# CDVMH language.
# CDVMH compiler.
# Compilation, execution and debugging of CDVMH programs.

**Introduction. 3**

**1 Glossary. 3**

1.1 Description of hardware computational systems 3
1.2 Description of virtual computational systems where DVMH program can be executed 3

**2 Programming model and parallelism model. 4**

**3 CDVMH language. 5**

**4 Data distribution. 6**

4.1 Directive array. 6
4.1.1 Data distribution. Specification distribute 7
4.1.2 Multidimensional distributions. 11
4.1.3 Distribution by alignment. Specification align. 12
4.1.4 Non-distributed data 14
4.2 Directive template. 14
4.2.1 Aligning dynamically allocated arrays 15

**5 Directives of data redistribution. 16**

5.1 Data redistribution. Directive redistribute. 16
5.2 Data realignment. Directive realign. 16

**6 Distribution of computations. Parallel loops. 17**

6.1 Parallel loop definition. 17
6.2 Distribution of loop iterations. Directive parallel. 18
6.2.1 Reduction operations and variables. Specification reduction. 20
6.2.2 Private variables. Specification private. 21
6.3 Computations outside parallel loop 21

**7 Remote data specification. 22**

7.1 Definition of remote references. 22
7.2 References of shadow type. 23
7.2.1 Specification of array with shadow edges. 23
7.2.2 Specification of renewing shadow edges for loop. 24
7.2.3 Specification across of dependent references of shadow type for single loop. Specification stage. 25
7.3 Remote references of remote type. Directive remote_access. 27

**8 Distribution of computations. Computational region. 28**

8.1 Directive region. ..... 28
8.2 Description of region constructions. ..... 30
8.2.1 Parallel loop. ..... 30
8.2.2 Serial group of statements. ..... 30
8.2.3 Host section. ..... 30

**9 Control of data movement, actualization. Actualization directives get_actual and actual. ..... 31**

**10 The directive specifying properties of declared functions. ..... 31**

**11 Functions. ..... 32**

11.1 Function call from parallel loop. ..... 32
11.2 Function call outside parallel loop. ..... 32
11.3 Formal arguments. ..... 32
11.4 Local arrays. ..... 33

**12 Input/output. ..... 33**

**13 CDVMH compiler. ..... 34**

**14 Modes of data and computation distribution on computational devices. 34**

14.1 DVMH program execution scheme. ..... 35
14.2 Modes of data and computation distribution. ..... 35
14.2.1 Simple static mode. ..... 36
14.2.2 Dynamic mode with selection of distribution scheme ..... 36
14.2.3 Dynamic mode with use of selected distribution scheme. ..... 37
14.3 Mechanism of dynamic rearrangement of arrays. ..... 37

**15 Compilation, execution and debugging of CDVMH programs. ..... 38**

15.1 Compilation. Obtaining ready-for-run program. ..... 38
15.2 Execution of CDVMH program. ..... 39
15.3 Comparative debugging. ..... 40
15.4 Performance debugging. ..... 41

**References. ..... 42**

**16 Examples of programs. ..... 43**

16.1 Example 1. Jacobi algorithm. ..... 43
16.2 Example 2. Gauss elimination method algorithm. ..... 46
16.3 Example 3. Red-Black Successive Over Relaxation. ..... 47

**Annex 1. Syntax of CDVMH directives. ..... 49**

**Annex 2. Environment variables for CDVMH programs. ..... 53**

**Annex 3. Description of output files of the compiler. ..... 57**

**Annex 4. User API. ..... 57**

# Introduction.

CDVMH language was designed to create portable and efficient parallel computational applications for clusters with accelerators.

The language is the extension of C language according to DVMH model (DVM for Heterogeneous systems) developed in Keldysh Institute of Applied Mathematics, Russian Academy of Sciences [1]. DVMH model is the extension of DVM model [1].

Using C-DVMH language the programmer deals with the single version of the program both for serial and parallel execution. Besides algorithm description by means of usual C the program contains rules for parallel execution of the algorithm. These rules are syntactically organized in such a manner that they are "transparent" for standard C compilers on serial computers and don’t prevent to execute and to debug CDVMH program as usual serial one at workstations.

# 1 Glossary.

The terms and abbreviations used in the document are described in this section.

## 1.1 Description of hardware computational systems

**Computational cluster** – a set of interconnected computational nodes (computers). The cluster node can contain some specialized processor devices – accelerators in addition to the central processor unit (CPU).

**CPU, central processor unit** – several universal multi-core processors with common random access memory (RAM).

**Accelerator – specialized processor device** attached to CPU and oriented on high-performance execution of some fragments of a program. As the accelerator has its own random access memory, and CPU memory isn’t accessible to it, then to launch a program fragment on the accelerator CPU moves the fragment and required for it data from CPU RAM to the accelerator RAM.

**GPU** - graphic processor, accelerator of AMD or NVIDIA types.

**CUDA device** - graphic processor of NVIDIA Corporation.

**Computational device, calculator** – CPU or accelerator.

## 1.2 Description of virtual computational systems where DVMH program can be executed

**Virtual multiprocessor system** (or **array of virtual processors**) is the machine provided to user DVMH program by hardware and basic system software. For distributed computer an example of such machine is MPI machine. In this case the multiprocessor system is a group of MPI processes created when the parallel program is launched. One or several MPI

processes can be mapped on the same node of hardware cluster. The number of processors of virtual multiprocessor system and its representation as multidimensional arrangement is specified in command line when the program is launched.

In DVMH model **virtual processor became heterogeneous** – it consists of a virtual host processor, virtual multiprocessor, and several virtual accelerators of different architectures. The virtual host processor and the virtual multiprocessor are mapped on CPU and work on common RAM, but each virtual accelerator has its own memory and is mapped on the hardware accelerator of appropriate architecture.

# 2 Programming model and parallelism model.

DVM [1] model is a basis of DVMH model. The constructions for organization of computations on clusters with accelerators and specifying of data streams, for control of data movement between RAM of central processor and memories of accelerators were added in DVM model.

The parallelism model is based on special form of data parallelism: one program – multiple data streams (OPMD). In the model the same program is executed on all virtual processors, but each processor executes its own subset of statements according to data distribution.

At first, a user defines multidimensional arrangement of virtual processors, on which sections data and computations will be mapped. The section can be varied from the whole processor arrangement up to a single processor.

Then the programmer defines arrays (*distributed data*) and iterations of the loops, that should be distributed on processors. Each iteration of the loop is executed completely on one processor.

The distributed arrays are specified by data mapping directives (see section 4), and parallel loops - by directives of computation distribution (see section 6).

Other variables (distributed by default) are mapped by one copy per each processor (*replicated data*). The replicated variable must have the same value on each processor except for reduction variables (see section 6.2.1) and private variables in parallel loops (including loop indexes).

*Replicated array* is an array that hasn't distributed dimensions – all array elements are located on each processor.

Data distribution defines a set of *local* or *own variables* for each processor. The set of own variables defines *the rule of own computations*: a processor assigns values only to its own variables.

When a processor calculates value of own variable it may need in values of as own variables as not own (*remote*) variables. Special directives are intended for remote data access (see section 7).

Then the programmer defines the code fragments which can be executed on accelerators. These fragments are called *computational regions* or simply *regions*.

A region may be performed on one or several accelerators of the same or different types and/or on CPU. A programmer can specify a list of computational device types the region is supposed to be executed on.

Program fragments out of regions are always executed on the central processor.

For each region data necessary for its execution (input, output, local) are specified.

Data movements between central processor and accelerators are performed automatically in general according to information about used by region data, containing in the region description. To control data movements between accelerators and the central processor special directives are provided.

There are several levels of parallelism in DVMH programs:

- Distribution of data and computations on processors. This level is specified by directives of distribution and redistribution of data and loops.
- Distribution of data and computations on computational devices at the entrance into computational region.
- Parallel processing within certain computational device. This level appears at entrance into a parallel loop inside the computational region.

These levels allows to map efficiently a program on a cluster with multi-core processors and accelerators in its nodes.

## 3 CDVMH language.

CDVMH language is standard C language extended by tools for specification of program parallel execution. Following directives and specifications are added in C language:

- directives of distribution of array elements;
- directives and specifications of distribution of loop iterations;
- directives and specifications of remote data access;
- directives of computational region specification;
- specifications of parallel loops within a region;
- directives of data movement outside the regions;
- directives to specify a program fragments within a region, which should be executed on CPU.

All **dvmh**-directives have the following form:

**dvm** *directive* [ *clause-list* ],

where: *directive* – directive name,
*clause-list* – specification list.

They are introduced by any of three ways processed by modern compilers:

**#pragma** *dvmh-directive*

**_Pragma(“***dvmh-directive***”)**

**__pragma(***dvmh-directive***)**

Below **#pragma** will be used everywhere in examples.

All directives are case sensitive: all keywords are typed only in lowercase letters, names of variables and constants are typed as they are named in serial program.

The syntax of dvmh-directives is described by extended Backus-Naur form. The following notations are used:

| | |
|---|---|
| **::=** | is by definition, |
| *x* \| *y* | alternatives, |
| [ *x* ] | encloses optional element/construction |
| [ *x* ]… | encloses optionally repeated construct that may occur zero or more times, |
| *x-list* | x [**,** x ]… |

```
dvmh-directive              ::= dvm directive

Directive                   ::= specification-directive
                            |   executable-directive
specification-directive     ::= array-directive
                            |   template-directive
                            |   inherit-directive
executable-directive        ::= redistribute-directive
                            |   realign-directive
                            |   getactual-directive
                            |   actual-directive
                            |   region-directive
                            |   parallel-directive
                            |   remote-directive
                            |   hostsection-directive
```

# 4 Data distribution.

CDVMH language supports distribution by blocks (equal and non-equal) and distribution via alignment.

## 4.1 Directive array.

Syntax:

```
array-directive             ::= array [ array-clause-list ]

array-clause                ::= distribute-align-clause
                            |   shadow-clause
distribute-align-clause     ::= distribute-clause
                            |   align-clause
```

Each specification is specified no more once. The order of specifications is arbitrary.

The specifications **distribute** and **align** are mutually exclusive and can't be present simultaneously in **one array directive**.

Specifications **distribute** and **align** specify a distribution of arrays and **shadow** specification – a width of array shadow edges (section 7.2.1). If no specifications or just **shadow** specification is specified in the directive, it means that the array will be distributed later by **redistribute** or **realign** directives (postponed distribution).

**Example 4.1.** Postponed distribution of A array.

**#pragma dvm array**
float A[20][20];

If **shadow** specification is absent the width of shadow edges is set to 1.

Syntax and semantics of separate parts of the directive are described in the following sections:

| | |
|---|---|
| *distribute-clause* | section 4.1.1 |
| *align-clause* | section 4.1.3 |
| *shadow-clause* | section 7.2 |

The directive is inserted before a line with standard declarations of arrays. It is applied only to one following declaration statement, to all arrays declared in the statement.

A number of dimensions specified in **distribute**, **align**, **shadow** specifications must be equal to the number of dimensions in the array declaration.

Before the description of external variables **array** directive is used without specifications:
**#pragma dvm array**

**Example 4.2.** Use of **array** directive for external variable.

**#pragma dvm array**
extern double C[][20];

### *4.1.1 Data distribution. Specification distribute.*

Syntax:

| | | |
|---|---|---|
| *distribute-clause* | ::= | **distribute [** *distribute-axis-rule* ]... |
| *distribute-axis-rule* | ::= | **[ ]** |
| | \| | **[ block ]** |
| | \| | **[ wgtblock (** *var-name* **,** *int-expr* ) **]** |
| | \| | **[ genblock (** *var-name* ) **]** |
| | \| | **[ multblock (** *int-expr* ) **]** |

**Constraint:**

- Distribution format must be specified for each dimension of an array, i.e. a number of *distribute-axis-rule* rules must be equal to the array rank.

Let’s consider distribution formats for one-dimensional arrays and for one-dimensional processor arrangement.

#### *4.1.1.1 Format **block.***

The array dimension is distributed mainly by equal blocks.

Distribution by blocks is performed as follows.

If a number of array elements (N) doesn't exceed the number of processors (P), either one array element or no one is located on each processor.

If a number of array elements (N) is more than the number of processors (P), the number of elements per processor is calculated by the formula:

**[k * N / P] - [(k - 1) * N / P]**

where:
N – a number of array elements,
P – a number of processors,
k – processor number, its value is varied from 1 till P. [x] means taking of integer part.

**Example 4.3.** Distribution by **block** format.

```
#pragma dvm array distribute[block]
  float A[12], B[11], C[5];
```

| Processor | A | B | C |
|---|---|---|---|
| R[0] | 0<br>1<br>2 | 0<br>1 | 0 |
| R[1] | 3<br>4<br>5 | 2<br>3<br>4 | 1 |
| R[2] | 6<br>7<br>8 | 5<br>6<br>7 | 2 |
| R[3] | 9<br>10<br>11 | 8<br>9<br>10 | 3<br>4 |

#### *4.1.1.2 Format* ***genblock.***

A distribution by blocks of different sizes allows to affect loading balance for algorithms performing different volume of computations on different parts of data area.

