# MAPPING DVMH-PROGRAMS WITH REGULAR DATA DEPENDENCIES ON CLUSTERS WITH GRAPHICS PROCESSORS

V.A. Bakhtin, A.S. Kolganov, V.A. Krukov, N.V. Podderyugina, M.N. Pritula

DVMH model (DVM for Heterogeneous systems) was introduced at Keldysh Institute of Applied Mathematics, Russian Academy of Sciences in 2011. High level programming languages which are standard FORTRAN and C languages, extended with directives to map on a parallel machine and implemented as special comments (or pragmas) were developed for new heterogeneous and hybrid supercomputer systems. The problems and methods of mapping of the loops with dependencies on graphics processors are described, and performance of the parallel Fortran DVMH programs with the regular data dependencies is demonstrated in the paper.

*Keywords: DVM for Heterogeneous systems, Fortran DVMH, hybrid systems with accelerators, graphics processors, CUDA.*

## 1. Difficulties of the GPU programming

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, 2012, 62 computers have accelerators. 50 computers have NVIDIA accelerators, 7 – Intel, 3 – AMD/ATI, 2 – IBM. 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 use CUDA (or OpenCL) and OpenMP technologies to load all the cores of the central and graphics processors.

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

Since GPU's massively parallel architecture is well suitable for processing a multidimensional loop without dependencies, its parallelization does not expose a great ideological problem, whether it would be manual parallelization or with use of high-level tools. Loops with dependencies can be parallelized with considerably higher difficulties, associated in particular with consistency model of GPU memory, limited support for execution flow synchronization on GPU, large time losses in case of uncoalesced random memory access.

## 2. Fortran DVMH language

DVM model has been extended to support clusters with accelerators [10] at Keldysh Institute of Applied Mathematics, Russian Academy of Sciences in 2011. The extension is called DVMH and allows to convert DVM-program for a cluster to DVMH-program for a cluster with accelerators with little changes.

The Fortran-DVM language has been expanded with following set of directives:

- to specify compute regions - parts of the program, which could be executed on GPU;
- to specify additional properties of parallel loops;
- to control actual state of data in CPU memory.

One of the important aspects of functioning of such a program model as DVMH is a problem to map source program on the all levels of parallelism and heterogeneous computing devices.

Important tasks of mapping mechanism are to ensure correct execution of all constructions supported by the language on heterogeneous computing devices, to balance load among computing devices, and to choose an optimal way to perform each part of the code on particular device.

Fortran-DVM supports loops with regular data dependencies through special ACROSS clause in the PARALLEL directive. Such loops can be executed correctly in parallel on a cluster in two modes: a pipeline mode or a hyperplane processing mode.

There was no efficient implementation for loops with ACROSS clause for GPU in Fortran-DVMH.

## 3. Extension of Fortran DVMH capabilities

The support for the loops with regular dependencies on NVIDIA GPU with use of CUDA programming technology was added in Fortran-DVMH language in 2013.

There is a capability to efficiently execute loops with regular dependencies on cluster architectures in DVM[3] model. To use graphics accelerator for such loops it is necessary to solve following set of problems:

- Data exchange between neighboring cluster nodes, including the pipeline mode
- Efficient mapping of chunks of a loop with dependencies on the CUDA architecture
- Optimization of global memory access for GPU

In the pipeline mode a portion of a parallel loop distributed on a particular MPI-process is divided on chunks in order to allow adjacent dependent processes to begin execution of the loop as soon as possible. Since there is the aim to use GPU, it is necessary to support such a data exchange during the loop processing.

To extract the maximum parallelism for the loops with more than one dependence-carrying dimension there is an opportunity not to transform these dimensions into sequential ones, but to use, for example, the method of hyperplanes.

Since the uncoalesced access to global memory is very slow, there is the problem of efficiency of loop execution if loop iterations are reordered during execution. The problem can't be solved by simple rearrangement of the array dimensions or the loop index variables in a source code as it will be shown on the examples of alternating directions and successive over relaxation methods. The mechanism of dynamic rearrangement of the arrays, implemented in Fortran-DVMH compiler is intended to solve this problem.

