Parallelization of NAS NPB3.3.1 tests on Fortran-DVMH for Intel Xeon Phi coproccessor [1]

V.F. Aleksahin, V.A. Bakhtin, O.F. Zhukova, A.S. Kolganov, V.A. Krukov,
I.P. Ostrovskaya, N.V. Podderugina, M.N. Pritula, O.A. Savitskaya

Keldysh Institute of Applied Mathematics RAS

The article analyzes the performance of NAS benchmarks from NPB 3.3.1 package (EP, MG, BT, SP, LU) on cluster nodes of different architecture using multi-core universal processors, NVidia graphics accelerators and Intel coprocessors. Characteristics of the tests developed in high-level Fortran-DVMH language (hereafter referred to as FDVMH), and their implementations in other languages are compared. The impact of different optimizations of FDVMH NAS benchmarks necessary for their effective work on Intel Xeon Phi coprocessor is researched. The results of their execution with simultaneous use of all cores of CPU, GPU and Intel Xeon Phi coprocessor are presented.

1. Introduction

In recent years more and more computational clusters whose nodes contain attached accelerators of different types in addition to universal multi-core processors are emerged. Most of them are Nvidia graphics processors and Intel Xeon Phi coprocessors. In the Top500 list [1] of the most powerful supercomputers of the world published in November, 2014, 75 computers have accelerators, 50 computers from them have NVIDIA accelerators, 25 – Intel. The combination of NVidia and Intel Xeon Phi accelerators is used in four computers. This tendency significantly complicates the process of cluster programming due to the requirement to know good several programming models and languages at once. Traditional approach is to use MPI technology for job distribution between cluster nodes, and then to use OpenMP, CUDA, OpenCL or OpenACC technologies to load all the cores of central and graphics processors. Therefore to develop the program for a supercomputer it is necessary to know exactly its architecture.

To simplify programming of distributed computing systems high-level programming languages based on the extension of standard languages by directives, such as HPF [2], Fortran-DVM [3,4], C-DVM [3,5] have been proposed. Programming models and appropriate directive-based extensions for programming languages such as OpenACC [5] и OpenMP 4.0 [6] have been also proposed to simplify exploiting of accelerators.

The parallelization of the loops without dependencies on graphic processors (GPUs) and coprocessors does not usually expose great ideological problems, whether it would be manual parallelization or with use of high-level tools because GPU's array-parallel architecture is well suitable for processing. When the loops with dependencies are parallelized the following problems arise: limited support of synchronization of execution streams on GPU, model of GPU global memory consistency, necessity to synchronize OpenMP threads on the coprocessor for pipeline organization.

2. Review of parallel architectures

In this chapter the following architectures are considered: central processor on the example of Intel Ivy Bridge-EP [7], Xeon Phi coprocessor (MIC – Many Integrated Cores) [8] on the example of Knights Corner and Nvidia Kepler GPU [9] on the example of GK110.

[1] The research was supported by grants of Russian Foundation for Basic Rsearch No. 13-07-00580, 14-01-00109, 14-07-31321_mol_ and Presidium of Russian Academy of Sciences, Programs №15, №16 and №18.

2.1. Архитектура Ivy Bridge-EP architecture

In the most complicated modification of this architecture three units including four CPU cores and a portion of third level cache memory of 2.5 MB per core are used. The doubled ring bus provides an interaction between units inside a chip, and multiplexers allow to pass the commands to the core they are addressed to. The external bus QPI (Quick Path Interconnect) used for the processor interconnection and connection with chipset works at the speed up to 9.6 GT/sec. The built-in PCI Express controller provides functioning of 40 lines of third generation. Two memory controllers are provided in 12-core chip, each of them supports work of memory up to DDR3-1866 in dual-channel mode.

