GPU parallelization of the tests from NAS NPB3.3.1 using Fortran DVMH programming language*

V.F. Aleksahin[1], V.A. Bakhtin[1], O.F. Zhukova[1], A.S. Kolganov[1], V.A. Krukov[1], N.V. Podderugina[1], M.N. Pritula[1], O.A. Savitskaya[1], A.V. Shubert[2],

Keldysh Institute of Applied Mathematics RAS[1]
Lomonosov Moscow State University[2]

The paper presents a number of transformations applied to serial versions of NAS Parallel Benchmarks, NPB3.3.1 (EP, MG, BT, LU, SP) and the parallel execution specifications of these tests by DVMH directives which are necessary for their high performance execution on clusters with GPUs. We explore the impact of different parallelization options on the program performance. The characteristics of the tests developed on a high-level language Fortran-DVMH (hereinafter FDVMH) are compared with their implementation on a low-level language OpenCL performed by researchers from Seoul National University.

1. Introduction

A lot of computational clusters with accelerators attached to their nodes are emerging in recent years. Most of them are graphics processors by Nvidia Corporation. Clusters with accelerators of other architecture – Xeon Phi by Intel Corporation – began to show up in 2012. In the Top500 list [1] of the most powerful supercomputers of the world published in November, 2013, 53 computers have accelerators, 39 computers from them have NVIDIA accelerators, 14 – Intel, 2 – AMD/ATI. This tendency significantly complicates the process of cluster programming due to 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 CUDA (or OpenCL) and OpenMP technologies to load all the cores of the central and graphics processors.

To simplify programming of distributed computing systems several high-level programming languages based on directives, such as HPF [2], Fortran-DVM [3,4], C-DVM [3,5] have been proposed. Programming models and appropriate directive-based extensions of programming languages such as HMPP [6], PGI Accelerator Programming Model [7], OpenACC [8], hiCUDA [9] have been also proposed for possibility to use accelerators.

Since GPU's array-parallel architecture is well suitable for processing of multidimensional loops without dependencies, its parallelization does not expose great ideological problems, whether it would be manual parallelization or with use of high-level tools. The loops with dependencies can be parallelized with considerably higher difficulties, associated with limited support of execution flow synchronization on GPU and consistency model of global memory.

2. Review of NASA NPB 3.3 Benchmarks

NAS Parallel Benchmarks [13] are suite of benchmarks allowing to estimate performance of supercomputers. They were developed and supported in NASA Advanced Supercomputing (NAS) Division (earlier NASA Numerical Aerodynamic Simulation Program) in NASA Ames Research Center. The NPB 3.3 Benchmarks consist of 11 tests.

* The research was supported by grants of the Russian Foundation for Basic Research No. 13-07-00580, 14-01-00109 and Presidium of Russian Academy of Sciences, Programs №15, №16 and №18

In the paper the parallelization of EP, MG, BT, LU, SP tests on GPU is considered. The parallel versions of the tests for clusters were earlier implemented in Fortran-DVM high-level language:

MG (Multi Grid) – a multiple 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 the given 20 points). The grid size N is determined by the test class. It tests the possibilities of the system to perform both long and short data transfer.

EP (Embarrassingly Parallel) – embarrassingly parallel. The test calculates an integral by Monte-Carlo method. It is the test of "complicated parallelism" for measurement of primary computational capacity of floating arithmetics. This test can be useful if the problems using the Monte-Carlo method will be solved on a cluster. The algorithm also takes into account the time of formatting and data output.

BT (Block Tridiagonal) – block three-diagonal scheme. The test solves synthetic system of nonlinear differential equations in partial derivatives (three-dimensional system of Navier-Stokes equations for compressible fluid or gas), using block three-diagonal scheme with the method of alternating directions.

SP (Scalar Pentadiagonal) – scalar pentadiagonal. The test solves synthetic system of nonlinear differential equations in partial derivatives (three-dimensional system of Navier-Stokes equations for compressible fluid or gas), using scalar pentadiagonal 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.

These tests may be divided into two groups: EP and MG – tests not requiring to parallelize loops with regular data dependencies (i.e. there are no parallel loops with ACROSS specifications in FDVM versions of the tests) and BT, SP, LU – tests requiring to parallelize loops with regular data dependencies (i.e. there are parallel loops with ACROSS specifications in FDVM versions of the tests).

Characteristics of the resulted tests after transformation using FDVMH are the following:

There is only 1 parallel loop in EP test.

There are 15 tightly-nested parallel loops in MG test.