## 4. Algorithm for mapping of loops with dependencies on GPU

Consider the program written in Fortran-DVMH which contains multidimensional tightly-nested loops with regular data dependencies. One of the known algorithms with data dependencies is SOR — the method of successive over relaxation. Let's consider a two-dimensional case for simplicity. Thus, for a square matrix of size *N* main loop looks as shown in Fig. 1.

```
DO J = 2,N-1
  DO I = 2,N-1
    S=A(I,J)
    A(I,J)=(W/4)*(A(I-1,J)+A(I+1,J)+A(I,J-1)+A(I,J+1))+(1-W)*A(I,J)
    EPS=MAX(EPS,ABS(S-A(I,J)))
  ENDDO
ENDDO
```

**Fig. 1.** Main loop of Successive Over Relaxation method

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 and J dimensions in the considered loop, therefore its space of iterations can't be simply mapped on the block of GPU threads since all threads are executed independently. Therefore, other method of mapping is needed.

One of the known methods of mapping for such loops is the method of hyperplanes. All elements on a hyperplane can be calculated independently from each other. We will calculate all space of iterations in several iterations: on the first iteration the elements of first hyperplane will be calculated, on the second one the elements of the second hyperplane will be calculated and so on, till all the elements are calculated. As this method applied to considered loop, the diagonals, parallel to the antidiagonal will be calculated. It will be performed about *2*N* iterations in total.

Let's generalize proposed algorithm for the multidimensional case. Let there is multidimensional tightly-nested loop of rank *k* with regular data dependencies on all of *k* dimensions. Then all the elements of hyperplane of rank *k-1* can be calculated independently. If *q* dimensions of *k* ones have no data dependencies, but *p* have, where $k = p + q$, then apply the proposed algorithm to hyperplanes of rank *p*, and calculate remaining *q* dimensions as independent ones.

As mentioned above, this execution order of the loop iterations may cause a problem of efficient access to a global memory since not adjacent elements of arrays are processed in parallel. That results in considerable losses of performance.

## 5. Mechanism for dynamic rearrangement of arrays

To optimize an access to global memory the mechanism of dynamic rearrangement of arrays has been implemented in the runtime system for Fortran DVMH programs. For each loop the mechanism uses information about mutual alignment of the loop and the array which is already available in DVM-program since it is needed for mapping to a cluster and distribution of computations. It determines the correspondence between loop dimensions and array dimensions and then rearranges the array in GPU memory so that the access to elements will be performed in the best way: adjacent threads of the thread block work with adjacent memory cells.

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. That enables applying technique of execution of the loop with dependencies with the hyperplane method without considerable performance losses on global memory access operations.

## 6. Example programs and characteristics of their execution

To illustrate new capabilities of Fortran DVMH language consider two simple programs: implementations of the method of alternating directions and the method of successive over relaxation. Figure 2 shows a Fortran DVMH program for alternating directions method.

There are loops with dependencies in the program, and each of three loops has dependence only on one dimension. This feature enables to achieve sufficient level of parallelism for them, but it is essentially that at any order of array dimensions one of these loops will work considerably slower (approximately 10 times) than other loops due to impossibility to process in parallel in this loop the array elements stored in adjacent memory cells. In this situation the mechanism of dynamic rearranging of arrays is very useful. For such program it is enough to rearrange the array one time for two loops. The program without additional modifications, compiled by the Fortran DVMH compiler and then run, will detect automatically necessity of such rearrangement and will speed up its execution in comparison with more traditional approach where an arrangement of an array is fixed and can't be changed during program execution. There was performed a series of runs on the K-100 cluster (designed at Keldysh Institute of Applied Mathematics, Russian Academy of Sciences) having two Intel Xeon X5670 CPUs and three NVIDIA Tesla C2050 GPUs in its computing nodes. Results of runs are shown in Table 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)
!DVM$ DISTRIBUTE(BLOCK,BLOCK,BLOCK) :: a
      call init(a,nx,ny,nz)
      do it = 1,itmax
       eps=0.D0
!DVM$  ACTUAL(eps)
!DVM$  REGION
!DVM$  PARALLEL (k,j,i) ON a(i,j,k),
!DVM$*  ACROSS (a(1:1,0:0,0:0))
       do k = 2,nz-1
        do j = 2,ny-1
         do i = 2,nx-1
          a(i,j,k) = (a(i-1,j,k) + a(i+1,j,k)) / 2
         enddo
        enddo
       enddo
!DVM$  PARALLEL (k,j,i) ON a(i,j,k),
!DVM$*  ACROSS (a(0:0,1:1,0:0))
       do k = 2,nz-1
        do j = 2,ny-1
         do i = 2,nx-1
          a(i,j,k) = (a(i,j-1,k) + a(i,j+1,k)) / 2
         enddo
        enddo
       enddo
!DVM$  PARALLEL (k,j,i) ON a(i,j,k),
!DVM$*  REDUCTION(MAX(eps)),ACROSS(a(0:0,0:0,1:1))
       do k = 2,nz-1
        do j = 2,ny-1
         do i = 2,nx-1
          eps = max(eps, abs(a(i,j,k) -
               (a(i,j,k-1)+a(i,j,k+1)) / 2))
          a(i,j,k) = (a(i,j,k-1)+a(i,j,k+1)) / 2
         enddo
        enddo
       enddo
!DVM$  END REGION
!DVM$  GET_ACTUAL(eps)
       if(eps.lt.maxeps) goto 3
      enddo
3     continue
      End
```

**Fig. 2.** Implementation of alternating directions method in Fortran DVMH

There is small deceleration on small problem sizes due to insufficient load of GPU and therefore decreased influence of memory access speed.

Now let’s consider Fortran DVMH program for a method of successive over relaxation. Its code is shown in Figure 3.

The main loop of the program has dependencies on all the dimensions leading to considerable difficulties for parallelizing even in OpenMP model. DVM-system allows to perform such loops on GPU and to obtain speedup using two described above techniques. It should be noted especially that the program not only can be executed on GPU efficiently, but it can be also executed on a cluster with GPU without any additional changes.

**Table 1**

Execution times for the method of alternating directions (100 iterations)

| Linear size (nx=ny=nz) | CPU, 1 core | GPU, without array rearrangement | | GPU, with array rearrangement | |
|---|---|---|---|---|---|
| | Time, seconds | Time, seconds | Speed-up | Time, seconds | Speed-up |
| 64 | 0,17 | 0,13 | 1,31 | 0,18 | 0,94 |
| 128 | 1,52 | 0,92 | 1,65 | 0,52 | 2,92 |
| 256 | 12,66 | 6,84 | 1,85 | 3,16 | 4,01 |
| 400 | 48,69 | 26,17 | 1,86 | 11,34 | 4,29 |

Efficiency characteristics of applying of mapping algorithm both with array rearrangement, and without array rearrangement are shown in Table 2.