There are SSE (Streaming SIMD Extensions) registers and AVX (Advanced Vector Extensions) registers at this architecture. SSE includes in processor architecture eight 128-bit registers and a set of instructions for operating with scalar and packed data types. Advantage in performance is reached in the case if it is necessary to perform the same sequence of operations with different data. The computational process is parallelized between data by SSE unit. AVX provides different improvings, new instructions and new encoding scheme of machine codes:

- Width of SIMD vector registers is increased from 128 to 256 bits. The existing 128-bit SSE instructions use a low half of new 256-bit registers, without changing a high part. For operating with these registers new 256-bit AVX instructions are added. It is possible to extend SIMD vector registers up to 512 or 1024 bits in the future;
- There are no requirements to align operands in the memory for most of new instructions. However it is recommended to oversee an alignment with the operand size, in order to avoid the considerable performance decrease;
- The set of AVX instructions contains the analogs of 128-bit SSE instructions for real numbers. But unlike originals, saving of 128-bit result will nullify the high half of 256-bit register. 128-bit AVX instructions keep other advantages of AVX, such as new coding scheme, three-operand syntax and not aligned access to the memory.

2.2. Knights Corner architecture

Up to 61 x86 cores with large 512-bit vector modules (containing 512-bit AVX registers) can be integrated in the coprocessor of this architecture. They have more than 1 GHz frequency and provide the speed of double precision computations more than 1 TFlops. They are located on two-slot PCI Express card with a special firmware on Linux base. Intel Xeon Phi contains up to 16 Gbytes of GDDR5 memory. Certainly, so constructed coprocessors aren't designed for processing of main tasks with which the processors of Xeon E family cope. They succeed in parallel tasks capable to use a large number of cores for maximum effect. Main characteristics are:

- X86 architecture with support of 64 bits, four threads per a core and up to 61 cores per coprocessor;
- 512-bit AVX instructions, 512 Kbytes of L2 cache per a core (up to 30,5 MB per whole Xeon Phi card);
- Linux support (special version for Phi), up to 16 Gbytes of GDDR5 memory per card.

It is possible to note that even the highest Intel Xeon Phi model has much less cores, than usual graphic processor. But it is impossible to compare MIC core with CUDA core in one to one ratio because one core of Intel Xeon Phi is four-thread module with 512-bit SIMD.

2.3. GK110 architecture

Kepler GK110 chip was developed first of all to expand Tesla model range. Its goal is to become the most fast parallel microprocessor in the world. GK110 chip not only exceeds in

performance the Fermi chip of previous generation, but also consumes much less energy. GK110 in the maximum configuration consists of 15 stream multiprocessors called SMX, and six 64-bit memory controllers.

Each stream SMX multiprocessor contains 192 cuda-cores for single precision operations, 64 cuda-cores for double precision operations, 32 special functional units and 32 units for data loading and saving, four schedulers. For all SMX on GPU one common second level cache of 1.5 MB size is provided. Also each SMX has its own first level cache of 64 KB size.

To use large capacities of GPU, it is necessary to map computationally independent code fragments on the groups of independent virtual threads. Each group will execute the same command with different input data. To control the group of virtual threads it is necessary to integrate them in the blocks of fixed size. These blocks in CUDA architecture are called cuda-blocks.

Such block has three dimensions. All blocks are integrated in a grid of blocks which can also have three dimensions. Eventually, each cuda-block will be processed by some stream multiprocessor, and each virtual thread will be matched with physical one. Minimum unit of parallelism on GPU is warp – the group of 32 independent threads which are physically executed in parallel and synchronously.

3. Review of NAS betchmarks

NAS Parallel Benchmarks [10] are suite of benchmarks allowing to estimate performance of supercomputers. The benchmarks were developed and are supported in NASA Advanced Supercomputing (NAS) Division (earlier NASA Numerical Aerodynamic Simulation Program) in NASA Ames Research Center. The NPB 3.3 package consists of 11 tests. The parallelization of EP, MG, BT, LU, SP tests on the CPU and coprocessors in DVMH [11] model is considered in the article. Parallel versions for clusters with use of Fortran-DVM language, and also parallel versions for graphic processors with use of Fortran-DVMH language were developed earlier. The tests are:

