Proceedings of the Federated Conference on Computer Science and Information Systems pp. 655–664
DOI: 10.15439/2016F436
ACSIS, Vol. 8. ISSN 2300-5963

Parallelizing nested loops on the Intel Xeon Phi on the example of the dense WZ factorization

Jarosław Bylina, Beata Bylina
Marie Curie-Skłodowska University,
Institute of Mathematics,
Pl. M. Curie-Skłodowskiej 5,
20-031 Lublin, Poland
Email: {jaroslaw.bylina,beata.bylina}@umcs.pl

***Abstract*—In this article we evaluate some strategies of parallelizing nested loops on Intel Xeon Phi on the example of the WZ factorization for dense matrices. We employ both parallelism and vectorization to accelerate nested loops on manycore coprocessor.**

For random dense square matrices with the dominant diagonal we report the execution time and the performance of the nested loops. Numerical experiments show that the vectorization that is efficiently exploiting SIMD vector units do not always improve the application performance on the coprocessor.

I. Introduction

MANYCORE computers with shared memory are used to solve the computational science problems. One of the machines with manycore architecture is Intel Xeon Phi [8], [9], which is based on Intel's Many Integrated Core (MIC). Intel Xeon Phi has got over 60 cores, hardware threading capabilities and wide vector units (VPU).

To implement parallel programs on manycore systems with shared memory, in particular on Intel Xeon Phi, programmers can use the OpenMP standard [12] as for the traditional multicore processors. The programming model provides a set of directives to explicitly define parallel regions in applications. The compiler translates these directives. One of its most interesting features in the language is the support for the nested parallelism.

In the scientific applications, loops are an important source of the parallelism — nested loops, in particular. Parallelizing nested loops require the programmer to make a decision about applying some strategies of the parallelization and the vectorization.

The research of the parallelization of nested loops have been undertaken by different scientists.

In the work [10], the authors study five different models for the nested parallel loops execution on shared-memory manyprocessors and show a simulation-based performance comparison of different techniques using a real application. The possibility to take advantage of the parallelism in nested parallel loops with the use of good scheduling and synchronization algorithms is described.

An automatic mechanism to dynamically detect the best way to exploit the parallelism when having nested parallel loops is presented in the study [5]. This mechanism takes into account the number of threads, the size of the problem, the number of iterations in a loop and it was implemented inside the IBM XL runtime library. That paper examined (among others) an LU kernel, which decomposes the matrix $\mathbf{A}$ into the matrices: $\mathbf{L}$ (a lower triangular matrix) and $\mathbf{U}$ (an upper triangular matrix).

An algorithm for finding good distributions of threads to tasks is provided and the implementation of the nested parallelism in OpenMP is discussed in the paper [1].

The focus of [7] was to investigate the possibility of dynamically choosing, at runtime, the loop which utilizes the available threads the best.

One of the direct methods of solving a dense linear system is to factorize the matrix into some simpler matrices — it is its decomposition into factor matrices of a simpler structure or of some specific properties — and then solving simpler linear systems. The most known factorization is the LU factorization (mentioned above). Another form of the factorization is the WZ factorization. In the work [2] we investigated four strategies of parallelizing nested loops on multicore architectures on the example of the WZ factorization [3], [6], [11]. We dealt with the following parallelism strategies for nested loops: *outer*, *inner*, *nested* and *split*.

For random dense square matrices with the dominant diagonal we reported the execution time, the performance, the speedup of the WZ factorization for these four strategies of parallelizing nested loops and we investigated the accuracy of such solutions. The *outer* and *split* approaches achieved the best speedup.

The goal of this paper is to study parallelized nested loops on Intel Xeon Phi. We research only two strategies, namely: *outer* and *split* due to their best results for multicore architectures. The efficient parallelizing of a nested loop is very difficult on Intel Xeon Phi because we must employ both a large number of threads and wide vector units available in Intel Xeon Phi. For the thread-level parallelism we use OpenMP [12], [4].

The OpenMP standard supports the loop parallelism. For the OpenMP standard, it is done by the utilization of the directive `#pragma omp parallel for`, which provides a shortcut for specifying a parallel region that contains a single `#pragma omp for`. We scale nested loops to a large number of threads and choose a good load balancing. The division of the work among threads is controlled with the

978-83-60810-90-3/$25.00©2016, IEEE

`schedule` clause.

We provide enough work for the coprocessor and we also try to efficiently exploit 512-bit vectors on Intel Xeon Phi using adequate pragmas.

The paper deals with the following issues: In Section II it describes the main characteristics of the Intel Xeon Phi architecture. Section III provides some information about some strategies of parallelizing nested loops and their application to the original WZ factorization. Section IV presents the results of our experiments. The time, the speedup, the performance of the WZ factorization for different strategies on Intel Xeon Phi are analyzed. Section V is a summary of our experiments.

