Язык C-DVMH.
C-DVMH компилятор.
Компиляция, выполнение и отладка CDVMH-программ.

Введение.

Язык C-DVMH предназначен для разработки мобильных и эффективных параллельных программ вычислительного характера для кластеров с ускорителями.

Он представляет собой расширение языка Си в соответствии с моделью DVMH (DVM for Heterogeneous systems), разработанной в ИПМ им М.В.Келдыша РАН. Модель является расширением модели DVM[1].

При использовании языка C-DVMH программист имеет один вариант программы и для последовательного, и для параллельного выполнения. Программа, помимо описания алгоритма обычными средствами языка Си, содержит правила параллельного выполнения этого алгоритма. Правила оформляются синтаксически таким образом, что они являются “невидимыми” для стандартных компиляторов с языка Си для последовательных ЭВМ и не препятствуют выполнению и отладке DVMH-программы на рабочих станциях как обычной последовательной программы.

1 Словарь терминов.

В данном разделе приведен список терминов и сокращений, используемых в документе.

1.1 Описание аппаратных вычислительных систем.

Вычислительный кластер – совокупность связанных между собой вычислительных узлов (компьютеров). В состав **узла кластера,** помимо центрального процессорного устройства (ЦПУ) может входить несколько специализированных процессорных устройств – ускорителей.

ЦПУ, центральное процессорное устройство – несколько универсальных многоядерных процессоров, имеющих общую оперативную память.

Ускоритель – **специализированное процессорное устройство**, подключаемое к ЦПУ и ориентированное на высокопроизводительное выполнение некоторых фрагментов программы. Поскольку ускоритель имеет собственную оперативную память, а память ЦПУ ему недоступна, то для запуска на нем фрагмента программы центральное процессорное устройство переписывает этот фрагмент и требуемые ему данные из оперативной памяти ЦПУ в оперативную память ускорителя.

ГПУ - графический процессор, один из видов ускорителей AMD или NVIDIA.

CUDA-устройство - графический процессор фирмы NVIDIA.

Вычислительное устройство, вычислитель – ЦПУ или ускоритель.

1.2 Описание виртуальных вычислительных систем, на которых может выполняться DVMH-программа.

Виртуальная многопроцессорная система (или **массив виртуальных процессоров)** – та машина, которая предоставляется DVMH-программе пользователя аппаратурой и базовым системным программным обеспечением. Для распределённой ЭВМ примером такой машины может служить MPI-машина. В этом случае многопроцессорная система – это группа MPI-процессов, которые создаются при запуске параллельной программы на выполнение. На один узел аппаратного кластера может быть отображен один или несколько MPI-процессов. Число процессоров виртуальной многопроцессорной системы и её представление в виде многомерной решетки задаются в командной строке при запуске программы.

В модели DVMH **виртуальный процессор стал гетерогенным** – он состоит из виртуального хост-процессора, виртуального мультипроцессора, и нескольких виртуальных ускорителей разной архитектуры. При этом виртуальный хост-процессор и виртуальный мультипроцессор отображаются на ЦПУ и работают на общей оперативной памяти, а каждый виртуальный ускоритель имеет свою оперативную память и отображается на аппаратный ускоритель соответствующей архитектуры.

2 Модель программирования и модель параллелизма.

При разработке модели DVMH за основу была взята модель DVM [1], в которую добавлены конструкции для организации вычислений на кластерах с ускорителями и спецификации потоков данных, для управления перемещением данных между оперативной памятью центрального процессора и памятями ускорителей.

Модель параллелизма базируется на специальной форме параллелизма по данным: одна программа – множество потоков данных (ОПМД). В этой модели одна и та же программа выполняется на всех процессорах, но каждый процессор выполняет свое подмножество операторов в соответствии с распределением данных.

Вначале пользователь определяет многомерный массив виртуальных процессоров, на секции которого будут распределяться данные и вычисления. Секция может варьироваться от полного массива процессоров до отдельного процессора.

Затем программист определяет массивы (распределенные данные) и витки циклов, которые должны быть распределены между процессорами. Каждый виток цикла выполняется полностью на одном процессоре.

Распределенные массивы специфицируются директивами отображения данных (раздел 4), а циклы – директивами распределения вычислений (раздел 6).

Остальные переменные (распределяемые по умолчанию) отображаются по одному экземпляру на каждый процессор (*размноженные данные*). Размноженная переменная должна иметь одно и то же значение на каждом процессоре за исключением редукционных переменных в параллельном цикле (раздел 6.2.1) и приватных переменных в циклах (в том числе счетчики циклов).

Размноженным массивом называется массив, в котором нет распределенных измерений – все элементы массива локализованы на каждом процессоре.

Распределение данных определяет множество *локальных* или *собственных переменных* для каждого процессора. Множество собственных переменных определяет *правило собственных вычислений*: процессор присваивает значения только собственным переменным.