- MG (Multi Grid) – the test calculates approximated solution of three-dimensional partial differential Poisson equation ("three-dimensional grid") on NxNxN grid with periodic boundary conditions (the function is equal 0 on the boundary except of given 20 points). The grid size N is determined by the test class. It tests possibilities of the system to perform both long and short data transfers.
- EP (Embarrassingly Parallel) - the test calculates an integral by Monte-Carlo method. It is intended for measurement of primary computational capacity of floating arithmetics. This test can be useful if the problems using Monte-Carlo method will be solved on a cluster. The algorithm also takes into account the time of data formatting and output.
- BT (Block Tridiagonal) – block three-diagonal scheme. The test solves synthetic system of non-linear differential equations in partial derivatives (three-dimensional system of Navier-Stokes equations for compressible fluid or gas), using the block three-diagonal scheme with method of alternating directions.
- SP (Scalar Pentadiagonal) – scalar pentadiagonal. The test solves synthetic system of non-linear differential equations in partial derivatives (three-dimensional system of Navier-Stokes equations for compressible fluid or gas), using scalar penta-diagonal scheme.
- LU (Lower-Upper) – decomposition by symmetric method of Gauss-Seidel. The test solves synthetic system of non-linear differential equations in partial derivatives (three-dimensional system of Navier-Stokes equations for compressible fluid or gas), using a method of symmetric successive over relaxation.

4. Mapping of programs on different architectures in DVM system

In 2011 in Keldysh Institute of Applied Mathematics of RAS high level programming model for support of clusters with graphic accelerators was developed. The model was called DVMH [11] (DVM for Heterogeneous systems). It is extension of DVM model and allows to convert DVM program for a cluster to DVMH program for the cluster with graphics accelerators with little changes.

In 2014 due to appearance of Intel Xeon Phi coprocessors it became necessary to support them by DVM system.

There are three modes of program execution on Xeon Phi [11] coprocessor:

- Native mode – the program is compiled and executed only on the coprocessor;
- Offload mode – the program is compiled for CPU, some code fragments are compiled both for CPU and Xeon Phi. These code fragments are marked by special OpenMP 4.0 directives (generally #pragma offload). The program is executed as well as in case of GPU use: a sequential part – on CPU, and parallel part with special directives – on the coprocessor. At this mode the same problem, as if GPU is used, arises – data loading and uploading in own memory of the coprocessor through PCIe. If there is no the coprocessor in a node, special code fragments will be executed on CPU.
- Symmetric the mode – the same program is compiled separately for CPU and separately for the coprocessor. The compiled programs are launched simultaneously on the coprocessor and CPU and can be synchronized by means of MPI technology.

The main mode of execution selected for implementation in DVMH is symmetric mode. This mode allows to tune balance between CPU and the coprocessor using MPI technology and to use OpenMP technology for the best loading of the cores inside each device. It is also possible to launch FDVMH program only on the coprocessor (native mode), using, for example, one MPI process and maximum quantity of OpenMP threads supported by the coprocessor.

As a result, parallelizing programs in DVMH model, it isn't necessary to select this or that architecture, whether it be graphic processors, central processors or Intel Xeon Phi coprocessors as this model supports usage of all listed architectures both separately, and simultaneously for the same program.

5. Implementation of Xeon Phi support in FDVMH compiler

The computational region in DVMH program is a fragment of the program with one entrance and one exit for possible execution on one or several computational devices. The region may contain several parallel loops. This program fragment is marked by REGION directive.

The region can be executed on one or several devices: accelerators, CPU, coprocessors. Any region can be executed on CPU or on the coprocessor. For execution of same region on different accelerators different additional restrictions can exist for the region. For example, on CUDA device any region which hasn’t input/output statements or calls of external procedures can be executed.

It is possible to specify type of the calculator where a region should be executed. The TARGETS specification is intended for this purpose. Enclosed (statically or dynamically) regions aren't allowed. DVM arrays are distributed on selected calculators (taking into account weights and high-speed performance of calculators), undistributed data are replicated. The iterations of parallel DVM loops inside the region are split between the calculators

selected for the region execution according to the rule of parallel loop mapping, specified in the parallel loop directive.

5.1. Generation of handlers for parallel loops

Fortran-DVMH compiler converts source program to the parallel program in Fortran language with function calls of DVMH runtime system (RTS – RunTime System). Besides, the compiler creates for each source program several additional modules to support execution of the program regions on GPU using CUDA technology and on CPU and coprocessor using OpenMP technology.

For each parallel loop from computational region the compiler generates procedure-handler and a kernel for computations on GPU and also procedure-handler for execution of this parallel loop on CPU and coprocessor. The handler is a subprogram processing parallel loop part on certain computational device. Arguments of the handler are the device descriptor and the parallel loop part.

