DOI 10.12694/scpe.v19i3.1373
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 19, Number 3, pp. 259–274. http://www.scpe.org © 2018 SCPE

AN EXPERIMENTAL EVALUATION OF THE OPENMP THREAD MAPPING FOR SOME FACTORISATIONS ON XEON PHI COPROCESSOR AND ON HYBRID CPU-MIC PLATFORM

BEATA BYLINA AND JAROSLAW BYLINA *

Abstract. Efficient thread mapping relies upon matching the behaviour of the application with system characteristics. The main aim of this paper is to evaluate the influence of the OpenMP thread mapping on the computation performance of the matrix factorisations on Intel Xeon Phi coprocessor and hybrid CPU-MIC platforms. The authors consider parallel LU factorisations with and without pivoting as well as parallel QR and Cholesky factorizations — all from MKL (Math Kernel Library) library. The results show that the choice of thread affinity, the number of threads and the execution mode have a measurable impact on the performance and the scalability of the factorisations.

Key words: thread mapping, LU factorisation, QR factorization, Cholesky factorization, execution mode, Intel Xeon Phi, hybrid platform, performance

AMS subject classifications. 15A06, 15A30, 15A23

1. Introduction.

Modern computing platforms are getting more and more efficient, but it comes with a price — computer architectures are getting more and more complicated. There are a lot of low-level details of machine architecture which have to be considered by HPC programmers and scientists to benefit from the promised performance. Thus, the efficient HPC software development is getting harder despite more and more capable hardware. Therefore, we have to identify and know well the existing software tools and their weak and strong points on hybrid platforms — as the one used in this paper, namely Intel Xeon CPU coupled with Intel Xeon Phi (called here a hybrid CPU-MIC platform). Such hybrid architectures add another layer of the complexity and thus, an effective level of parallelism is difficult to achieve in heterogeneous architectures — especially, when both such different units are to perform computing-intensive parts of the algorithm. This causes more and more difficulties in the optimisation of the code. One of the techniques for optimising the code in order to effectively exploit the potential of the coprocessors and the hybrid CPU-MIC platform is the thread mapping.

However, the hybrid nature of the hardware hinders the efficient use of the thread mapping in practice. There is a similar problem with the proper choice of number of threads and the prospective use of various modes (native and automatic offload). Our goal is to experimentally answer these questions. The objects of our study are some well-known and widely used algorithms, namely the LU (without and with pivoting), QR and Cholesky factorisations. We investigate the practical use of the thread mapping for different modes and the number of threads.

Operating systems on Intel Xeon Phi and on the hybrid CPU-MIC platform run numerous software threads and these threads share a complex hierarchical memory. Since the architecture consists of many processing units, these software threads have to be assigned to appropriate processing units (that is, hardware threads). Such an assignment is called thread mapping [5]. This assignment should be used to efficiently exploit the potential of modern multiprocessors. Efficient parallel numerical algorithms and their implementations on different contemporary parallel machines are crucial for engineering applications and computational science.

Determining the efficiency of the thread mapping depends on the machine and the application. There is not a single thread mapping strategy that suits all the applications. We studied the OpenMP thread mapping strategies for matrix decompositions on multicore architectures in our work [3]. The results showed that the choice of thread affinity has the measurable impact on the executed time of the matrix factorisations. Here, we extend this investigation by an experimental evaluation of the OpenMP thread mapping for the LU (without and with pivoting), QR and Cholesky factorisations from MKL library (Math Kernel Library) [15] on the Intel Xeon Phi coprocessor and on the hybrid CPU-MIC platform. While the determining of the OpenMP thread

*Institute of Mathematics, Marie Curie-Sklodowska University, Pl. M. Curie-Sklodowskiej 5, Lublin, 20-031, Poland, beata.bylina@umcs.pl, jaroslaw.bylina@umcs.pl

mapping on Intel Xeon Phi is not very difficult, the same task on a hybrid CPU-MIC platform remains a challenging issue. The contribution of this paper to areas of the scalable algorithms on coprocessors and hybrid platform is an experimental evaluation of the LU factorisations, QR and Cholesky factorizations from MKL library in two modes, namely native on coprocessor and automatic offload on the hybrid CPU-MIC platform. This assessment takes into account the performance for the different settings of the OpenMP thread mapping and for the different number of threads on coprocessor and the different matrix size.

The rest of this paper is organised as follows. Section 2 describes related works regarding the thread mapping and the LU factorisation. Section 3 reviews the matrix decomposition, namely the block LU factorisation with and without pivoting, QR and Cholesky factorisations. Section 4 contains the overview of Intel Xeon Phi and an introduction to the programming model on Intel Xeon Phi and the hybrid CPU-MIC platform. Section 5 presents different thread mapping strategies on Intel Xeon Phi and the hybrid CPU-MIC platform. Section 6 shows the results of numerical experiments carried out on Intel Xeon Phi and on the hybrid CPU-MIC platform for the LU factorisation with and without pivoting, QR and Cholesky factorisations and Section 7 contains some considerations about the impact of various factors on the algorithms' performance. Finally, Section 8 concludes our research and presents the future plans.

2. Related work.

2.1. Thread Affinity. In the last years, the issue of the thread mapping control in OpenMP on different parallel architectures for different applications has been researched. The authors of [14] investigated the possibilities to improve thread mapping in OpenMP programs for several simple applications (for example, SpMV — sparse matrix-vector multiplication — and Jacobi solver) and presented the ways to apply this knowledge to larger application codes on ccNUMA and multicore architecture. In the work [10], a solution to control thread mapping in OpenMP programs was presented and shown to be compatible with MPI in hybrid use cases. The authors of [12] discussed effective thread mapping strategies through comparing the computing performance and analysing the performance differences between various mapping methods using the k-means application program to fully exploit the computing potential of the MIC (Many Integrated Core) coprocessor, as well as the hybrid system consisting of MIC and a traditional multicore CPU. Results of these papers showed that there is no single thread mapping strategy adapted for all the applications.

2.2. Factorisation. Recently, several groups have been working on the efficient parallel linear algebra libraries, particularly the Gaussian elimination. The Gaussian elimination on multicore and manycore architectures was studied, among others, in works [9], [2], [6] and [8]. In the work [9] the authors investigated the parallelization of sub-cubic Gaussian elimination. They focused on the parallelization of three subroutines, namely, the matrix multiplication, the triangular equation solver and the LU factorisation with pivoting. In [2], a class of parallel tiled linear algebra algorithms for multicore architectures is presented, the LU factorisation with pivoting, Cholesky among others. The article [6] describes recent developments in parallel implementations of Gaussian elimination for shared memory architecture. Four different approaches to pivot in the LU factorisation are investigated — partial pivoting among others, and all approaches were compared with the implementation of the LU without pivoting. The comparison given in that article gives a good insight into the performance properties of the different LU factorisation algorithms using relatively large shared memory systems. In the work [8] the design and implementation of several fundamental dense linear algebra (DLA) algorithms for multicore with Intel Xeon Phi coprocessors were presented. In particular, algorithms for solving linear systems were considered, namely the LU factorisation with pivoting. The research by Intel [11] shows a great performance of LINPACK benchmark. In this work, we research the LU factorisation, QR and Cholesky factorisations implementation from a vendor library, namely MKL.

3. Factorisations. The LU decomposition with pivoting factorises a matrix into matrices, namely a lower triangular matrix $\mathbf{L}$, an upper triangular matrix $\mathbf{U}$ and a permutation matrix $\mathbf{P}$. It has the following form:

$$\mathbf{PA} = \mathbf{LU}$$

For improving computing performance on the contemporary computer architecture, a block version of the LU decomposition is applied. The block LU decomposition is a matrix decomposition of a block matrix into a lower block triangular matrix **L**, an upper block triangular matrix **U** and block permutation matrix **P**. The block version of the LU decomposition is implemented in LAPACK [1]. That implementation is based on BLAS. The parallelism of that block version of the LU factorisation arises from the use of a multithreaded BLAS. The MKL library provides exactly this kind of implementation of BLAS and exactly this kind of parallel version of the block LU decomposition. The block LU algorithm is described in detail in [4]. The LAPACK LU algorithm is described in the following steps:

- A panel of b columns is factorised along with creation of a pivoting pattern (DGETF2 routine).
- Panel factorisation gives elementary transformations which are performed as block operations in the rest of the matrix — some rows are swapped correspondingly to the pivoting pattern (DLASWP) and top b rows are treated with the triangular solver (DTRSM).
- A matrix factorisation is performed (DGEMM) — the square remainder of the matrix is updated with the product of the panel (without top b rows) and the top b rows without the panel items.

The LU decomposition without pivoting factorises a matrix into two matrices, namely a lower triangular matrix **L** and an upper triangular matrix **U**. It has the following form:

$$\mathbf{A} = \mathbf{LU}$$

The implementation of LU without pivoting can be carried out very rarely in practice without risking serious numerical consequences. The LU without pivoting exists if the matrix **A** has a strict dominant diagonal. Giving up the pivoting improves performance — because we get rid of the rows swapping and because the panel operations can be easily parallelized now.