При вычислении значения собственной переменной процессору могут потребоваться как значения собственных переменных, так и значения несобственных (удаленных) переменных. Для организации доступа к удаленным данным служат специальные директивы (раздел 7).

Далее программист определяет фрагменты кода, которые могут быть выполнены на ускорителях. Такие фрагменты называются *вычислительными регионами* или просто *регионами*.

Регион может выполняться на одном или нескольких ускорителях одного или разных типов и/или на центральном процессоре. Программист может указать список типов вычислительных устройств, на которых предполагается выполнять регион.

Фрагменты программы вне регионов всегда выполняются на центральном процессоре.

Для каждого региона указываются данные, необходимые для его выполнения (входные, выходные, локальные).

Перемещения данных между вычислительными регионами осуществляются в основном автоматически в соответствии с имеющейся в описании регионов информацией об используемых ими данных. Для управления перемещениями данных между регионами и фрагментами программы вне регионов предусмотрены специальные директивы.

Параллелизм в C-DVMH-программах проявляется на нескольких уровнях:

- Распределение данных и вычислений по процессорам. Этот уровень задается директивами распределения и перераспределения данных и циклов.
- Распределение данных и вычислений по вычислительным устройствам при входе в вычислительный регион.
- Параллельная обработка в рамках конкретного вычислительного устройства. Этот уровень появляется при входе в параллельный цикл, находящийся внутри вычислительного региона.

Наличие этих уровней дает возможность эффективно отобразить программу на кластер с многоядерными процессорами и ускорителями в узлах.

3 Язык C-DVMH.

Язык C-DVMH представляет собой стандартный язык Си, дополненный средствами спецификации параллельного выполнения программы. В язык Си добавлены следующие директивы и спецификации:

- директивы распределения элементов массивов;
- директивы и спецификации распределения витков циклов;
- директивы и спецификации доступа к удаленным данным;
- директивы задания вычислительных регионов;
- спецификации параллельных циклов внутри региона;
- директивы управления перемещениями данных вне регионов;
- директивы задания фрагментов программы внутри региона, которые надо выполнить на центральном процессоре.

Все **dvmh**-директивы построены следующим образом:

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

где:

directive – имя директивы,

clause-list –список спецификаций.

Вводятся они любым из трех практикуемых современными компиляторами способов:

#pragma *dvmh-directive*

_Pragma(“*dvmh-directive***”)**

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

Далее везде в примерах будет использоваться **#pragma**.

Регистр букв имеет значение: все ключевые слова набираются только в нижнем регистре, имена переменных и констант набираются так, как они названы в последовательной программе.

Синтаксис dvmh-директив описывается расширенной БНФ. Используются следующие обозначения:

::=	есть по определению,
x | *y*	альтернативы,
[*x*]	необязательный элемент/конструкция,
[*x*]…	повторение 0 или более раз,
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 Распределение данных.

Язык C-DVMH поддерживает распределение массивов блоками (равными и неравными) и распределение через выравнивание.

4.1 Директива array.

Синтаксис:

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

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

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

Каждая спецификация указывается не более одного раза. Порядок спецификаций не важен.

Спецификации **distribute** и **align** являются взаимоисключающими и не могут одновременно присутствовать в одной директиве **array**.

Спецификации **distribute** и **align** задают распределение массивов, а спецификация **shadow** – ширину теневых граней массивов (раздел 7.2.1). Если в директиве не указана ни одна спецификация или указана только спецификация **shadow,** это означает, что массив будет распределен позже при помощи директив **redistribute** или **realign** (отложенное распределение)**.**

Пример 4.1. Отложеннное распределение массива A.

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

При отсутствии спецификации **shadow** ширина теневых граней устанавливается равной 1.

Синтаксис и семантика отдельных частей директивы описаны в следующих разделах:

distribute-clause раздел 4.1.1
align-clause раздел 4.1.2
shadow-clause раздел 7.2

Директива вставляется перед строкой со стандартными объявлениями массивов. Ее действие распространяется только на один следующий оператор объявления, на все объявленные этим оператором массивы.

Количество измерений, указанное в спецификациях **distribute**, **align**, **shadow** должно совпадать с количеством измерений в объявлении этого массива.

Перед описанием внешних переменных директива **array** используется без спецификаций:

#pragma dvm array

Пример 4.2. Использование директивы **array** для внешней переменной.

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

4.1.1 Распределение данных. Спецификация distribute.

Синтаксис:

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

Ограничение:

- Для каждого измерения массива должен быть задан формат распределения, т.е. количество правил *distribute-axis-rule* должно быть равно количеству измерений массива.

Рассмотрим форматы распределения для одномерных массивов и для одномерной решетки процессоров.

4.1.1.1 Формат block.