The handler requests from RTS a portion for execution (the loop boundaries and a step), a configuration of parallel processing (quantity of threads), an initialization of reduction variables and other system information. To execute the parts of the loop CUDA handler calls CUDA kernel, generated in special way, with parameters, received from RTS during execution. CUDA kernel is executed on the graphic processor, performing computations of the loop body. To process the loop parts CPU handler uses OpenMP technology for distribution of computations. It is supposed by default that the computational region can be executed on architectures of all types which exist at the cluster node, and the compiler generates handlers for CPU, the coprocessor and CUDA device.

One of the main reasons of deceleration of the programs executed on CPU or on the coprocessor is a linearization of distributed arrays performed by Fortran-DVMH compiler. The memory for elements of such arrays is allocated by RTS system. For local section of the array the memory is allocated on each processor according to data distribution format and taking into account shadow edges. The compiler replaces ARRAY (I, J, K) references to elements of distributed arrays by the following references:

*BASE(ARRAY_OFFSET+I+C_ARRAY1*J+C_ARRAY2*K),*

where BASE – a base for addressing of all distributed arrays; ARRAY_OFFSET – offset of the array beginning relative to the base; C_ARRAYi – coefficients of addressing of the distributed array. Such program is worse recognized and optimized by standard Fortran compilers.

To solve this problem FDVMH compiler by special option can generate the optimized handlers in order to distributed arrays will be passed to host handlers as assumed shape arrays. Such approach allows not to convert the references to the arrays in linear form inside host handlers and essentially increases the program performance on CPU or coprocessor.

5.2. Use of collapse clause

Unlike central processors, which have no more than 24 threads (for newest Intel Xeon E), Intel Xeon Phi coprocessors can have 244 threads. There are a lot of tasks having not only one-dimensional loops, but also two-dimensional and three-dimensional. Many of NPB Benchmarks [9], in particular, are such tasks. OpenMP directive *!$OMP PARALLEL FOR*, generated by FDVMH compiler in host handlers, is applied only to the most upper loop of the nest. If a number of iterations of such loop is less, than a quantity of available threads, then performance will be decreased.

To optimize the work of host handlers on Xeon Phi it is necessary to generate an additional COLLAPSE clause in OpenMP directive. COLLAPSE clause allows to distribute

execution of several loops of the nest that allows to use all cores of the coprocessor. The parameter of this clause is a quantity of nested loops to which it is applied. To apply this clause to all nested loops is not efficiently as the same thread will process the elements not lying in a row and in this case there will be more misses in L2 cache.

Static analysis during DVMH program compilation can't determine a quantity of nested loops to which COLLAPSE clause can be applied. Therefore during compilation several host handlers are generated. In each of them COLLAPSE (N) clause is inserted, where N = 1, 2, 3, … , *m,* and m is the quantity of loops in the nest. During the program execution RTS selects the handler which will execute the parallel loop.

A user can set the option *–collapsN* for DVMH compiler, where N is integer positive number. Then COLLAPSE(N) clause will be added in OpenMP directive for each parallel loop in the host handler.

5.3. Loading balance in DVM system

One of the important aspects of DVMH program model functioning is the problem to map a source program on all parallelism levels and heterogeneous computational devices. The important tasks of mapping mechanism are a support of correct execution of all constructions provided by language on heterogeneous computational devices, loading balance between computational devices, and also selection of optimum method of each code fragment execution on this or that device. There are several levels of parallelism in DVMH programs:

- Distribution of data and computations on MPI processes. This level is specified by data distribution and redistribution directives and by specifications of parallel subtasks and loops. At this level the program is mapped on cluster nodes. Several MPI processes can exist inside each node, and they can be mapped on CPU cores or coprocessor cores;
- Distribution of data and computations on computational devices at the entrance into computational region.
- Parallel processing within computational device. This level appears at entrance into parallel loop inside computational region. At this level the computations are mapped on OpenMP threads and CUDA architecture.

According to this division in DVMH model there are two levels of balancing between computational devices: specifying of weight of each MPI process mapped on CPU or coprocessor cores, and specifying of a ratio of computational capability between cores of CPU and GPU for each MPI process.

To tune DVMH program performance its recompilation isn't required. To define a ratio of MPI processes, user should specify a vector of weighs in the special file with parameters of DVMH program startup. According to specified weights, RTS will separate data between MPI processes during the program execution. By default all MPI processes are identical.