The total number of floating point operations (add, multiply, divide) for the LU factorisations without and with pivoting are the same and equal approximately $\frac{2}{3}n^3$. The number of the floating point comparisons for the LU factorisation with pivoting equals approximately $\frac{1}{2}n^2$ and for the LU factorisation without pivoting equals zero. Flops measurement gives only an approximated performance — because of the differences in kernels and dynamic of the parallelism. Section 6 shows the experiments which give a better comparison.

In this work, we investigate the LAPACK implementation of the LU factorisation from MKL library, namely `dgetrf` (LU with pivoting) and `dgetrfnpi` (LU without pivoting) routines.

The QR factorisation is a decomposition of a form $\mathbf{A} = \mathbf{QR}$, where **R** is a usual upper triangular matrix, and **Q** is an orthogonal matrix (that is, $\mathbf{Q}^T\mathbf{Q} = \mathbf{Q}\mathbf{Q}^T = \mathbf{I}$). It is used to solve least square problems and eigenvalues problems. The number of floating-point operations in the QR factorisation is $\frac{4}{3}n^3 + o(n^2)$ for a given matrix **A** of the size $n \times n$. Here, we use and study the LAPACK implementation of the QR factorisation from MKL library (`dgeqrf`).

The Cholesky factorisation is a decomposition of a form $\mathbf{A} = \mathbf{L}\mathbf{L}^T$, where **L** is a lower triangular matrix — and it is defined only for **A** being Hermitian and positive-definite. The number of floating-point operations in the Cholesky factorisation is $\frac{1}{3}n^3 + o(n^2)$ for a given matrix **A** of the size $n \times n$. Here, we use and study the LAPACK implementation of the Cholesky factorisation from MKL library (`dpotrf`).

4. Intel Xeon Phi and its programming models. Intel Xeon Phi coprocessors [13] are multicore coprocessors designed on the basis of Intel MIC (Many Integrated Cores) architecture, where more than 50 redesigned Intel CPU cores are connected. The cores allow running up to 4 hardware threads per each core. The cores ensure hardware support for the FMA (Fused Multiply-Add) instruction and also have their own vector processing unit (VPU). Additionally, the cores are enriched with 64-bit service instructions and a cache memory. In this work, we address the first generation of Intel Xeon Phi devices known as Knight Corner (KNC). KNC is connected to CPU through the PCIe bus. Contrary, the second generation call Knight Landing (KNL) is a separate processor.

MIC provides a general-purpose programming environment similar to that provided for CPUs. It supports the source-code portability between coprocessor and CPU allowing running the same code using CPU or MIC. The Intel company offers a set of programming tools assisting programming process — such as compilers, debuggers, libraries that allow creating parallel applications (e.g. OpenMP, Intel TBB) and different kinds of mathematical libraries (e.g. Intel MKL) similarly to conventional multicore CPUs.

The MKL library on MIC can be used in two ways: native and offload. The native mode does not require changing the multithreaded code, but only adding the `-mmic` option during compilation. In the native mode,

Table 5.1
Number of Intel Xeon Phi cores used for various affinity settings

number of threads	**cores used in the affinity setting** `compact`	`balanced`	`scatter`
60	15 (4 thr./core)	60 (1 thr./core)	60 (1 thr./core)
120	30 (4 thr./core)	60 (2 thr./core)	60 (2 thr./core)
180	45 (4 thr./core)	60 (3 thr./core)	60 (3 thr./core)
240	60 (4 thr./core)	60 (4 thr./core)	60 (4 thr./core)

the MKL routines are called from the program which runs directly on the coprocessor, treated as a separate processor.

In the offload mode, the indicated parts are executed on the coprocessor and the rest on CPU and thus, this platform is treated as a hybrid CPU-MIC computing platform. Typically, the CPU controls the code execution and the data transfer between the CPU and MIC. The programmer can indicate by himself which part of the program will be executed on the coprocessor with the use of suitable pragmas or using the automatic offload version of the MKL library (which is studied in this work). Only some computationally intensive level 3 BLAS routines (`GEMM`, `TRSM`) and LAPACK functions (for example `dgetrf` and `dgetrfnpi` routine) can be called in the automatic offload mode.

To obtain the good performance for these routines we need to use square matrices of the huge size. In the automatic offload mode, the runtime system is responsible for workload division between the host (CPU) and coprocessor (MIC). Moreover, it sends data between processing units. The programmer must only make some alternations in the code. Calls to the `mkl_mic_enable()` routines in the code enable switching on the automatic offload mode of the MKL library and switching off this mode is realised by `mkl_mic_disable()`. The programmer may set the percentage of workload between the host and coprocessor by calling `mkl_mic_set_workdivision()` with proper parameters. In our code, we use `MKL_MIC_AUTO_WORKDIVISION` which indicates that division of workload between the host and coprocessor will be determined by the runtime system.

5. Thread Mapping.

In this section, we briefly describe the thread mapping on MIC and on a hybrid CPU-MIC platform. The thread mapping (which is included in the Intel runtime library) provides different ways to bind the OpenMP threads to the hardware threads (we have 2 hardware threads per core on CPU and 4 hardware threads per core on MIC). On CPU, there are three types, and on MIC, there are four types of distribution of the OpenMP threads between hardware threads. The first one is `compact` type: threads sequentially bound (one after another) to successive hardware threads. A single core is filled by two OpenMP threads on CPU and four ones on MIC. The second type `scatter`: threads are bound sequentially to the successive cores as evenly as possible across the entire system. `Scatter` is the opposite of `compact`. The third type is `balanced`: threads are bound evenly to the successive hardware threads, which are the neighbouring threads; this type does not exist on CPU. Using the fourth type, `none`, we leave out the order in which threads are bound to the operating system.

In this research, we control the thread affinity using the environment variable `KMP_AFFINITY` on CPU and `PHI_KMP_AFFINITY` on MIC. We studied the OpenMP thread mapping strategies for matrix decompositions on multicore architectures in our work [3]. The results showed that the choice of `scatter` has the measurable impact on the executed time of the matrix factorisations on CPU. Thus, we set `scatter` for CPUs and change only the value of the environment variable `PHI_KMP_AFFINITY`. To avoid threads migration between cores we set the value `granularity=thread` for both the environment variables.

Table 5.1 shows the usage of the system with different affinity settings. We can see that for `compact` affinity, the load balance is only ensured for 240 threads. For `scatter` and `balanced` settings, the load is always the same, although the thread arrangement is different. We can also observe that the `balanced` with 60 threads should be equivalent to `scatter` with 60 threads, and the `balanced` with 240 threads should be equivalent to `compact` with 240 threads. It is because the threads in `balanced` mode are put on cores in sequence (e.g. for 120 threads — first and second on the first core etc.), and in `scatter` mode they are put in a round robin

Table 6.1
Hardware and software used in the experiments

	CPU	MIC
	2 × Intel Xeon E5-2670 v.3 (Haswell)	Intel Xeon Phi 7120 (Knights Corner)
# cores	24 (12 per socket)	61
# threads	48 (2 per core)	244 (4 per core)
clock	2.30 GHz	1.24 GHz
level 1 instruction cache	32 kB per core	32 kB per core
level 1 data cache	32 kB per core	32 kB per core
level 2 cache	256 kB per core	512 kB per core
level 3 cache	30 MB	—
SIMD register size	256 b	512 b
compiler	Intel ICC 16.0.0	Intel ICC 16.0.0
BLAS/LAPACK libraries	MKL 2016.0.109	MKL 2016.0.109

fashion (first on the first core, second on the second one etc.). Hence, the access to the memory is different. We can see that some combinations can be eliminated at sight (like `compact` with 60 threads), because only a part of the system works. Moreover, we should expect the best results for the full workload, that is for 240 threads. However, the `scatter` mode is not equivalent to `compact` and `balanced`. Thus, we expect it to behave poorer because `scatter` is less cache-friendly and the threads in the tested algorithms prefer access to neighbouring memory areas. On the other hand, the `balanced` mode reduces the data flow between caches of different cores what gives a higher throughput and lower latency. So, the `balanced` mode should give the best results, regardless of the number of threads.

6. Numerical Experiments. We tested the performance of two matrix factorisations, namely the block LU factorisation with and without pivoting from the MKL library on Intel Xeon Phi using native mode and on hybrid CPU-MIC platform using automatic offload mode. We compared four implementations:

- an optimised multithreaded implementation of the `dgetrfnpi` routine from the MKL library, which computes the complete LU factorisation of a general matrix without pivoting. In our case, the matrices are square, diagonally dominant and their size is $n \times n$. In the implementation of the `dgetrfnpi` routine, the panel factorisation (factorisation of a block of columns) is used, as well as the level 3 BLAS routines (`DTRSM` and `DGEMM`). We denoted this LU factorisation implementation by *LU without piv.*
- an optimised multithreaded implementation of the `dgetrf` routine from the MKL library, which computes the complete LU factorisation of a general matrix with pivoting. We denoted this LU factorisation implementation by *LU with piv.*
- an optimised multithreaded implementation of the `dgeqrf` routine from the MKL library, which computes the QR factorisation of a general matrix with pivoting. We denoted this QR factorisation implementation by *QR.*
- an optimised multithreaded implementation of the `dpotrf` routine from the MKL library, which computes the Cholesky factorisation of a symetric positive-definite matrix. We denoted this Cholesky factorisation implementation by *Cholesky.*

Table 6.1 shows details of the specification of the hardware and software used in the numerical experiments. All the experiments reported below were performed with the use of the double-precision arithmetic. In the automatic offload mode, we used all available cores on CPU and thus the number of threads was set to 24, and we changed only the number of threads on the coprocessor.

