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

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 7. Prime factorization. Vectorization and load balancing.*

Nizhni Novgorod

2014

# Objectives

The objective of this practice is to study optimization on the example of prime factorization. This objective includes the following activities:

1. Studying a simple prime factorization algorithm.
2. Sequential implementation and porting to Xeon Phi.
3. Parallel implementation that uniformly distributes numbers being factorized between threads.
4. Approaches to load balancing to boost efficiency.
5. Optimization for Xeon Phi.
6. Hybrid scheme of computation on CPU and Xeon Phi simultaneously.

# Abstract

This practice is devoted to boosting performance of applications of Xeon Phi. Unlike the previous practice, which focused on standard approaches, here we illustrate some non-standard techniques.

We use a simple algorithm for prime factorization, which is very suitable for demonstration of optimization techniques. We describe an implementation and porting to Xeon Phi, discuss efficiency of parallel implementation and load balancing issues. Optimization aspects are considered with vectorization of a complicated loop. Finally, we demonstrate a hybrid CPU + Xeon Phi computational scheme.

# BRIEF OVERVIEW

We consider a problem of prime factorization of all natural numbers in [1, N]. There are various algorithms; we use a simple algorithm of dividing a number by all previous natural numbers. If any remainder is zero, we save the divisor and continue with the quotient, until it is 1. We stress that this algorithm is far from ideal, but is well suited for demonstration of load balancing issues. The typical implementation is also interesting from optimization point of view. A sequential implementation of the algorithm is as follows:

void factorization()

{

for (int i = 1; i < NUM\_NUMBERS; i++)

{

int number = i;

int idx = number;

for (int j = 2; j < idx; j++)

{

if (number == 1) break;

int r;

r = number % j;

if (r == 0)

{

number /= j;

divisors[idx].push\_back(j);

j--;

}

}

}

}

We discuss porting this implementation to Xeon Phi using offload mode. Performance of this version on Xeon Phi is significantly lesser compared to CPU, which is no surprise as the code is scalar and sequential, and a single core of Xeon Phi is weaker compared to CPU.

We proceed to parallel implementation and discuss load balancing issues. First we parallelize using OpenMP with default static schedule, so that all numbers are divided between threads with the first thread handling the first k numbers, etc. Speedup of this version on Xeon Phi is about 47 times, which is fair but not ideal given the fact there are 60 cores. The reason of non-ideal speedup is load balancing: factorization of larger numbers takes longer, thus the last thread has the maximum workload.

Our next approach is using static schedule of OpenMP with the chunk = 2. However, it leads to almost 2 times performance degradation. The reason is that the first thread processes the first number, the second thread – the second number, etc. Threads with that process even numbers get less work because the number is halved on the first iteration. As a result, this introduces even more imbalance to the previous scheme. This problem can be resolved by distributing groups of numbers instead of single numbers. This is done using chunk parameter, its value is chosen empirically.

We analyze performance of implementation. Despite solving load balancing issues, parallel implementation on Xeon Phi is still slower than sequential version on CPU. This is caused by absence of vectorization: vectorization report shows that the compiler was unable to vectorized the loop with conditions and data dependencies. The code should be significantly modified to eliminate dependencies. The main operation is comparison of remainders with zero. Obviously, there are much more such checks than actual divisors and remainders are mostly non-zero. We can exploit this fact by computing remainders from 16 sequential numbers and multiplying them. If the product is non-zero, all 16 numbers can be skipped, otherwise the standard test has to be performed. The loops for operating with 16 numbers are vectorized with overall speedup of 14.8. Vectorization yields performance benefit on host as well. The final version on Xeon Phi outperforms host by a factor of 3.

Finally, we construct a hybrid CPU + Xeon Phi version. A straightforward implementation results in performance that is average between CPU and Xeon Phi. We demonstrate that load balancing that takes into account different performance of the devices helps to outperform the implementation on Xeon Phi.

# FOR STUDENTS

An exhaustive study of factorization algorithms is given in [1]. Examples of vectorization and load balancing on Xeon Phi are given in [2].

# References

1. Knuth D. The art of computer programming. Volume 2: Seminumerical Algorithms. Third Edition. Massachusetts: Addison-Wesley, 1997.
2. Jeffers J., Reinders J. Intel Xeon Phi Coprocessor High Performance Programming. – Morgan Kaufmann, 2013.

# Individual work

1. Adjust number of OpenMP thread for the maximum performance.
2. Check if performance of our implementation increases with odd chunk size.
3. Study ways of algorithmic optimization of prime factorization.
4. Study ways of further optimization of our implementation.
5. Find the optimal distribution of workload between CPU and Xeon Phi in the hybrid scheme.