To balance loading between CPU and GPU the following mechanisms exist. For all settings environment variables are used. The user can define required amount of threads on CPU (or on coprocessor) in OpenMP terms. It is also possible to turn on or turn off a parallel loop execution on one or several GPUs. To specify CPU and GPU performance two environment variables are used:

- DVMH_CPU_PERF – relative performance of CPU. There is a possibility to specify different values of this parameter for different MPI processes executed on CPU cores or on Intel Xeon Phi coprocessor cores.
- DVMH_CUDAS_PERF – relative performance of GPU. It is set by the list of real numbers through a space or a comma. The list is circular. It allows not to specify performances of all GPUs.

Thus, DVM system allows to use efficiently all devices of different architecture attached to cluster nodes by two-level balancing: specifying of each MPI process weight and

specifying of performance a ratio between OpenMP threads and CUDA devices inside each of MPI processes.

6. Applying of AVX instructions

Minimum parallelism unit in terms of GPU is warp consisting of 32 independent threads. When applied programmer writes a parallel program using CUDA program model he himself defines the warps, integrates them in blocks, and then in grids of blocks, thereby specifying where and what will be executed in parallel and vectorially. It is possible to consider that the minimum size of a vector is equal to 32 elements in GPU architecture. Further problems of optimization, planning and data loading/uploading will be solved by the NVidia compiler and GPU hardware.

Minimum parallelism unit of one core of CPU or coprocessor is AVX vector register which can process 256 or 512 bit of data per one operation correspondently. In this case the applied programmer works at first with a serial code of the program, and then applies high level OpenMP directive extensions. And unlike GPU, in the program oriented on the CPU or the coprocessor there are no explicit specifications of vector operations – the compiler performs all vectorization job. Therefore there are two methods of efficient parallel program writing using AVX vector registers:

- to use the compiler or OpenMP directives (for example, *"omp simd"*);
- to use low level commands or AVX instructions (at the level of intrinsic-functions).

The first method is realized not always because the compiler not always can perform vectorization even if it is possible. Let's consider two versions of the same loop (see Fig. 1).

```
#if VER1 /*** first version of the loop ***/
  double rhs[5][162][162][162], u[5][162][162][162];
  #define rhs(m,i,j,k) rhs[(m)][(i)][(j)][(k)]
  #define u(m,i,j,k)     u[(m)][(i)][(j)][(k)]
#endif
#if VER2 /*** second version of the loop ***/
     double rhs[162][162][162][5], u[162][162][162][5];
     #define rhs(m,i,j,k) rhs[(i)][(j)][(k)][(m)]
     #define u(m,i,j,k)     u[(i)][(j)][(k)][(m)]
#endif
#pragma omp parallel for
for(int i = 0; i < Ni; i++)
{
     for(int j = 0; j < Nj; j++)
     {
        for(int k = 0; i < Nk; k++)
        { /*** first group of statements ***/
          rhs(0, i, j, k) = F(u(0, i, j, k), u(0, i+1, j, k));
          rhs(1, i, j, k) = F(u(1, i, j, k), u(1, i-2, j, k));
          rhs(2, i, j, k) = F(u(2, i, j, k), u(2, i, j+1, k));
          rhs(3, i, j, k) = F(u(3, i, j, k), u(3, i, j-1, k));
          rhs(4, i, j, k) = F(u(4, i, j, k), u(4, i+1, j, k+1));
          /***second group of statements ***/
               for (int m = 0; m < 5; m++)
                rhs(m, i, j, k) += F(u(m, i, j, k), u(m, i, j+1, k));
        }
     }
}
```

Fig.1. Two versions of parallel loop

These loops perform identical computations, but they differ in data location in memory. In the first version (VER1) the compiler can't vectorize any group of statements as the fastest

indexed dimension is the dimension of parallel loop. In the second version (VER2) the compiler successfully vectorizes the second group of statements as the fastest indexed dimension isn't dimension of parallel loop and data along this dimension are located in memory in a row.

Nevertheless, there are situations when it is favorable to use data location as in the first version (VER1, Fig. 1). One of such situations is reordering of arrays in previous parallel loop of the program for the better localization of data. And in order to avoid a deceleration of the considered loop execution, it is necessary to use AVX vector instructions directly in the program code.