6.1. LU factorisation without pivoting. Figure 6.1 presents the performance of the LU factorisation without pivoting in the function of matrix size on Intel Xeon Phi in native mode for the four values of `PHI_KMP_AFFINITY` for a different number of the threads. For the native mode, we achieved the best performance for the `scatter` value of this environment variable for 120 threads or the `compact` value for 240 threads. All the

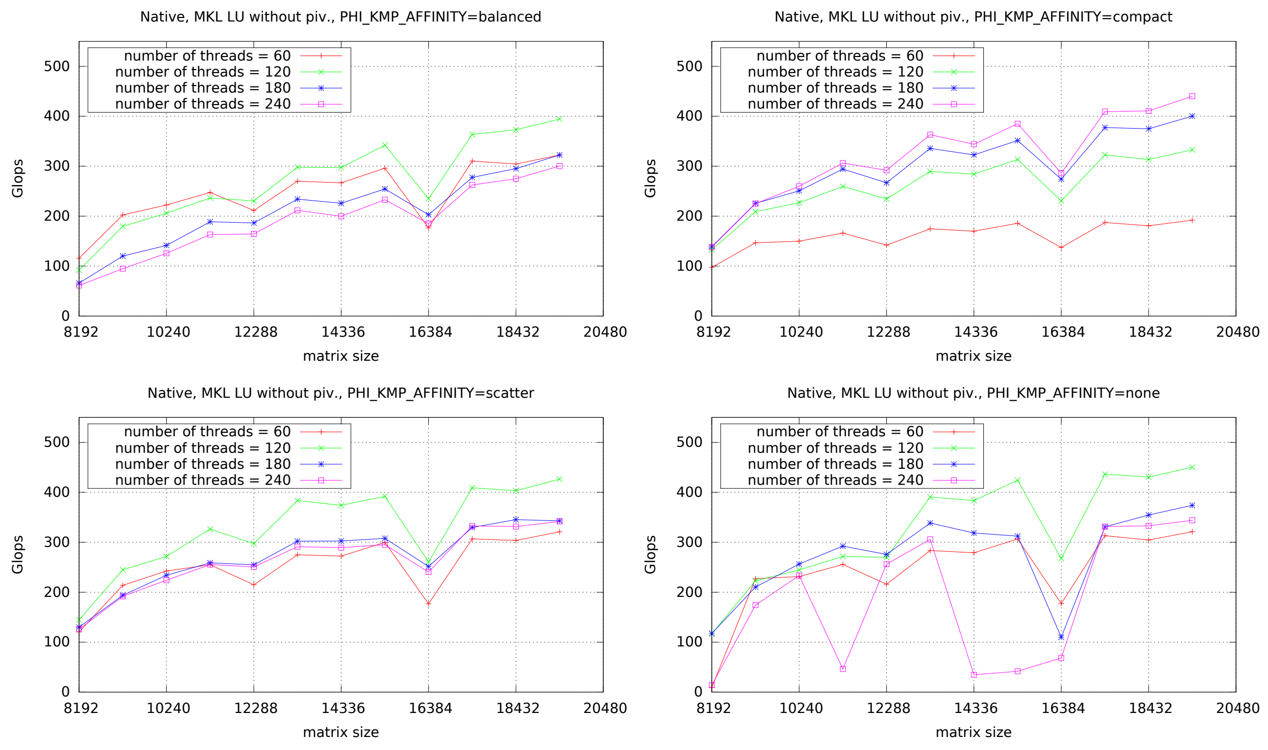

Fig. 6.1. *The performance of the LU factorisation without pivoting (MKL library's implementation) in the native mode on Intel Xeon Phi — for different matrix sizes, number of the threads, and the thread mapping settings.*

results are expected from Sect. 5 — besides `balance` with 240 threads (it should be the same as `compact` with 240 threads). The algorithm scales well with respect to the matrix size — except at the size of 16384. However, when we consider scaling with respect to the number of the threads, it is only the case for the `compact` affinity; other values give poor scalability with respect to the number of the threads. When we choose the `none` value of the affinity settings, the performance is chaotic and the scalability is poor both with respect to the size and to the threads — it is caused by the fact that the thread affinity is controlled by the operating system which makes decisions about it not suitable for computing. The last issue demanding an explanation is a sudden drop in performance at the size of $16384 = 2^{14}$. It seems to be caused by the cache size — for the matrix size of 16384 the blocks fit in cache ideally and there is no room for other data.

Figure 6.2 presents the performance of the LU factorisation without pivoting in the function of matrix size on hybrid CPU-MIC platform (with AO — automatic offload). In Fig. 6.2 (as well as in Figs. 6.3, 6.5 and 6.6), `cpu_aff/mic_aff` denotes the affinity settings both for CPU (the first value) and for coprocessor (the second value). The run-time reports say that the algorithm works exclusively on CPU up to the size of 14336 (the run-time systems believes that including MIC cannot improve the performance for such small data), so there is a poor scalability here. For bigger matrices, there is a performance drop, because the MIC gets some work and it is somewhat slower then sole CPU; however, after that, the performance grows almost up to the earlier level.

Table 6.2 shows the percentage work division between CPU and MIC for the LU decomposition without pivoting (for 24 threads on CPU and 240 threads on MIC). It is hard to determine the best number of threads because the computations — even for big matrices — are performed mainly on CPU. Thus, all the sizes except 14336 perform similarly (with the performance of about 600 Gflops) — the matrix size and the Xeon Phi settings (the number of threads and the affinity) matter little.

Figure 6.3 shows the performance of the LU factorisation in the function of the number of the threads for the matrix size of 19456 on Intel Xeon Phi in native mode and on the hybrid CPU-MIC platform in automatic offload mode for `KMP_AFFINITY=scatter` on CPU and the different values for `PHI_KMP_AFFINITY`. We can see

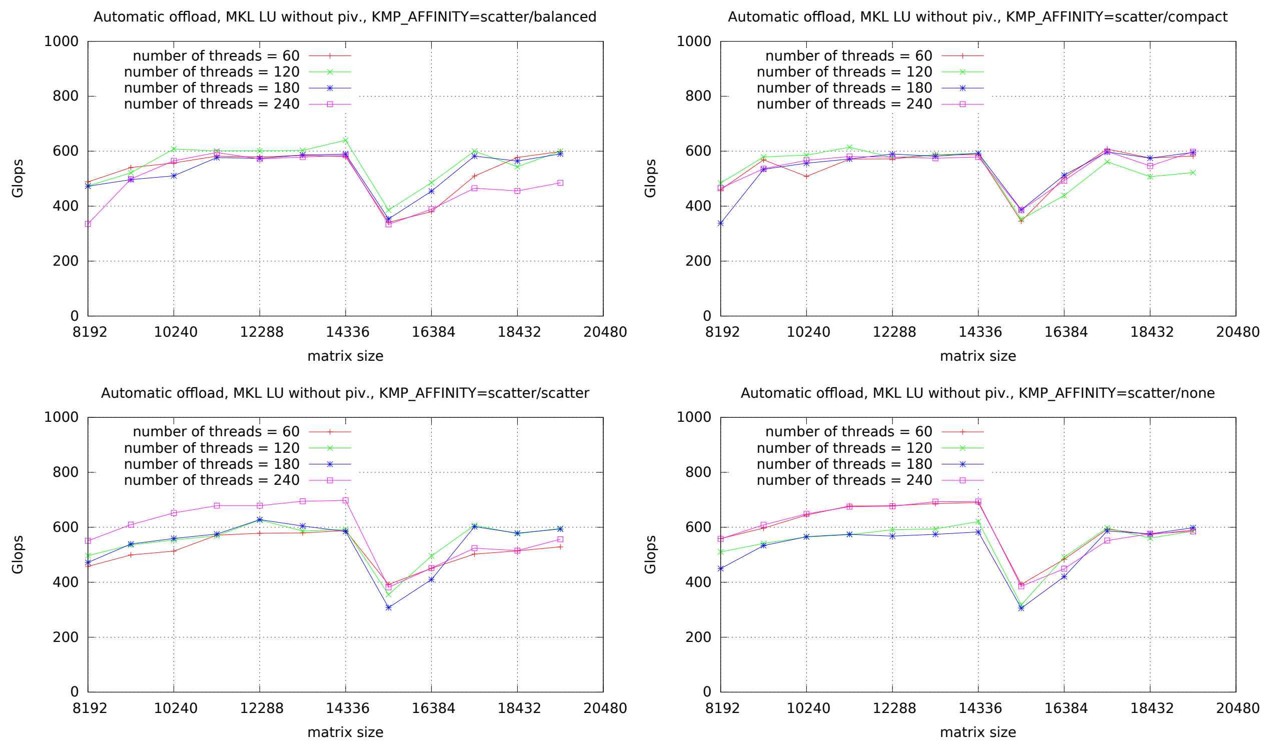

Fig. 6.2. *The performance of the LU factorisation without pivoting (MKL library's implementation) in the automatic offload mode — for different matrix sizes, number of the threads, and the thread mapping settings.*