II. Intel Xeon Phi architecture

Intel Xeon Phi [9] coprocessors are one of the state-of-the-art architectures which goal is to execute parallel codes. Intel Xeon Phi is a manycore coprocessor created on the basis of the Intel MIC (Many Integrated Cores) technology. The first generation the Intel MIC architecture is based on the Knights Corner chips. Many redesigned Intel CPU cores are connected by a bi-directional 512-bit ring bus. The cores are enriched with 64-bit instructions and the L1 and L2 cache memories. Each of the cores contains a vector processing unit (VPU), which together with 32 512-bit vector registers allows to process many data with the use of one instruction (SIMD instructions).

A single Intel Xeon Phi has 61 cores of 1,238 GHz frequency and it serves 244 threads and communicates through PCI-Express 2.0.

Intel Xeon Phi enhances the performance of applications written in the C/C++ and Fortran languages. The Intel company offers a set of programming tools assisting programming processes such as compilers, debuggers, libraries, that allow creating parallel applications (e. g. Pthread, OpenMP, Intel Cilk plus, MPI). Intel Xeon Phi is able to use standard parallel programming models such as OpenMP.

The Intel Xeon Phi coprocessors can work in two executing modes: the native mode or the offload mode (for one Intel Xeon Phi).

In the native mode the task is executed directly by a coprocessor, which makes it a separate computing node. The compilation of the source code for the accelerator architecture demands so-called cross-compiling, which produces a file executable on Intel Xeon Phi. The native application can be started by hand on the coprocessor or by the *micnativeloadex* tool which automatically copies the program together with necessary files and then starts it.

III. Nested loops parallelization

An application with nested loops can be performed in parallel in different ways depending on compilers, hardware and run-time system support available. Nested loops require a programmer to take a decision concerning details of the parallelism. Nested loops can have a few levels of the parallelism. The outermost loop contains other loops. Next, each of these loops may also consist of loops. It is a reason for the increased complexity of the implementation.

Frequently, the parallelization is applied to the outer loop levels and the vectorization to the inner levels. If you are certain that the vectorization is a safe alternative (gives the same results as the non-vectorized code) in a particular loop where the compiler itself does not vectorize, using `#pragma simd` often provides the best and the most predictable benefits.

In this work we deal with the following parallelization strategies for the nested loops:

1) *outer*;
2) *split*.

All variables used in a parallel region are by default `shared`; in each strategy we declare explicitly all variables as `private` or `shared` for all directives respectively. Using the `private` clause, we specify that each thread has its own copy of variables.

To ensure a good load balancing for all threads we use the `schedule` clause, which specifies how the iterations of the loop are assigned to the threads. In the directive `#pragma omp parallel for` in the clause `schedule` we set values `static` or `dynamic`. We research the impact of this clause on the efficiency.

A. Outer

Outer — the simplest parallelization strategy of nested loops is the parallel execution of the outermost loop (not counting the loop which cannot be parallelized). This approach gives good results if the number of iterations in a loop is big and the iteration's granularity is coarse enough.

Figure 1 presents a listing of the outer strategy for the WZ factorization. The outermost k-loop cannot be parallelized. However, we can parallelize the i-loop. In this simple parallelization strategy the loop is divided equally between threads using both the static and dynamic scheduler.

Figure 2 also presents a listing of the outer strategy for the WZ factorization with the use of the vectorization. Again the outermost k-loop cannot be parallelized, however, we can parallelize the i-loop. The loop is divided equally between threads. The inner loop is vectorized. The compiler is unable to automatically vectorize the inner loop due to the vector dependences. To vectorize this code we use `#pragma simd`.

B. Split

The second (and final) strategy consists in the division of the i-loop into two separate loops and we denote it by *split*. Figure 3 shows a listing of the *split* strategy for the WZ factorization. The first loop is parallelized. The second loop is a nested loop and we execute only its outer loop in parallel.

Figure 4 also shows a listing of the *split* strategy for the WZ factorization. Here, the first loop is parallelized. The second loop is a nested loop and we execute its outer loop in parallel and we vectorize it at the same time. For the OpenMP standard, it is done by the utilization of the directive `#pragma omp parallel for simd`, which provides a shortcut for

```
for (k=1; k<=n/2-1; k++) {
  k2=n-k+1;
  det=a[k2,k]*a[k,k2]-a[k,k]*a[k2,k2];
#pragma omp parallel for default(none) private(i,j) shared(w,k,k2,n,a,det) \
schedule(static/dynamic)
  for (i=k+1; i<=(k2-1); i++) {
    w[i,k]=(a[k2,k] * a[i,k2] - a[k2,k] * a[i,k])/det;
    w[i,k2]=(a[k,k2] * a[i,k]-a[k,k] * a[i,k2])/det;
    for (j=k+1; j<=k2-1; j++)
        a[i,j]=a[i,j] - w[i,k]*a[k,j] - w[i,k2] * a[k2,j];
  }//for i
}//for k
```

Fig. 1. A fragment of the WZ factorization algorithm — the outer strategy with the static or dynamic scheduler without the vectorization

```
for (k=1; k<=n/2-1; k++) {
    k2=n-k+1;
    det=a[k2,k]*a[k,k2]-a[k,k]*a[k2,k2];
#pragma omp parallel for default(none) private(i,j) shared(w,k,k2,n,a,det) \
schedule(static/dynamic)
    for (i=k+1; i<=(k2-1); i++) {
      w[i,k]=(a[k2,k] * a[i,k2] - a[k2,k] * a[i,k])/det;
      w[i,k2]=(a[k,k2] * a[i,k]-a[k,k] * a[i,k2])/det;
#pragma simd
      for (j=k+1; j<=k2-1; j++)
        a[i,j]=a[i,j] - w[i,k]*a[k,j] - w[i,k2] * a[k2,j];
    }//for i
}//for k
```