Измерение массива распределяется преимущественно равными блоками.

Распределение блоками выполняется следующим образом.

Если количество элементов массива (N) не превышает число процессоров (P), то на каждый процессор попадает либо один элемент массива, либо ни одного.

Если количество элементов массива (N) больше числа процессоров, то количество элементов на процессоре определяется по формуле:

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

где N – число элементов массива, P – число процессоров, k-номер процессора, принимает значение от 1 до P. [x] означает взятие целой части числа.

Пример 4.3. Распределение по формату **block**.

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

процессор	A	B	C
R[0]	**0** **1** **2**	**0** **1**	**0**
R[1]	**3** **4** **5**	**2** **3** **4**	**1**
R[2]	**6** **7** **8**	**5** **6** **7**	**2**
R[3]	**9** **10** **11**	**8** **9** **10**	**3** **4**

*4.1.1.2 Формат **genblock.***

Распределение блоками разных размеров позволяет влиять на балансировку загрузки для алгоритмов, которые выполняют разное количество вычислений на различных участках области данных.

Пусть **NB[P]** – массив неотрицательных целых чисел. Для описания массива можно использовать типы int и long.

Следующая директива

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

разделяет массив ***A*** на ***P*** блоков. Блок **i** размера **NB[i]** распределяется на процессор **R[i]**. При этом должно выполняться равенство

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

Пример 4.4. Распределение неравными блоками.

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

процессор	**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 Формат **wgtblock.***

Формат **wgtblock** определяет распределение блоками по их относительным “весам”.

Пусть задан формат **wgtblock (WB, NBL)**.

WB(i) – вещественное (float или double) неотрицательное число, определяет вес i-го блока для 0≤ i ≤ NBL-1. Если число заданных весов равно NBL, то массив делится на NBL блоков, а блоки распределяются на P процессоров с балансировкой сумм весов блоков на каждом процессоре.

Формат **block** является частным случаем формата **wgtblock(WB, NBL)**, где *WB(i) = 1* для *0≤ i ≤NBL-1*.

Пример 4.4 можно переписать с использованием формата **wgtblock** следующим образом.

Пример 4.5. Распределение блоками по весам.

```
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];
```

В примере 4.5 *P = 4* и распределение идентично примеру 4.4.

В отличие от распределения неравными блоками, распределение по формату **wgtblock** можно выполнить для любого числа процессоров.

Ограничений на размер распределяемого массива не накладывается. Размер массива может быть больше, равен или меньше размера массива с весами.

Пример 4.6. Распределение блоками по весам.

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

*4.1.1.4 Формат **multblock.***

Формат multblock (m) указывает, что размер блока на каждом процессоре должен быть кратен заданному числу m. Для распределения по этому формату размер массива (N) должен быть кратен m.

Распределение осуществляется следующим образом – сначала массив разбивается на блоки по m элементов, а потом эти блоки распределяются по правилам распределения block, только распределяются не элементы массива, а блоки.

Пример 4.6. Распределение по формату **multblock**.

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

процессор	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 Формат не задан.

Если формат не задан (заданы пустые скобки **[ ]**), это означает, что измерение будет полностью локализовано на каждом процессоре (нераспределенное измерение).

4.1.1.6 Многомерные распределения.

При многомерных распределениях формат распределения указывается для каждого измерения. Между измерениями распределяемого массива и массива процессоров устанавливается следующее соответствие.

Пусть массив процессоров имеет **n** измерений. Пронумеруем распределенные измерения массива (без формата **[ ])** слева направо $\mathbf{d_1, ..., d_k}$.. Тогда измерение $\mathbf{d_i}$ будет распределяться на *i*-ое измерение массива процессоров. При этом должно выполняться условие $\mathbf{k \le n}$.

Пример 4.7. Одномерное распределение.

2 процессора

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

	Блоки А	Процессоры
1	**A[0: 49][0:99]**	**0**
2	**A[50:99][0:99]**	**1**

Пример 4.8. Двумерное распределение.

2 * 2 процессора

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

Блоки А

1	**2**
3	**4**

Процессоры

	0	1
0	**1**	**2**
1	**3**	**4**

.

4.1.1.7 Распределение динамически размещаемых массивов

Для динамических массивов распределение может быть только отложенным Директива **redistribute** (раздел 5.1) для динамически размещаемого массива может выполняться только после обращения к функции malloc, которая разместит массив в памяти.

Пример 4.9. Распределение динамически размещаемых массивов.

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

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

4.1.2 Распределение выравниванием. Спецификация align.

Выравнивание массива A на распределенный массив B ставит в соответствие каждому элементу массива A элемент или секцию массива B. Секция массива – подмножество массива, которое само является массивом. Распределение элементов массива B определяет распределение элементов массива A. Если на данный процессор распределен элемент B, то на этот же процессор будет распределен элемент массива A, поставленный в соответствие выравниванием.