There are 44 tightly-nested parallel loops in BT test. 6 loops from them have a dependence on one of three dimensions, and dependent dimension in different loops corresponds to different dimensions of processed arrays.

There are 25 tightly-nested parallel loops in SP test. 6 loops from them have dependence on one of three dimensions, and dependent dimension in different loops corresponds to different dimensions of processed arrays.

There are 107 tightly-nested parallel loops in LU test. 2 loops from them have dependence on all three dimensions.

3. Implementation and features of EP and MG tests

3.1 EP test

In the test the pairs of pseudorandom normally distributed numbers are generated and the frequencies of their hit in each of ten selected half-intervals [k, k+1) are calculated, where k is varied from 0 to 9. The test contains the single loop, and it is possible to perform its iterations independently from each other. During parallel execution of the loop on cluster no exchanges between processors are required, and when the loop is finished the results of sum reduction operation accumulated on each processor are jointed in the array of 10 elements. For efficient mapping on GPU the parallel loop in DVM program is declared as computational region.

3.2 MG test

MG test implements the algorithm of multi-grid method of Poisson task solution. The test contains two types of loops where the main computational loading is concentrated. The loops of first type are loops of projection on more rough grid – during descent on V-loop and of approximation on refined grid during ascension on the V-loop. The loops of the second type are the loops of the solution interpolation based on a discrepancy.

When the values in one grid are calculated using values of other grid it is necessary appropriate distribution of these grids on processors. To solve this problem REALIGN directive allowing to redistribute already distributed array was used in FDVM version. As transition from one grid to other grid is performed very often, their redistributions cause the program deceleration. The deceleration during execution on GPU was very significant because of the low speed of exchanges between CPU memory and GPU memory (in comparison with the speed of computations on GPU). Therefore it was decided to reduce the number of REALIGN directives during computations in FDVMH version and instead of them to use additional memory to store several copies of the grid with different distribution of its elements on processors. The possibilities of Fortran DVMH language for operation with dynamic arrays and pointers were used.

3.3 Optimization of MG test loops

There are four main computational loops in MG test. Each of them is constructed by the same way. Therefore we will consider only one loop and its optimization. PARALLEL directive allows to match the iterations of tightly-nested loops to elements of distributed arrays. In this case tightly-nested loops will be executed in parallel (see **Fig. 3.1**).

```
!DVM$ PARALLEL (i3,i2,i1) ON u(i1,i2,i3)
    do i3=2,n3-1)
      do i2=2,n2-1
        do i1=2,n1-1
          u(i1,i2,i3) = u(i1,i2,i3) + c(0) * r(i1,i2,i3) + c(1) *
  ( r(i1-1,i2,i3) + r(i1+1,i2,i3) + r(i1,i2-1,i3) + r(i1,i2+1,i3)
  + r(i1,i2,i3-1) + r(i1,i2,i3+1) )  + c(2) * (r(i1,i2-1,i3-1)
  + r(i1,i2+1,i3-1) + r(i1,i2-1,i3+1) + r(i1,i2+1,i3+1)
  + r(i1-1,i2-1,i3) + r(i1-1,i2+1,i3) + r(i1-1,i2,i3-1)
  + r(i1-1,i2,i3+1) + r(i1+1,i2-1,i3) + r(i1+1,i2+1,i3)
  + r(i1+1,i2,i3-1) + r(i1+1,i2,i3+1))
        enddo
      enddo
    enddo
```

Fig. 3.1. The loop of psinv procedure before transformations

It isn't difficult to see that, for example, for each fixed iteration on i2 the data of R array on remained not fixed indexes (i1, i3) and all their combinations (i1±1, i3±1) are required. As a result, each thread executing the loop iteration on i2 will read the repeated data of adjacent threads on i1 and i3 indexes. To reuse the data we make the loop on i2 the most internal loop and change the directive as follows:

!DVM$ PARALLEL (i3, i1) ON u(i1, *, i3)