Fig. 2. A fragment of the WZ factorization algorithm — the outer strategy with the static or dynamic scheduler with the vectorization

```
for (k=1; k<=n/2-1; k++) {
  k2=n-k+1;
  det=a[k2,k]*a[k,k2]-a[k,k]*a[k2,k2];
#pragma omp parallel for  default(none) private(i) shared(k2,k,w,n,a,det) \
schedule(static/dynamic)
   for (i=k+1; i<=(k2-1); i++) {
     w[i,k]=(a[k2,k]*a[i,k2]-a[k2,k2]*a[i,k])/det;
     w[i,k2]=(a[k,k2]*a[i,k]-a[k,k]*a[i,k2])/det;
   }
#pragma omp parallel for  default(none) shared(k2,k,a,w) private(i,j) \
schedule(static/dynamic)
   for (i=k+1; i<=(k2-1); i++)
     for (j=k+1; j<=k2-1; j++)
       a[i,j]=a[i,j]-w[i,k]*a[k,j]-w[i,k2]*a[k2,j];
}//for k
```

Fig. 3. A fragment of the WZ factorization algorithm — the split strategy with the static or dynamic scheduler without the vectorization

```
for (k=1; k<=n/2-1; k++) {
  k2=n-k+1;
  det=a[k2,k]*a[k,k2]-a[k,k]*a[k2,k2];
#pragma omp parallel for  default(none) private(i) shared(k2,k,w,n,a,det) \
schedule(static/dynamic)
  for (i=k+1; i<=(k2-1); i++) {
    w[i,k]=(a[k2,k]*a[i,k2]-a[k2,k2]*a[i,k])/det;
    w[i,k2]=(a[k,k2]*a[i,k]-a[k,k]*a[i,k2])/det;
  }
#pragma omp parallel for simd default(none) shared(k2,k,a,w) private(i,j) \
schedule(static/dynamic)
  for (i=k+1; i<=(k2-1); i++) {
    for (j=k+1; j<=k2-1; j++)
      a[i,j]=a[i,j]-w[i,k]*a[k,j]-w[i,k2]*a[k2,j];
  }
}//for k
```

Fig. 4. A fragment of the WZ factorization algorithm — the split strategy with the static or dynamic scheduler with the parallelization of the first loop and with the parallelization and the vectorization of the second outer loop

specifying a parallel region that contains a single `#pragma omp for simd`.

Figure 5 shows a listing of another version of the *split* strategy for WZ factorization. The first loop is parallelized. The second loop is a nested loop and we execute the outer loop in parallel and we vectorize the inner loop using `#pragma simd`.

IV. Numerical experiments

In this section we show how we tested the time and the performance of the parallelized nested loops for Intel Xeon Phi. Our intention was to investigate different nested loops parallelization strategies for nested loops on manycore architectures. We examined 10 versions of the parallelized nested loops:

1) *outer*:
   - *static* (Figure 1),
   - *dynamic* (Figure 1),
   - *static+simd* (Figure 2),
   - *dynamic+simd* (Figure 2);
2) *split*:
   - *static* (Figure 3),
   - *dynamic* (Figure 3),
   - *static+forsimd* (Figure 4),
   - *dynamic+forsimd* (Figure 4)
   - *static+simd* (Figure 5),
   - *dynamic+simd* (Figure 5).

The input matrices were generated (by the authors). They were random, dense, square matrices with a dominant diagonal of even sizes (1000, 2000, ..., 12000).

A. Test environment

The tests were carried out using a computing node of the following parameters:

- Platform: Intel Server Chassis R2000WTXXX, Intel Server Board S2600WT2.
- CPU: 2×Intel Xeon E5-2670 v3 (2x12 cores, 2.3 GHz).
- Memory: 128 GB DDR4 2133MT/s (8×Crucial CT16G4RFD4213).
- Network card: FDR InfiniBand ConnectX-3 Mellanox AXX1FDRIBIOM (FDR 56GT/S).
- Coprocessor: Intel Xeon Phi Coprocessor 7120P.
- Software: Intel Parallel Studio XE 2016 Cluster Edition for Linux (Intel C++ Compiler, Intel Math Kernel Library, Intel OpenMP).

The algorithms were implemented with the use of the C language and with the use of the double precision. Our codes were compiled by INTEL C Compiler (icc) with the optimization flag `-O3` and with the cross-compiling option `-mmic`. Additionally, all algorithms were linked with the OpenMP library. To run the native executable on the co-processor, the `micnativeloadex` command was used. We set the number of threads using the environment variable `OMP_NUM_THREADS`.

B. The run-time

All the processing times are reported in seconds. The time was measured with the OpenMP function `open_get_wtime()`. They were tested in the double precision.