```
      PROGRAM  SOR
      PARAMETER(N1=16000,N2=16000,ITMAX=100,MAXEPS=0.5E-6,W=0.5)
      REAL A(N1,N2), EPS, W
      INTEGER  ITMAX
!DVM$ DISTRIBUTE  A(BLOCK, BLOCK)
      call init(A, N1, N2)
      DO IT = 1,ITMAX
       EPS = 0.
!DVM$  ACTUAL(EPS)
!DVM$  REGION
!DVM$  PARALLEL (J, I) ON A(I, J), PRIVATE (S),
!DVM$*  REDUCTION(MAX(EPS)), ACROSS (A(1:1,1:1))
       DO J = 2,N2-1
        DO I = 2,N1-1
         S = A(I,J)
         A(I,J) = (W/4)*(A(I-1,J)+A(I+1,J)+
     *          A(I,J-1)+A(I,J+1)) + (1-W)*A(I,J)
         EPS = MAX(EPS, ABS(S - A(I,J)))
        ENDDO
       ENDDO
!DVM$  END REGION
!DVM$  GET_ACTUAL(EPS)
       IF (EPS .LT. MAXEPS) GOTO 4
      ENDDO
4     continue
      END
```

**Fig. 3.** Implementation of the method of successive over relaxation in Fortran DVMH

**Table 2**

Execution times for the method of successive over relaxation (100 iterations)

| Linear size (N1=N2) | CPU, 1 core | GPU, without array rearrangement | | GPU, with array rearrangement | |
|---|---|---|---|---|---|
| | Time, seconds | Time, seconds | Speedup | Time, seconds | Speedup |
| 2000 | 2,76 | 2,23 | 1,24 | 1,8 | 1,53 |
| 4000 | 11,10 | 5,6 | 1,98 | 3,9 | 2,85 |
| 8000 | 44,50 | 19,01 | 2,34 | 10,76 | 4,14 |
| 16000 | 178,02 | 80,24 | 2,22 | 32,91 | 5,41 |

## 7. Testing on NPB benchmarks: BT, SP, LU

NAS Parallel Benchmarks bundle contains three benchmarks with loops with data dependencies: BT (Block Tridiagonal), SP (Scalar Pentadiagonal) and LU (Lower - Upper). These benchmarks solve the synthetic problem of differential equations in partial derivatives (three-dimensional system of Navier-Stokes equations for compressible fluid or gas), using the block three-diagonal scheme with the method of alternating directions (BT), the scalar pentadiagonal method (SP), and a method of the symmetric successive over relaxation (SSOR, algorithm of LU by means of Gauss–Seidel symmetric method).

There are 57 tightly-nested loops in the BT benchmark which can be executed in parallel. 6 of them have a dependence on one of three mapped dimensions, and dependence-carrying dimension in different loops corresponds to different dimensions of processed arrays. The resulted Fortran program consists of 3850 lines in fixed format, 76 of them are DVMH directives. For the best access to global memory of GPU the source code was optimized in following manner: the dimensions in all arrays were reordered so that the innermost loop dimension became corresponding to the first dimension of the array, in accordance with the way of array element order used in Fortran.