Let **NB[P]** is integer array of non-negative numbers. It is allowed to use **int** and **long** types for the array description.

The following directive

```
#pragma dvm array distribute[genblock(NB)]
float A[N];
```

splits array **A** on **P** blocks. The **i**-th block of **NB[i]** size is mapped on processor **R[i]**. The equality

$$\sum_{i=0}^{P-1} NB[\,i\,] = N$$

must be satisfied.

**Example 4.4.** Distribution by blocks of different sizes.

```
  int BS[4]={2,4,4,2};
#pragma dvm array distribute[genblock(BS)]
  float A[12];
```

| Processor | A |
|---|---|
| R[0] | 0 |
| | 1 |
| R[1] | 2 |
| | 3 |
| | 4 |
| | 5 |
| R[2] | 6 |
| | 7 |
| | 8 |
| | 9 |
| R[3] | 10 |
| | 11 |

#### *4.1.1.3 Format* ***wgtblock.***

The **wgtblock** format specifies a distribution by blocks according to their relative "weights".

Let **wgtblock** (**WB, NBL**) format is specified.

WB(i) is real (**float** or **double**) non-negative number, defines the weight of i-th block for 1≤ i ≤ NBL-1. If the number of specified weights is equal to NBL, the array is split on NPB blocks and the blocks are distributed on P processors with balancing of sums of block weights on every processor.

Format **block** is special case of **wgtblock (WB, NBL)** format, where WB(i) = 1 for 1≤ i ≤ NBL-1.

The example 4.4 can be rewritten using **wgtblock** format in the following way.

**Example 4.5.** Distribution by blocks according to weights.

```
float WS[12]={2.,2.,1.,1.,1.,1.,1.,1.,1.,1.,2.,2}
#pragma dvm array distribute[wgtblock(WS,12)]
float A[12];
```

In the example 4.5 P = 4 and distribution is identical to the distribution in the example 4.4.

As distinct from distribution by non-equal blocks, distribution by **wgtblock** format can be performed for any number of processors.

There are no restrictions on the size of distributed array. The array size can be larger, equal or smaller than the size of the array with the weights.

**Example 4.6.** Distribution by blocks according to weights.

```
float WS[6]={2.,1.,1.,1.,1.,2.}
#pragma dvm array distribute[wgtblock(WS,6)]
float A[12];
```

#### *4.1.1.4 Format **multblock.***

Format **multblock ( m )** specifies that a block size on each processor must be multiple to the given number **m**. For distribution by this format the array size (N) must be multiple to **m**.

A distribution is performed as follows – at first the array is divided into blocks by **m** elements, and then these blocks are distributed according to the rules of **block** distribution, only not the array elements but the blocks are distributed.

**Example 4.6.** Distribution by **multblock** format.

```
#pragma dvm array distribute[multblock(2)]
  float A[4], B[8], C[16];
```

| Processor | A | B | C |
|---|---|---|---|
| R[0] | **0** | **0** | **0** |
| | **1** | **1** | **1** |
| | | | **2** |
| | | | **3** |
| R[1] | **2** | **2** | **4** |

| | | |
|---|---|---|
| | | |
| 3 | 3 | 5 |
| | | 6 |
| | | 7 |
| R[2] | 4 | 8 |
| | 5 | 9 |
| | | 10 |
| | | 11 |
| R[3] | 6 | 12 |
| | 7 | 13 |
| | | 14 |
| | | 15 |

#### *4.1.1.5 The format isn't specified.*

If a format isn't specified (square brackets [ ] are empty), it means that the dimension will be completely localized on each processor (non-distributed dimension).

### ***4.1.2 Multidimensional distributions.***

For multidimensional distributions mapping format is specified for each dimension. The following correspondence is established between dimensions of distributed array and the processor arrangement.

Let the processor arrangement has **n** dimensions. Number the distributed dimensions (not formatted as **[ ]**) from left to right $\mathbf{d_1, ..., d_k}$. Then dimension $\mathbf{d_i}$ will be mapped on *i*-th dimension of processor arrangement. The condition **k≤n** must be satisfied.

**Example 4.7.** One-dimensional distribution.

**2 processors**

```
#pragma dvm array distribute[block][]
  float  A[100][100];
```

| | A blocks | Processors |
|---|---|---|
| 1 | **A[0: 49][0:99]** | **0** |
| 2 | **A[50:99][0:99]** | **1** |

**Example 4.8.** Two-dimensional distribution.

**2 * 2 processors**

```
#pragma dvm array distribute[block][block]
  float  A[100][100];
```

A blocks

| | |
|---|---|
| **1** | **2** |
| **3** | **4** |

Processors

| | 0 | 1 |
|---|---|---|
| 0 | **1** | **2** |
| 1 | **3** | **4** |

.

#### 4.1.2.1 Distribution of dynamically allocated arrays

For dynamic arrays the distribution can be only postponed. The **redistribute** directive (section 5.1) for dynamically allocated array can be executed only after malloc function call which will allocate the array in the memory.

**Example 4.9.** Distribution of dynamically allocated arrays.

```
....................................
#pragma dvm array
 float (*A)[N + 1];

  /* create array */
 A = malloc(N * (N + 1) * sizeof(float));
#pragma dvm redistribute (A[block][block])
....................................
   free (A);
```

### 4.1.3 Distribution by alignment. Specification align.

Alignment of array **A** with a distributed array **B** brings in accordance to each element of array **A** an element or section of array **B**. Array section is the array subset, which is the array itself. Distribution of array **B** elements defines the distribution of array **A** elements. If element of array **B** is distributed on a processor, the element of **A**, corresponding to element of **B** via alignment, will be also distributed on the same processor.

The method of distribution via alignment performs the following two functions.

1. Identical distribution of arrays of same shape to the same processor arrangement does not always guarantee that correspondent elements will be located on the same processor. It forces to specify remote access (see section 7) where it isn't exist perhaps. Only alignment of corresponding elements of the arrays guarantees their location on the same processor.
2. Several arrays can be aligned with the same array (base array). Redistribution of base array by **redistribute** directive (see section 5.1) will cause corresponding redistribution of all arrays of the group.

Syntax:

| | | |
|---|---|---|
| *align-clause* | ::= | **align (** [ *align-axis-name* ]... **with** *templ-align-spec* **)** |
| *align-axis-name* | ::= | **[ ]** |
| | \| | **[** *align-dummy* **]** |
| *align-dummy* | ::= | *Name* |
| *templ-align-spec* | ::= | *var-name* [ *templ-axis-spec* ]... |
| *templ-axis-spec* | ::= | **[ ]** |
| | \| | **[** *int-expr* **]** |
| | \| | **[** *align-dummy-use* **]** |

| | | |
|---|---|---|
| *align-dummy-use* | ::= | [ *primary-expr* * ] *align-dummy* [ *add-op primary-expr* ] |
| *primary-expr* | ::= | *int-constant* |
| | \| | *var-name* |
| | \| | ( *int-expr* ) |
| *add-op* | ::= | + |
| | | - |

**Constraints:**

- A length of *align-axis-name* list must be equal to a rank of aligned array.
- A length of *templ-axis-spec* list must be equal to a rank of base array *var-name.*

Let the alignment of two arrays is specified by the directive

**#pragma dvm array align ( [$d_0$]…[$d_n$] with B[$ard_0$]…[$ard_m$] )**
**float A[**$A_0$**]…[**$A_n$**];**

$\mathbf{d_i}$ – specification of *i*-th dimension of aligned array **A**,
$\mathbf{ard_j}$ – specification of *j*-th dimension of base array **B**.
$\mathbf{A_i}$ – size of *i*-th dimension.

If $\mathbf{d_i}$ is specified by integer variable I, then there must be one and only one dimension of array **B**, specified by linear function $\mathbf{ard_j = a*I + b}$.Therefore, the number of array **A** dimensions, specified by identifiers (*align-dummy-use*) must be equal to the number of array **B** dimensions specified by the linear function.

Let a size of *i*-th dimension of array **A** is $A_i$, and the size of *j*-th dimension of array **B** specified by the linear function **a*I + b** is $B_j$. Since parameter I is defined over a set of values 0 … $A_i$-1, the following conditions must be satisfied:

$b \geq 0$; $a*(A_i - 1) + b < B_j$

If $\mathbf{d_i}$ isn't set ([ ] are specified), then *i*-th dimension of array **A** will be local on each processor independently from any distribution of array **B** (analog of local dimension in **distribute** specification).

If $\mathbf{ard_j}$ **=[ ]**, then array **A** will be replicated along *j*-th dimension of array **B**.

If $\mathbf{ard_j}$ = ***int-expr***, then array **A** is aligned with the section of array **B**.

**Example 4.10.** Aligning arrays.

```
#pragma dvm array distribute [block][block]
 float B[10][10];
#pragma dvm array distribute [block]
 float D[20];
/* aligning the vector A with the first line of B) */
#pragma dvm array align ([i] with B[0][i])
 float A[10];
/* replication of the vector aligning it with each line */
#pragma dvm array align ([i] with B[][i])
 float F[10];
```

```
/* collapse: each column corresponds to the vector element */
#pragma dvm array align ([][i] with D[i])
 float C[20][20];
/* alignment using stretching */
#pragma dvm array align ([i] with D[2*i])
float E[10];
/* alignment using rotation and stretching */
#pragma dvm array align ([i][j] with C[2*j][2*i])
float H[10][10];
```

Let the alignments *f1* and *f2* are specified; *f2* aligns array **B** with array **C**, and *f1* aligns array **A** with array **B**. The arrays **A** and **B** are considered as aligned with array **C** by definition. The array **B** is aligned directly by the function *f2* and array **A** is aligned indirectly by composite function *f1(f2)*. Therefore applying **realign** directive to the array **B** doesn't cause redistribution of array **A**.

#### *4.1.4 Non-distributed data*

If **array** directive isn't specified for data, then these data are distributed on each processor (full replication). The same distribution can be specified by **array** directive with **distribute** specification, with format **[ ]** for all dimensions. But in this case an access to data will be less effective.

### 4.2 Directive template.

If values of linear function **a*I** + **b** exceed the limits of base array dimension, it is necessary to define a dummy array named an alignment template. Then it is necessary to align both arrays with the template. The template is distributed by **distribute** specification or **redistribute** directive. The template elements don’t require real memory, they specify a processor on which the elements of aligned arrays must be mapped. If **distribute** specification isn't specified in the directive, it means that the template will be distributed later by **redistribute** directive.

The alignment template is defined by the following directive:

Syntax:

| | |
|---|---|
| *template-directive* | **::= template** [ *template-decl* ] |
| *template-decl* | **::=** [ *size-spec* ]… [ *distribute-clause* ] |
| *size-spec* | **::= [ ]** |
| | \| **[** *int-expr* **]** |

**Constraint.**

- Size of *size-spec* list should be equal to a number of dimensions of aligned array.

If empty brackets "[ ]" are specified instead of a template dimension size, it means that the template is dynamic. In this case empty brackets should be specified for all dimensions of the array.

The template should have void* type in a program, and the name of this variable is the template name.

**Note.**
The possibility to specify dynamic templates is in implementation process.

Consider the following example.

**Example 4.10.** Aligning with template.

```
#pragma dvm template[102] distribute [block]
void *TABC;
#pragma dvm array align ([i] with TABC[i])
float B[100];
#pragma dvm array align ([i] with TABC[i+1])
float A[100];
#pragma dvm array align ([i] with TABC[i+2])
float C[100];
        for (i = 1; i < 98; i++)
            A[i] = C[i-1] + B[i+1];
```

To avoid exchanges between processors, the elements A[i], C[i-1] and B[i+1] must be located on the same processor. It is impossible to align arrays C and B with array A, because alignment functions i-1 and i+1 cause bounds violation of array A. Therefore the template TABS is declared. The elements of arrays A, B and C, that must be allocated on the same processor, are aligned with the same element of the template.

Before a description of external variables **template** directive is used without specifications:

```
#pragma dvm template
extern void *templName;
```

### *4.2.1 Aligning dynamically allocated arrays*

For dynamic arrays the aligning can be only postponed. The **realign** directive for dynamically allocated array can be executed only after execution of the statement which will allocate the array in memory.

Let sequence of alignments by **align** directives is specified

$A_1\ f_1\ A_2\ f_2\ .\ .\ .\ f_{N-1}\ A_N$

where: $f_i$ – aligning function,
$A_i$ – dynamically allocated array.

Then dynamic arrays must be allocated in reverse order, i.e. $A_N$ will be allocated at first and $A_1$ will be allocated latest. The memory for $A_N$ array is freed latest, after the memories for all arrays aligned on it were freed.