Table 6.2
An exemplary percentage work division between CPU and MIC for the LU decomposition for the automatic offload

matrix	DTRSM		DGEMM	
size	CPU	MIC	CPU	MIC
15360	91%	9%	97%	3%
	74%	26%	84%	16%
16384	89%	11%	93%	7%
	75%	25%	83%	17%
17408	97%	3%	92%	8%
	93%	7%	83%	17%
19456	93%	7%	90%	10%
	83%	17%	82%	18%

that the performance is better for the AO mode than the native mode. It is caused by the fact that our CPU is generally faster than the Intel Xeon Phi and even employing both of them (as in AO mode), it is not easy to boost the efficiency.

6.2. LU factorisation with pivoting. Figure 6.4 presents the performance of the LU factorisation with pivoting in the function of matrix size on Intel Xeon Phi in native mode for the four values of `PHI_KMP_AFFINITY` for a different number of the threads. For the native mode, we achieved the best performance for the `balanced` and `compact` values of this environment variable (both for 240 threads). These were expected from the analysis from Sect. 5. For the `balanced` and `compact` affinities, the algorithm scales very well with respect to both the size of the matrix and the number of threads. The `scatter` affinity gives quite a nice scalability only up to the

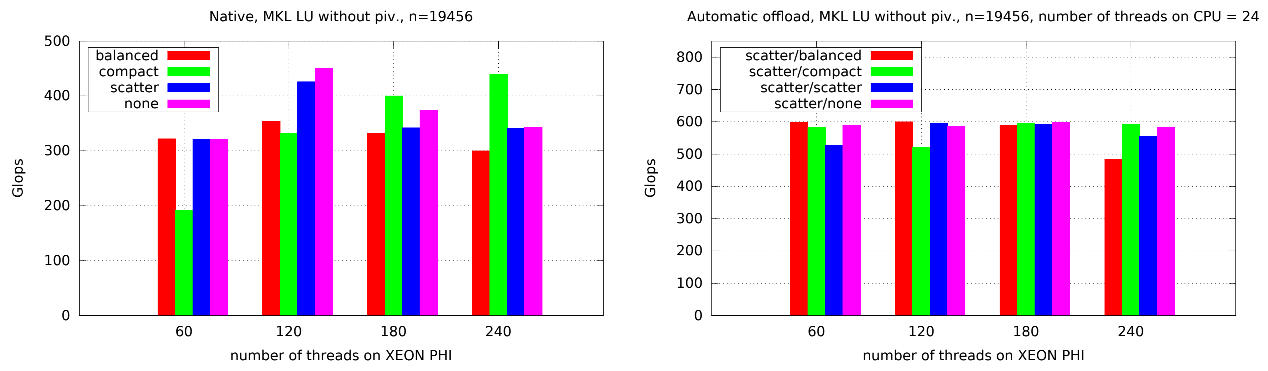

FIG. 6.3. *The performance of the LU factorisation without pivoting (MKL library's implementation) for the matrix size of 19456 on Intel Xeon Phi in the native mode (left) and on the hybrid CPU-MIC platform in the automatic offload mode (right).*

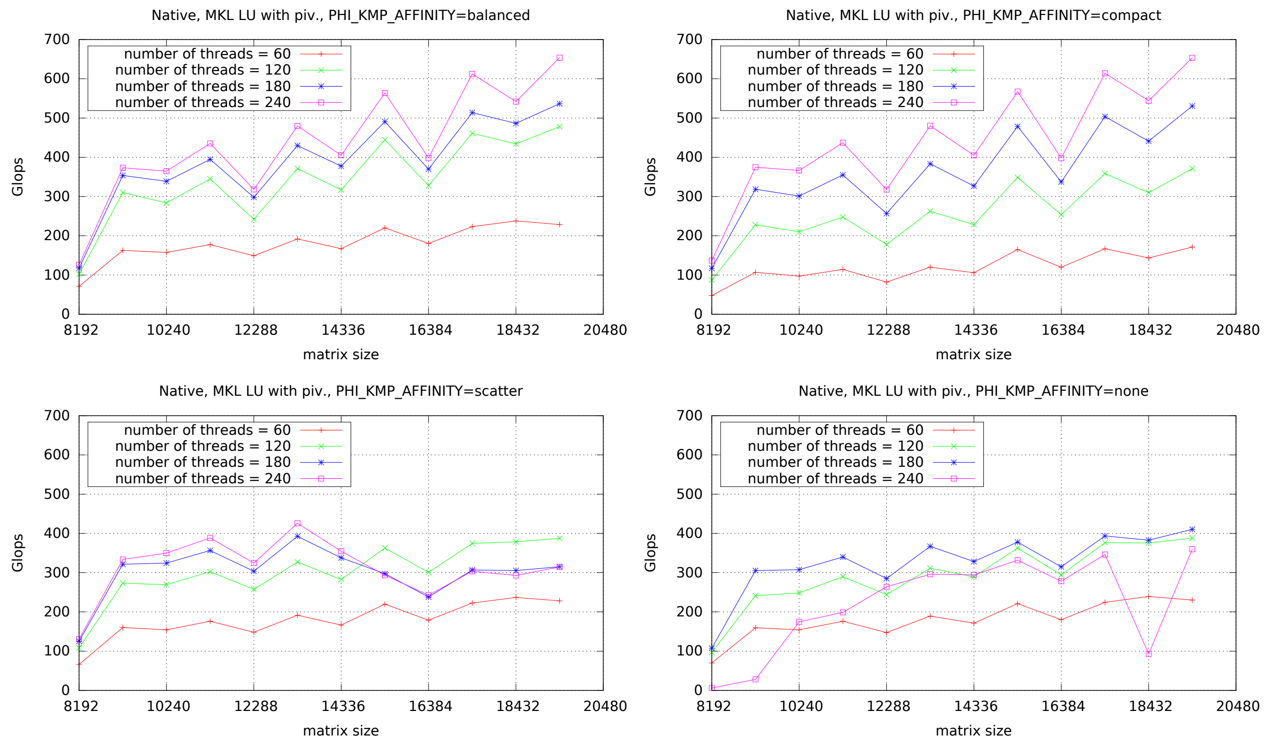

FIG. 6.4. *The performance of the LU factorisation with pivoting (MKL library's implementation) in the native mode on Intel Xeon Phi — for different matrix sizes, number of the threads, and the thread mapping settings.*

size 14336. The `none` affinity scales poor and is chaotic — just like for the version without pivoting, and for the same reason. For all affinity settings, we can see a saw shape of the chart — these spikes and drops are results of the relationship between the cache size and the size of the matrix.

Figure 6.5 presents the performance of the LU factorisation with pivoting in the function of matrix size on the hybrid CPU-MIC platform (with AO) for `KMP_AFFINITY=scatter` and 24 threads on CPU and the different values of `PHI_KMP_AFFINITY` on MIC. The algorithm scales very well with respect to both the size of the matrix and the number of threads. It seems that a lot of work is done on CPU (the report for this routine does not show the percentage work division, although, it shows the time used by both parts of the hybrid system — see

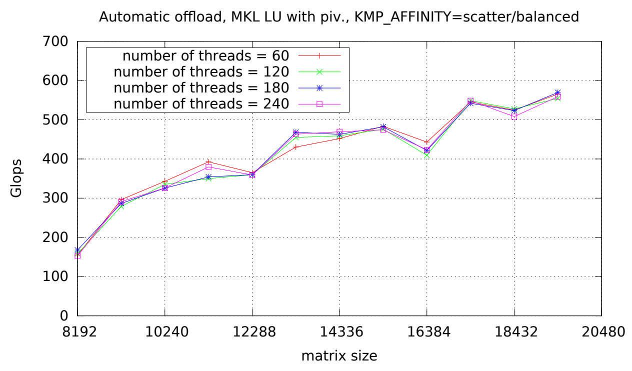

Fig. 6.5. *The performance of the LU factorisation with pivoting (MKL library's implementation) in the automatic offload mode — for different matrix sizes, number of the threads, and the thread mapping settings.*