To reuse the values in the loop, for each fixed pair (i1, i3) and (i1±1, i3±1) we introduce three variables declared as private – r1, r1_p1 and r1_m1 which will correspond to i2, i2+1, i2-1indexes. Then it is necessary to read data in r1_m1 and r1 before main loop, to add reading in r1_p1 in the loop before computations, to change the appropriate expressions in the loop by read ones, and after computations – to save r1 and r1_p1 values in r1_m1 and r1 variables correspondently for next iteration. After all transformations there will be 5 groups in the loop (see **Fig. 3.2**).

```
!DVM$ PARALLEL (i3,i1) ON u(i1,*,i3)
!DVM$& ,PRIVATE (i2, r1,r1_m1,r1_p1, r2,r2_m1,r2_p1, r3,r3_m1,r3_p1,
r4,r4_m1,r4_p1, r5,r5_m1,r5_p1)
      do i3=2,n3-1
        do i1=2,n1-1
          r1_m1 = r(i1,1,i3) ! loading of initial values before the loop
          r1 = r(i1,2,i3)
          r2_m1 = r(i1-1,1,i3)
          r2 = r(i1-1,2,i3)
          r3_m1 = r(i1+1,1,i3)
          r3 = r(i1+1,2,i3)
          r4_m1 = r(i1,1,i3+1)
          r4 = r(i1,2,i3+1)
          r5_m1 = r(i1,1,i3-1)
          r5 = r(i1,2,i3-1)

          do i2=2,n2-1
               r1_p1 = r(i1,i2+1,i3) ! loading of next values
               r2_p1 = r(i1-1,i2+1,i3)
               r3_p1 = r(i1+1,i2+1,i3)
               r4_p1 = r(i1,i2+1,i3+1)
               r5_p1 = r(i1,i2+1,i3-1)

               u(i1,i2,i3) = u(i1,i2,i3)+
               c_0 * r1 + c_1 * (r2 + r3 + r1_m1 + r1_p1 + r4 + r5) +
               c_2 * (r4_m1 + r4_p1 + r5_m1 + r5_p1 + r2_m1 + r2_p1 +
               r(i1-1,i2,i3-1) + r(i1-1,i2,i3+1) + r3_m1 + r3_p1 +
               r(i1+1,i2,i3-1) + r(i1+1,i2,i3+1))

               r1_m1 = r1  ! saving of read values
               r1 = r1_p1
               r2_m1 = r2
               r2 = r2_p1
               r3_m1 = r3
               r3 = r3_p1
               r4_m1 = r4
               r4 = r4_p1
               r5_m1 = r5
               r5 = r5_p1
          enddo

        enddo
      enddo
```

Fig. 3.2. The loop of psinv procedure after transformations

Thereby the number of necessary readings from global memory significantly reduced due to saving already read values in i2 loop upon transition from one iteration to another. On the class C the program runtime was reduced by 50% in comparison with the program without these optimizations, however the program runtime on the CPU became much worse.

4. Implementation and features of the BT, SP, LU tests

4.1 Review of dependence types

To understand nature of data dependency better we abstract from algorithm of each of the tests and will consider standard programs, namely – the method of alternating directions used in BT and SP tests and the method of successive over relaxation (Successive over relaxation - SOR) used in LU test.

The sample program for method of alternating directions is shown below (see **Fig. 4.1**).

```
       program adi
      parameter(nx=400,ny=400,nz=400,maxeps=0.01,itmax=100)
      integer nx,ny,nz,itmax
      double precision eps,relax,a(nx,ny,nz)
      call init(a,nx,ny,nz)
      do it = 1,itmax
       eps=0.D0
       do k = 2,nz-1
        do j = 2,ny-1
         do i = 2,nx-1
          a(i,j,k) = (a(i-1,j,k) + a(i+1,j,k)) / 2
         enddo
        enddo
       enddo
       do k = 2,nz-1
        do j = 2,ny-1
         do i = 2,nx-1
          a(i,j,k) = (a(i,j-1,k) + a(i,j+1,k)) / 2
         enddo
        enddo
       enddo
       do k = 2,nz-1
        do j = 2,ny-1
         do i = 2,nx-1
          eps = max(eps, abs(a(i,j,k) - a(i,j,k-1)+a(i,j,k+1)) / 2))
          a(i,j,k) = (a(i,j,k-1)+a(i,j,k+1)) / 2
         enddo
        enddo
       enddo
      if(eps.lt.maxeps) goto 3
      enddo
3     continue
      end
```

Fig.4.1. Implementation of the method of alternating directions

There are three tightly-nested loops in the program and there is a dependence on one of three dimensions in each of the loops. In the first nest of loops the most inner loop can be executed only sequentially since there is the dependency on data of A array. In this regard – if this loop is executed on GPU –a time of reading from memory will be large because data for parallel iterations of the remained two loops will be located not in a row. One of the ways to solve this problem is to reorder two first dimensions of A array. But then the similar problem will arise in the second nest of loops. It is possible to solve the problem similarly, for example, to reorder first two dimensions back. Therefore, it is impossible to select initial arrangement of the array so that all three nests of the loops will be executed most efficiently.