**Example 4.11.** Alignment of dynamically allocated arrays.

```
…………………………………….
#pragma dvm array
```

```
   float (*A)[N + 1];
 #pragma dvm array
   float (*X);

   /* create arrays */
   A = malloc(N * (N + 1) * sizeof(float));
   X = malloc(N * sizeof(float));
#pragma dvm redistribute (A[block][])
#pragma dvm realign(X[i] with A([i][])
……………………………………
   free(X);
   free(A);
```

# 5 Directives of data redistribution.

## 5.1 Data redistribution. Directive redistribute.

Syntax:

| | |
|---|---|
| *redistribute-directive* | **::= redistribute (** *var-name* [ *distribute-axis-rule* ]... ) |
| *distribute-axis-rule* | **::= [ ]** |
| | **\|  [ block ]** |
| | **\|  [ wgtblock** ( *var*-name **,** *int-expr* ) **]** |
| | **\|  [ genblock** ( *var-name* ) **]** |
| | **\|  [ multblock (** *int-expr* ) **]** |

**Example 5.1.** Data redistribution.

```
#pragma dvm array distribute[block][]
double A[L][L];

…………………………………
#pragma dvm redistribute (A[block][block])
```

## 5.2 Data realignment. Directive realign.

Syntax:

| | |
|---|---|
| *realign-directive* | **::= realign (** *var-name align-rule* ) [ **new_value** ] |
| *align-rule* | **::=** [ *align-axis-name* ]... **with** *templ-align-spec* |
| *align-axis-name* | **::= [ ]** |
| | \|  **[** *align-dummy* **]** |
| *align-dummy* | **::=** *name* |
| *templ-align-spec* | **::=** *var-name* [ *templ-axis-spec* ]**...** |

*templ-axis-spec* ::= [ ]
| [ *int-expr* ]
| [ *align-dummy-use* ]

*align-dummy-use* ::= [ *primary-expr* * ] *align-dummy* [ *add-op primary-expr* ]

*primary-expr* ::= *int-constant*
| *var-name*
| ( *int-expr* )
*add-op* ::= +
| -

The **new_value** specification in **realign** directive means that the array contents may be not saved after the directive execution, and values of the array elements will be undefined.

**Example 5.2.** Data realignment.

**#pragma dvm array distribute[block][block]**
double A[L][L];
**#pragma dvm array align([i][] with B[i][])**
double B[L][L];
……………………………………
**#pragma dvm realign(B[i][j] with A([i][j])**

# 6 Distribution of computations. Parallel loops.

## 6.1 Parallel loop definition.

The model of CDVMH program execution is SPMD model (single program, multiple data streams). The same program is loaded on all the processors, but according to *own computation rule* each processor performs only those assignment statements that update the variables located on the processor (own variables).

Thus computations are distributed in accordance with data mapping (data parallelism). If a variable is replicated, an assignment statement is performed by all the processors. In the case of distributed array the assignment statement is performed only by the processor (or the processors) where the corresponding array element is located.

Identification of "own" statements and missing "others" ones can cause essential overhead during a program execution. Therefore the specification of computation distribution is permitted only for loops, satisfying the following requirements:

- the loop is tightly nested loop with rectangular index space;
- distributed dimensions of arrays are indexed only by regular expressions of the form a*I + b , where I is the loop index;
- left sides of assignment statements of one loop iteration are located at the same processor and, therefore, the loop iteration is executed on the processor entirely.

- there is no data dependencies except for reduction dependence and regular dependence along distributed dimensions;
- left side of assignment statement is a reference to distributed array, reduction variable (section 6.2.1) or private variable (see section 6.2.2);
- there are no I/O statements, exit statements outside the loop and dvmh-directives inside the loop body and in all functions called in the loop.

A loop, satisfying these requirements, will be called *parallel loop*. An index variable of serial loop, surrounding a parallel loop or nested in the loop, can index only the local (replicated) dimensions of distributed arrays.

The following restrictions are also imposed on the loop:

- The loop should be described by **for** construction.
- Three sections - loop variable with initialization, a completion condition and expression modifying loop variable - should be specified.
- Exactly one assignment should be in initialization section, a description of the variable is allowed in the section (but only one and only with initialization): for (int i = getStart()+abs(S); i < N; i++). The right side of the assignment should be calculated without side effects, doesn't depend on the loop variables or data updated in the loop, and also on values of distributed arrays.
- Continuation condition must be a comparison on <, <=, >, >= with the same variable in the left part which was in the assignment in initialization section. The right side of continuation condition should be calculated without side effects and doesn't depend on the loop variables, or data updated in the loop, and also on values of distributed arrays.
- There should be an operator -=, - ++, += in incremental section. Prefix and postfix expressions are permitted. This operator should update the same variable which is assigned in the first section and which is compared in middle section. The right side should be calculated without side effects and doesn't depend on the loop variables, or data updated in the loop, and also on values of distributed arrays.

## 6.2 Distribution of loop iterations. Directive parallel.

Parallel loop is specified by the following directive:

| | |
|---|---|
| *parallel-directive* | **::= parallel** *parallel-map* [ *parallel-clause-list* ] |
| *paralle-map* | **::= (** *int-constant* **)** |
| | \| **(** [ *align-axis-name* ]... **on** *templ-align-spec* **)** |
| *templ-align-spec* | **::=** *var-name* [ *templ-axis-spec* ]... |
| *templ-axis-spec* | **::= [ ]** |
| | \| **[** *int-expr* **]** |
| | \| **[** *align-dummy-use* **]** |

*parallel-clause* ::= *private-clause*
| *reduction-clause*
| *shadow-renew-clause*
| *across-clause*
| *remote-clause*
| *cuda-clause*
| *stage-clause*

Directive **parallel** is inserted just before the loop header (outer loop of a nest). If the mapping rule is specified as *parallel-map*, the directive distributes the loop iterations in accordance with array or template distribution. The directive semantics is similar to the semantics of **align** specification, where index space of aligned array is replaced by the loop index space. The order of loop indexes in *align-axis-name* list must corresponds to the order of corresponding loop statements in tightly nested loop. If a number is specified instead of the mapping rule, then the loop will be parallel, but not distributed, and the number will specify a quantity of the nest loops associated with the directive.

The syntax and semantics of the directive clauses are described in the following sections:

| | |
|---|---|
| *private-clause* | section 6.2.2, |
| *reduction-clause* | section 6.2.1, |
| *shadow-renew-clause* | section 7.2.2, |
| *remote-access-clause* | section 7.3, |
| *across-clause* | section 7.2.3, |
| *cuda-clause* | section 8.2.1, |
| *stage-clause* | section 7.2.3. |

**Example 6.1.** Distribution of loop iterations.

```
#pragma dvm array distribute [block][block]
float B[N][M+1];
#pragma dvm array align ([i][j] with B[i][j+1])
float A[N][M], C[N][M], D[N][M];
. . .
#pragma dvm parallel ([i][j] on B[i][j+1])
for (i = 0; i < N; i++)
   for (j = 0; j < M; j++)
   {
         A[i][j] = D[i][j] + C[i][j];
         B[i][j+1] = D[i][j] – C[i][j];
   }
```

The loop satisfies all conditions of parallel loop. In particular, both assignment statements are executed on the same processor due to the alignment of arrays **A** and **B**.

If left sides of assignment statements are located on different processors (distributed iteration of the loop) then the loop must be split on several loops.

### 6.2.1 Reduction operations and variables. Specification reduction.

Programs often contain loops with so called *reduction operations*: array elements are summarized in some variable, or maximum (minimum) value of them is calculated. The iterations of such loops can be distributed and executed in parallel, if to use **reduction** specification.

Syntax:

| | |
|---|---|
| *reduction-clause* | ::= **reduction** **(** *red-spec-list* **)** |
| *red-spec* | ::= *red-func* **(** *red-variable* **)** |
| | \| *red-loc-func* **(** *red-variable***,** *loc-variable* [ **,** *size* ] **)** |
| *red-variable* | ::= *array-name* |
| | \| *scalar-variable-name* |
| *loc-variable* | ::= *array-name* |
| | \| *scalar-variable-name* |
| *Size* | ::= *int-constant* |
| | |
| *red-func* | ::= **sum** |
| | \| **product** |
| | \| **max** |
| | \| **min** |
| | \| **and** |
| | \| **or** |
| | \| **xor** |
| *red-loc-func* | ::= **maxloc** |
| | \| **minloc** |

**Constraints:**

- Distributed arrays can’t be used as reduction variables.
- Reduction variable is calculated in each iteration of the loop in statements of certain type - reduction statements. For **sum** and **product** reduction operations the reduction statements are the statements of sum and product calculation correspondently (for example, "+=" and "*=" operators). For **max** and **min** reduction operations they are conditional statements where minimum or maximum value are calculated (for example, "v=(v<A?A:v)", "v=(v>A?A:v)"). For **and**, **or** and **xor** reduction operations they are assignment statements where a value of logical expression with use of appropriate bit-by-bit operation is calculated (for example, "& =", "|=", "^="). For **maxloc** and **minloc** reduction operations they are conditional statements where maximum or minimum value of an array are calculated and assignment statements where coordinates of maximum or minimum value of array elements are kept. Second parameter of **maxloc** and **minloc** functions (*loc-variable*) is a variable describing coordinates of found array element with maximum (correspondently, minimum) value. For one-dimensional array it is a scalar, for multidimensional array – one-dimensional array which size is equal to a number of the array dimensions. A value of i-th element of the array is a coordinate along i-th dimension. The value of second parameter (*loc-variable*) can be modified only together with modifying of reduction variable value (*red-variable*).

**Example 6.2.** Reduction specification.

```
. . . . . . .
S = 0;
/* array A is distributed. According to own computation rule (section
6.3) it is necessary to use remote access directive (section 7.3) */
#pragma dvm remote_access (A[0])
{
X = A[0];
Y = A[0];
}
MINi = 0;
#pragma dvm parallel ([i] on A[i]) reduction
(sum(S), max(X), minloc(Y, MINi))
  for (i = 1; i < N; i++)
{
     S = S + A[i];
      if(A[i] > X)
        X = A[i];
        if(A[i] < Y) {
                Y = A[i];
                MINi = i;
        }
}
```

**Note.**
A preprocessor substitutes macro-definitions instead of all dvmh-directives. If **min** and/or **max** reductions are used in a program, then it is impossible to use the macros with the same names. It will cause compilation errors.

### *6.2.2 Private variables. Specification private.*

The **private** specification declares a variable as private. The variable is called private if its usage is localized within one iteration of a loop.

The reduction variable can't have **private** attribute.

Syntax:

*private-clause* ::= **private (** *var-name-list* **)**

## 6.3 Computations outside parallel loop

The computations outside parallel loop are performed according to own computation rule.

Assignment statement

**lh = rh**;

can be executed by some processor, only if **lh** is located on it.

If **lh** is an element of distributed array, then such statement (statement of own computations) will be executed only by that processor (or those processors) where this element of the distributed array is located. All data used in **rh** expressions should be located also at the same processor.

If any data in **rh** expressions are not located on the processor, they must be specified in remote access directive prior this statement (see section 7.3).

**Example 6.3.** Own computations.

```
#define N 100
#pragma dvm array distribute [block][]
float A[N][N+1];
#pragma dvm array align ([i] with A[i][N])
float X[N];
. . .
/* Reverse substitution of Gauss algorithm
   own computations are outside the loops

   own computation statement
   left and right sides are located on the same processor */
X[N-1] = A[N-1][N] / A[N-1][N];
for (j = N - 2; j >= 0; j--)
/* remote_access specification (section 7.3) in the parallel loop
specifies remote data (X[j+1]) for all processors the array A is
distributed on. */
#pragma dvm parallel([i] on A[i][]) remote_access(X[j+1])
 for (i = 0; i <= j; i++)
  A[i][N] = A[i][N] – A[i][j+1] * X[j+1];
/* Own computations in serial loop,
        surrounding the parallel loop */
 X[j] = A[j][N] / A[j][j];
}
```

Note, that A[j],[N+1] and A[j],[j] are localized on the processor, where X[j] is located.

# 7 Remote data specification.

## 7.1 Definition of remote references.

Data, located on one processor, but used on other one are called *remote data*. The references to such data are called *remote references*. Consider the assignment statement in a loop body:

A[inda] = B[indb]

where:
A, B - distributed arrays,
inda, indb - index expressions.

In DVMH model this statement will be executed by the processor, where A[inda] element is located. B[indb] reference isn't a remote reference, if A[inda] and B[indb] elements are located on the same processor. This is guarantied only if array B is aligned with distributed array A. If the alignment is impossible or wasn't performed, B[indb] reference should be specified as remote one. For multi-dimensional arrays this rule is applied to every distributed dimension.

By degree of processing efficiency remote references are divided on two types: **shadow** and **remote**.

If A and B arrays are aligned and

inda = indb ± d ( d – positive integer constant),

then remote reference B[indb] belongs to **shadow** type.

Remote reference to multi-dimensional array belongs to **shadow** type, if the arrays are aligned and the rule

inda = indb ± d ( d – positive integer constant),

is performed for each dimension.