In Figures 6 and 7 we have compared the average running time of the four versions of the *outer* strategy on Intel Xeon Phi.

Figure 6 shows the dependence of the time on the number of threads for the matrix of the size 12000 on Intel Xeon Phi.

Figure 7 shows the dependence of the time on the matrix size for 240 threads on Intel Xeon Phi.

In Figures 8 and 9 we have compared the average running time of the six versions *split* strategy on Intel Xeon Phi.

```
for (k=1; k<=n/2-1; k++) {
  k2=n-k+1;
  det=a[k2,k]*a[k,k2]-a[k,k]*a[k2,k2];
#pragma omp parallel for  default(none) private(i) shared(k2,k,w,n,a,det) \
schedule(static/dynamic)
   for (i=k+1; i<=(k2-1); i++) {
     w[i,k]=(a[k2,k]*a[i,k2]-a[k2,k2]*a[i,k])/det;
     w[i,k2]=(a[k,k2]*a[i,k]-a[k,k]*a[i,k2])/det;
    }
#pragma omp parallel for default(none) shared(k2,k,a,w) private(i,j) \
schedule(static/dynamic)
   for (i=k+1; i<=(k2-1); i++) {
#pragma simd
     for (j=k+1; j<=k2-1; j++)
       a[i,j]=a[i,j]-w[i,k]*a[k,j]-w[i,k2]*a[k2,j];
   }
}//for k
```

Fig. 5. A fragment of the WZ factorization algorithm — the split strategy with the static or dynamic scheduler with the parallelization of the first loop and the second loop and with the vectorization of the second outer loop and with vectorization of the inner loop

Figure 8 shows the dependence of the time on the number of threads for the matrix of the size 12000 on Intel Xeon Phi.

Figure 9 shows the dependence of the time on the matrix size for 240 threads on Intel Xeon Phi .

Figure 10 shows the dependence of the time on the number of threads for the matrix of the size 12000 on Intel Xeon Phi comparing the best of the *outer* and *split* strategies.

Figure 11 shows the dependence of the time on the matrix size for 240 threads on Intel Xeon Phi comparing the best of the *outer* and *split* strategies.

Using obtained results we conclude that:

- The *outer* strategy gives better results without the vectorization.
- For the *split* strategy the best results were obtained for the *static+forsimd* version (Figure 4).
- The *split* strategy achieves better execution time than the *outer* strategy.
- The worse execution time was achieved for the *outer* strategy in the *static+simd* version (Figure 2).
- The choice of the scheduler usually makes small differences in the execution time.
- If the size of the matrix is increased, then the runtime is increased too and it may become more profitable to use a big number of threads.
- Our approaches (both *outer* and *split* strategies) are scalable to a large number of threads.

C. The performance

Figures 12 and 13 compare the performance (in Gflops) results obtained for both the *outer* and the *split* strategies — in double precision on Intel Xeon Phi. The performance is based on the number of floating-point operations in the WZ factorization, namely:

$$\sum_{k=1}^{\frac{n}{2}-1}\left(3+\sum_{i=k+1}^{n-k}\left(8+\sum_{j=k+1}^{n-k}4\right)\right)=\frac{4n^3-7n-18}{6}.$$

Figure 12 shows the dependence of the performance (of the best algorithms of both strategies) on the size of the matrix for 240 threads.

Figure 13 shows the dependence of the performance (of the best algorithms of both strategies) on the number of threads for the matrix size of 12000.

We can see that the best performance (about 11 Gflops) is achieved by the *split* strategy for the matrix of the size 12000 for 240 threads, and worst (less than 2 Gflops) is for the *outer* strategy for the smallest sizes. The performance increases fast with the growth of the number of threads.

V. Conclusion

In this paper we examined several practical aspects of the nested parallel loop execution on Intel Xeon Phi. Our approach exploits the thread-level and SIMD parallelism of the Intel Xeon Phi coprocessor. We used different strategies for executing nested parallel loops on the examples of the WZ factorization.

Both the *outer* and the *split* algorithms exploited the available number of threads. The *split* strategy achieves the best performance. The performance of 11 Gflops was achieved for 240 threads on Intel Xeon Phi. The implementation of the *split* strategy presented in this paper achieves high performance results, which has a direct impact on the solution of linear systems. Using the split strategy can help programmers with loop parallelization in multithreading environments.

This paper is another example of the successful use of OpenMP for solving scientific applications on Intel Xeon Phi.

