syntax of Array notation and list of available operations, discuss applications to static and dynamic arrays, and present some examples. We also consider Elemental functions that are somewhat similar to kernels in CUDA and OpenCL. Elemental function is a scalar function declared with \_\_declspec(vector) and called within a loop body. The compiler generates vectorized version of the function and calls it within vectorized loops. This operation can be used from a single thread or in a multi-threaded situation. We discuss basic rules of writing Elemental functions and demonstrate a relation with Array notation.

We discuss compiler vectorization reports and their interpretation and pay a special attention on the latest features. In particular, level 6 allows to retrieve very specific information about reasons of failed loop vectorization and recommendations of the compiler, and level 7 allows to retrieve expected speedup and memory access pattern.

It is important to consider efficiency of accessing vector registers. Due to hardware reasons reading and writing works most efficiently for aligned data. Alignment for Xeon Phi is 64 bytes, which is both vector register size and L1/L2 cache lane size. We discuss ways to ensure alignment and directives to notify the compiler about alignment. We investigate an example when an array is aligned but in parallel processing threads handle non-aligned chunks of the array.

We consider auto vectorization of external loops. By default auto vectorization is only applied to the inner-most loop. This might be not efficient if the inner-most loop has few iterations (less than vector register size). In this case the external loop can be vectorized using #pragma simd with Elemental functions or Array notation.

The next segment of the lecture is devoted to process of optimization of applications for Xeon Phi. Coprocessors allow relatively simple code porting, but it does not guarantee good performance on Xeon Phi. Thus, additional optimization steps are usually required. They mostly yield benefit for both CPUs and Xeon Phi.

Optimization process usually starts with detection of hot spots, which can be done using Intel® VTune™ Amplifier XE. There is another tool for loop profiling in Intel Compiler that allows to detect functions and loops taking significant computational time. The loops with high iteration count and short body are good candidates for vectorization. In case such a loop has intricate control code with lots of conditional statements, a thread-level or functional parallelism is an option. The loop profiler is applicable to sequential code only. For parallel application more powerful tools have to be used, including Intel® VTune™ Amplifier XE which will be a topic of the following lectures. In this lecture we describe the loop profiler which is enabled by special compiler directives that collect statistics of all functions and loops. An application built with these directives is then launched on a representative benchmark. The collected statistics will be output in a table and xml file that can be later analyzed using graphical Loop Profile Viewer.

Another powerful tool is compiler vectorization report. Besides the previously considered diagnostics, the compiler can give recommendations (-guide-vec[=n]) and report on optimization (-opt-report [n]).

Usually an important issue is load balancing. Here we discuss distribution of threads between cores of Xeon Phi. The coprocessor allows up to 4 threads per core. It is recommended to run at least 2 threads per core. Sometimes 2 or 3 threads per core might be beneficial compared to 4 because of the following reasons:

* reduced load on caches (L1, L2, TLB) due to lesser competition between threads;
* reduced competition for the only VPU;
* reduced competition for memory access.

However, sometimes 4 threads per core might be advantageous because of increased benefit of data locality in case all threads on the same core process the same data. The optimal amount of thread per core is usually found empirically.

In case the total number of threads in an application is lesser than number of threads supported by the coprocessor, one needs to address mapping of threads to cores. In OpenMP it can be controlled by KMP\_AFFINITY environment variable. We discuss compact, scatter and balanced modes.

Finally, we give some general recommendations on memory access efficiency: alignment, appropriate data organization (SoA instead of AoS), software prefetch, efficient utilization of L1 and L2 caches, huge pages.

# FOR STUDENTS

Some useful optimization techniques for offload mode are described in [6]. Detailed discussion of vectorization is given in [5, 8-10].

# References

1. Intel Corporation. Beginning Intel Xeon Phi Coprocessor Workshop, Offload Compilation, September 2012.
2. Fourestey G. Intel Xeon Phi Programming Models [<https://hpcforge.org/plugins/mediawiki/wiki/pracewp8/images/6/68/XeonPhi.pdf>]
3. Intel Corporation. Intel® C++ Compiler XE 13.1 User and Reference Guides: User-mandated or SIMD Vectorization [<http://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-42986DEF-8710-453A-9DAC-2086EE55F1F5.htm>]
4. Intel Corporation. Intel® C++ Compiler XE 13.1 User and Reference Guides: SIMD [<http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011Update/compiler_c/cref_cls/common/cppref_pragma_simd.htm>]
5. Intel Corporation. Advanced Intel Xeon Phi Coprocessor Workshop, Extracting Vector Performance with Intel Compilers, September 2012.
6. Green R.W. Effective Use of the Intel Compiler's Offload Features: [<http://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features>]
7. Green R.W. Overview of Vectorization Reports and new vec-report6: [<http://software.intel.com/en-us/articles/overview-of-vectorization-reports-and-new-vec-report6>]
8. Krishnai R. Data Alignment to Assist Vectorization: [<http://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization>]
9. Green R.W. Outer Loop Vectorization via Intel Cilk Plus Array Notations: [<http://software.intel.com/en-us/articles/outer-loop-vectorization-via-intel-cilk-plus-array-notations>]
10. Green R.W. Outer Loop Vectorization: [<http://software.intel.com/en-us/articles/outer-loop-vectorization>]
11. Intel Corporation. Beginning Intel Xeon Phi Coprocessor Workshop, Optimization, September 2012.
12. Intel Corporation. A case study comparing AoS (Arrays of Structures) and SoA (Structures of Arrays) data layouts for a compute-intensive loop run on Intel® Xeon® processors and Intel® Xeon Phi™ product family coprocessors: [<http://software.intel.com/en-us/articles/a-case-study-comparing-aos-arrays-of-structures-and-soa-structures-of-arrays-data-layouts>]
13. J. Jeffers, J. Reinders. Intel Xeon Phi Coprocessor High Performance Programming. -Morgan Kaufmann, 2013.
14. Intel® Cilk Plus Evaluation guide – [<http://software.intel.com/sites/products/evaluation-guides/docs/cilk-plus-evaluation-guide.pdf>]
15. Intel® Cilk Plus language specification –[<http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf>]
16. Intel Developer Zone [<http://software.intel.com/en-us/mic-developer>]

# Individual work

1. Compare explicit and implicit data transfers in offload mode.
2. Explain how does #pragma simd directive work and when is it used.
3. Overview main options of #pragma simd directive. Describe advantages and disadvantages of this way of vectorization.
4. Describe Array notation and its advantages for vectorization.
5. Describe Elemental functions and its advantages for vectorization.
6. How to use compiler vector reports?
7. What is data alignment and why is it used?
8. How to vectorize an external loop using compiler directives?
9. How to use loop profiler?
10. How to provide thread mapping to cores on Intel Xeon Phi?