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

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 4. Vector extensions of Intel Xeon Phi*

Nizhni Novgorod

2014

# Objectives

The objective of this lecture is to introduce vector extensions supported by Intel Xeon Phi coprocessors. Efficient usage of vector extensions is necessary for good performance on Xeon Phi.

# Abstract

In this lecture we consider vector extensions, which are one of the most important directions of development of instruction sets in modern architectures. Vector extensions augment instruction sets with specialized data types, registers and instructions oriented at SIMD (Single Instruction Multiple Data) paradigm. The essence of SIMD paradigm is to perform an instruction for multiple sets of data in parallel which allows to significantly improve performance of computational applications. The layout of materials on vectorization is as follows: first we give an introduction about vector extensions, and then consider features of vector extensions on Xeon Phi. We briefly describe the main data types and registers, give an overview of main operations and extended support for mathematical functions. The next segment of the lecture is devoted to access to vector extensions from high level programming languages. We consider several ways of vectorization. Finally we discuss an important topic of vectorization of mathematical functions.

Elements of vectorization will be considered in almost every following lecture or practice. A technique of vectorization is demonstrated in the next practice that illustrates typical problems related to vectorization and common solutions. The next lecture considers more sophisticated topics of vectorization in applied programs using Intel compiler. Throughout the following practice vectorization is consistently used as one of the key ways to enhance the performance.

# BRIEF OVERVIEW

First we introduce a general concept of vectorization and vector extensions that augment instruction set with specialized data types, registers and instructions oriented at SIMD (Single Instruction Multiple Data) paradigm. An efficient usage of vector instructions (vectorization) can significantly boost performance of applications. We consider a simple example with vector add operation and demonstrate that vectorized add of 4 numbers generally does not lead to 4x speedup compared to scalar code due to overheads related to data packing and unpacking, data dependencies, and other reasons. Nevertheless, for a wide set of applications vectorization is one of the most important resources of performance improvement. Typical algorithms used in multimedia tend to perform lots of similar operations on large sets of data. Many engineering computations benefit from at least partial vectorization. Modern CPUs support vector operations in single precision with 4 (SSE instruction set) or 8 (AVX instruction set) numbers, in double precision – with 2 (SSE) or 4 (AVX) numbers. Thus by virtue of ignoring vectorization one can lose up to 87.5% of CPU performance which is very significant. Vectorization is even more important on Xeon Phi, as vector registers on Xeon Phi are two times larger compared to AVX. The bottom line is that efficient usage of vector extensions is one of the key mechanisms that affect performance on both CPUs and particularly on Xeon Phi coprocessors.

The next segment is devoted to a brief overview of vector extensions and their historical evolution in Intel processors: MMX – SSE – SSE2 – SSE3 – SSE4 – AVX. Then we describe vector extensions for Intel Xeon Phi coprocessors. This description is aimed to create a general impression of vectorization and an understanding of how does C/C++ code translate to machine/assembly code generated by a compiler. This understanding is rather useful for program optimization. Additional information on this topic is given in [1, 3].

Xeon Phi architecture is a new step towards development of vector extensions. Each core contains a specialized vector processing unit with 32 512-bit registers (zmm-registers, 2 times larger than in AVX) and supports vector FMA (fused multiply–add) instruction that performs operation *a = a + b \* c* with a single round-off. Each core of Xeon Phi is capable of vector operations on 16 32-bit integers, or 8 64-bit integers, or 16 single precision floating point numbers, or 8 double precision floating point numbers. Additionally, there is a support for operations with complex floating-point numbers. Unlike other vector extensions, operations on Xeon Phi are ternary – they have 2 arguments and 1 result which according to some sources (e.g. [1]) yields 20% performance increase compared to the traditional 2-operand scheme. For FMA instructions all 3 operands are arguments, one of which is also a result.

The overview is concluded by a description of the main types of vector operations:

1. *Arithmetic operations*: add, subtract, multiply, divide, FMA (for floating-point computations).

2. *Type cast operations*, that perform type transformations according to certain rules (see [1, 3]).

3. *Logic operations*, that allow to perform vector comparisons, find min and max, etc.

4. *Data access operations*. This group contains operations to load data from memory to a register and store data from a register to memory. The new instruction set supports scatter/gather instructions that operate with data stored with a stride.

Additionally there is a support for software prefetch, streaming stores, swizzle, shuffle, extended support for 4 mathematical functions and other capabilities [3].

In the next segment we discuss support for vectorization in high level programming languages and the role of an optimizing compiler. The following ways of vectorization are considered:

1. Specialized high performance libraries that utilize vector instructions. This is a good choice in case a library meets most needs of an application.
2. Program development using C/C++ or Fortran and building using a vectorizing compiler. We study this way in detail, discuss the conditions required for Intel compiler vectorization of loops (minimization of data dependencies, function inlining, data alignment, uniform memory access, no mixed-type computations, elimination of conditionals if possible, etc.), and give some examples.
3. Compiler directives and options. We give some examples.
4. Array Notation and Elemental Functions from Intel Cilk Plus. We give a brief overview, this way will be considered in detail in the next lectures.
5. Intrinsic C++ classes for SIMD. One can develop a special C++ class for storing and handling packed data using inline assembly or intrinsics – special functions that directly map to vector instructions.
6. Vector intrinsic functions. This approach uses special data types that map to hardware-friendly data types and the previously mentioned intrinsic functions.
7. Program development using assembly.

A detailed discussion of approaches 5-7 is out of the limits of this course.

Another segment of the lecture is devoted to an important topic of mixing vectorization and mathematical functions called in loops, as computation of mathematical functions often takes significant time. We discuss the following ways of computing mathematical functions.

First, C compiler has a special module LibM that implements the main mathematical routines. In case the loop was not vectorized (or there was no loop at all) compiler inserts calls to LibM in the machine code. A simple recompilation with Intel compiler often leads to significant reduction of computational time due to the highly optimized mathematical module. Second, Intel compilers have SVML (short vector math library) module that contains vectorized implementations of mathematical routines for a short vector of arguments. Vector lengths are defined by a number of arguments that can be packed in xmm (for SSE), ymm (for AVX), or zmm (for Xeon Phi vector extensions) register. One can reduce an average time of function computation for a point by computing it in several points in parallel. In this case SVML is preferable over LibM. A compiler will automatically use SVML in case the rest of the loop can be vectorized. Third, some applications need to compute function values for many points at once which can be efficiently done using VML (vector math library) which is a part of Intel MKL (Math Kernel Library). For large lengths of vectors VML can be provide a significant performance benefit.

# FOR STUDENTS

A detailed description of VPU architecture and Xeon Phi vector extensions is given in [1]. A series of works [2] contains exhaustive information about various aspects of vectorization in computational programs. Book [4] demonstrates examples of vectorization and studies its influence on performance on Xeon Phi.

# 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. List the main technical characteristics of Xeon Phi vector extensions.
2. List the main Xeon Phi vector instruction groups.
3. List the main ways of vectorization in C and Fortran programming languages. Formulate advantages and disadvantages of each way.
4. Implement dense matrix-vector product on Xeon Phi. Ensure vectorization using different ways. Analyze the correctness and performance compared to scalar code.