Remote references, that don't belong to **shadow** type, are **remote** type references.

Special remote references are references to reduction variables (see 6.2.1). They belong to **reduction** type references. These references can be used only in a parallel loop.

## 7.2 References of shadow type.

### *7.2.1 Specification of array with shadow edges.*

Remote reference of **shadow** type means, that remote data processing will be performed using "shadow" edges. Shadow edge is a buffer, that is continuous prolongation of the array local section in a processor memory (see **fig. 7.1**). Consider the statement

**A[i] = B[i + d2] + B[i – d1]**

where **d1**, **d2** - integer positive constants. If both references to array **B** are remote references of **shadow** type, it is necessary to specify for array **B** specification **shadow[ d1 : d2 ]** where **d1** is left edge width, and **d2** is right edge width. For multidimensional arrays the edges for each dimension should be specified. In the array description maximum width among all remote references of **shadow** type is set in shadow edge specification. The width of both shadow edges of distributed array is equal to 1 for each distributed dimension by default.

The **shadow** specification is specified in **array** directive.

Syntax:

*shadow-clause* **::= shadow** [ *shadow-edge* ]...

*shadow-edge* **::= [** *width* **]**
**|** **[** *low-width* **:** *high-width* **]**
*width* **::=** *int-expr*
*low-width* **::=** *int-expr*
*high-width* **::=** *int-expr*

**Constraints:**

- The width of left shadow edge (*low-width*) and width of right shadow edge (*high-width*) must be integer non-negative expressions.
- If one width (*width*) is specified, then the expression should be without side effects.

The specification of shadow edge width as *width* is equivalent to *width* : *width*.

#### 7.2.2 Specification of renewing shadow edges for loop.

The specification of shadow edge renewing is a part of **parallel** directive:

*shadow-renew-clause* **::= shadow_renew (** *shadow-renew-spec-list* **)**

*shadow-renew-spec* **::=** *specvar-name* [ *shadow-edge* ]... [ **( corner )** ]
*shadow-edge* **::= [** *low-width* [ **:** *high-width* ] **]**
*low-width* **::=** *int-expr*
*high-width* **::=** *int-expr*

**Constraints:**

- The width of renewed shadow edges must not exceed the maximum width described in **shadow** specification.
- If shadow edge widths aren't specified, then maximum widths are used.

Specification performing consists in shadow edge renewing by the values of remote variables prior loop execution.

**Example 7.1.** Specification of **shadow** references without corner elements

```
#pragma dvm array distribute [block]
float A[100];
#pragma dvm array align ([i] with A[i]), shadow[1:2]
float B[100];
.  .  .
#pragma dvm parallel ([i] on A[i]) shadow_renew(B)
for (i = 1; i < 98; i++)
        A[i] = (B[i-1] + B[i+1] + B[i+2]) / 3.;
```

To renew shadow edges the maximum widths 1:2 specified in **shadow** specification are used.

Two buffers, that are continuous prolongation of the array local section, are created on each processor. The width of left shadow edge is equal to 1 element (for B[i-1]), the width of right shadow edge is equal to 2 elements ( for B[i+1] and B[i+2]). If before the loop execution to perform exchange between processors, the loop can be executed on each

processor without replacement of the references to the arrays by the references to the buffer.

For multidimensional distributed arrays shadow edges can be specified for each dimension. A special case is when it is required to renew "a corner" of shadow edges. In this case additional **corner** parameter is required.

**Example 7.2.** Specification of **shadow** references with corner elements.

```
#pragma dvm array distribute [block][block]
float A[100][100];

#pragma dvm array align ([i][j] with A[i][j])
 float B[100][100];
. . .
#pragma dvm parallel ([i][j] on A[i][j]) shadow_renew(B(corner))
for (i = 1; i < 99; i++)
    for (j = 1; j < 99; j++)
        A[i][j] = (B[i][j+1] + B[i+1][j] + B[i+1][j+1]) / 3.;
```

The width of shadow edges of array B is equal to 1 element for each dimension by default. Since "corner" reference B[i+1][j+1] exists, the **corner** parameter is specified.

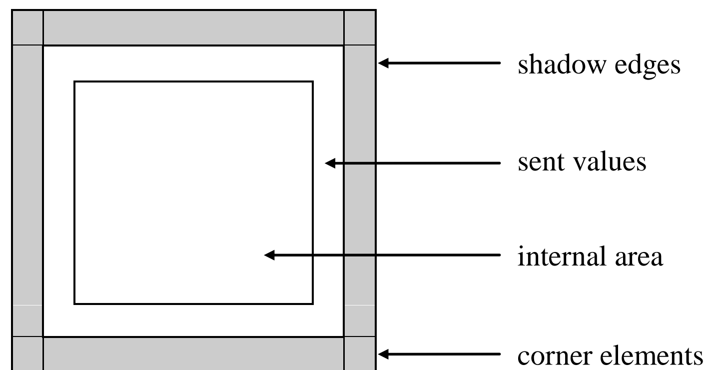


**Fig. 7.1**. Scheme of array local section with shadow edges.

### *7.2.3 Specification across of dependent references of shadow type for single loop. Specification stage.*

Consider the following loop

```
for (i = 1; i < N-1; i++)
    for (j = 1; j < N-1; j++)
        A[i][j] =(A[i][j-1]+A[i][j+1]+A[i-1][j]+A[i+1][j])/4.;
```

A dependence on A array data (information dependence) exists between loop iterations with indexs *i1* and *i2* ( $i1<i2$ ), if both these iterations access to the same array element by write-read or read-write scheme.

If iteration *i1* writes a value and iteration *i2* reads this value, then flow dependence or simply dependence $i1 \rightarrow i2$ exists between the iterations.

If iteration *i1* reads the "old" value and iteration *i2* writes the "new" value, then anti dependence *i1*← *i2* exists between the iterations.

In both cases iteration *i2* can be executed only after iteration *i1*.

The value *i2 - i1* is called a range or length of the dependence. If for any iteration *i* dependent iteration *i + d* (*d* is integer constant) exists, then the dependence is called regular one, or dependence with constant length.

The loop with regular dependencies on distributed arrays can be distributed by **parallel** directive, using **across** specification.

Syntax:

| | |
|---|---|
| *across-clause* | **::= across (** *across-spec-list* **)** |
| *across-spec* | **::=** *var-name* [ *subscript-range* ]... |
| *subscript-range* | **::= [** *flow-dep-length* [ **:** *anti-dep-length* ] **]** |
| *flow-dep-length* | **::=** *int-expr* |
| *anti-dep-length* | **::=** *int-expr* |

All distributed arrays with regular data dependences are listed in **across** specification. The length of flow dependence (*flow-dep-length*) and the length of anti dependence (*anti-dep-length*) are specified for each dimension of the array. Zero value of dependence length means that there is no data dependence.

**Example 7.3.** Specification of loop with regular data dependence.

```
#pragma dvm parallel([i][j] on A[i][j]) across(A[1:1][1:1])
  for (i = 1; i < N-1; i++)
    for (j = 1; j < N-1; j++)
         A[i][j]=(A[i][j-1]+A[i][j+1]+A[i-1][j]+A[i+1][j])/4.;
```

Flow and anti dependencies of length 1 exist for all dimensions of A array.

Specification **across** is implemented via shadow edges. The length of anti dependence defines a width of high edge renewing, and length of flow-dependence defines width of low edge renewing. High edges are renewed prior the loop execution (as for **shadow_renew** specification). Low edges are renewed during the loop execution in process of calculation of remote data values. It allows to organize so-called wave calculations for multidimensional arrays. Actually, **across-**references are a subset of **shadow**-references, that have data dependence. Such data dependence generally requires serial execution of loop iterations with passing of renewed common data values from each processor already performed its job to the next one. However in many cases, for example, when the two-dimensional loop is distributed on a line of processors along one dimension it is possible to parallelize such loop efficiently if to organize its pipeline execution. The idea of pipeline execution is the first processor executes a part of "own" loop iterations and sends the corresponding portion of updated common data to the next processor. After receiving these data the second processor begins to execute "own" loop iterations in parallel with execution of second portion of iterations by the first processor, etc. The optimal size of iteration portion depends on a number of the loop iterations, each iteration runtime, a number of

processors, and also latency and capacity of communication environment. The runtime system supports the calculation of optimal portion of iterations and the appropriate splitting of one or several dimensions of the loop, and also passing of updated common data between processors.

The user can specify a quantity of the pipeline steps. The **stage** specification is provided for this purpose.

Syntax:

*stage-clause* **::= stage (** *int-expr* **)**

### 7.3 Remote references of remote type. Directive remote_access.

Remote references of **remote** type are specified by **remote_access** directive.

Syntax:

*remote-directive* **::= remote_access (** *rma-spec-list* **)**

*rma-spec* **::=** *templ-align-spec*

The **remote_access** directive can appear as a separate directive (its operating area is a next statement) or as additional specification in **parallel** directive (its operating area is parallel loop body).

In the limits of statement below or of a parallel loop the compiler replaces all occurrences of remote reference by the reference to the buffer. The remote data are passed just before execution of the statement or the loop.

**Example 7.4.** Specification of remote references of **remote** type.

```
#pragma dvm array distribute[][block]
float A[100][100], B[100][100];
.  .  .
#pragma dvm remote_access(A[50][50])
{
X = A[50][50];
}
.  .  .

#pragma dvm remote_access(B[99][99])
{
 A[0][0] = B[99][99];
}
.  .  .
#pragma dvm parallel([i][j] on A[i][j]) remote_access(B[][N])
   for (i = 0; i < 100; i++)
      for (j = 0; j < 100; j++)
            A[i][j] = B[i][j] + B[i][N];
```

First two **remote_access** directives specify remote references for separate statements. The **remote_access** specification in the parallel loop specifies remote data (the matrix column) for all processors the array A is distributed on.

**Note.** The **remote_access** directive for parallel loops and regions isn't implemented yet.

# 8 Distribution of computations. Computational region.

The computational region specifies a fragment of a program (with one entrance and one exit) for possible execution on one or several computational devices.

The region can be executed on one or several accelerators at once and/or on CPU. Any region can be executed on CPU. For performance on different accelerators the various additional restrictions can be imposed on the region. For example, any region without I/O statements and calls of external procedures can be executed on CUDA device.

Nested (statically or dynamically) regions aren't permitted.

The distributed arrays are distributed on selected calculators (taking into account weights (see section 14.2.1) and high-speed performance of calculators), non-distributed data are replicated. The iterations of parallel loops inside the region are divided between the calculators selected for the region execution according to a rule of parallel loop mapping, specified in the parallel loop directive.

## 8.1 Directive region.

Syntax.

| | |
|---|---|
| *region-directive* | ::= **region** [ *region-clause-list* ] |
| *region-clause* | ::= *in-out-local-clause* |
| | \| *targets-clause* |
| | \| *async_clause* |
| *in-out-local-clause* | ::= **in** ( *array-range-list* ) |
| | \| **out** ( *array-range-list* ) |
| | \| **local** ( *array-range-list* ) |
| | \| **inout** ( *array-range-list* ) |
| | \| **inlocal** (*array-range-list*) |
| *array-range* | ::= *var-name* [ *subscript-range* ]… |
| *subscript-range* | ::= **[** *int-expr* [ **:** *int-expr* ] **]** |
| *targets-clause* | ::= **targets** ( *target_name-list* ) |
| *target_name* | ::= **CUDA** |
| | \| **HOST** |
| *async_clause* | ::= **async** |

The *in-out-local-clause* specifications are intended to specify a direction of data usage in a region:

| | | |
|---|---|---|
| **in** | - | input data: newest values of these data should be in the region; |
| **out** | - | output data: the values of specified variables are updated in the region and may be used further; |

**local** - local data: the values of specified variables are updated in the region, but these updates won't be used further;
**inout** - abbreviated notation of two specifications **in** and **out**;
**inlocal** - abbreviated notation of two specifications **in** and **local**.

It isn't necessary to specify all variables used in a region in *in-out-local-clause* specifications. For the variables used in the region, but not listed in specifications, the following rules are applied by default:

- all used arrays are considered to be used wholly (subarrays aren't selected);
- **in** attribute is assigned to any variable used for reading;
- **inout** attribute is assigned to any variable used for writing;
- **inout** attribute is assigned to any variable, whose direction of usage isn't determined;
- **local** and **out** attributes aren't assigned.

If only **in** attribute is specified for a variable (**out** or **local** isn't specified), it means that there are no any writing to such variable in the region and it doesn't updated during the region execution.

In *in-out-local-clause* specifications for each dimension of an array it is possible to use a section (*subscript-range*) by specifying of a range of indexes.

Composite specifications, for example, **out** (s[1:5]), **out** (s[7:10]) or **in** (s[1:5]), **out** (s[6:10]) and the crossed specifications, for example, **out** (s[1:6]), **out** (s[3:10]) or even **out** (s[1:6]), **out** (s[3:5]) are allowed.