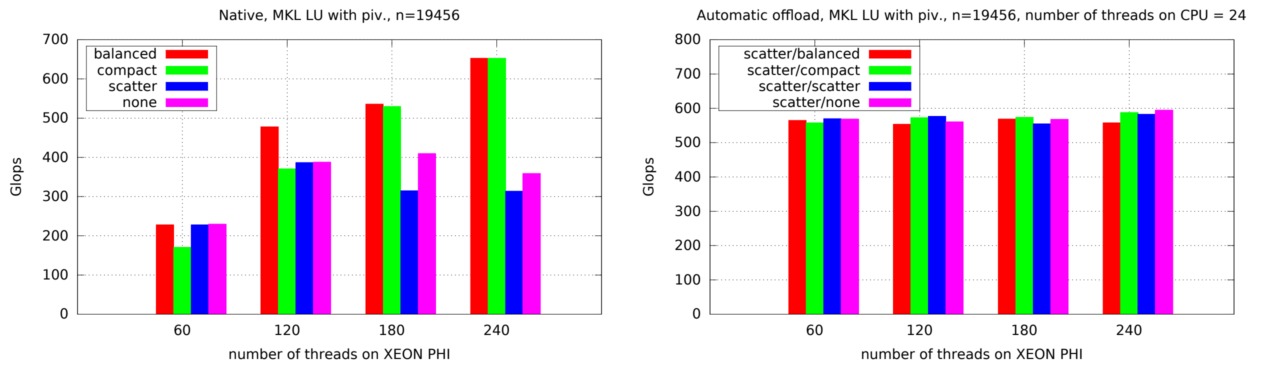

Fig. 6.6. *The performance of the LU factorisation with pivoting (MKL library's implementation) for the matrix size of 19456 on Intel Xeon Phi in the native mode (left) and on hybrid CPU-MIC Platforms in the automatic offload mode (right).*

Fig. 6.7; the time on CPU is much bigger).

Figure 6.6 shows the performance of the LU factorisation with pivoting in the function of the number of the threads for the matrix size of 19456 on Intel Xeon Phi in native mode and on hybrid CPU-MIC platform in automatic offload mode for `KMP_AFFINITY=scatter` on CPU and different values of `PHI_KMP_AFFINITY`. As we can see, the native mode version achieves the better performance than the automatic offload mode one. It seems that the former is very well optimised: it scales well with respect to both the number of threads and the matrix size. The AO version demands some more development, because (potentially) it could achieve even 1000 Gflops — taking into account combined forces of both the processing units.

```
[MKL] [MIC --]   [AO Function] DGEMM
[MKL] [MIC --]   [AO DGEMM Workdivision] 0.91 0.09
[MKL] [MIC 00] [AO DGEMM CPU Time] 0.275540 seconds
[MKL] [MIC 00] [AO DGEMM MIC Time] 0.170474 seconds
[MKL] [MIC 00] [AO DGEMM CPU->MIC Data] 33423360 bytes
[MKL] [MIC 00] [AO DGEMM MIC->CPU Data] 3932160 bytes
[MKL] [MIC --]   [AO Function] DTRSM
[MKL] [MIC --]   [AO DTRSM Workdivision] 0.97 0.03
[MKL] [MIC 00] [AO DTRSM CPU Time] 0.277691 seconds
[MKL] [MIC 00] [AO DTRSM MIC Time] 0.041793 seconds
[MKL] [MIC 00] [AO DTRSM CPU->MIC Data] 125829120 bytes
[MKL] [MIC 00] [AO DTRSM MIC->CPU Data] 7864320 bytes

[...9 analogous calls hidden...]

[MKL] [MIC --]   [AO Function] DGEMM
[MKL] [MIC --]   [AO DGEMM Workdivision] 0.91 0.09
[MKL] [MIC 00] [AO DGEMM CPU Time] 0.066654 seconds
[MKL] [MIC 00] [AO DGEMM MIC Time] 0.044188 seconds
[MKL] [MIC 00] [AO DGEMM CPU->MIC Data] 33423360 bytes
[MKL] [MIC 00] [AO DGEMM MIC->CPU Data] 3932160 bytes
```

```
[MKL] [MIC --]   [AO Function] DGETRF
[MKL] [MIC --]   [AO DGETRF Workdivision] -1.00 -1.00
[MKL] [MIC 00] [AO DGETRF CPU Time] 4.737727 seconds
[MKL] [MIC 00] [AO DGETRF MIC Time] 3.010779 seconds
[MKL] [MIC 00] [AO DGETRF CPU->MIC Data] 2796011520 bytes
[MKL] [MIC 00] [AO DGETRF MIC->CPU Data] 1242562560 bytes
```

FIG. 6.7. *Automatic offload reports generated by MKL's LU factorisation routines for the matrix size 15360 — without pivoting (top; fragments) and with pivoting (bottom; whole). The reports for MKL's QR and Cholesky factorisations are analogous to the one for MKL's LU factorisation with pivoting (bottom).*

6.3. Comparison of the pivot and non-pivot version. The peak performance of the AO mode is about 600 Gflops — the same for LU without pivoting and LU with pivoting. It arises from the fact that much more computations are performed on CPU (see Fig. 6.7) and both CPU versions seems equally optimised. However, in the native mode, LU without pivoting performs much worse (about 400 Gflops) than LU with pivoting (about 650 Gflops) — which is very surprising. Moreover, if we expected any differences, they would be in favour of the version without pivoting. However, from Fig. 6.7 we can see that the implementations of both factorisations are substantially different. It seems that they are very various algorithms, the pivot one being an implementation of the original LAPACK algorithm. Also, [16] says that this algorithm uses Intel Threaded Building Blocks and nothing like that is said about the non-pivot routine.

6.4. QR and Cholesky factorisations. Figures 6.8 and 6.9 present the performance of the QR and Cholesky factorisations in the function of matrix size on Intel Xeon Phi in native mode for the four values of `PHI_KMP_AFFINITY` for a different number of the threads.

Figures 6.10 nad 6.11 present the performance of the QR and Cholesky factorizationthe function of matrix size on the hybrid CPU-MIC platform (with AO) for `KMP_AFFINITY=scatter` and 24 threads on CPU and the different values of `PHI_KMP_AFFINITY` on MIC.

Figures 6.12 and 6.13 show the performance of the QR and Cholesky factorisations in the function of the number of the threads for the matrix size of 19456 on Intel Xeon Phi in native mode and on hybrid CPU-MIC platform in automatic offload mode for `KMP_AFFINITY=scatter` on CPU and different values of `PHI_KMP_AFFINITY`.

The performance results of the QR and Cholesky factorisations are consistent with the results of the LU factorisation with pivoting.

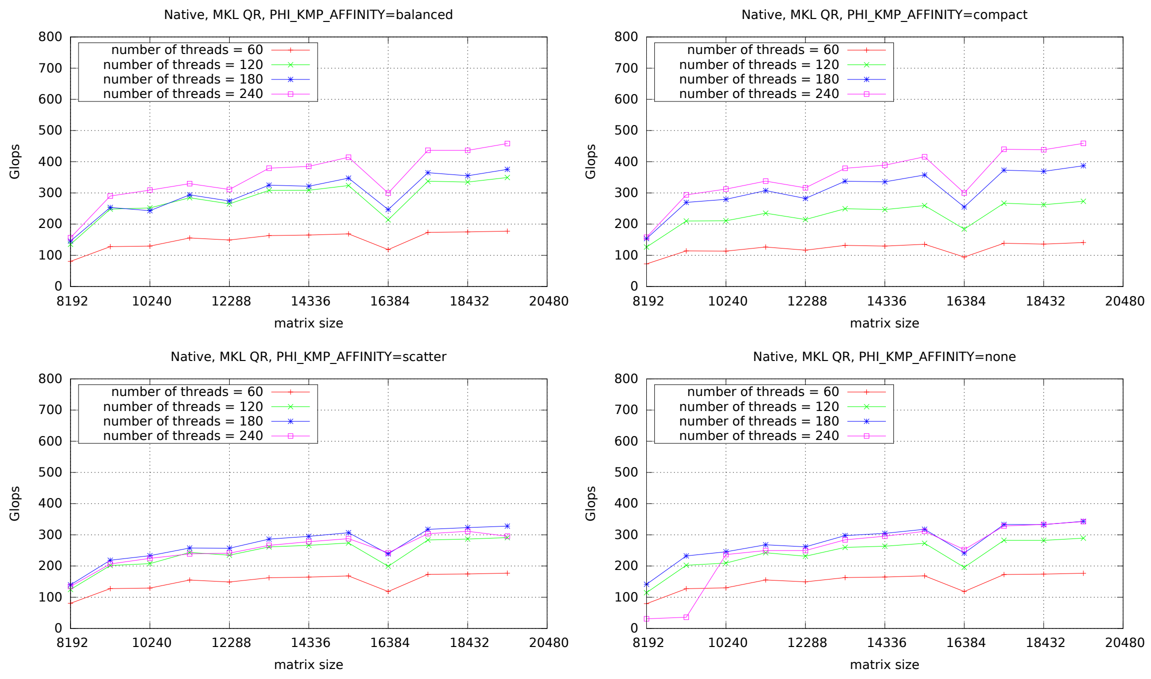

Fig. 6.8. *The performance of the QR factorisation (MKL library's implementation) in the native mode on Intel Xeon Phi — for different matrix sizes, number of the threads, and the thread mapping settings.*