Let's consider the example of other program implementing a method of successive over relaxation (see **Fig. 4.2**). There is only one nest of loops in this program where computations are performed. Unlike a method of alternating directions in this method the main loop has dependence on all its dimensions. It results in considerable difficulties of parallelization even in OpenMP model.

```
            program  sor
            parameter(n1=1000,n2=1000,n3=1000,itmax=100,
            maxeps=0.5e-6,w=0.5)
            real a(n1,n2,n3), eps, w
            integer  itmax

            call init(a, n1, n2, n3)
            do it = 1,itmax
```

```
        eps = 0.
       do k = 2, n3-1
        do j = 2, n2-1
         do i = 2, n1-1
          s = a(i,j,k)
          a(i,j,k) = (w/4)*( a(i-1,j,k)+a(i+1,j,k)+
            a(i,j-1,k)+a(i,j+1,k)+ a(i,j,k-1)+a(i,j,k+1))
            + (1-w)*a(i,j,k)
          eps = max(eps, abs(s - a(i,j,k)))
         enddo
        enddo
       enddo
        if (eps .lt. maxeps) goto 4
       enddo
4      continue
       end
```

Fig.4.2. Implementation of successive over relaxation method

4.2 Algorithm of mapping of loops with dependences in DVM system

The following algorithm is used in DVM system [12] to map these loops. A set of tuples of all possible values of index loop variables will be called a space of the loop iterations. There are flow- and anti- dependencies for I, J and K dimensions in the considered loop, therefore its space of iterations can't be mapped on the block of GPU threads since all threads are executed independently. One of the known methods of such loop mapping is the method of hyperplanes. All elements of a hyperplane can be calculated independently from each other.

Such order of execution of the loop iterations causes a problem of efficient access to global memory because not adjacent elements of arrays are processed in parallel. That results in considerable performance losses (approximately 10 times). And unlike the first considered program it will be even more difficult to specify initial arrangement of the array as the elements are calculated by hyperplanes. Existence in the program of the loops with dependences on more than one dimension and loops without dependences complicates the problem since different arrangement of data for such loops is required for their efficient execution.

To free a programmer from the described above difficulties, the following possibilities are in FDVMH language: the support of loops with dependences using ACROSS specification in PARALLEL directive and dynamic rearrangement of arrays.

4.3 ACROSS specification

The implementation of ACROSS specification is based on the method of hyperplanes described above. To specify that the loop has a dependence, it is necessary to add the list of arrays with dependence to ACROSS specification (see **Fig. 4.3**).

```
!example for the method of alternating directions
      !DVM$ PARALLEL (k,j,i) on A(i,j,k), ACROSS(A(1:1,0:0,0:0))
      do k = 2,nz-1
        do j = 2,ny-1
         do i = 2,nx-1
          a(i,j,k) = (a(i-1,j,k) + a(i+1,j,k)) / 2
         enddo
        enddo
       enddo
```

```
!example for SOR method
      !DVM$ PARALLEL (K,J,I) on A(I,J,K), ACROSS(A(1:1,1:1,1:1))
      !DVM$& ,PRIVATE(s), REDUCTION(MAX(eps))
            do k = 2, n3-1
             do j = 2, n2-1
              do i = 2, n1-1
               s = a(i,j,k)
               a(i,j,k) = (w/4)*(
     >           a(i-1,j,k)+a(i+1,j,k)+
     >           a(i,j-1,k)+a(i,j+1,k)+
     >           a(i,j,k-1)+a(i,j,k+1)+
     >                    ) + (1-w)*a(i,j,k)
               eps = max(eps, abs(s - a(i,j,k)))
              enddo
             enddo
            enddo
```

Fig. 4.3. Usage of ACROSS specification for two methods

4.4 Mechanism of dynamic rearrangement of arrays in DVMH

To optimize an access to global memory of GPU the mechanism of dynamic rearrangement of arrays has been implemented in DVMH runtime system. Before parallel loop execution on GPU the correspondence between the loop dimensions and dimensions of arrays used in the loop is verified and if it is necessary some arrays are rearranged so that the access to elements will be performed in the best way: adjacent threads of the block will operate with adjacent cells of global memory.

The mechanism performs any necessary rearrangement of array dimensions and also so called diagonal transformation, which makes adjacent elements on diagonals (on the plane of certain two dimensions) to be stored in adjacent memory cells. It enables to apply a technique of execution of the loop with dependencies by the hyperplanes without considerable performance losses on access operations to GPU global memory.