The conflicting specifications, such as **out (**v), **local** (v), aren't allowed.

The **targets** specification is intended to specify a list of calculator types where it is supposed to execute the region. Now *target_name* in **targets** specification can have only two values:

**CUDA** - the region is supposed to be executed on CUDA device;
**HOST** - the region is supposed to be executed on CPU.

Only one **targets** specification can be in **region** directive.

This specification restricts a set of types of computational devices for which use the region will be prepared by the compiler. Really the region can be executed only on accelerators available to DVMH program (or on CPU), their number and types are specified in environment variables at program startup. The concrete performers of the region (from a number of available calculators, the region programs were generated for) are selected during the program execution.

The **async** specification is intended to specify asynchronous execution of a region. When the region is launched in any mode (synchronous, asynchronous) it is necessary to wait a completion of earlier launched region if the region will update data (according to **in**, **out**, **local**, **inout**, **inlocal** specifications) used by earlier launched region or use for writing or reading the data (according to **out**, **inout**, **local**, **inlocal** specifications) updated by earlier launched region. Control will go to next statement after synchronous region only when the current region will be completed. Control can go to the next statement after asynchronous region, without waiting for its completion (or even its start).

**Note.** The **async** specification is in implementation process now.

## 8.2 Description of region constructions.

The computational region has the form:

**#pragma dvm region** [ *clause-list* ]

{
<region inner>
}

Region contents (<region inner>) is a sequence of the constructions following one after another - in random order and in any quantity:

- parallel loop;
- serial group of statements;
- host section.

The region can be empty, in this case <region inner> doesn't contain any construction.

### *8.2.1 Parallel loop.*

Parallel loop is the most important part of computational region.

In a parallel loop inside a region the additional *cuda-clause* specification can be specified.

Syntax:

*cuda-clause* **::= cuda_block (** *int-expr* [ **,** *int-expr* [ **,** *int-expr* ] ] )

The **cuda_block** specification is intended to specify the size of thread block of CUDA device. If one integer expression is specified then the block is one-dimensional. If two or three integer expressions through a comma are specified then the block will have correspondent rank.

### *8.2.2 Serial group of statements.*

Each statement of serial group of statements is executed on all calculators, selected for the region execution, except for the case of distributed data modification - then the rule of own computation works.

**Note.** Control of own computations in serial group of statements in a region isn't implemented yet. The statements where the distributed data are modified should be out of the region. It is possible to interrupt the region execution in this point of program or to move the statements outside the region.

### *8.2.3 Host section.*

Host section is a fragment of a program with one entrance and one exit inside a region which always will be executed on CPU.

Syntax:

*hostsection-directive* **::= host_section**

Host section has the form:

**#pragma dvm host_section**

{
<hostsection inner>
}

Host sections are intended for intermediate control of variable values during the region execution.

Output operations and calls of external procedures are allowed inside host sections. It is possible to use **get_actual** directives (section 9), **actual** directives are forbidden.

# 9 Control of data movement, actualization. Actualization directives get_actual and actual.

The control of data movement between CPU random access memory and memories of accelerators outside computational regions is specified by special actualization directives

Syntax:

*getactual-directive* **::= get_actual (** *array-range-list* **)**
*actual-directive* **::= actual (** *array-range-list* **)**
*array-range* **::=** *var-name* [ *subscript-range* ]...
*subscript-range* **::= [** *int-expr* [ **:** *int-expr* ] **]**

The **get_actual** directive performs all necessary updates in order to actual (i.e. newest) values of data in subarrays and scalars specified in the list were in CPU memory.

The **actual** directive declares that the subarrays and scalars specified in the list have the newest values in CPU memory. The values of these variables and array elements located in memory of accelerators are considered outdated and will be updated if necessary before use.

# 10 The directive specifying properties of declared functions.

If an actual argument of called function is a distributed array, then corresponding formal argument must have *inherited* distribution.

Inherited distribution means that the formal argument inherits the distribution of the actual argument for each procedure call.

Inherited distribution is described by **inherit** directive which is inserted prior function declaration and prior its description.

Syntax:

*inherit-directive* ::= **inherit (** *var-name-list* **)**

All variables from *var-name-list* must be present in function arguments. And those arguments which have the void* type are considered as templates, and others are the arrays. If the template is specified in the list, the formal and actual arguments must have the void* type.

**Example 10.1.** Distribution of local arrays and formal arguments.

```
#pragma dvm inherit(A, B)
double relax(int n, int m, double A[n][m], double B[n][m]);
………..
#pragma dvm inherit(A, B)
double relax(int n, int m, double A[n][m], double B[n][m]) {
………..
}
```

# 11 Functions.

## 11.1 Function call from parallel loop.

A function, called from a parallel loop, must't have side effects and contains exchanges between processors (*purest procedure*). As a result of it, the purest procedure doesn't contain input/output statements and DVMH directives;

## 11.2 Function call outside parallel loop.

If a function argument is distributed array, then the argument should refer to the array beginning, and actual and formal arguments must have equal ranks.

## 11.3 Formal arguments.

The formal arguments can have only inherited distribution (section 10). If the formal argument is a template, it should be listed in **inherit** directive and have void* type.

**Example 11.1.** Distribution of formal arguments.

```
#pragma dvm inherit(A, TA)
double test (int n, double A[n], void *TA);
………..
#pragma dvm inherit(A, TA)
double test(int n, double A[n], void *TA) {
………..
}
```

## 11.4 Local arrays.

Local arrays may be distributed in a function (by **distribute** and **align** specifications, **redistribute** and **realign** directives). A local array can be aligned with formal argument. The **distribute** specification distributes the local array on the processor subsystem, on which the function was called (*current subsystem*).

**Example11.2.** Distribution of local arrays and formal arguments.

```
#pragma dvm inherit(A, B, C, templ)
void dist(int N, float A[][N], float B[][N], float C[][N], void *templ)
{
#pragma dvm array
float (*X)[N];
X = malloc(N * N * sizeof(float));
#pragma dvm array redistribute(X[][block])
...
}
```

# 12 Input/output.

C99 standard with some restrictions and extensions is implemented in CDVMH.

The following input/output functions are allowed:

**remove, rename, tmpfile, tmpnam, fclose, fflush, fopen, freopen, setbuf, setvbuf, fprintf, fscanf, printf, scanf, vfprintf, vfscanf, vprintf, vscanf, fgetc, fgets, fputc, fputs, getc, getchar, gets, putc, putchar, puts, ungetc, fread, fwrite, fgetpos, fseek, fsetpos, ftell, rewind, clearer, feof, ferror, perror**.

Input/output of whole distributed array is performed by **fread/fwrite** functions. The **remote_access** directive should be inserted prior statement of distributed array input/output by elements (see section 7.3).

A set of file open modes is extended for **fopen** and **freopen** functions:

- If to add "l" or "L" letter to the file open mode then each processor opens its own local file and all operations are performed by each processor independently from others. In this mode a construction "%d" can be used in file name to specify different names for different processors.

  Following statement

  FILE *f = fopen(“out_%04d.txt”, “wl”);

  opens on writing N files (where N – a number of processors in current multiprocessor arrangement) with the names out_0000.txt, out_0001.txt, out_0002.txt, out_0003.txt, …

  All files of the group are deleted by the function

  **int dvmh_remove_local (const char *filename);**

- If "p" or "P" letter is added to the file open mode all processors perform parallel input/output in single global file.

**Constrants.**

- The “%” format can't be used in functions of **printf** family.
- Writing to local files and reading from local files is performed on the same processor arrangement and to the arrays with identical distribution.

# 13 CDVMH compiler.

CDVMH compiler converts a source program to parallel program in C language with function calls of DVMH runtime system (Lib-DVMH library). Besides, for each source program the compiler creates several additional modules to support execution of the program regions on GPU with use of CUDA technology.

For each parallel loop from a region the compiler generates procedure-handler and a kernel for computations on GPU and procedure-handler for execution of the parallel loop on CPU.

The handler is a subprogram processing a parallel loop part on concrete computational device. Arguments of the handler are the device descriptor and the parallel loop part. The handler requests a portion for execution (the loop boundaries and a step), a configuration of parallel processing (quantity of threads), initialization of reduction variables, and after execution of the portion returns result of partial reduction to runtime system.

So, to process a loop parts CUDA handler calls CUDA kernel generated in special way. CUDA kernel is executed on graphic processor and performs the computations of the loop body.

By default the compiler generates handlers for all types of calculators for which it hasn't found contraindications during their contents analyzing. Using **targets** specification in **region** directive a user can specify supposed calculator types for the region execution. In this case the compiler generates only the specified types of handlers.

The main job to implement parallel program execution model (for example, distribution of data and computations) is performed dynamically. It allows to tune DVMH programs dynamically at startup (without recompilation) on a parallel computer configuration (a number of processes and accelerators, their performance and type, and also latency and capacity of communication channels). Thereby the programmer has an opportunity to have single version of the program for execution on serial and parallel computers of different configuration.

# 14 Modes of data and computation distribution on computational devices.

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 the 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.

### 14.1 DVMH program execution scheme.

DVMH program execution can be considered as the execution of sequence of computational regions and code fragments between them, which we will call out-region space. The code in out-region space is executed on the central processor whereas computational regions can be executed on heterogeneous computational devices. Parallel loops and serial fragments of the program can be both inside and outside regions.

DVMH program execution begins synchronously by all launched processors. One main serial thread of execution is provided for each process.

When entering the computational region each processor independently performs additional distribution of the data used by the computational region on computational devices, selected for the region execution. At this stage dynamic scheduling is performed to balance load and minimize the time costs of data movements due to the data redistribution.

When entering a parallel loop inside a region each processor divides job among computational devices according to data distribution. Then it selects the handler for each device, and also optimization parameters for selected handler. At this stage dynamic tuning of optimization parameters, including quantity of CPU threads for handlers on CPU, and also the size and the form of thread block for CUDA handlers to minimize execution time on each separate device is performed.

During execution a parallel loop inside a region is split into several parts, each of them is processed by some handler on some computational device. At this stage main information for DVMH program performance debugging is accumulated.

When exiting from the computational region the additional information is accumulated for dynamic scheduling of the region execution, and also comparative debugging can be performed to check a correctness of the results obtained on accelerators.

### 14.2 Modes of data and computation distribution.

In DVMH programs all distributed data are distributed by blocks: each distributed dimension is divided into segments. Along each dimension of the distributed array it is possible to specify either equal block distribution or distribution by weighted blocks, i.e. according to the given vector of weights. These instructions are performed immediately during data distribution between processors. The nested distributions appearing at an entrance to the computational region are built with use of this information, but due to heterogeneity of computational devices, can have distribution scheme differing from external one.

DVMH runtime system supports four modes of data and computation distribution on computational devices in points of entry to regions:

1. Simple static mode.
2. The simple dynamic mode (there is no description yet).

3. The dynamic mode with selection of distribution scheme.
4. The dynamic mode with use of selected distribution scheme.

Consider these distribution modes in more details.

### 14.2.1 Simple static mode.

In simple static mode a distribution is performed identically in each region. A user specifies a vector of relative performances of computational devices available in each node of a cluster (environment variable **DVMH_CUDAS_PERF**, see Annex 2). Also they can be approximately determined automatically at program startup. Then these performances are superimposed on parameters of inter-process distribution of data, which can be as by equal blocks, and with the given vector of weights along each distributed dimension. In this mode the data movements due to their redistribution are minimized, but different ratio of performances of computational devices on different fragments of the program isn't considered.

In this mode default handlers are used, and optimization parameters have the following values:

- A number of used CPU threads – the most available for the current processor.
- A method of CPU thread scheduling is automatic (in OpenMP terms).
- The sizes and the form of CUDA block of threads for each loop are selected based on a number of parallel loop dimensions, the quantity and the location of dimensions with dependences, the quantity of used hardware resources per one CUDA thread, the quantity of available hardware resources of graphic processor and a way of their distribution on CUDA threads.
- The way of reduction operation processing in CUDA handler is selected dynamically for each loop, based on available memory size of graphic processor – or more fast and more exacting on memory volume, or less fast and less exacting on memory volume.

### 14.2.2 Dynamic mode with selection of distribution scheme

In this mode in each computational region data and computation distribution is selected on the basis of permanently replenished history of launches of the region and its neighbors which are determined dynamically.

For each variant of the region launch the following is determined:

- Dynamically, previous variants of the region launch, being suppliers of actual input data the (newest values), and earlier completed regions which used the actualized data only on reading are considered also. The **actual** and **get_actual** directives are suppliers of actual data also. They are considered as the regions which are executed only on CPU and have empty region-inner.
- Runtime dependence from data distribution, and all variants of the same computational region launches are taken into account.
- A number of entrances into the region.

Runtime dependence from data distribution is built as a table time function from distributions of distributed arrays. The function is a sum of same table functions built for parallel loops, serial fragments and host sections contained in given variant of the region launch.

Processing time of serial fragments is considered constant.