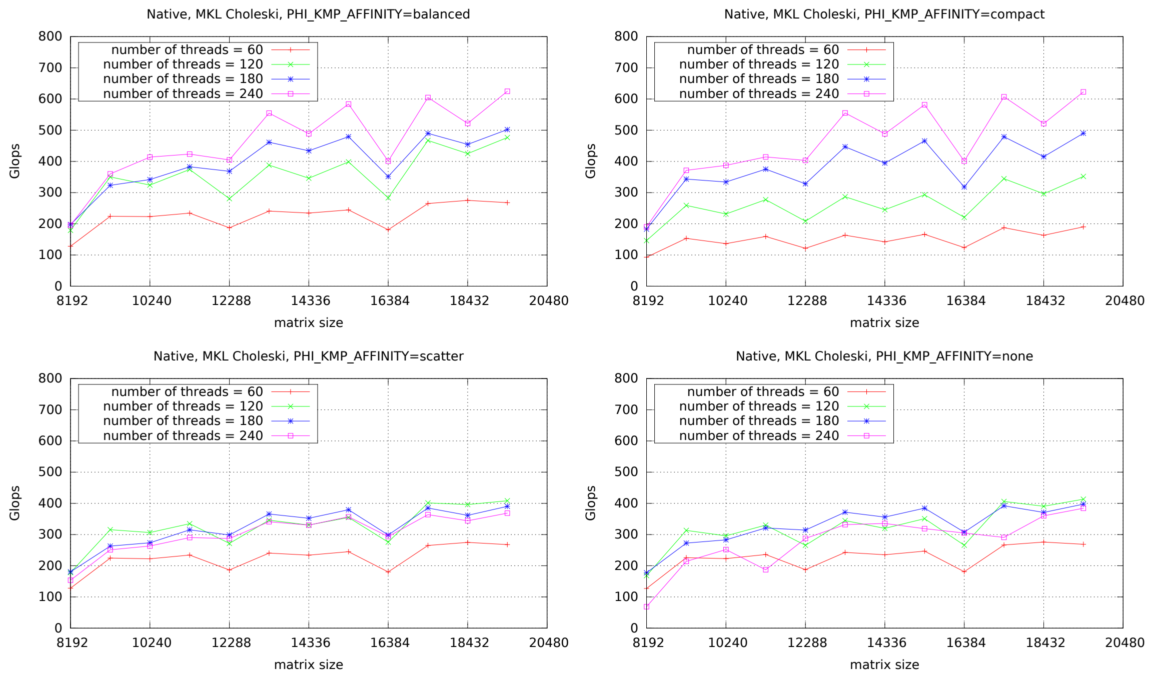

Fig. 6.9. *The performance of the Cholesky factorisation (MKL library's implementation) in the native mode on Intel Xeon Phi — for different matrix sizes, number of the threads, and the thread mapping settings.*

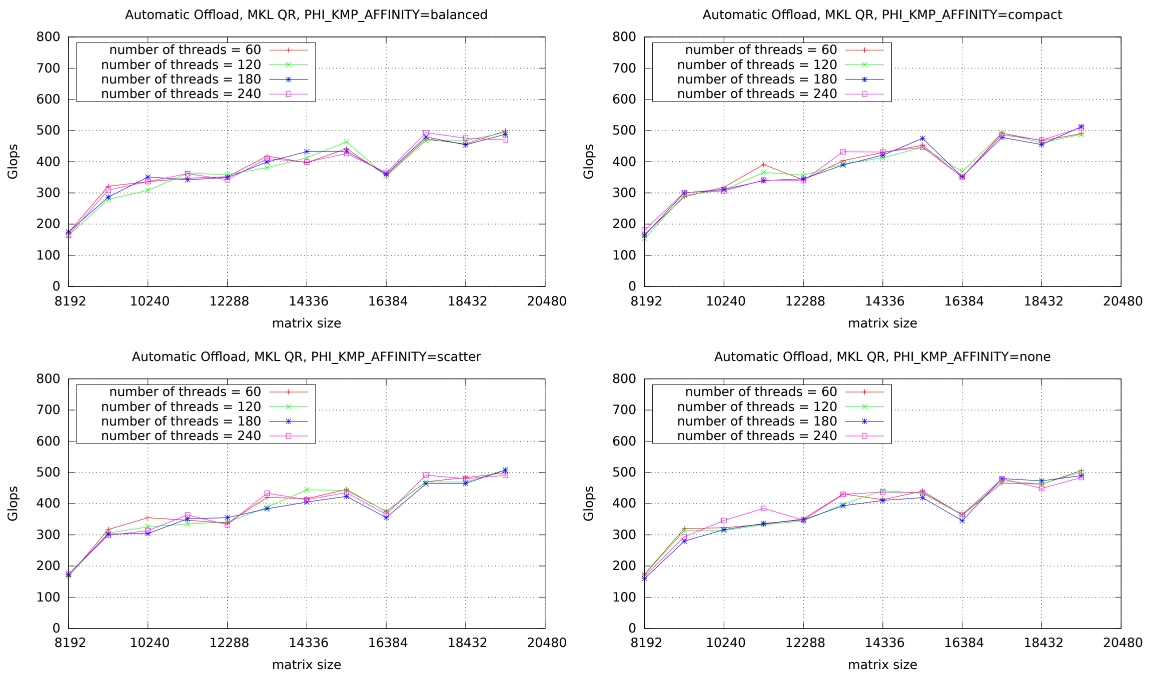

Fig. 6.10. *The performance of the QR factorisation with pivoting (MKL library's implementation) in the automatic offload mode — for different matrix sizes, number of the threads, and the thread mapping settings.*

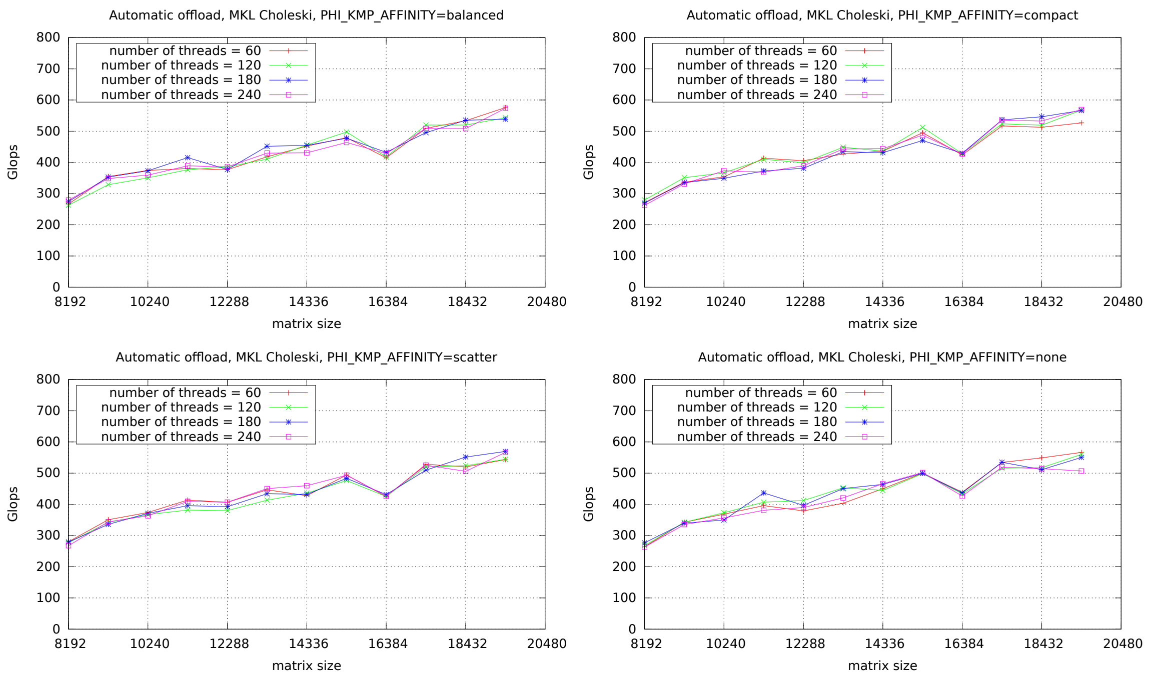

Fig. 6.11. *The performance of the Cholesky factorisation with pivoting (MKL library's implementation) in the automatic offload mode — for different matrix sizes, number of the threads, and the thread mapping settings.*