4.5 Transformation of source program codes

In BT and SP tests the method of alternating directions is used. In LU test SSOR (symmetric successive over relaxation) method is used. It consists of two three-dimensional loops with positive and negative steps with three dependent dimensions.

To parallelize the loops using FDVMH it is necessary to transform the program so that the loops will be tightly-nested. For example, in LU test the blts and buts procedures were substituted in order to obtain three-dimensional tightly-nested loops. In SP and BT tests the loops in compute_rhs procedure were combined for more effective execution. In x_solve, y_solve and z_solve procedures implementing a method of alternating directions temporary arrays were expanded by one dimension in order to three-dimensional loops could be executed on GPU.

4.6 Optimization of obtained programs

A programmer, using high-level FDVMH language, operates with the serial version of the program. Therefore, it is necessary to write the "good" serial program in order to obtain from it efficient parallel program. The knowledge of target architecture can be useful for optimization performing.

The modern CPUs have three-level cache. A size of first level cache is equal to 64KB and it exists on all computing cores of the processor. The size of the second level cache is varied from 1 to 2 MB. The cache of the third level is common for whole CPU and its size is 12-15MB.

The modern GPUs have a two-level cache. A size of the first level cache is equal to

64KB. It is used for shared memory and displacement of registers. Not more than 48KB is available for shared memory. It exists in each computational block. The maximum size of the second level cache is 1,5MB and it is common for whole GPU. It is used to cache the data loaded from global memory of GPU. There are 15 computational blocks in modern GPU chip GK110. So there are about 48KB of first level cache and 102KB of second level cache per one block. In comparison with CPU, it isn't enough, therefore read operations from global memory of graphic processor are more expensive than from RAM of central processor. All optimizations will be directed to decrease a number of readings from global memory and to increase computational loading. Thereby the ratio of the number of computing operations to read operations from global memory of GPU is raised.

4.6.1 LU test

The main computational loading is concentrated in four loops of blts and bult procedures. In each of the procedures at first the initialization of four arrays of double precision type is performed. The size of the arrays is equal to 25 * $Class^3$, where Class is the size of the considered task class (for example, Class = 162 for class C). As a result, the class C requires about 3 GB of memory. As after initialization the next nest of the loops has dependence on three dimensions, then according to described above principle of DVMH program functioning, the diagonalization of these arrays will be performed. As after initialization the arrays are used only in blts and bult procedures, the expressions for initialization of the array elements were substituted directly in the body of the loop nest with dependence, and the arrays were deleted that caused a recalculation of the values of these arrays on each iteration. Thereby the execution of this test on C class requires by 3 GB less memory that allowed to launch the program on Tesla c2050 GPU.

One more transformation applied to the test is the use of private variables for frequently used elements of arrays mapped on registers by NVCC compiler. The array element is frequently used if, first of all, the quantity of writings and readings is more, than the number of readings for its loading on registers and writing for saving. To explain it let’s consider the following fragment of the program (see **Fig. 4.4**).

```
!DVM$ PARALLEL (k,j,i) on u(i,j,k,*), PRIVATE(m1,m2,m)
do k = 2,nz-1
  do j = 2,ny-1
   do i = 2,nx-1
    m1 = 2
    m2 = 3
    do m = 1,5
      u(i,j,k,m) = u(i,j,k,m1) + u(i,j,k,m2)
    enddo
    do m = 1,5
      u(i,j,k,m) = u(i,j,k,m1+1) + u(i,j,k,m2+1)
    enddo
    do m = 1,5
      u(i,j,k,m) = u(i,j,k,m1-1) + u(i,j,k,m2-1)
    enddo
   enddo
  enddo
 enddo
```

Fig. 4.4. Source loop

In this fragment U array is used on reading and writing, and the array isn't distributed on last dimension. 15 writings and 45 readings are performed in total. And for loading on registers 5 readings are required and the same number of readings will be required to save the result. As a result, it is necessary to add in the program code two loops for the array loading and saving, and also to replace all accesses in the loop body by new variable (see **Fig. 4.5**)

```
!DVM$ PARALLEL (k,j,i) on u(i,j,k,*), PRIVATE(m1,m2,m,u_)
do k = 2,nz-1
  do j = 2,ny-1
   do i = 2,nx-1
     do m=1,5
       u_(m) = u(i,j,k,m)
     enddo
    m1 = 2
    m2 = 3
    do m = 1,5
      u_(m) = u_(m1) + u_(m2)
    enddo
    do m = 1,5
      u_(m) = u_(m1+1) + u_(m2+1)
    enddo
    do m = 1,5
      u_(m) = u_(m1-1) + u_(m2-1)
    enddo
     do m=1,5
       u(i,j,k,m) = u_(m)
     enddo
   enddo
  enddo
 enddo
```