Runtime of host sections is calculated as a sum of runtime of host section statements (constant value) and runtime of **get_actual** directives contained in the host section.

In this mode data distribution scheme building in all variants of launches of computational regions is periodically performed. It uses accumulated for the whole program information based on the analysis of sequence of launch variants of regions and characteristics of their execution. When such schemes are built there are considered as internal characteristics of region launch variants in the form of dependence of runtime from data distribution, and a sequence of launch variants of computational regions to minimize time cost of data redistribution. After the scheme has been built, it is applied, and characteristics and information about computational regions and parallel loops are accumulated further.

Built distribution schemes can be written to the file for use in next program launches both in this mode, and in the third mode of data and computation distribution. The vector of performances of computational devices in same form as for simple static mode can be used as initial approximation.

The transition from this mode to third mode is possible in any point of the program execution. It provides simplified scheme of selection and use of distribution scheme without a need to save parameters in a file and restart the program.

#### *14.2.3 Dynamic mode with use of selected distribution scheme.*

In the dynamic mode with use of selected distribution scheme the distribution is selected in each computational region on the basis of provided distribution scheme built during the program execution in second mode. There is possibility of transition to this mode directly from the second one, or to use distribution scheme from the file obtained as a result of the program execution in the second mode.

If the distribution scheme from the file is used its correct usage isn't guaranteed in the case if the program parameters have been changed. In particular it concerns those parameters which influence on a way of the program execution (the choice of other calculation method, excluding or including of calculation stages).

### 14.3 Mechanism of dynamic rearrangement of arrays.

To optimize an access to global memory of GPU the mechanism of dynamic rearrangement of arrays has been introduced in DVMH runtime system. Before each loop execution the mechanism uses information about mutual alignment of the loop and the array. This information is already available in DVMH program for mapping on a cluster and distribution of computations. The mechanism determines a correspondence between the loop dimensions and the array dimensions and then rearranges the array so that when the

array will be mapped on CUDA architecture the access to its elements will be performed in the best way — the adjacent threads of a thread block will work with the adjacent memory cells.

The mechanism performs any necessary rearrangement of array dimensions, and also so-called diagonal transformation. As result of the transformation the adjacent elements on diagonals will be stored in adjacent memory cells. It enables to apply technique of hyper-plane execution for the loop with dependences without considerable performance loses on memory access operations.

# 15 Compilation, execution and debugging of CDVMH programs.

The parallel program is ordinary serial program, in which DVMH directives specifying its parallel execution are inserted.

CDVMH program is one or several files with source codes in CDVMH language with **.cdv, .c, .h** extensions.

To compile and launch CDVMH program it is necessary to copy to working directory where the program is placed, the startup file of dvm commands (**dvm**) from **dvm_sys/user** directory of DVM system or to execute **dvminit** command.

The environment variables that can be modified by the user are defined in **dvm** file. The environment variables for DVMH programs are described in Annex 2.

To support a program execution as serial and parallel at the same time **_DVMH** macro definition is introduced.

It is defined when CDVMH program is converted, and also when the converted text is compiled further. A value of this macro definition is integer positive number.

Also during conversion and/or compilation of CDVMH program the file with API headers (see Annex 4)

```
#include <dvmh_runtime_api.h>
```

is included implicitly.

**Example 14.1. Use of _DVMH.**

```
#ifdef _DVMH
   dvmh_barrier();
   startt = dvmh_wtime();
#else
   startt = 0;
#endif
```

## 15.1 Compilation. Obtaining ready-for-run program.

The command to compile and get executable CDVMH program has the form:

**dvm c <compilation options> < CDVMH program name >**

If a program has **.cdv** extension, then the program name can be issued without extension.

If the program consists of several files, then to combine the files it is possible to use **makefile** or to list all the files in compilation line:

**dvm c <compilation options> -o <name of executable file of CDVMH program> <file list>**

where:

**<file list>** - a list of file names of the program with **.cdv, .c, .h** extensions.

*Processing result*: ready-for-run program (executable file <C**DVMH program name**> or with the name from command line) in current directory. Besides, the compiler creates several additional files for each source program. These modules are described in Annex 3.

If the compiler detected errors, the executable file isn't created.

For CDVMH programs the new modes of compilation specified by the following options are provided:

| | | |
|---|---|---|
| -noH | - | the mode to ignore the directives providing the program execution on clusters with graphic processors |
| -oneThread | - | the mode of serial execution of all loops on GPU |
| -autoTfm | - | the mode of array dynamic rearrangement (the mode of reordering of the array dimensions to optimize an access to GPU memory) |
| -noCuda | - | a control of compilation process – the compiler doesn't prepare region execution on CUDA devices. |

When CDVMH program is compiled with –noH option the following directives and specifications are ignored:

- directives of computational region specifications;
- some specifications of parallel loops inside a region: specification of CUDA block size for CUDA device;
- directives to control data movement outside regions;
- directives to specify the program fragments inside a region which should be executed on central processor.

When such program is executed the computations are distributed only on processors. It is reasonable to use the option for the program debugging.

Usage of optimization option (–autoTfm) can increase the program performance (section 15.4).

Remaining options are intended to debug DVMH system.

## 15.2 Execution of CDVMH program.

**dvm run [N1 [N2 [N3[N4]]]] <name of executable file of CDVMH program >**

where:

**N1, N2, N3, N4** – sizes of processor matrix (by default – **1 1 1 1**).

When CDVMH program is launched the sizes of virtual processor matrix determine a number of processes (**N1*N2*N3*N4**), that will be executed in parallel.

## 15.3 Comparative debugging.

A possibility of comparative debugging was implemented for CDVMH programs. It is a special mode of DVMH program execution when all computations in regions are executed simultaneously on CPU and GPU.

Data comparison at entrance in a region enables to detect the absence of **out** specification in earlier executed regions or **actual** directive.

Comparison of output data, obtained in a region during execution on GPU, with the data, obtained in the region during execution on CPU, allows to detect and localize the errors which occur during the region execution on accelerators.

All output data of a computational region participate in comparison. Integer data are compared on equality, and real numbers are compared with specified accuracy on absolute and relative error. If the discrepancies were found the information about it is issued. The data version obtained on CPU is used further in the program.

When a region is executed on an accelerator there are several reasons of errors:

1. Programmer performed incorrect parallelization not suitable for array-parallel execution on shared memory.
2. Programmer incorrectly specified private or reduction variables in a parallel loop.
3. Arithmetical operations or mathematical functions give on an accelerator other result in comparison with result on CPU. It can occur due to distinctions in command system leading to different results (in limits of accuracy rounding).
4. Programmer specified wrong data actualization directives **get_actual** and **actual** therefore processed data on CPU and on the accelerator became different.

Turning on and use of comparative debugging mode doesn't require from a programmer to change anything in the program, and to recompile it.

To turn on the mode of comparative debugging it is necessary to assign 1 to environment variable **DVMH_COMPARE_DEBUG** in startup file of dvm-commands, or to launch the program by the command:

**dvm cmph [N1 [N2 [N3[N4]]]] <name of executable file of CDVMH program >**

where:

**N1, N2, N3, N4** – sizes of virtual processor matrix (by default – **1 1 1 1**).

If errors were detected the information about them is output to standard error stream or to a file. The name of the file can be specified in environment variable **DVMH_LOGFILE**.

Accuracy of variable comparison can be changed, if to set values of environment variables **DVMH_COMPARE_FLOATS_EPS, DVMH_COMPARE_DOUBLES_EPS, DVMH_COMPARE_LONGDOUBLES_EPS**.

To perform comparative debugging a value of environment variable **DVMH_LOGLEVEL** specifying the detail level should be at least 1. In this case the information about existence of errors and a set of indexes with discrepancies in compact form are output. If the errors exist the information issued on 4 or 5 detail level may be useful.

### 15.4 Performance debugging.

If detail level **DVMH_LOGLEVEL** is equal to 4 or more the information about performance of each computational device for each parallel loop is written to output file. The name of the file can be specified in **DVMH_LOGFILE** environment variable. If the file name isn't set, the information is output to standard error stream.

The program performance can be improved if to use optimization option:

-autoTfm - the mode of array dynamic rearrangement (the mode of the array dimension reordering to optimize an access to global memory).

Option – autoTfm may be the most effective if a program has the loops with regular data dependencies.

A choice of data and computation distribution mode can also affect program performance improving. The following environment variables are used to select a mode:

**DVMH_SCHED_TECH** - scheduling mode. There are several modes. It is possible to set them using a number or a word, the letter case is ignored. It is equal 1 by default. It is possible to set the following modes:

| | | | | |
|---|---|---|---|---|
| 0 | - | Single | - | "one device" mode. If at least one accelerator is assigned to a process, then only it will be used. If more accelerators are assigned, then only one from them is selected. If no accelerators are assigned to the process, then the process will be executed on CPU. |
| 1 | - | static | - | "simple static" mode. Data distribution (to be exact, distribution of templates – the roots of alignment trees) will be performed in each region on devices in a ratio specified by **DVMH_CPU_PERF** and **DVMH_CUDAS_PERF** environment variables. |
| 2 | - | dynamic1 | - | "simple dynamic" mode. Data distribution (to be exact, distribution of templates – the roots of alignment trees) will be selected during a program execution so that to find such **DVMH_CPU_PERF** and **DVMH_CUDAS_PERF** values which minimize total time of the program execution in the simple static mode if the variables were set. |
| 3 | - | dynamic2 | - | "dynamic with selection" mode. Data distribution (to be exact, distribution of templates – the roots of alignment trees) will be |

| | | | | |
|---|---|---|---|---|
| | | | | selected during a program execution so that to determine weights of devices for each of regions which minimize total time of the program execution in the mode with use of distribution scheme. |
| 4 | - | scheme | - | static mode with use of the scheme obtained in the dynamic mode with selection. The file containing the weights and values of optimization parameters for each region and parallel loop is used. |

**DVMH_SCHED_FILE** - file name with saved distribution scheme for scheduling mode 4. The name Scheme.dvmh is used by default.

The following environment variables can also affect program performance in a certain mode:

**DVMH_CUDAS_PERF** - ring list of not-negative numbers specifying relative performance of CPU (all working CPU threads together) in each of the processes. It is indexed by process number. It is equal to 1 by default.

**DVMH_CUDAS_PERF** - ring list of non-negative numbers specifying relative performance of CUDA devices of a node. It is indexed by CUDA device number according to CUDA Runtime numbering on each node independently. It is equal to1 by default.

**DVMH_CUDA_PREFER_SHARED** – sign to specify the mode of shared memory preference for kernel launch. It is set by Boolean value. It is equal to 0 by default.

The values of **DVMH_NUM_CUDAS** and **DVMH_NUM_THREADS** environment variables can also affect the execution performance.

## References.


1. DVM system[Electronic resource] - : [web-site] http://dvm-system.org/ (accessed 28.05.2016)
2. 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.
3. 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”, Chelyabinsk: publishing center SUSU, 2012, № №18 (277), Vypusk 12, P. 82–92.

# 16 Examples of programs.

Small scientific programs are presented to illustrate CDVMH language features. They are intended to solve a system of linear equations:

**A x = b**

where:

**A** - matrix of coefficients,
**b** - vector of free members,
**x** - vector of unknowns.

The following basic methods are used to solve this system.

**Direct methods**. Well-known Gaussian Elimination method is the most commonly used algorithm of this class. The main idea of the algorithm is to transform the matrix A to upper triangular matrix and then to diagonalize it by use of back substitution.

**Explicit iteration methods**. Jacobi Relaxation method is the most known algorithm of this class. The algorithm performs the following iterative computations:

$$x_{i,j}^{new} = ( x_{i-1,j}^{old} + x_{i,j-1}^{old} + x_{i+1,j}^{old} + x_{i,j+1}^{old} ) / 4$$

**Implicit iteration methods**. Successive Over Relaxation (SOR) method belongs to this class. The iterative approximation is calculated by the formula:

$$x_{i,j}^{new} = ( w / 4 ) * ( x_{i-1,j}^{new} + x_{i,j-1}^{new} + x_{i+1,j}^{old} + x_{i,j+1}^{old} ) + (1-w) * x_{i,j}^{old}$$

When using «red-black» coloring of variables each iterative step is split on two Jacobi half-steps. One half-step processes «red» variables and the other processes «black» variables. Red-black coloring of variables allows to overlap calculations and data communications.