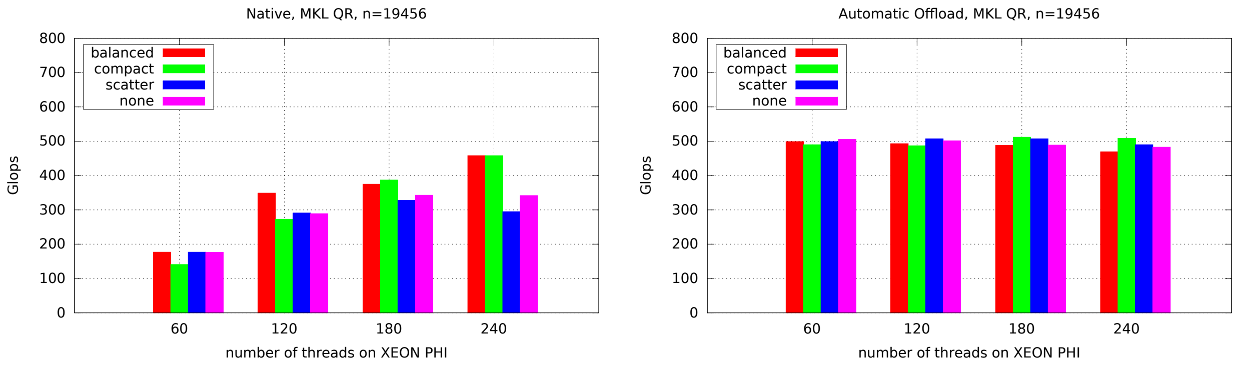

FIG. 6.12. *The performance of the QR factorisation (MKL library's implementation) for the matrix size of 19456 on Intel Xeon Phi in the native mode (left) and on hybrid CPU-MIC Platforms in the automatic offload mode (right).*

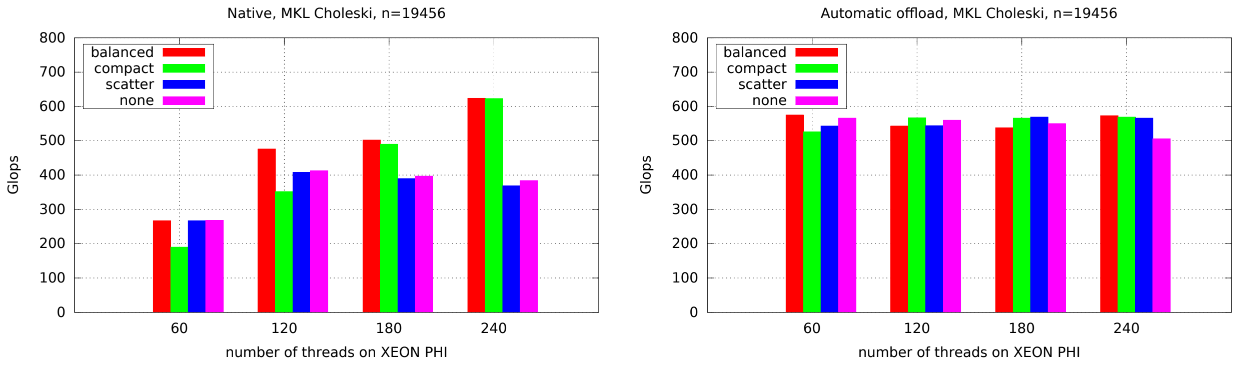

FIG. 6.13. *The performance of the Cholesky factorisation (MKL library's implementation) for the matrix size of 19456 on Intel Xeon Phi in the native mode (left) and on hybrid CPU-MIC Platforms in the automatic offload mode (right).*

7. Discussion.

7.1. Thread Mapping. It is obvious from the experiments that the proper setting of the thread mapping improves the performance. Moreover, the performance of the three factorisations is sensitive to the thread mapping. The best setting is `balanced` (see also Sect. 5), for all the tested number of threads and modes. The `balanced` thread affinity is the best because it uses well the system computing power and the cache at the same time. The `none` setting is the worst and largely unpredictable (because all the decisions are passed to the operating system and the threads can wonder freely between cores) and it should not be used in serious computational applications. The `scatter` affinity gives the second worst performance. All the performance results of the LU factorisation with pivoting, as well as QR and Cholesky facorisations (especially in native mode) confirm our analysis from Sect. 5.

7.2. Number of threads. All the factorisations effectively utilize a large number of cores in the native mode. Thus, it is the best to use all the cores with hyperthreading (that gives 240 threads, that is, 4 threads per core). That way, we use the computing power of the Intel Xeon Phi the most efficiently. On the other hand, the number of the threads has no impact on the performance in the automatic offload mode at all — the AO mode is controlled by the library.

7.3. Mode. If we can afford the native mode, we should rather use it — the native mode is better optimised than the automatic offload for the LU and Cholesky factorisations with pivoting. On the contrary, the AO is better than the native mode for the QR factorisation. It shows that this factorisation could be more optimised for the MAC architecture.

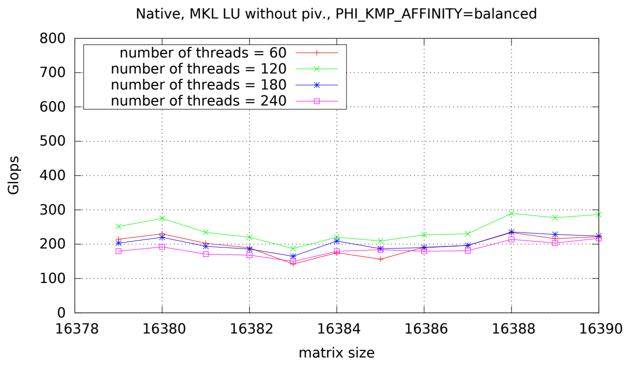

FIG. 7.1. *The performance of the LU factorisation without pivoting (MKL library's implementation) in the native mode on Intel Xeon Phi — for different matrix sizes, number of the threads, and the* `balanced` *thread mapping setting (cache associativity).*

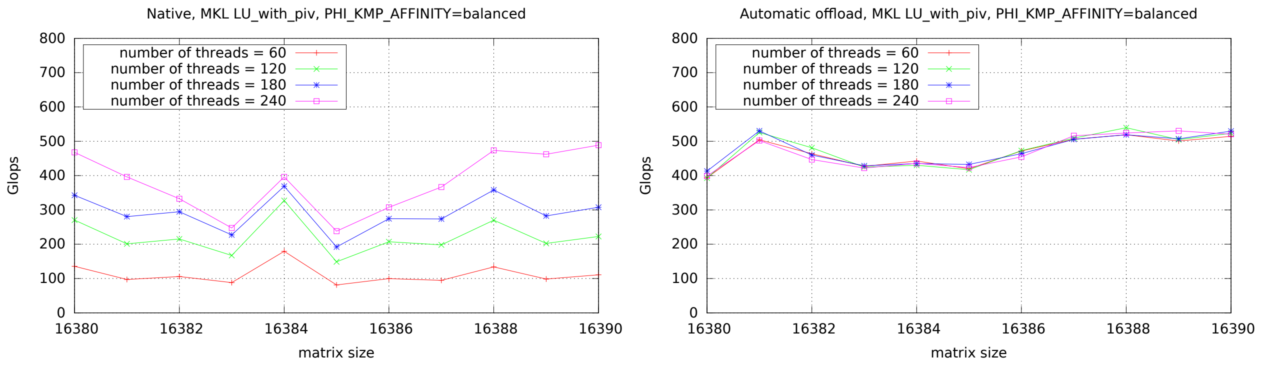

FIG. 7.2. *The performance of the LU factorisation with pivoting (MKL library's implementation) in the native mode (left) and in the automatic offload mode (right) on Intel Xeon Phi — for different matrix sizes, number of the threads, and the* `balanced` *thread mapping setting (cache associativity).*

The LU factorisation with pivoting is the most highly optimised of these three factorizations in native mode.

On the other hand, the LU factorisation without pivoting performs completely differently than three other factorisations — and it is caused by the fact that it works (and was written) completely differently — which is proven by Fig. 6.7.

7.4. Cache associativity. To test the influence of the cache associativity on the performance we should investigate the behaviour of the subject algorithms for the matrix of the size around 8192×8192 — because 8192 double-precision floats occupies 32 kB which is the size of the L1 cache. However, for this size, the algorithms do not utilize all the computing power (they enter the automatic offload mode only for significantly larger matrices). On the other hand, we got some performance drop around the size 16384×16384 (and 16384 double-precision floats is twice the size of the L1 cache), and that is why we decided to take a look at sizes around this number. In this manner we can investigate the cache associativity.

All the tests for cache associativity were performed only for the `balanced` thread affinity, as it proved the best for tested algorithms.

7.4.1. LU without pivoting. Figure 7.1 shows the performance of the LU facorisation without pivoting for the matrix sizes about 16384. We present only the native mode, because — as we see in Figure 6.2 — in the automatic offload mode, there is no efficiency drop around this size. The figure shows that there is a performance drop around 16384 (although the number itself is a weak local maximum). The performance minimum does not have to be precisely at the multiple of cache size, because there are some more auxiliary variables, but it is clearly visible for all the thread mappings.