Fig. 4.5. Applying of optimization – replacement of u array by u_ array

4.6.2 BT test

As it was described above the largest computational loading in the test is concentrated in three procedures – x_solve, y_solve, z_solve. Since these procedures differ only in the dependent dimension (x, y and z correspondently), the optimizations, applied to one of them, will be also applied also to all remaining. Let's consider x_solve procedure. It is possible to see that in x_solve procedure the large temporary lhs array of double precision type is used, its size is equal to 75 * Class^3, where Class is the task class (for example, Class = 162 for the class C). Thereby, this array on the class C requires about 2,3 GB of memory. We have the same problems, as for LU test: lack of memory and long time of dynamic rearrangement. It is possible to solve the problem by described above method – to reduce the array size by re-initialization of the array elements directly in loop body of x_solve procedure. Thus, due to the task specifics the auxiliary array can be reduced by one dimension, namely, in the number of times corresponding to the task class. For the class C the array is reduced by 162 times and will require about 15 MB.

4.6.3 SP test

Since this test differs from BT in the algorithm used in x_solve procedure, and the dependence is the same, all described optimizations are applied similarly. The temporary lhs array of the double precision type is used initially, its size is 15 * Class^3. For the class C the array requires about 500 MB memory and it is impossible to reduce it in the program.

5. The obtained results. Comparison with OpenCL.

To estimate of efficiency of FDVMH parallelization two versions of each of programs were used: source serial version without any transformations and transformed and optimized version with FDVMH directives. Testing was performed on K100 supercomputer [14] with Intel Xeon X5670 processors and NVIDIA Tesla C2050 GPUs of Fermi architecture and on Titan server with Intel core i7 processors and NVIDIA GeForce GTX GPU of the newest

Kepler architecture. For comparison, there were obtained execution times of these tests implemented on low-level OpenCL language by researchers from the Seoul national university [11].

The results of comparison of parallelization efficiency for BT, LU, SP, MG, EP tests on A, B, C classes are shown below (see **Table 5.1**; **Fig. 5.1**). The acceleration of FDVMH versions of the tests in comparison with OpenCL is shown in **Fig. 5.1**. The times and the accelerations of FDVMH and OpenCL programs in comparison with source serial versions of the tests executed on one core of Intel Xeon X5670 processor are shown in **Table 5.1**. A dash in the table cells means that for corresponding variants of the test launches there wasn't enough memory on GPU. Note that implementation of OpenCL version of BT test on C class requires more than 6 GB of memory (4 times more, than FDVMH program requires) that didn't allow to compare OpenCL and FDVMH versions of the test.

Table 5.1. Efficiency of test parallelization

Program		CPU, Xeon X5670	FORTRAN DVMH				OpenCL			
			Tesla C2050 (with ECC)		GeForce GTX Titan (without ECC)		Tesla C2050 (with ECC)		GeForce GTX Titan (without ECC)	
Test	Class	Time, seconds	Time, seconds	Acceleration	Time, seconds	Acceleration	Time, seconds	Acceleration	Time, seconds	Acceleration
BT	A	52,69	15,63	3,37	5	10,54	75,5	0,7	22,41	2,35
	B	221,9	61,74	3,59	18,7	11,87	272,42	0,81	71,7	2,97
	C	951,0	192,29	4,94	56,86	16,73	-	-	-	-
SP	A	36,6	9,82	3,73	3,23	11,33	22,95	1,59	6,09	6,01
	B	154,7	33,4	4,63	11,37	13,61	86,35	1,79	28,53	5,42
	C	637,73	117,55	5,43	35	18,22	433,77	1,47	137,7	4,63
LU	A	40,31	7,0	5,76	5,14	7,84	8,86	4,55	4,49	8,9
	B	170	21,0	8,10	12,51	13,59	30,23	5,62	13,49	12,6
	C	779	70,5	11,05	37,46	20,80	127,95	6,09	44,65	17,4
MG	A	1,41	0,18	7,8	0,11	12,8	0,20	7,05	0,09	20
	B	6,62	0,82	8,07	0,50	13,24	0,94	7,04	0,42	15,7
	C	55,17	5,62	9,8	3,08	17,9	7,83	7,04	3,71	14,8
EP	A	7,97	0,24	33,2	0,38	20,9	0,25	31,88	0,43	18,5
	B	31,85	0,73	43,6	1,12	28,4	0,82	38,8	1,26	25,2
	C	55,17	2,68	20,5	4,14	13,3	2,89	19	4,51	12,2