For the first version (VER1) vectorization of two groups of statements is possible because data on the fastest dimension of a parallel loop are located in memory in a row. The example of transformation of one of statements using AVX instructions is shown in Fig. 2.

```
double rhs[5][162][162][162], u[5][162][162][162];
#pragma omp parallel for
for(int i = 0; i < Ni; i++)
{
     for(int j = 0; j < Nj; j++)
     {
     /*** изменение индексного пространства цикла ***/
        for(int k = 0; i < Nk; k+=8)
        { /*** rhs[0][i][j][k] = u[0][i][j][k] + u[0][i+1][j][k]; ***/
          _mm512_store_pd(&rhs[0][i][j][k],
               _mm512_add_pd(
                _mm512_load_pd(u[0][i][j][k]),
                _mm512_load_pd(u[0][i+1][j][k]
               )
          );
        }
     }
}
```

Fig. 2. AVX transformation

In one operation 512-bit AVX command can process 8 elements of double type. Use of this approach allows to accelerate the program approximately by 8 times. To confirm this fact one of computationally complex procedures of NPB SP test was implemented in C++ language with use of AVX instructions and without them. The obtained programs were compiled by Intel compiler with optimization flag -O3.

As a result of the experiment there was obtained the acceleration about 6.3 times in comparison with the same procedure without use of AVX instructions (see Table 1). It shows high efficiency of this approach. The possibility to use AVX instructions in Fortran-DVMH compiler is a subject for further researches.

Table 1. Comparison of runtime of different implementations of compute_rhs procedure from SP program

	Xeon Phi 240th	Xeon Phi 240th + AVX512	SpeedUp
CLASS A	2,9 sec	1,8 sec	1,55
CLASS B	13,4 sec	4,8 sec	2,76
CLASS C	118,1 sec	18,7 sec	6,31

7. Obtained results. Comparing with MPI and OpenMP versions of NASA benchmarks.

To estimate parallelization efficiency of the programs using DVMH model the runtimes of Fortran-DVMH versions of NAS benchmarks (MG, EP, SP, BT and LU from NPB 3.3 package) were compared with runtimes of standard versions of the tests using MPI and OpenMP technologies. There is single version of parallel FDVMH program for each of the tests which can be compiled for each of the architectures: CPU, coprocessor or GPU. All optimizations applied to the tests were described in article [12].

Тестирование производилось на сервере с установленными на нем 6-ядерным (12-поточным) процессором Intel Xeon E5-1660v2 с 24Гб оперативной памяти типа DDR3, сопроцессором Intel Xeon Phi 5110P с 8Гб оперативной памяти типа GDDR5 и ГПУ NVidia GTX Titan с 6Гб оперативной памяти типа GDDR5. Основные результаты представлены в Таблице 2.

Testing was performed on the server with 6-core (12-threads) Intel Xeon E5-1660v2 processor with 24 GB of DDR3 RAM, Intel Xeon Phi 5110P coprocessor with 8 GB of GDDR5 RAM and NVidia GTX Titan GPU with 6 GB of GDDR5 RAM. The main results are shown in Table 2.

Table 2. Runtimes of different versions of NASA tests (in seconds)

Application		NAS					Fortran-DVMH		
Test	Class	Xeon E5 (4-12th)			Xeon Phi (64-240th)		Xeon E5 12th	Xeon Phi 240th	GTX Titan
		Serial	MPI	OpenMP	MPI	OpenMP			
BT	A	40,7	12,	10,2	11,08	11,5	7,8	7,68	2,84
	B	166,9	54,	43,07	33,1	32,15	32,2	20,9	9,16
	C	713,3	223	176,7	119,4	105,7	125	74	31,05
SP	A	28,6	17,	14,6	12,03	13,3	15,5	11,6	2,4
	B	116,9	96,	57,1	33,4	38,7	37	27	10,2
	C	483,2	408	425,2	124,0	128,2	174,1	120	3
LU	A	35,07	9,6	8,31	15,05	16,5	18,9	33,75	4,18
	B	148,5	35,	31,10	47,01	44,5	77,7	89,8	11,69
	C	852,3	291	351,9	162,4	134,2	312,5	192,5	34,32
MG	A	1,06	0,5	0,7	0,36	0,22	0,8	0,61	0,13
	B	4,96	2,7	3,22	1,7	1,13	3,8	2,8	0,58
	C	42,3	25,	34,6	10,9	6,39	29,7	15,5	3,36
EP	A	16,73	1,6	1,76	0,94	0,89	1,5	0,78	0,48
	B	67,33	6,6	7,03	3,94	3,31	5,99	2,99	1,17
	C	266,3	26,	26,3	14,8	13,31	23,96	11,6	4,27