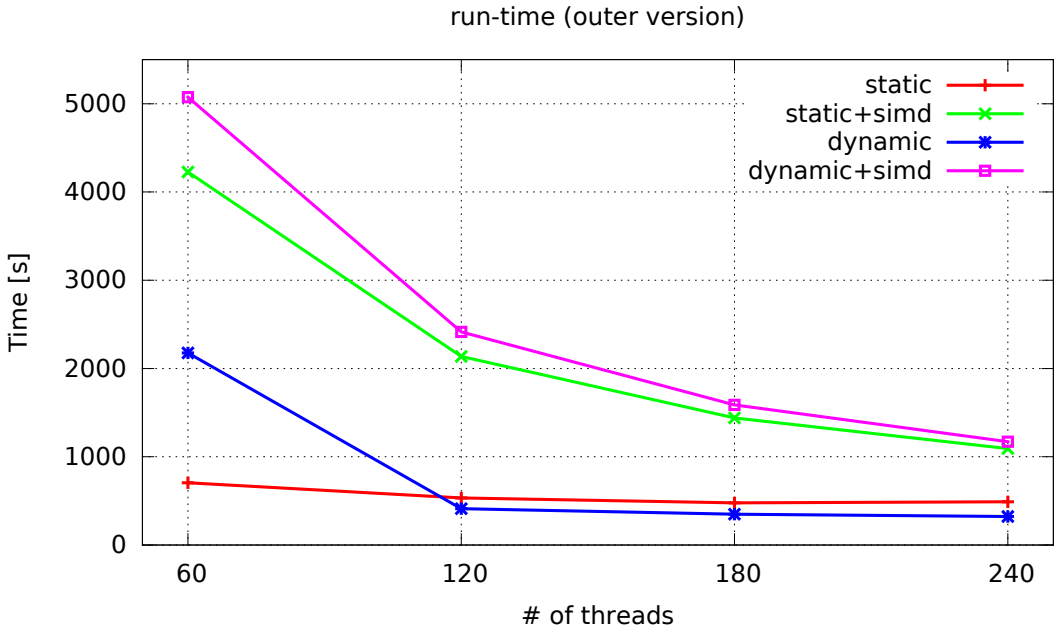

Fig. 6. The average running time of the WZ matrix decomposition as a function of the number of threads — for the algorithms using the double precision on Intel Xeon Phi for the matrix of the size 12000.

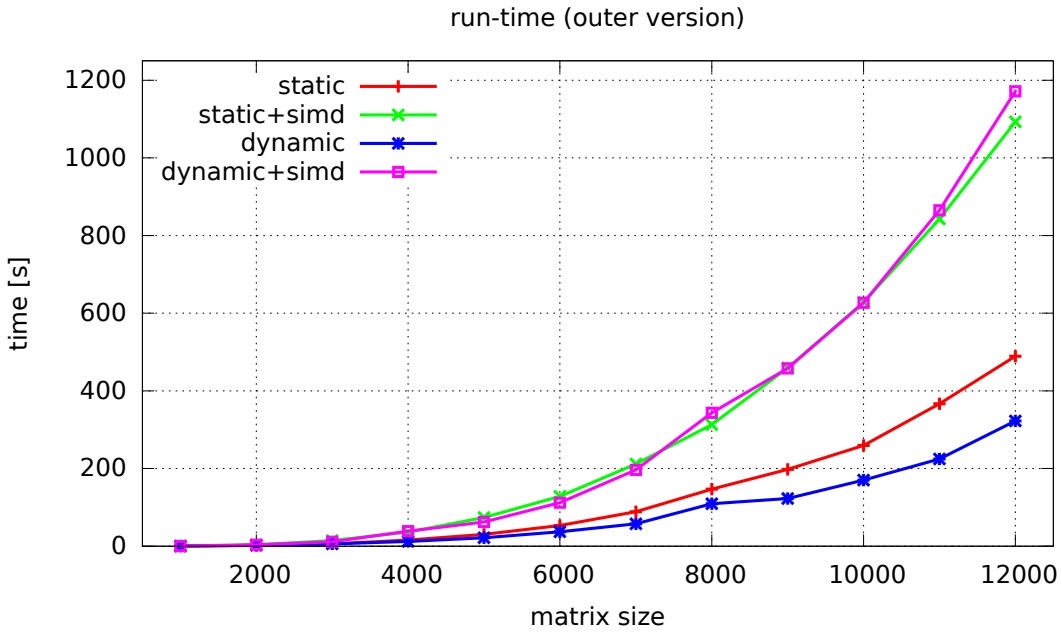

Fig. 7. The average running time of the WZ matrix decomposition as a function of the matrix size — for 240 threads on Intel Xeon Phi