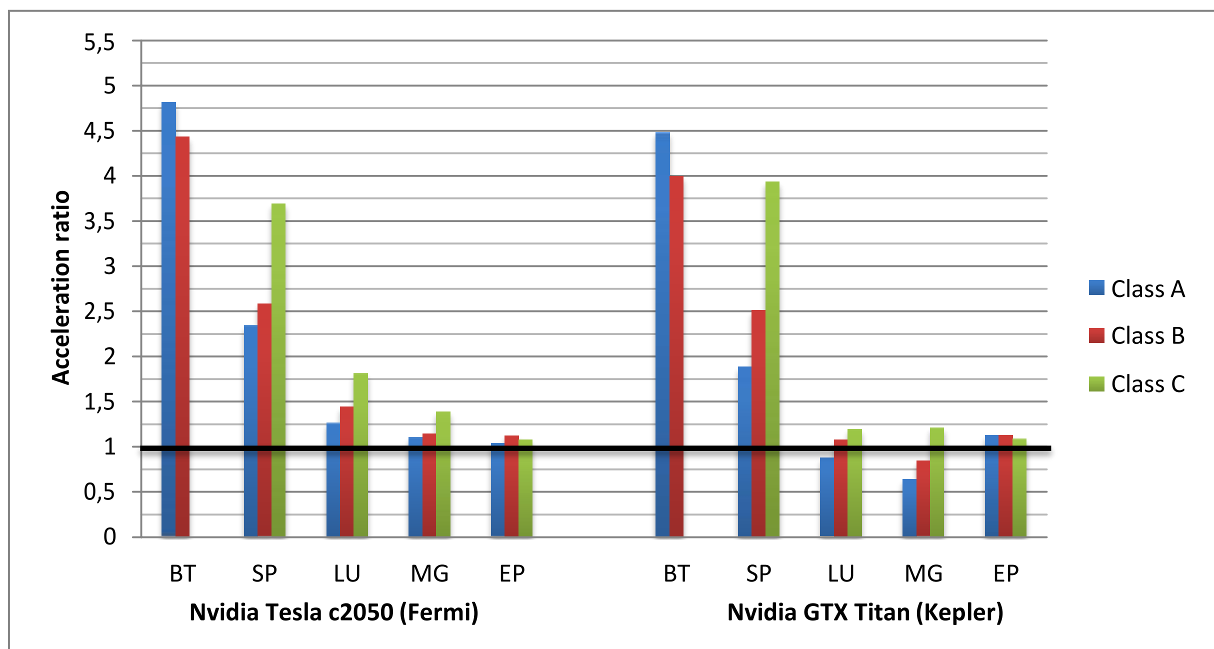

Fig. 5.1. Acceleration of DVHM programs in comparison with OpenCL programs.

The quantity of lines and words in the considered programs, and also amount of used memory are shown in **Table 5.2**. In certain cases DVMH requires more memory for performing of dynamic rearrangement of the arrays. In the table the amount of memory for DVMH programs consists of amount of memory without use of dynamic rearrangement and amount of memory, necessary for dynamic rearrangement.

Table 5.2. Use of memory and quantity of lines and words

Task		Sourse version		DVM version		OpenCL version	
Test	Class	memory, GB	quantity of lines, words	memory, GB	quantity of lines, words	memory, GB	quantity of lines, words
BT	A	0,041	4092, 13588	0,088+0,044	3478, 18427	0,346	14678, 55932
	B	0,166		0,356+0,175		1,2	
	C	0,665		1,425+0,79		> 6	
SP	A	0,043	3469, 10547	0,063+0,005	3437, 10744	0,154	11179, 41714
	B	0,174		0,253+0,04		0,340	
	C	0,698		1,014+0,158		1,38	
LU	A	0,036	4148, 18173	0,039+0,005	2682, 15772	0,116	9439, 41426
	B	0,142		0,158+0,04		0,249	
	C	0,558		0,634+0,158		0,776	
MG	A	0,440	1686, 5028	0,518	2315, 7713	0,503	4030, 13791
	B	0,440		0,518		0,503	
	C	3,31		3,50		3,55	
EP	A	0,07	659, 2614	0,1	704, 2865	0,1	928, 3792
	B	0,07		0,1		0,1	
	C	0,07		0,1		0,1	