Acceleration of FDVMH version of EP test in comparison with serial version of the test executed on one core of CPU is shown in Fig. 3. The test was executed on different architectures separately, and also in the following combinations: CPU + GPU, CPU + coprocessor and coprocessor + CPU + GPU. For each configuration the number of MPI

processes and amount of OpenMP threads inside each of the processes is shown in Fig. 3. The cases marked by red and magenta colors are the cases when loading balance was additionally used by specifying of weight ratio of all cores of CPU and GPU and weight ratio of MPI processes mapped on CPU and coprocessor.

As a result, in case of simultaneous use of GPU and CPU we could obtain this test performance 17% more than its performance only on GPU. At that the ratio of CPU and GPU performances was set 1: 5,7 correspondently.

When the coprocessor and CPU were used simultaneously we could obtain performance 30% more than the test performance on one coprocessor. The most favorable ratio of MPI processes is following: two MPI processes on Xeon Phi and one MPI process on Xeon E5, and MPI processes have equal weights, or one MPI process on CPU and on coprocessor, and ratio of the process weights is 1: 2 correspondently.

When all devices attached to the node were used we could obtain performance which is 28% more than EP test performance on GPU and exceeds of the test performance on the coprocessor by 3.7 times.

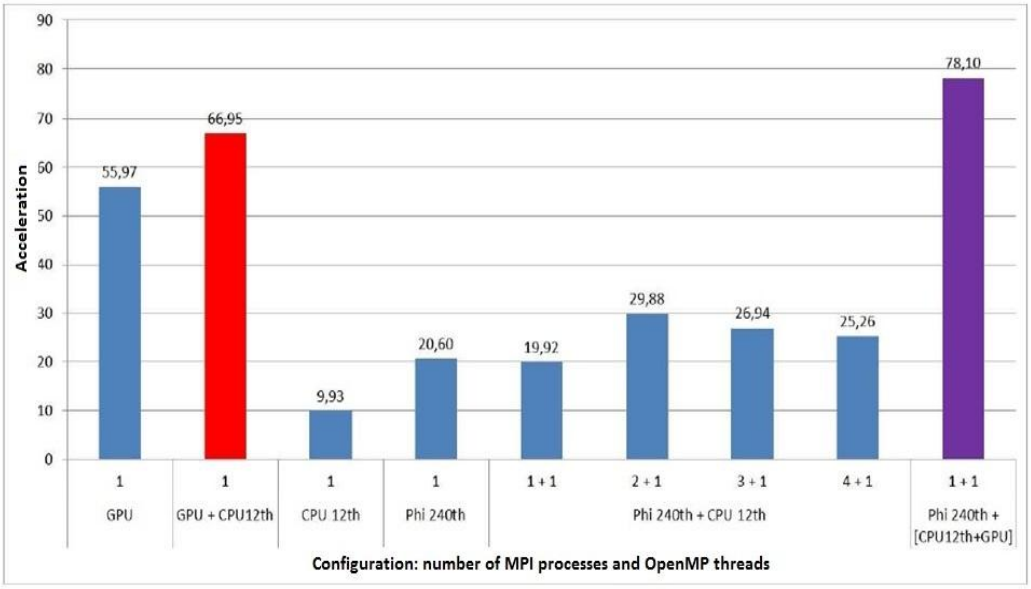

Fig. 3. Use of loading balance in DVMH program on the example of EP test, class C

8. Conclusions

As a result of this research the extension of DVM system was implemented to support Intel Xeon Phi coprocessors. Thus, the important step was made to provide efficient portability of FDVMH programs when the same parallel program can be executed efficiently on the clusters of different architecture using multi-core universal processors, graphic accelerators and Intel Xeon Phi coprocessors. The implemented mapping of DVMH program on CPU and coprocessor allows to apply many of those optimizations of serial program which were described in article [14] and allow to map these programs efficiently on graphic processors.