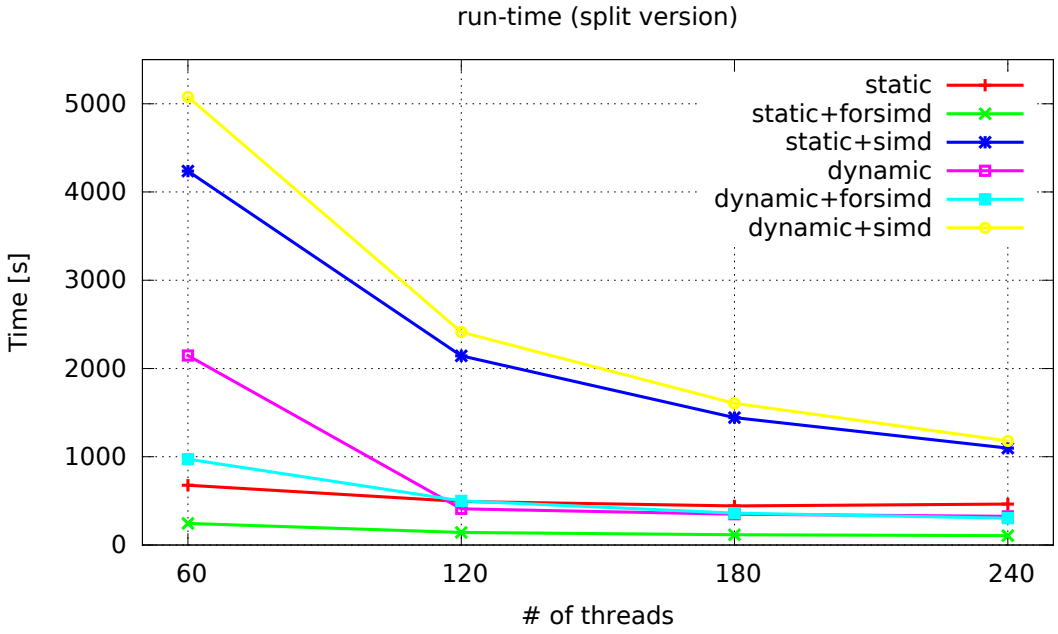

Fig. 8. The average running time of the WZ matrix decomposition as a function of the number of threads — for the *split* strategy using the double precision on Intel Xeon Phi for the matrix of the size 12000.

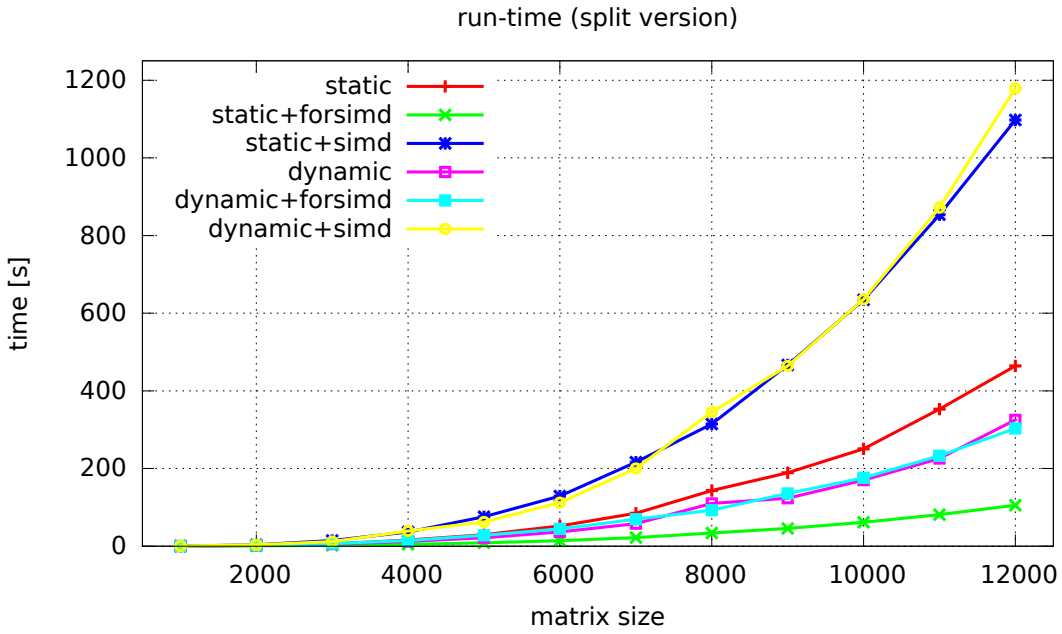

Fig. 9. The average running time of the WZ matrix decomposition as a function of the matrix size — for the *split* strategy for 240 threads on Intel Xeon Phi