## 16.1 Example 1. Jacobi algorithm.

```
/* Jacobi-2 program */

#include <math.h>
#include <stdio.h>

#define Max(a, b) ((a) > (b) ? (a) : (b))

#define L 8
#define ITMAX 20

int i, j, it;
double eps;
double MAXEPS = 0.5;
FILE *f;

/* 2D arrays block distributed along 2 dimensions */
```

```
#pragma dvm array distribute[block][block]
double A[L][L];
#pragma dvm array align([i][j] with A[i][j])
double B[L][L];

int main(int an, char **as)
{
   #pragma dvm region
   {
   /* 2D parallel loop with base array A */
   #pragma dvm parallel([i][j] on A[i][j]) cuda_block(256)
   for (i = 0; i < L; i++)
      for (j = 0; j < L; j++)
      {
         A[i][j] = 0.;
         if (i == 0 || j == 0 || i == L - 1 || j == L - 1)
            B[i][j] = 0.;
         else
            B[i][j] = 3. + i + j;
      }
   }

   /* iteration loop */
   for (it = 1; it <= ITMAX; it++)
   {
      eps = 0.;
      #pragma dvm actual(eps)

      #pragma dvm region
      {
      /* Parallel loop with base array A */
      /* calculating maximum in variable eps */
      #pragma dvm parallel([i][j] on A[i][j]) reduction(max(eps)),  cuda_block(256)
      for (i = 1; i < L - 1; i++)
         for (j = 1; j < L - 1; j++)
         {
            eps = Max(fabs(B[i][j] - A[i][j]), eps);
            A[i][j] = B[i][j];
         }

      /* Parallel loop with base array B and */
      /* with prior updating shadow elements of array A */
      #pragma dvm parallel([i][j] on B[i][j]) shadow_renew(A), cuda_block(256)
      for (i = 1; i < L - 1; i++)
         for (j = 1; j < L - 1; j++)
            B[i][j] = (A[i - 1][j] + A[i + 1][j] + A[i][j - 1] + A[i][j + 1]) / 4.;
      }

      #pragma dvm get_actual(eps)
```

```
        printf("it=%4i   eps=%e\n", it, eps);
        if (eps < MAXEPS)
            break;
    }
    f = fopen("jacobi.dat", "wb");
    #pragma dvm get_actual(B)
    fwrite(B, sizeof(double), L * L, f);
    fclose(f);
    return 0;
}
```

As a result of execution of the directive

**#pragma dvm array distribute[block][block]**

array A will be distributed on calculators. The quantity and type of used calculators is set using environment variables and command line options when program is launched.

The directive

**#pragma dvm array align([i][j] with A[i][j])**
double B[L][L];

specifies joint distribution of two arrays A and B. The array B elements will be distributed on the same calculator where the corresponding elements of array A will be located.

The directive

**#pragma dvm parallel([i][j] on A[i][j])**

specifies the distribution of computations. The loop iterations will be executed on that calculator where the corresponding elements of array A are located.

The **reduction(max(eps))** specification organizes effective execution of reduction operation (maximum value finding) - global operation with data located on different calculators.

The **shadow_renew(A)** specification indicates the need of remote data (shadow edges) swapping from other calculators before the loop execution.

The **cuda_block** specification specifies thread block size of CUDA device.

As there are no any additional specifications in **region** directives, the compiler automatically defines the directions of variable use - **inout**(A,B,eps).

When executing the first computational region (initialization loop) required memory will be allocated on accelerators for the distributed parts of arrays A and B.

When entering into the second computational region (in iteration loop) a check is performed, whether actual representatives for arrays A and B are present on the calculator. As such representatives are already present, no additional operations of actual data copying to calculators are performed.

When exiting from the computational region the data in host memory isn't updated. Before eps and MAXEPS comparing it is necessary to execute **get_actual(eps)** directive,

and before the array B output to the file it is necessary to copy the last updates of the array from calculator memory using **get_actual (B)** directive.

## 16.2 Example 2. Gauss elimination method algorithm.

The coefficients of matrix A are stored in the array section A[0:N-1][0:N-1], and B vector – in section A[0:N-1][N] of the same array.

```
/* GAUSS program */
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#define N 10

int main(int an, char **as)
{
   int i, j, k;
   /* declaration of dynamic distributed arrays */
   #pragma dvm array
   float (*A)[N + 1];
   #pragma dvm array
   float (*X);
   /* create arrays */
   A = malloc(N * (N + 1) * sizeof(float));
   X = malloc(N * sizeof(float));
   #pragma dvm redistribute (A[block][])
   #pragma dvm realign(X[i] with A([i][])
   /* initialize array A */
   #pragma dvm region
   {
   #pragma dvm parallel([i][j] on A[i][j])
   for (i = 0; i < N; i++)
      for (j = 0; j < N + 1; j++)
         if (i == j || j == N)
            A[i][j] = 1.f;
         else
            A[i][j] = 0.f;
   }
   /* elimination */
   for (i = 0; i < N - 1; i++)
   {
      #pragma dvm region
      {
      #pragma dvm parallel([k][j] on A[k][j]) remote_access(A[i][])
```

```
        for (k = i + 1; k < N; k++)
            for (j = i + 1; j < N + 1; j++)
                A[k][j] = A[k][j] - A[k][i] * A[i][j] / A[i][i];
        }
    }
    /* reverse substitution */
    #pragma dvm region in(A[N - 1][N - 1 : N]), out(X[N - 1])
    {
    X[N - 1] = A[N - 1][N] / A[N - 1][N - 1];
    }
    for (j = N - 2; j >= 0; j--)
    {
        #pragma dvm region
        {
        #pragma dvm parallel([k] on A[k][]) remote_access(X[j + 1])
        for (k = 0; k <= j; k++)
            A[k][N] = A[k][N] - A[k][j + 1] * X[j + 1];
        X[j] = A[j][N] / A[j][j];
        }
        #pragma dvm remote_access(X[j])
        {
        printf("j=%4i   X[j]=%e\n", j, X[j]);
        }
    }
    free(X);
    free(A);
    return 0;
}
```

## 16.3 Example 3. Red-Black Successive Over Relaxation.

The points are processed in chessboard order: at first "red", then "black".

```
/* RED-BLACK program */
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#define Max(a, b) ((a) > (b) ? (a) : (b))
#define N 8
#define ITMAX 10

int main(int an, char ** as)
{
    int i, j, it;
```

```
float MAXEPS = 0.5E-5f;
float w = 0.5f;

#pragma dvm array
float (*A)[N];

/* Create array */
A = malloc(N * N * sizeof(float));
#pragma dvm redistribute (A[block][block])

#pragma dvm region
{
/* Initialization parallel loop */
#pragma dvm parallel([i][j] on A[i][j]) cuda_block(256)
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      if (i == j)
         A[i][j] = N + 2;
      else
         A[i][j] = -1.0f;
}

/* iteration loop */
for (it = 1; it <= ITMAX; it++)
{
   float eps = 0.f;
   #pragma dvm actual(eps)

   #pragma dvm region
   {
   /* Parallel loop with reduction on RED points */
#pragma dvm parallel([i][j] on A[i][j]) shadow_renew(A),
                                           reduction(max(eps)),cuda_block(256)
    for (i = 1; i < N - 1; i++)
      for (j = 1; j < N - 1; j++)
         if ((i + j) % 2 == 1)
         {
            float s;
            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(fabs(s - A[i][j]), eps);
         }

   /* Parallel loop on BLACK points (without reduction) */
   #pragma dvm parallel([i][j] on A[i][j]) shadow_renew(A), cuda_block(256)
   for (i = 1; i < N - 1; i++)
      for (j = 1; j < N - 1; j++)
         if ((i + j) % 2 == 0)
```

```
            {
                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];
            }
        }
        #pragma dvm get_actual(eps)
        printf("it=%4i   eps=%e\n", it, eps);
        if (eps < MAXEPS)
            break;
    }
    free(A);
    return 0;
}
```

## Annex 1. Syntax of CDVMH directives.

The syntax of CDVMH directives is described by the extended Backus-Naur form. The following notations are used:

| | |
|---|---|
| **::=** | is by definition |
| *x* \| *y* | Alternatives |
| [ ] | encloses optional construct |
| [ ]… | construction repetition of 0 or more times |
| *x-list* | *x* [ **,** *x* ]… |

*Directive* **::= dvm** *dvmh-directive*

*dvmh-directive* **::=** *specification- directive*
**|** *executable-directive*

*specification-vaiable – directive* **::=** *array-directive*
**|** *template-directive*
**|** *inherit-directive*

*executable-directive* **::=** *redistribute-directive*
*|* *realign-directive*
*|* *getactual-directive*
*|* *actual-directive*
*|* *region-directive*
*|* *parallel-directive*
*|* *remote-directive*
*|* *hostsection-directive*

*array-directive* **::= array** [ *array-clause-list* ]
*array-clause* **::=** *distribute-aline-clause*
**|** *shadow-clause*

*distribute-aline-clause* **::=** *distribute-clause*
**|** *align-clause*

*distribute –clause* **::= distribute** [ *distribute-axis-rule* ]...

*distribute-axis-rule* **::= [ ]**
**| [ block ]**
| **[ wgtblock (** *var-name* **,** *int-expr* ) **]**
**| [ genblock (** *var-name* ) **]**
**| [ multblock (** *int-expr* ) **]**

*align –clause* **::= align (** [ *align-axis-name* ]... **with** *templ-align-spec* **)**
*align-axis-name* **::= [ ]**
*/* **[** *align-dummy* **]**

*align-dummy* **::=** *name*
*templ-align-spec* **::=** *var-name* [ *templ-axis-spec* ]...
*templ-axis-spec* **::= [ ]**
**| [** *int-expr* **]**
**| [** *align-dummy-use* **]**

*align-dummy-use* **::=** [ *primary-expr* ***** ] *align-dummy* [ *add-op primary-expr*]

*primary-expr* **::=** *int-constant*
*/ var-name*
*/* **(** *int-expr* **)**
*add-op* **::=** +
| ..-

*template-directive* **::= template** [ *template-decl* ]
*template-decl* **::=** [ *size-spec* ]... [ *distribute-clause* ]
*size-spec* **::= [ ]**
**| [** *int-expr* **]**

*redistribute-directive* **::= redistribute (** *var-name* [ *distribute-axis-rule* ]... **)**
*distribute-axis-rule* **::= [ ]**
**| [ block ]**
**| [ wgtblock (** *var-name* **,** *int-expr* ) **]**
**| [ genblock (** *var-name* ) **]**
**| [ multblock (** *int-expr* ) **]**

*realign-directive* **::= realign (** *var-name align-rule* **)** [ **new_value** ]
*align-rule* **::=** [ *align-axis-name* ]... **with** *templ-align-spec*
*align-axis-name* **::= [ ]**
**| [** *align-dummy* **]**
*align-dummy* **::=** *name*
*templ-align-spec* **::=** *var-name* [ *templ-axis-spec* ]…

*templ-axis-spec* **::= [ ]**
| **[** *int-expr* **]**
| **[** *align-dummy-use* **]**
*align-dummy-use* **::=** [ *primary-expr* ***** ] *align-dummy* [ *add-op primary-expr* ]
*primary-expr* **::=** *int-constant*
| *var-name*
| **(** *int-expr* **)**
*add-op* **::=** +
| **-**

*parallel-directive* **::= parallel** *parallel-map* [ *parallel-clause-list* ]
*parallel-map* **::= (** *int-constant* **)**
| **(** [ *align-axis-name* ]... **on** *templ-align-spec* **)**
*parallel-clause* **::=** *private-clause*
| *reduction-clause*
| *shadow-renew-clause*
| *across-clause*
| *remote-clase*
| *cuda-clause*
| *stage-clause*

*private-clause* **::= private (** *var-name-list* **)**
*cuda-clause* **::= cuda_block (** *int-expr* [ **,** *int-expr* [ **,** *int-expr* ] ] **)**

*reduction-clause* **::= reduction (** *red-spec-list* **)**
*red-spec* **::=** *red-func* **(** *red-variable* **)**
| *red-loc-func* **(** *red-variable* **,** *loc-variable* [ **,** *size* ] **)**
*red-variable* **::=** *array-name*
| *scalar-variable-name*
*loc-variable* **::=** *array-name*
| *scalar-variable-name*
*Size* **::=** *int-constant*

*red-func* **::= sum**
| **product**
| **max**
| **min**
| **and**
| **or**
| **xor**
*red-loc-func* **::= maxloc**
| **minloc**

shadow-clause **::= shadow** [ *shadow-edge* ]...
*shadow-edge* **::= [** *width* **]**
| **[** *low-width* [ **:** *high-width* ] **]**
*low-width* **::=** *int-expr*

*high-width* ::= *int-expr*

*shadow-renew-clause* ::= **shadow_renew (** *shadow-renew-spec-list* **)**
*shadow-renew-spec* ::= *var-name* [ *shadow-edge* ]... [ **( corner )** ]
*shadow-edge* ::= **[** *width* **]**
| **[** *low-width* [ **:** *high-width* ] **]**
*low-width* ::= *int-expr*
*high-width* ::= *int-expr*

*across-clause* ::= **across (** *across-spec-list* **)**
*across-spec* ::= *var-name* [ *subscript-range* ]...
*subscript-range* ::= **[** *flow-dep-length* [ **:** *anti-dep-length* ] **]**
*flow-dep-length* ::= *int-expr*
*anti-dep-length* ::= *int-expr*

*stage-clause* ::= **stage (** *int-expr* **)**

*remote-clause* ::= **remote_access (** *rma-spec-list* **)**
*rma-spec* ::= *templ-align-spec*

*remote-directive* ::= **remote_access (** *rma-spec-list* **)**

*region-directive* ::= **region** [ *region-clause-list* ]
*region-clause* ::= *in-out-local-clause*
/ *targets-clause*
/ *async_clause*
*in-out-local-clause* ::= **in (** *array-range-list* **)**
/ **out (** *array-range-list* **)**
/ **local (** *array-range-list* **)**
/ **inout (** *array-range-list* **)**
| **inlocal (** *array-range-list* **)**
*targets-clause* ::= **targets (** *target_name-list* **)**
*target_name* ::= **CUDA**
/ **HOST**
*async_clause* ::= **async**

*hostsection-directive* ::= **host_section**

*getactual-directive* ::= **get_actual (** *array-range-list* **)**
*actual-directive* ::= **actual (** *array-range-list* **)**
*array-range* ::= *var-name* [ *subscript-range* ]...
*subscript-range* ::= **[** *int-expr* [ **:** *int-expr* ] **]**

*inherit-directive* ::= **inherit (** *var-name-list* **)**

## Annex 2. Environment variables for CDVMH programs.

All environment variables of DVM system have **DVMH_** prefix.