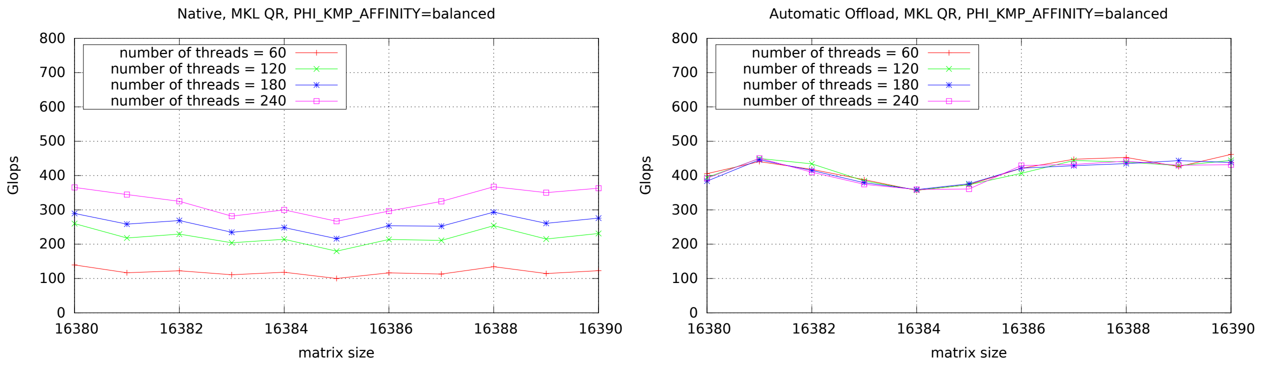

FIG. 7.3. *The performance of the QR factorisation (MKL library's implementation) in the native mode on Intel Xeon Phi (left) and in the automatic offload mode (right) — for different matrix sizes, number of the threads, and the* `balanced` *thread mapping setting (cache associativity).*

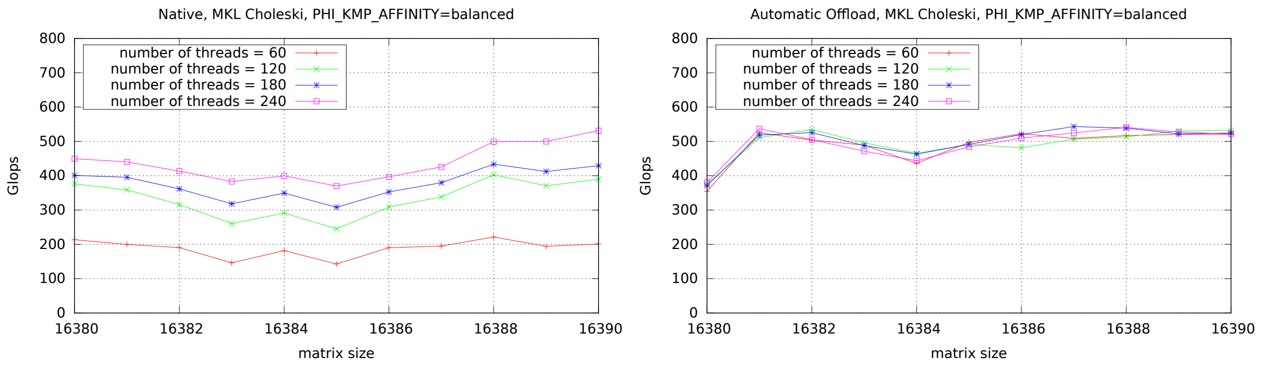

FIG. 7.4. *The performance of the Choleski factorisation (MKL library's implementation) in the native mode on Intel Xeon Phi (left) and in the automatic offload mode (right) — for different matrix sizes, number of the threads, and the* `balanced` *thread mapping setting (cache associativity).*

7.4.2. LU with pivoting. Figure 7.2 shows the performance of the LU facorisation with pivoting for the matrix sizes about 16384. However, here we present both the native mode and the automatic offload mode. In the native mode, there is an efficiency drop around 16384, but again, in the 16384 we have a clear although local maximum. On the other hand, in the automatic offload mode we can see almost a flat line around 16384, which is lower than the neighbourhood. Again, all the thread mappings behave similarly.

7.4.3. QR and Cholesky. Figures 7.3 and 7.4 show the behaviour of the QR and Cholesky (respectively) factorisations around the size 16384 in both modes (left: native, right: automatic offload). The plots are quite similar to the respective plots for the LU factorisation with pivoting, although the local maximum in 16384 in the native mode is very slight. So, the cache associativity is shown in both modes, independent of the thread mapping.

8. Conclusion. The paper reports the effect of thread-mapping for the LU (without and with pivoting), QR and Cholesky factorisations from MKL library on Intel Xeon Phi and the hybrid CPU-MIC platform. Our results showed that there is one thread mapping strategy adapted for all optimised factorisation on Xeon Phi, namely `balanced`. Determining the most efficient OpenMP thread mapping depends highly on the number of thread and it sets the system load. It is surprised that the performance of MKL's `dgetrf` (LU factorisation with pivoting) is much better than MKL's `dgetrfnpi` (LU factorisation without pivoting) on KNC in native mode. This situation indicates that Intel does not optimise `dgetrfnpi` for KNC. However, it should be very easy for them to make optimised `dgetrfnpi`, by just removing the pivoting code from `dgetrf`. In the native mode, the LU with pivoting, QR and Cholesky factorisations are scalable on Intel Xeon Phi but the LU factorisation without pivoting is not. The comparison given here gives good insight into the performance properties of the different factorisation algorithms on Intel Xeon Phi and hybrid CPU-MIC platform. These results can be generalised as the paper gives the performance analysis of some other similar algorithms (namely, the QR and Cholesky factorisations). In future works, the authors plan to research the impact of the thread mapping on the performance and the energy saving for other applications from the domain of the dense linear algebra on shared memory multicore and manycore architectures and to compare it with the results obtained in this work.

REFERENCES

[1] Anderson E., Bai Z., Bischof C., Blackford S., Demmel J., Dongarra J., Du Croz J., Greenbaum A., Hammarling S., McKenney A., Sorensen D.: LAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, (1999)

[2] Buttari A., Langou J., Kurzak J., Dongarra J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Computing, 35(1):38–53, (2009)

[3] Bylina B., Bylina J.: OpenMP Thread Affinity for Matrix Factorization on Multicore Systems, Proceedings of the 2017 Federated Conference on Computer Science and Information Systems, Annals of Computer Science and Information Systems 11, 489–492, (2017)

[4] Demmel J. W.: Applied Numerical Linear Algebra. SIAM, (1997)

[5] Diener M., Cruz E. H. M., Alves M. A. Z., Navaux M. A. Z., Koren I.: Affinity-based thread and data mapping in shared memory systems. ACM Comput. Surv., 49(4):64:1–38, (2016)

[6] Donfack S., Dongarra J., Faverge M., Gates M., Kurzak J., Luszczek P., Yamazaki I.: A Survey of Recent Developments in Parallel Implementations of Gaussian Elimination Concurrency and Computation, Practice and Experience, 27:1292–1309, (2015)

[7] Dongarra J., DuCroz J., Duff I. S., Hammarling S. : A set of level-3 Basic Linear Algebra Subprograms. ACM Trans. Math. Software, **16**: 1–28, (1990)

[8] Dongarra J., Gates M., Haidar A., Jia Y., Kabir K., Luszczek P., Tomov S.: Portable hpc programming on intel many-integrated-core hardware with magma port to xeon phi. In PPAM 2013, Warsaw, Poland, (2013)

[9] Dumas J. G, Gautier T., Pernet C., Roch J. L, Sultan Z.: Recursion based parallelization of exact dense linear algebra routines for Gaussian elimination. Parallel Computing, 57:235–249, (2016)

[10] Eichenberger A. E, Terboven Ch, Wong M., Mey D.: The design of openmp thread affinity. In Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World, IWOMP'12, Berlin, Heidelberg, 15–28 (2012)

[11] Heinecke, et al.: Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems, https://software.intel.com/en-us/articles/design-and-implementation-of-the-linpack-benchmark-for-single-and-multi-node-systems-based, IPDPS 2013 (2013).

[12] Ju T., Zhu Z., Wang Y., Li L., Dong X.T.: Thread Mapping and Parallel Optimization for MIC Heterogeneous Parallel Systems. In: Sun X. et al. (eds) Algorithms and Architectures for Parallel Processing. ICA3PP 2014. Lecture Notes in Computer Science, vol 8631. Springer, 300–311, (2014).

[13] Rahman R.: Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers, 1st edn. Apress, Berkely, CA, USA (2013)

[14] erboven Ch., Mey D., Schmidl D., Jin H., Reichstein T.: Data and thread affinity in openmp programs. In Proceedings of the 2008 workshop on Memory access on future processors: a solved problem? (MAW '08). ACM, New York, NY, USA, 377-384, (2008)

[15] Intel Corporation, Intel Math Kernel Library (MKL), (2017)

[16] https://software.intel.com/en-us/articles/intel-mkl-113-release-notes

Edited by: Sasko Ristov
Received: Mar 1, 2018
Accepted: August 24, 2018