There are 56 tightly-nested loops in the SP benchmark which can be executed in parallel. 12 of them have a dependence on one of three mapped dimensions, and dependence-carrying dimension in different loops corresponds to different dimensions of processed arrays. The resulted Fortran program consists of 3500 lines in fixed format, 215 of them are DVMH directives. No optimizations of source code in comparison with the original serial program were performed.

There are 107 tightly-nested loops in the LU benchmark which can be executed in parallel. 2 of them have a dependence on all three mapped dimensions. The resulted Fortran program consists of 4500 lines in fixed format, 56 of them are DVMH directives. The same optimization of the source code, as for the BT benchmark, was performed for LU. Temporary arrays used in two large loops with dependencies were excluded from the loops, but expressions to calculate its element values were inlined to reduce number of reads from global memory of GPU and amount of allocated memory.

Testing was performed on the K100 supercomputer equipped with Intel Xeon X5670 and GPU NVIDIA Tesla C2050 processors with turned on ECC and on the workstation with Intel Core i7-3770 and GPU NVIDIA GeForce GTX TITAN without ECC. These GPU have different processor architectures which will allow us to compare their applicability for considered benchmarks. Serial versions of the programs were executed on K100 supercomputer. For comparison purposes the execution times of parallel DVM-programs on 12 processor cores (one K100 node) were also obtained.

The results of performance testing for these benchmarks are shown in Table 3 and in Figure 4 (for each target hardware the program showing the best result was evaluated).

**Table 3**

Efficiency of parallelizing of NPB benchmarks: BT, SP, LU

| | | CPU Intel Xeon 5670, 1 core | Tesla C2050 (with ECC) | | GeForce GTX Titan (without ECC) | |
|---|---|---|---|---|---|---|
| Benchmark | Class | Time, seconds | Time, seconds | Speedup | Time, seconds | Speedup |
| BT | A | 52,6 | 14,96 | 3,52 | 2,25 | 23,42 |
| | B | 221,9 | 66,07 | 3,35 | 8,34 | 26,61 |
| | C | 951,0 | 200,9 | 4,7 | 25,57 | 37,19 |
| SP | A | 36,6 | 8,6 | 4,2 | 2,27 | 16,12 |
| | B | 154,7 | 31,9 | 4,8 | 9,44 | 16,39 |
| | C | 637,7 | 114,24 | 5,5 | 29,82 | 21,39 |
| LU | A | 40,3 | 6,5 | 6,2 | 4 | 10 |
| | B | 170 | 20,36 | 8,3 | 11,4 | 14,9 |