The environment variables which can be changed by a user are defined in dvm-commands startup file.

All variables with **DVMH_** prefix are passed from zero process to all other processes by RTS tools, and the variable value already specified for the process has a priority.

The format of each variable value depends on a type of specified parameter:

For string parameters not changed contents of the variable is used.

For numerical parameters the variable value is passed to sscanf function for conversion to appropriate type. A point is used as decimal symbol.

For Boolean parameters the variable value can be (letter case is ignored):

1, on, yes, enable – true;

0, off, no, disable – false.

For parameters-lists it is possible to use both numerical values, and verbal ones. Specific cases are considered separately below.

For parameters-lists their elements are separated by commas or spaces or their any combination as it is convenient to a user, the empty elements in the list aren't allowed (in particular, several spaces in a row are equivalent to one).

**DVMH_PPN** – a number of processes per a node (positive integer number or list of non-negative integer numbers). This variable differs from all others: it is used only by DVM system driver. If there are nodes of different types on a computational cluster, then the value can be specified by a list of integer numbers. The value is equal to 1 by default.

**DVMH_LOGLEVEL** - detail level of a log. It can be both a number and a word, the letter case is ignored. The value is equal to 1 by default. The following levels can be specified:

| | | | | |
|---|---|---|---|---|
| 0 | - | Fatal | - | to output only fatal errors |
| 1 | - | error | - | the errors that aren't the reason of abnormal termination. |
| 2 | - | Warning | - | warnings. They usually show the potential mistakes made during parallelization. |
| 3 | - | Info | - | informational messages. Some help information about DVM system version, a way of a program launch, used computational devices is output. |
| 4 | - | Debug | - | debug messages. Information for DVM system developers. |
| 5 | - | Trace | - | trace messages. Information for DVM system developers. Very large volume of output information is possible. |

**DVMH_FLUSHLOGLEVEL** - level of messages whose output leads to pushing out of input/output buffer from the memory of user process. It is set in the same format, as **DVMH_LOGLEVEL**. The value is equal to 0 by default.

**DVMH_LOGFILE** - log file name. The variable value is string. It is possible to use construction %d (for example, 'dvmh_%d.log') in file name; in this case the log of each process will be in separate file. If the variable isn't set or the file name isn't specified, the standard error stream is used.

**DVMH_FATAL_TO_STDERR** - boolean value specifying whether it is necessary to duplicate fatal messages to standard error stream if the file was specified for the log. The value is equal to 1 by default.

**DVMH_NUM_THREADS** - ring list of non-negative integer numbers specifying a number of working CPU threads in each of the processes. It is indexed by process number. If the variable isn't set, its optimal value is determined by runtime system. Runtime system supports effective usage of all resources of a node. Optimal value depends on a number of devices in the node, a number of launched on the node processes and a number of CUDA devices (**DVMH_NUM_CUDAS**) for use by one process.

**DVMH_NUM_CUDAS** - ring list of non-negative integer numbers specifying the number of virtual CUDA accelerators in each of the processes. It is indexed by process number. If the variable isn't set, its optimal value is determined by runtime system. Runtime system supports effective usage of all resources of a node. Optimal value depends on a number of devices in the node and a number of launched on the node processes.

**DVMH_CPU_PERF** - ring list of non-negative real numbers specifying the relative performance of CPU (all working CPU threads together) in each of the processes. It is indexed by process number. It is equal 1 by default.

**DVMH_CUDAS_PERF** - ring list of non-negative real numbers specifying the relativel performance of CUDA devices of a node. It is indexed by CUDA device number according to CUDA Runtime numbering on each node independently. It is equal 1 by default.

**DVMH_CUDA_PREFER_SHARED** - a sign to specify the mode of shared memory preference for launch of kernels. It is set by Boolean value.

**DVMH_SCHED_TECH** - scheduling mode. There are several modes. It can be specified by both a number and a word, the letter case is ignored. It is equal 1 by default. The following modes are possible:

| | | | | |
|---|---|---|---|---|
| 0 | - | Single | - | "one device" mode. If at least one accelerator is assigned to a process, then only it will be used. If more accelerators are assigned, then only one from them is selected. If no accelerators are assigned to the process, then the process will be executed on CPU. |
| 1 | - | Static | - | "simple static" mode. Data distribution (to be exact, distribution of templates – the roots of alignment trees) will be performed in each region on devices in a ratio specified by **DVMH_CPU_PERF** and **DVMH_CUDAS_PERF** environment variables. |

| | | | | |
|---|---|---|---|---|
| 2 | - | dynamic1 | - | "simple dynamic" mode. Data distribution (to be exact, distribution of templates – the roots of alignment trees) will be selected during a program execution so that to find such **DVMH_CPU_PERF** and **DVMH_CUDAS_PERF** values which minimize total time of the program execution in the simple static mode if the variables were set. |
| 3 | - | dynamic2 | - | "dynamic with selection" mode. Data distribution (to be exact, distribution of templates – the roots of alignment trees) will be selected during a program execution so that to determine weights of devices for each of regions which minimize total time of the program execution in the mode with use of distribution scheme. |
| 4 | - | Scheme | - | static mode with use of the scheme obtained in the dynamic mode with selection. The file containing the weights and values of optimization parameters for each region and parallel loop is used. |

**DVMH_SCHED_FILE** - file name with saved distribution scheme for scheduling mode 4. The name Scheme.dvmh is used by default.

**DVMH_REDUCE_DEPS** - Boolean value indicating whether to consider dimensions of a parallel loop of length 1 as the dimensions with dependences if the dimensions are dependent according to **across** specifications. It is equal to 0 by default (not to consider).

**DVMH_ALLOW_ASYNC** - a sign of asynchronous execution of regions (regions with **async** specification). It is set by Boolean value. It is equal to 0 by default. (It isn't implemented yet.)

**DVMH_COMPARE_DEBUG** - a sign to turn on the comparative debugging mode of program execution on accelerators. The **cmph** command of DVM system driver is more convenient way to start comparative debugging. It is set by Boolean value. It is equal to 0 by default.

**DVMH_COMPARE_FLOATS_EPS** - non-negative real number specifying the comparison accuracy of variables with floating point of single precision on relative and absolute error during comparative debugging. Null sets a requirement of full equality. By default it is equal FLT_EPSILON*1000, where FLT_EPSILON is minimum positive x such that 1.0+x=1.0.

**DVMH_COMPARE_DOUBLES_EPS** - non-negative real number specifying the comparison accuracy of variables with floating point of double precision on relative and absolute error during comparative debugging. Null sets a requirement of full equality. By default it is equal to DBL_EPSILON*10000, where DBL_EPSILON is minimum positive x such that 1.0+x=1.0.

**DVMH_COMPARE_LONGDOUBLES_EPS** - non-negative real number specifying the comparison accuracy of variables with floating point of long double precision on relative and absolute error during comparative debugging. Null sets the requirement of full equality. By default it is equal to LDBL_EPSILON*100000, where LDBL_EPSILON – minimum positive x such that 1.0+x=1.0.

**DVMH_SYNC_CUDA** - a sign of synchronization with the CUDA device after each launch of a kernel. If this parameter is set the localization of execution error arising in CUDA kernel is improved, and if it isn't set less time is spent on overheads. It is used for search of the system errors. It is set by Boolean value. It is equal to 0 by default.

**DVMH_FORTRAN_NOTATION** – a sign to use Fortran notations when array sections are output to the log (for example, after detecting of differences during comparative debugging). It is set by Boolean value. Default value depends on the language of main program unit (PROGRAM in Fortran or main in C/C ++).

**DVMH_CACHE_CUDA_ALLOC** - a sign of memory allocation and freeing caching on CUDA device. The caching can considerably increase the program execution whereas the refusal from it makes behavior of the program more predictable. It is set by Boolean value. It is equal to 1 by default.

**DVMH_USE_GENBLOCK** - a sign of unconditional use of **GENBLOCK** distribution method when the distribution is performed by runtime system tools. It is set by Boolean value. It is equal to 0 by default.

**DVMH_SET_AFFINITY** - a sign of thread binding to processors by runtime system tools. It is set by Boolean value. It is equal to 1 by default.

**DVMH_OPT_PARAMS** - a sign to perform the selection of optimization parameters, such as amount of the threads used for each parallel loop or the size of CUDA block of threads used for each parallel loop. This sign setting will lead to the variation of optimization parameters during the program execution, and also to the output of selection results in the file of distribution scheme (see **DVMH_SCHED_FILE**). It is possible to use it independently on the distribution mode. It is set by Boolean value. It is equal to 0 by default.

**DVMH_NO_DIRECT_COPY** - a sign to prohibit direct addressing of CPU memory from GPU to perform data actualization. Note that the absence of the prohibition doesn't mean direct addressing use. There are required the support from hardware, OS, and also more strong requirements for the form and data type of arrays. It is set by Boolean value. It is equal to 0 by default.

**DVMH_SPECIALIZE_RTC** – a sign to permit a specialization of parameters if runtime-compilation of CUDA kernels is used. It is set by Boolean value. It is equal to 1 by default.

**DVMH_PREFER_CALL_SWITCH** - a sign to call a handler via a switch on a number of parameters if it is possible. Two ways of the handler call are provided by RTS: via the switch, it have limits on argument quantity (256 - at the time of 13.05.2016) and with use of libffi library which has no such limits, but it is available not on all platforms. It is set by Boolean value. It is equal to 1 by default.

**DVMH_NUM_VARIANTS_FOR_VAR_RTC** - positive integer number specifying maximum quantity of variable specialization variants before a refusal from its specialization. It is equal to 3 by default.

**DVMH_PARALLEL_IO_THRES** - positive integer number specifying minimum size of output or input data (in bytes) which RTS will try to input or output in parallel (only if all basic restrictions are satisfied). It is equal 104857600, i.e. 100 MB by default.

**DVMH_IO_BUF_SIZE** - positive integer number specifying maximum size of input/output buffer if input/output is performed via input/output processor. It is equal 104857600, i.e. 100 MB by default.

**DVMH_IO_THREAD_COUNT** - non-negative integer number specifying a number of input/output threads executing asynchronous input/output operations. Zero value turns off asynchronous input/output. It is equal to 5 by default.

## Annex 3. Description of output files of the compiler.

In addition to the file containing executable CDVMH program some more files are created as a result of DVMH program compilation. Consider them on the sample program prog.cdv.

1. prog.DVMH.c - the file contains a code for all out-region space, host sections, control code for organization of parallel execution and data distribution. The file also contains the handlers of parallel loops and serial fragments of regions intended for execution on CPU (multiprocessor). The language is standard C + OpenMP (OpenMP support from the compiler isn't required).
2. prog.DVMH_cuda.cu - the file contains the handlers of parallel loops and serial sections of a region intended for execution on GPU of NVIDIA corporation with use of CUDA technology. CUDA kernels for these handlers are also included in the file. The language is CUDA C ++.

## Annex 4. User API.

1. Barrier synchronization within the current multiprocessor system.

   **void dvmh_barrier();**

2. The total number of processors in the current multiprocessor system.

   **int dvmh_get_total_num_procs();**

3. The effective number of dimensions in the current multiprocessor system.

   **int dvmh_get_num_proc_axes();**

4. The number of processors in the current multiprocessor system on axis dimension (numbering from one).

   **int dvmh_get_num_procs(int axis);**

5. Time request from wall clock, without synchronization.

   **double dvmh_wtime();**

6. The functions of local input/output which don't deal with file descriptors (for the functions dealing with file descriptors see the note about **fopen** and **freopen** functions in section 12).

    **int dvmh_remove_local(const char *filename);**

    **int dvmh_rename_local(const char *oldFN, const char *newFN);**

    **void *dvmh_tmpfile_local(void);**

    **char *dvmh_tmpnam_local(char *s);**