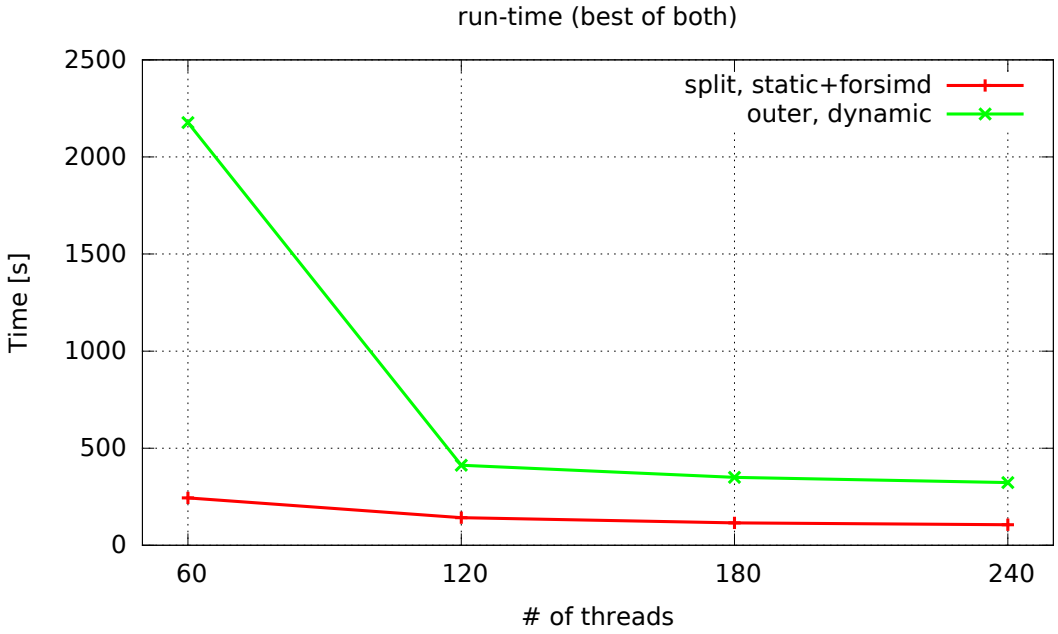

Fig. 10. The average running time of the WZ matrix decomposition as a function of the number of threads — the best of the *split* and *outer* strategies for the matrix of the size 12000

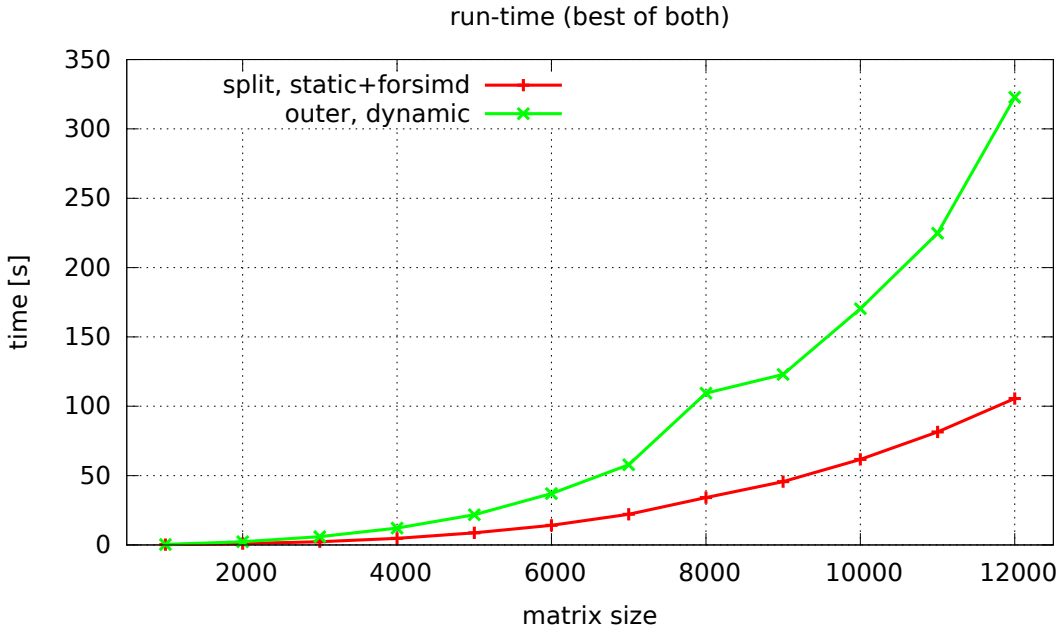

Fig. 11. The average running time of the WZ matrix decomposition as a function of the matrix size — the best of the *split* and *outer* strategies for 240 threads

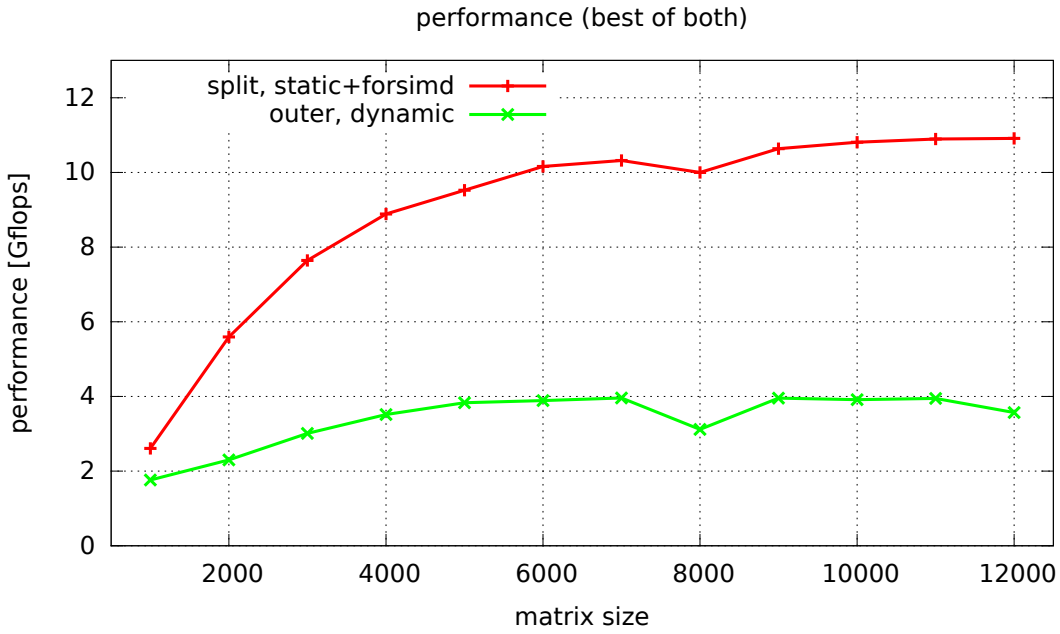

Fig. 12. The performance results for nested loops — for 240 threads for the best of both strategies as a function of the matrix size