Метод распределения через выравнивание выполняет следующие две функции.

1. Одинаковое распределение массивов одной формы на один массив процессоров не всегда гарантирует, что соответствующие элементы будут принадлежать одному процессору. Это вынуждает специфицировать удаленный доступ (раздел 7), там, где его, возможно, нет. Только выравнивание соответствующих элементов массивов гарантирует их размещение на одном процессоре.
2. На один массив могут быть выравнены несколько массивов. Изменение распределения этого массива директивой **redistribute** (раздел 5.1) вызовет соответствующее изменение распределения всей группы массивов.

Синтаксис:

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	**::=**	+
		-

Ограничения:

- Длина списка *align-axis-name* должна быть равна количеству измерений выравниваемого массива.
- Длина списка *templ-axis-spec* должна быть равна количеству измерений базового массива *var-name*.

Пусть задано выравнивание двух массивов с помощью директивы

#pragma dvm array align ([d_0]…[d_n] with B[ard_0]…[ard_m])

float A[A_0**]…[**A_n**];**
d_i – спецификация *i*-го измерения выравниваемого массива A,
ard_j – спецификация *j*-го измерения базового массива B,
A_i – размер i-го измерения.

Если **d_i** задано целочисленной переменной I, то обязательно должно существовать одно и только одно измерение массива B, специфицированное линейной функцией **ard_j = a*I + b**. Следовательно, количество измерений массива A, специфицированных идентификаторами (*align-dummy-use*) должно быть равно количеству измерений массива B, специфицированных линейной функцией.

Пусть i-ое измерение массива A имеет размер A_i, а j-ое измерение массива B, специфицированное линейной функцией **a*I + b**, имеет размер B_j. Т.к. параметр I определен над множеством значений 0…A_i-1, то должны выполняться следующие условия:

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

Если **d_i** не задано (заданы **[ ]**), то i-ое измерение массива A будет локальным на каждом процессоре при любом распределении массива B (аналог локального измерения в спецификации **distribute**).

Если **ard_j =[ ]**, то массив A будет размножен по j -му измерению массива B.

Если **ard_j = *int-expr***, то массив A выравнивается на секцию массива B.

Пример 4.10. Выравнивание массивов.

```
#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];
```

Пусть заданы выравнивания *f1* и *f2*, причем *f2* - выравнивает массив B по массиву C, а *f1* - выравнивает массив A по массиву B. По определению массивы A и B считаются выровненными на массив C. Массив B выравнен непосредственно функцией *f2*, а массив *A* выравнен опосредовано составной функцией *f1*(*f2*). Поэтому применение директивы **realign** к массиву B не вызовет перераспределения массива A.

4.1.3 Нераспределяемые данные

Если для данных не указана директива **array,** то эти данные распределяются на каждый процессор (полное размножение). Такое же распределение можно задать директивой **array** со спецификацией **distribute**, указав для каждого измерения формат **[ ]**. Но в этом случае доступ к данным будет менее эффективным.

4.2 Директива template.

Если при выравнивании массива значения линейной функции **a*I + b** выходят за пределы измерения базового массива, то необходимо определить фиктивный массив - шаблон выравнивания. Затем необходимо произвести выравнивание обоих массивов на этот шаблон. Шаблон распределяется с помощью спецификации **distribute** или директивы **redistribute**. Элементы шаблона не требуют памяти, они указывают процессор, на который должны быть распределены элементы выравненных массивов. Если в директиве не указана спецификация **distribute,** то это означает, что шаблон будет распределен позже при помощи директивы **redistribute.**

Шаблон выравнивания определяется следующей директивой:

Синтаксис:

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

Ограничение.

- Длина списка *size-spec* должна быть равна количеству измерений выравниваемого массива.

Если вместо длины измерения шаблона указаны пустые скобки - " **[]**" , то это означает, что шаблон является динамическим. В этом случае пустые скобки должны быть заданы для всех измерений массива.

Шаблон в программе должен иметь тип void *, а имя этой переменной и есть имя шаблона.

Замечание.

Возможность задания динамических шаблонов находится в процессе реализации.

Рассмотрим следующий пример.

Пример 4.10. Выравнивание по шаблону.

```
#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];
```

Чтобы не было обменов между процессорами, элементы A[i], C[i-1] и B[i+1] должны размещаться на одном процессоре. Выравнивание массивов C и B на массив A невозможно, т.к. функции выравнивания i-1 и i+1 выводят за пределы индексов измерения массива A. Поэтому описывается шаблон TABC и на один элемент шаблона TABC выравниваются те элементы массивов A, B и C, которые должны быть размещены на одном процессоре.

Перед описанием внешних переменных директива **template** используется без спецификаций:

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

4.3 Выравнивание динамически размещаемых массивов.