The runtimes of implemented FDVMH programs given above show that the performance of MG and LU tests on Intel Xeon Phi accelerator and CPU is low. LU test contains loops with dependences on more than one dimension. It is possible that the diagonal execution scheme together with diagonal transformation of arrays by RTS during execution can give the same effect, as on GPU [13]. This problem is a subject for further researches.

References

1. Top500 List - November 2014. URL: http://top500.org/list/2014/11/ (accessed: 30.11.2014).
2. High Performance Fortran. URL: http://hpff.rice.edu (accessed: 30.11.2014).
3. N.A. Konovalov, V.A. Krukov, A.A . Pogrebtsov, N.V. Podderyugina , Y.L. Sazanov. Parallel'noe programmirovanie v sisteme DVM. Yazyki Fortran-DVM i C-DVM.[Parallel programming in the DVM system. Fortran-DVM and C-DVM languages.] // Proceedings of international conference "Parallel Computations and Control Problems" (PACO'2001), Moscow, October 2 – 4, 2001, P. 140-154
4. OpenACC. URL: http://www.openacc-standard.org/ (accessed: 30.11.2014).
5. OpenMP 4.0 Specifications. URL: http://openmp.org/wp/openmp-specifications/ (accessed: 30.11.2014).
6. Intel Ivy Bridge-EP architecture. URL: http://www.intel.ru/content/www/ru/ru/secure/intelligent-systems/privileged/ivy-bridge-ep/xeon-e5-1600-2600-v2-bsdl.html (accessed: 30.11.2014).
7. Intel MIC architecture. URL: https://software.intel.com/mic-developer (accessed: 30.11.2014)
8. Nidia Kepler architecture. URL: http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf (accessed: 30.11.2014)
9. NAS Parallel Benchmarks. URL: http://www.nas.nasa.gov/publications/npb.html (accessed: 30.11.2014).
10. Bakhtin V.A, Klinov M.S., Krukov V.A., Podderugina N.V., Pritula M.N., Sazanov Yu.L. Rasshirenie DVM-modeli parallel'nogo programmirovaniya dlya klasterov s geterogennymi uzlami [Extension of the DVM-model of parallel programming for clusters with heterogeneous nodes]. Vestnik Yuzho-Uralskogo gosudarstvennogo universiteta. Seriya "Matematicheskoe modelirovanie i programmirovanie" [Bulletin of South Ural State University. Series: Mathematical Modeling, Programming & Computer Software]. 2012, No. 18 (277). P 82–92.
11. Intel Xeon Phi programming environment. URL: https://software.intel.com/en-us/articles/intel-xeon-phi-programming-environment (accessed: 30.11.2014).
12. Aleksahin V.F, Bakhtin V.A, Zhukova O.F., Kolganov A.S., Krukov V.A., Podderugina N.V., Pritula M.N., Savitskaya O.A., Shubert A.V. Rasparallelivanie na graficheskie processory testov NAS NPB3.3.1 na jazyke Fortran DVMH [GPU parallelization of NPB 3.3 NAS tests on Fortran DVMH language] // Proceedings of international conference "Parallel Computational Technologies (PCT'2014)" / Rostov-na-Donu, 31.03 - 3.04 2014/ Chelyabinsk: Publishing center SUSU, 2014. P. 30-41.
13. Bakhtin V.A. Otobrazhenie na klastery s graphicheskimi processorami DVMH-programm s regulyarnymi zavisimostyami po dannym [Mapping DVMH-programs with regular data dependencies on clusters with graphics processors] / Bakhtin V.A., Kolganov A.S., Krukov V.A., Podderyugina N.V., Pritula M.N. // Vestnik Yuzhno-Ural'skogo gosudarstvennogo universiteta, seriya "Vychislitel'naya matematika i informatika". 2013. V.2 No. 4, P. 44–56.
14. Arunmoezhi Ramachandran, Jerome Vienne, Rob Van Der Wijngaart, Lars Koesterke, Ilya Sharapov. "Performance Evaluation of NAS Parallel Benchmarks on Intel Xeon Phi". In Proceedings of the 42nd International Conference on Parallel Processing, 2013, pp. 736-743