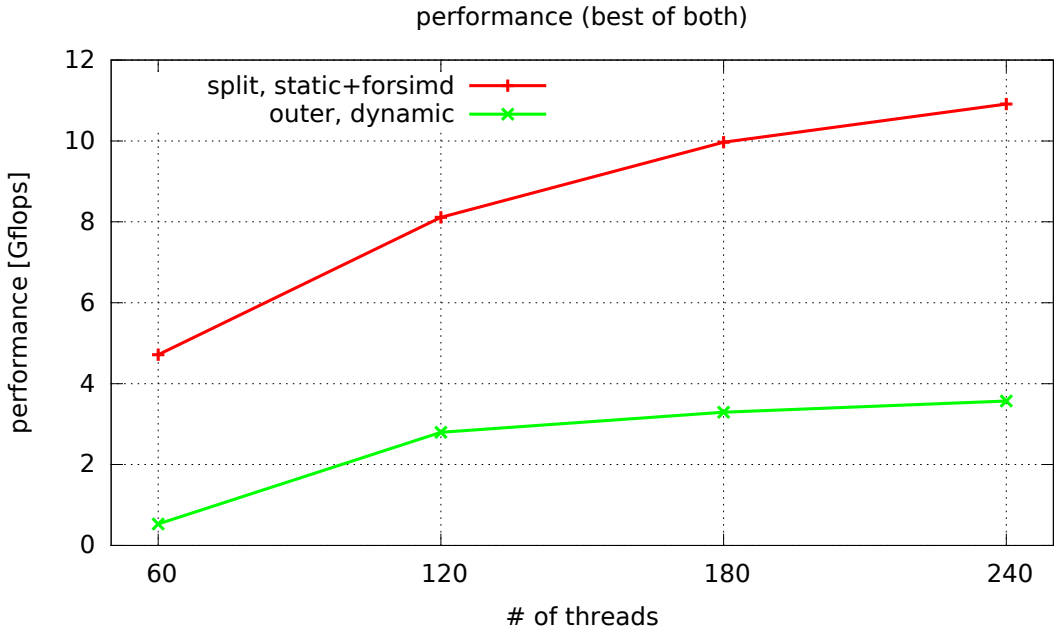

Fig. 13. The performance results for nested loops — for the matrix of the size 12000 for the best of both strategies as the function of number of threads

This paper is also example that the vectorization is not always the only key to achieving high performance on Intel Xeon Phi architecture.

The achieved results provide the basis for the further research on optimization of nested loops using the loop tiling technique in such a way that a data element used once is reused as soon as possible. Another field of further research is the use of thread affinity strategies which allow improving the performance and the scalability of our solution.

REFERENCES

[1] R. Blikberg, T. Sørevik, "Load balancing and OpenMP implementation of nested parallelism", *Parallel Computing* 31, Elsevier, 2005, pp. 984–998.

[2] B. Bylina, J. Bylina, "Strategies of parallelizing nested loops on the multicore architectures on the example of the WZ factorization for the dense matrices", Proceedings of the Federated Conference on Computer Science and Information Systems, Annals of Computer Science and Information Systems 5, 2015.

[3] S. Chandra Sekhara Rao, "Existence and uniqueness of WZ factorization", *Parallel Computing* 23, (1997), pp. 1129–1139.

[4] T. Cramer, D. Schmidl, M. Klemm, D. Mey, "OpenMP programming on Intel Xeon Phi coprocessors: An early performance comparison, Proc. of the Many-core Applications Research Community Symposium at RWTH Aachen University, (2012), pp. 38–44.

[5] A. Duran, R. Silvera, J. Corbalan, J. Labarta, "Runtime adjustment of parallel nested loops", *Proceedings of the 5th international conference on OpenMP Applications and Tools: shared Memory Parallel Programming with OpenMP*, Houston, 2004, pp. 137–147.

[6] D. J. Evans, M. Hatzopoulos, "The parallel solution of linear system", *Int. J. Comp. Math.* **7** (1979), pp. 227–238.

[7] A. Jackson, O. Agathokleous, "Dynamic Loop Parallelisation", arXiv: 1205.2367v1, 10 May 2012.

[8] J. Jeffers,J. Reinders, "Intel Xeon Phi Coprocessor High Performance Programming", Morgan Kaufmann Publishers Inc, 2013.

[9] R. Rahman, "Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers",Apress , Berkely, USA, 2013.

[10] A. Sadun, W. W. Hwu: "Executing nested parallel loops on shared-memeory multiprocessors", *Proceedings of the 21st Annual International Conference on Parallel Processing*, 1992.

[11] P. Yalamov, D. J. Evans: "The WZ matrix factorization method", *Parallel Computing* 21, 1995, pp. 1111–1120.

[12] OpenMP, http://openmp.org/wp/, April 2015.