Для динамических массивов выравнивание может быть только отложенным. Директива **realign** для динамически размещаемого массива может выполняться только после выполнения оператора , который разместит массив в памяти.

Пусть задана цепочка выравниваний директивами **align**

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

где f_i - функция выравнивания,
A_i – динамически размещаемый массив.

Тогда размещение массивов должно происходить в обратном порядке, т.е. сначала A_N , а самый последний A_1. Освобождение памяти, выделенной под массив

A_N, осуществляется в последнюю очередь, после освобождения памяти под массивы, которые выравнены на него.

Пример 4.11. Выравнивание динамически размещаемых массивов.

```
……————————————————————————————
#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 Директивы изменения распределения данных.

5.1 Изменение распределения данных. Директива redistribute.

Синтаксис:

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

Пример 5.1. Изменение распределения данных.

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

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

5.2 Изменение выравнивания данных. Директива realign.

Синтаксис:

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 **::= +**
| **-**

Спецификация **new_value** в директиве **realign** означает, что содержимое массива при выполнении этой операции можно не сохранять, значения элементов массива в результате выполнения этой операции будут не определены.

Пример 5.2. Изменение выравнивания данных.

```
#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 Распределение вычислений. Параллельные циклы.

6.1 Определение параллельного цикла.

Моделью выполнения C-DVMH программы является модель SPMD (одна программа - множество потоков данных). На все процессоры загружается одна и та же программа, но каждый процессор в соответствии с *правилом собственных вычислений* выполняет только те операторы присваивания, которые изменяют значения переменных, размещенных на данном процессоре (собственных переменных).

Таким образом, вычисления распределяются в соответствии с размещением данных (параллелизм по данным). В случае размноженной переменной оператор присваивания выполняется всеми процессорами, а в случае распределенного массива - только процессором (или процессорами), где размещен соответствующий элемент массива.

Определение "своих" и пропуск "чужих" операторов может вызвать значительные накладные расходы при выполнении программы. Поэтому спецификация распределения вычислений разрешается только для циклов, которые удовлетворяют следующим условиям:

- цикл является тесно-гнездовым циклом с прямоугольным индексным пространством;
- распределенные измерения массивов индексируются только регулярными выражениями типа a*I + b, где I - индекс цикла;
- левые части операторов присваивания одного витка цикла размещены на одном процессоре и, следовательно, виток цикла полностью выполняется на этом процессоре;
- нет зависимости по данным кроме редукционной зависимости и регулярной зависимости по распределенным измерениям;
- левая часть оператора присваивания является ссылкой на распределенный массив, редукционную переменную (раздел 6.2.1), приватную переменную (раздел 6.2.2);
- нет операторов ввода-вывода, операторов перехода за границу цикла и dvmh-директив непосредственно в теле цикла и во всех функциях, вызываемых в цикле;

Цикл, удовлетворяющий этим условиям, будем называть *параллельным циклом.* Управляющая переменная последовательного цикла, охватывающего параллельный цикл или вложенного в него, может индексировать только локальные (размноженные) измерения распределенных массивов.

На сам цикл также накладываются следующие ограничения:

- Цикл должен быть описан for-конструкцией.
- Должны быть заданы три секции - переменная цикла с инициализацией, условие завершения и выражение, которое модифицирует переменную цикла.
- В инициализационной секции должно быть ровно 1 присваивание, причем допускается иметь описание переменной (но только одной и только с инициализацией): for (int i = getStart()+abs(S); i < N; i++). Правая часть должна вычисляться без побочных эффектов, не зависеть от переменных цикла, либо данных, изменяемых в нем, а также от значений распределенных массивов.
- Условием продолжения должно быть сравнение на <, <=, >, >= с той же самой переменной в левой части, что была в присваивании в инициализационной секции. Правая часть должна вычисляться без побочных эффектов и не зависеть от переменных цикла, либо данных, изменяемых в нем, а также от значений распределенных массивов.
- В инкрементальной секции должен быть оператор -=, --, ++, +=. Префиксные и постфиксные выражения допускаются. Этот оператор должен модифицировать ту же переменную, в которую стоит присваивание в первой секции и которая сравнивается в средней секции. Правая часть должна вычисляться без побочных эффектов и не зависеть от переменных цикла, либо данных, изменяемых в нем, а также от значений распределенных массивов.

6.2 Распределение витков цикла. Директива parallel.