| | C | 779 | 63,97 | 12,1 | 33,8 | 23,04 |

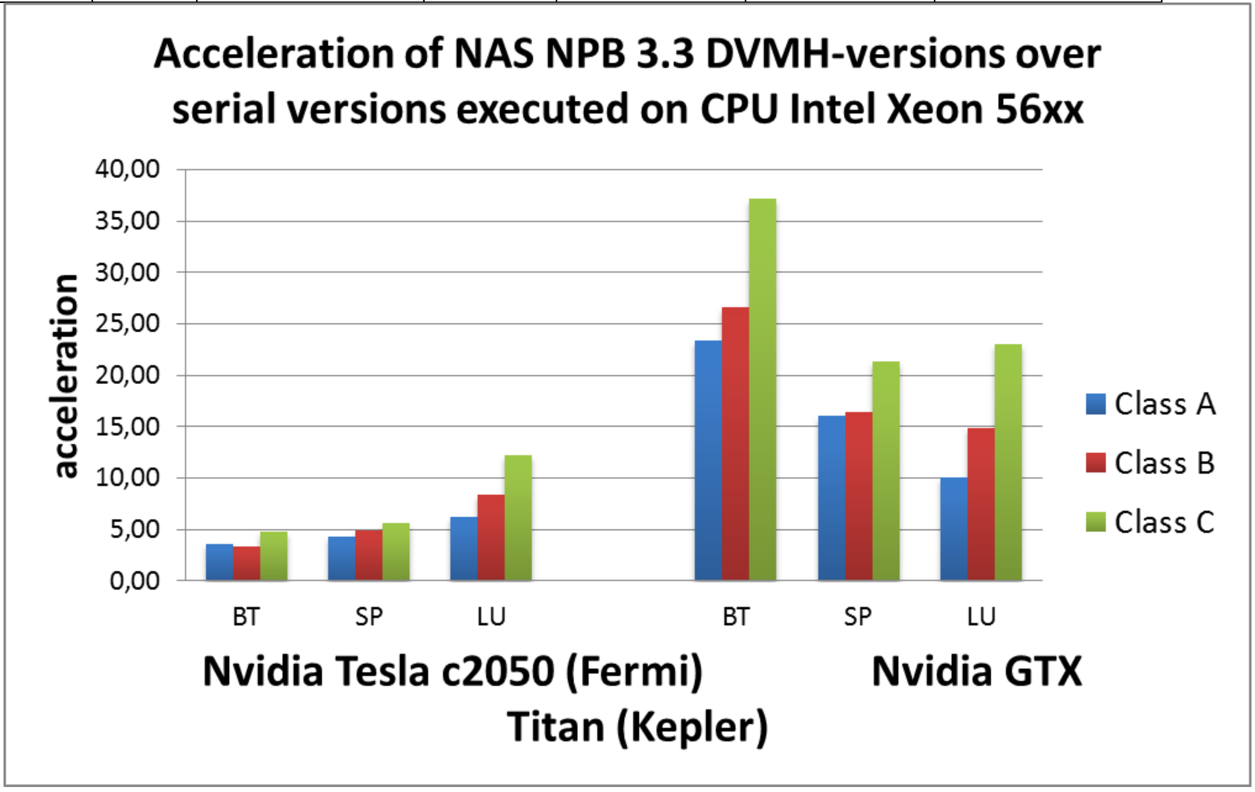

**Fig. 4.** Efficiency of BT, LU, SP parallelizing

## Conclusion

DVM-system and Fortran DVMH language were extended with support for execution of loops with dependencies on GPU. Approbation on NAS benchmarks was performed. It shows performance results, which are close to the results of manually optimized versions of the tests described in [11] and [12].

*The research was supported by grants of the Russian Foundation for Basic Research No. 11-01-00246, 12-01-33003 mol_a_ved, 12-07-31204-mol_a and a grant of the President of the Russian Federation of NSh-4307.2012.9.*

## References

1. Top500 List – November 2012 TOP500 Supercomputer Sites. URL: http://top500.org/list/2012/11/ (accessed 01.12.2012).
2. High Performance Fortran. URL: http://hpff.rice.edu/ (accessed 01.12.2012).
3. Konovalov N.A., Krukov V.A., Pogrebtsov A.A., Podderyugina N.V., Sazanov Y.L. Parallel'noe programmirovanie v sisteme DVM. Yazyki Fortran-DVM i C-DVM [Parallel programming in the DVM system. Fortran-DVM and C-DVM languages]. Trudy Mezhdunarodnoy konferencii "Parallel'nye vychisleniya i zadachi upravleniya" (PACO'2001) [Proceedings of International conference "Parallel computations and control problems" (PACO'2001)]. Moscow, 2001. P. 140–154.
4. Konovalov N.A., Krukov V.A., Mihailov S.N., Pogrebtsov A.A. 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. Konovalov N.A., Krukov V.A., Sazanov Y.L. 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. Dolbeau R., Bihan S., Bodin F. HMPP™: A Hybrid Multi-core Parallel Programming Environment URL: http://www.caps-entreprise.com/wp-content/uploads/2012/08/caps-hmpp-gpgpu-Boston-Workshop-Oct-2007.pdf (accessed 02.12.2012).
7. The Portland Group. PGI Accelerator Programming Model for Fortran & C. URL: http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.3.pdf (accessed 02.12.2012).
8. OpenACC. URL: http://www.openacc-standard.org/ (accessed 01.12.2012).
9. Han T.D., Abdelrahman T.S. hiCUDA: High-Level GPGPU Programming. IEEE Transactions on Parallel and Distributed Systems. 2011. Vol. 22, No. 3. P. 78–90.
10. Bakhtin V.A., Klinov M.S., Krukov V.A., Podderyugina N.V., Pritula M.N., Sazanov Y.L. 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". Chelyabinsk, Publishing of the South Ural State University, 2012. No. 18 (277), Issue 12. P. 82–92.
11. 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.
12. 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.