6. Conclusion

As a result of researches LU, BT, SP, MG and EP tests optimized and parallelized by FDVMH were obtained. A laboriousness of parallelization by FDVMH and OpenCL can be estimated roughly by quantity of lines or words added in the source program. In **Table 5.2** it is possible to see that the size of the obtained DVMH programs differs from size of source serial ones not more than by 45%. The size of OpenCL programs is differed by 2-3 times. In tests with dependences performance of OpenCL programs is extremely small. Parallelization efficiency of FDVMH version of EP test is insignificantly higher then parallelization efficiency of OpenCL version of the test. Parallelization efficiency of DVMH version of MG test is higher than parallelization efficiency of OpenCL version of the test by 50% if to use Fermi architecture, and by 25% if to use Kepler architecture, on C class. If to consider resource-intensive tasks (class C), then FDVMH programs showed higher efficiency, than OpenCL on all considered tests.

In the further it is intended to parallelize remaining tests by FDVMH and to compare them with implementations on OpenCL and OpenACC. The researches of parallelization efficiency of the tests on Intel Xeon Phi platform are intended also.

References

1. Top500 List – November 2013 | TOP500 Supercomputer Sites URL: http://top500.org/list/2013/11/ (accessed 24.11.2013)

2. High Performance Fortran URL: http://hpff.rice.edu/ (accessed 01.12.2013)

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. N.A. Konovalov, V.A.Krukov, S.N. Mihailov, A.A. Pogrebtsov. Fortran DVM – yazyk razrabotki mobil'nyh parallel'nyh programm [Fortran DVM – a language for mobile parallel programs development]. // Programmirovanie [Programming]. 1995. No. 1. P. 49–54.

5. N.A.Konovalov, V.A.Krukov, Y.L. Sazanov. C-DVM – yazyk razrabotki mobil'nyh parallel'nyh programm [C-DVM – a language for mobile parallel programs development] // Programmirovanie [Programming]. 1999. No. 1. P. 54–65.

6. Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP™: A Hybrid Multi-core Parallel Programming Environment.

7. OpenACC
URL: http://www.openacc-standard.org/ (accessed 24.11.2013)

8. T.D. Han and T.S. Abdelrahman. hiCUDA: High-Level GPGPU Programming. IEEE Transactions on Parallel and Distributed Systems, vol. 22, no. 1, pp. 78-90, Jan. 2011

9. V.A. Bakhtin, M.S. Klinov, V.A. Krukov, N.V. Podderyugina, M.N. Pritula, Y.L. Sazanov. Rasshirenie DVM-modeli parallel'nogo programmirovaniya dlya klasterov s geterogennymi uzlami [Extension of the DVM parallel programming model for clusters with heterogeneous nodes] // Vestnik Yuzhno-Ural'skogo gosudarstvennogo universiteta, seriya "Matematicheskoe modelirovanie i programmirovanie", No. 18 (277), vypusk 12. Chelyabinsk: publishing center SUSU, 2012. P. 82–92.

10. Pennycook S.J., Hammond S.D., Jarvis S.A., Mudalige G.R. Performance Analysis of a Hybrid MPI/CUDA Implementation of the NAS-LU Benchmark. ACM SIGMETRICS Performance Evaluation Review – Special issue on the 1st international workshop on performance modeling, benchmarking and simulation of high performance computing systems (PMBS 10). 2011. Vol. 38, Issue 4. P. 23–29.

11. Seo S., Jo G., Lee J. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. 2011 IEEE International Symposium on. Workload Characterization (IISWC). 2011. P. 137–148.

12. V.A. Bakhtin, A.S. Kolganov, V.A. Krukov, N.V. Podderyugina, M.N. Pritula. Otobrazhenie na klastery s graphicheskimi processorami tsiklov s zavisimostyami po dannym v DVMH-programmax [Mapping the loops with data dependencies in DVMH-programs on clusters with graphics processors] // Proceedings of international conference "Scientific service on the Internet: all verges of parallelism", September 2013, Novorossiysk. Moscow, Moscow University Press, 250–257.

13. NAS Parallel Benchmarks
URL: http://www.nas.nasa.gov/publications/npb.html (accessed 25.11.2013)

14. Hybrid supercomputer K-100
URL: http://www.kiam.ru/MVS/resourses/k100.html (accessed 25.11.2013)