Параллельный цикл специфицируется следующей директивой:

parallel-directive **::= parallel** *parallel-map* [*parallel-clause-list*]
parallel-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*

Директива **parallel** размещается перед заголовком цикла (внешнего цикла гнезда). Если в качестве *parallel-map* задано правило отображения, то директива распределяет витки циклов в соответствии с распределением массива или шаблона. Семантика директивы аналогична семантике спецификации **align**, где индексное пространство выравниваемого массива заменяется индексным пространством цикла. Индексы циклов в списке *align-axis-name* перечисляются в том порядке, в котором размещены соответствующие операторы цикла в тесно-гнездовом цикле. Если вместо правила отображения указано число, то цикл будет параллельным, но нераспределенным, а число будет указывать количество циклов гнезда, которое ассоциируется с данной директивой.

Синтаксис и семантика отдельных частей директивы описаны в следующих разделах:

private-clause	раздел 6.4,
reduction-clause	раздел 6.2.1,
shadow-renew-clause	раздел 7.2.2,
remote-access-clause	раздел 7.3,
across-clause	раздел 7.2.3,
cuda-clause	раздел 8.2.1,
stage-clause	раздел 7.2.3.

Пример 6.1. Распределение витков цикла.

```
#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];
}
```

Цикл удовлетворяет всем условиям параллельного цикла. В частности, оба оператора присваивания выполняются одним процессором благодаря выравниванию массивов *A* и *B*.

Если левые части операторов присваивания размещены на разных процессорах (распределенный виток цикла), то цикл необходимо разделить на несколько циклов.

6.2.1 Редукционные операции и переменные. Спецификация reduction.

Очень часто в программе встречаются циклы, в которых выполняются *редукционные операции* - в некоторой переменной суммируются элементы массива или вычисляется максимальное (минимальное) значение. Витки таких циклов можно распределять и выполнять параллельно, если указать спецификацию **reduction**.

Синтаксис:

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

Ограничения:

- Редукционными переменными не могут быть распределенные массивы.
- Редукционная переменная вычисляется на каждом витке параллельного цикла в операторах определенного вида – редукционных операторах. Для редукционных операций sum и product - это операторы вычисления суммы и произведения соответственно (например, операциями “+=”, “*=”), для

редукционных операций max и min - это условные операторы в которых вычисляются минимальное или максимальное значение (например, "v=(v<A?A:v) ", "v=(v>A?A:v) "), для редукционных операций and, or и xor - это операторы присваивания, в которых вычисляется значение логического выражения с использованием соответствующей побитовой операции (например, "&=", "|=", "^="). Для редукционных операций maxloc и minloc - это условные операторы, в которых вычисляются максимальное или минимальное значение массива и операторы присваивания, в которых запоминаются координаты максимального и минимального элементов массива. Второй параметр функций maxloc и minloc (loc-variable) - это переменная, описывающая координаты элемента массива с найденным максимальным (соответственно, минимальным) значением. Для одномерного массива это скаляр, для многомерного – одномерный массив, размер которого (size) равен числу измерений массива. Значение i-го элемента массива – координата по i-му измерению. Изменять значение второго параметра (loc-variable) можно только вместе с изменением значения редукционной переменной (red-variable).

Пример 6.2. Спецификация редукции.

```
. . . . . .
S = 0;
/* массив A – распределенный. По правилу собственных вычислений (раздел 6.3)
необходимо использовать директиву удаленного доступа (раздел 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;
        }
}
```

Замечание.

Все dvmh-директивы подвергаются подстановке макроопределений препроцессором. Если в программе используется редукции **min** и/или **max**, то нельзя использовать одноименные макросы. Это приведет к ошибкам компиляции.

6.2.2 Приватные переменные. Спецификация private.

Спецификация **private** объявляет переменную приватной. Переменная называется приватной, если ее использование локализовано в пределах одного витка цикла.

Редукционная переменная не может иметь атрибут **private.**

Синтакис:

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

6.3 Вычисления вне параллельного цикла.

Вычисления вне параллельного цикла выполняются по правилу собственных вычислений.

Оператор присваивания

lh = rh;

может быть выполнен некоторым процессором, только если *lh* присутствует на нем.

Если **lh** – элемент распределенного массива, то такой оператор (оператор собственных вычислений) будет выполняться только тем процессором (или теми процессорами), где присутствует данный элемент распределенного массива. Все данные, используемые в выражениях **rh**, должны также присутствовать на этом процессоре.

Если какие-либо данные из выражений **rh** отсутствуют на этом процессоре, то их необходимо указать в директиве удаленного доступа (раздел 7.3) перед этим оператором.

Пример 6.3. Собственные вычисления.

```
#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];
. . .
/*                Обратная подстановка алгоритма Гаусса
                 собственные вычисления вне циклов
```

```
                оператор собственных вычислений
                левая и правая части – на одном процессоре*/

X[N-1] = A[N-1][N] / A[N-1][N];
for (j = N - 2; j >= 0; j--)
```

/* Спецификация **remote_access** (раздел 7.3) в параллельном цикле специфицирует удаленные данные (X[j+1]**)** для всех процессоров, на которые распределен массив *A*.*/

```
#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];

/* Собственные вычисления в последовательном цикле,
                охватывающем параллельный цикл*/
 X[j] = A[j][N] / A[j][j];
}
```

Отметим, что A[j][N] и A[j][j] локализованы на том процессоре, где размещается X[j].

7 Спецификация удаленных данных.

7.1 Определение удаленных ссылок.

Удаленными данными будем называть данные, которые размещены не на том процессоре, на котором используются. Ссылки на такие данные будем называть *удаленными ссылками*. Рассмотрим оператор присваивания в теле цикла:

A[inda] = B[indb]

где
A, B - распределенные массивы,
inda, indb - индексные выражения.

В модели DVMH этот оператор будет выполняться процессором, на котором размещен элемент A[inda]. Ссылка B[indb] не является удаленной ссылкой, если элементы A[inda] и B[indb] размещены на одном процессоре. Единственной гарантией этого является выравнивание массива B на распределенный массив A. Если выравнивание невозможно или не было выполнено, то ссылку B[indb] необходимо специфицировать как удаленную. В случае многомерных массивов данное правило применяется к каждому распределенному измерению.

По степени эффективности обработки удаленные ссылки разделены на два типа: **shadow** и **remote**.

Если массивы A и B выровнены и

inda = indb ± d (d – положительная целочисленная константа),

то удаленная ссылка B[indb] принадлежит типу **shadow**.

Удаленная ссылка на многомерный массив принадлежит типу **shadow**, если массивы выровнены и правило

inda = indb ± d (d – положительная целочисленная константа),

выполняется для каждого измерения.

Удаленные ссылки, не принадлежащие типу **shadow**, составляют множество ссылок типа **remote**.

Особым множеством удаленных ссылок являются ссылки на редукционные переменные (см 6.2.1), которые принадлежат типу **reduction**. Эти ссылки могут использоваться только в параллельном цикле.

7.2 Удаленные ссылки типа shadow

7.2.1 Спецификация массива с теневыми гранями

Удаленная ссылка типа **shadow** означает, что обработка удаленных данных будет происходить через “теневые” грани. Теневая грань представляет собой буфер, который является непрерывным продолжением локальной секции массива в памяти процессора (см. рис.7.1). Рассмотрим оператор

A[i] = B[i + d2] + B[i – d1]

где d1, d2 – целые положительные константы. Если обе ссылки на массив B являются удаленными ссылками типа **shadow**, то для массива B необходимо указать спецификацию **shadow [** d1 **:** d2 **]**, где d1 – ширина левой грани, а d2 – ширина правой грани. Для многомерных массивов необходимо специфицировать грани по каждому измерению. При спецификации теневых граней в описании массива указывается максимальная ширина по всем удаленным ссылкам типа **shadow**. По умолчанию, распределенный массив имеет теневые грани шириной 1 с обеих сторон каждого распределенного измерения.

Спецификация **shadow** задается в директиве **array**.

Синтаксис:

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

Ограничения:

- Размер левой теневой грани (*low-width*) и размер правой теневой грани (*high-width*) должны быть целыми выражениями, значения которых больше или равны 0.
- Если задана одна ширина(*width*), то выражение должно быть без побочных эффектов.

Задание размера теневых граней как *width* эквивалентно заданию *width : width*.

7.2.2 Спецификация обновления теневых граней для цикла

Спецификация обновления теневых граней является частью директивы parallel:

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*

Ограничения:

- Размер обновляемых теневых граней не должен превышать максимального размера, описанного в спецификации **shadow**.
- Если размеры теневых граней не указаны, то используются максимальные размеры.

Выполнение спецификации заключается в обновлении теневых граней значениями удаленных переменных перед выполнением цикла.

Пример 7.1. Спецификация **shadow-**ссылок без угловых элементов.

```
#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.;
```

При обновлении значений в теневых гранях используются максимальные размеры 1:2**,** заданные в спецификации shadow.

На каждом процессоре создаются два буфера, которые являются непрерывным продолжением локальной секции массива. Левая теневая грань имеет размер в 1 элемент (для B[i-1]), правая теневая грань имеет размер в 2 элемента (для B[i+1] и B[i+2]). Если перед выполнением цикла произвести обмен между процессорами, то цикл может выполняться каждом процессором без замены ссылок на массивы ссылками на буфер.

Для многомерных распределенных массивов теневые грани могут распределяться по каждому измерению. Особая ситуация возникает, когда необходимо обновлять "угол" теневых граней. В этом случае требуется указать дополнительный параметр **corner**.

Пример 7.2.. Спецификация **shadow**-ссылок с угловыми элементами.

```
#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.;
```

Теневые грани для массива *B* распределяются по умолчанию размером в 1 элемент по каждому измерению. Т.к. имеется удаленная "угловая" ссылка *B[i+1][j+1]*, то указывается параметр **corner**.

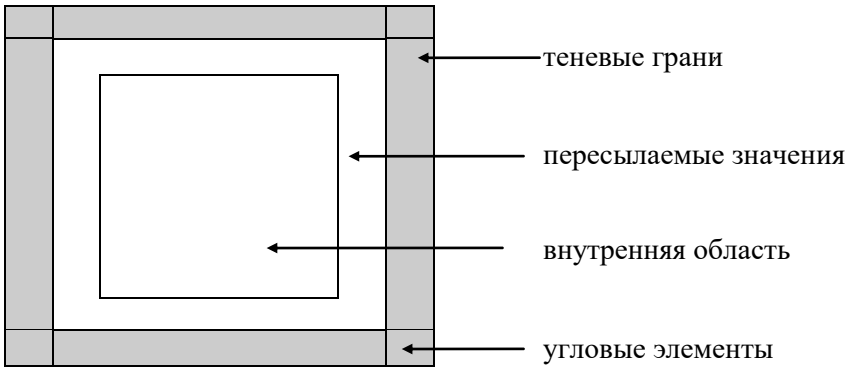

Рис. 7.1. Схема локальной секции массива с теневыми гранями.

7.2.3 Спецификация across зависимых ссылок типа shadow для одного цикла. Спецификация stage.

Рассмотрим следующий цикл

```
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.;
```

Между витками цикла с индексами *i1* и *i2 (i1<i2)* существует зависимость по данным (информационная связь) массива *A*, если оба эти витка осуществляют обращение к одному элементу массива по схеме запись-чтение или чтение-запись.

Если виток *i1* записывает значение, а виток *i2* читает это значение, то между этими витками существует потоковая зависимость или просто зависимость $i1 \rightarrow i2$.

Если виток *i1* читает “старое” значение, а виток *i2* записывает “новое” значение, то между этими витками существует обратная зависимость $i1 \leftarrow i2$.

В обоих случаях виток *i2* может выполняться только после витка *i1*.

Значение *i2* - *i1* называется диапазоном или длиной зависимости. Если для любого витка *i* существует зависимый виток *i* + *d* (*d* - константа), тогда зависимость называется регулярной или зависимостью с постоянной длиной.

Цикл, в котором существуют регулярные зависимости по распределенным массивам, можно распределять с помощью директивы **parallel**, указывая спецификацию **across**.

Синтаксис:

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*

В спецификации **across** перечисляются все распределенные массивы, по которым существует регулярная зависимость по данным. Для каждого измерения массива указывается длина прямой зависимости (*flow-dep-length*) и длина обратной зависимости (*anti-dep-length*). Нулевое значение длины зависимости означает отсутствие зависимости по данным.

Пример 7.3.. Спецификация цикла с регулярной зависимостью по данным.

```
#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.;
```

По каждому измерению массива *A* существует прямая и обратная зависимость длиной 1.

Спецификация **across** реализуется через теневые грани. Длина обратной зависимости определяет ширину обновления правой грани, а длина прямой

зависимости – ширину обновления левой грани. Обновление значений правых граней производится перед выполнением цикла (как для директивы **shadow_renew**). Обновление левых граней производится во время выполнения цикла по мере вычисления значений удаленных данных. Это позволяет организовать так называемые волновые вычисления для многомерных массивов. Фактически, **across**-ссылки являются подмножеством **shadow**–ссылок, между которыми существует зависимость по данным. Такая зависимость по данным в общем случае требует последовательного выполнения витков цикла с передачей значений обновленных общих данных от каждого отработавшего процессора к следующему. Однако во многих случаях, например, когда двумерный цикл распределен по одному измерению на линейку процессоров, можно эффективно распараллелить такой цикл, если организовать его конвейерное выполнение. Идея конвейерного выполнения заключается в том, что первый процессор выполняет часть “своих” витков цикла и передает следующему процессору соответствующую порцию обновленных общих данных. После получения этих данных второй процессор начинает выполнять “свои” витки цикла параллельно с выполнением первым процессором второй порции витков, и т.д. Оптимальный размер порции витков зависит от количества витков цикла, времени выполнения каждого витка, числа процессоров, а также латентности и пропускной способности коммуникационной среды. Система поддержки обеспечивает вычисление оптимальной порции витков и соответствующее дробление одного или нескольких измерений цикла, а также передачу обновленных общих данных между процессорами.

Пользователь имеет возможность указать количество шагов конвейера. Для этого он должен указать спецификацию **stage.**

Синтаксис:

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

7.3 Удаленные ссылки типа remote. Директива remote_access.

Удаленные ссылки типа **remote** специфицируются директивой **remote_access.**

Синтаксис:

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

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

Директива **remote_access** может быть отдельной директивой (область действия - следующий оператор) или дополнительной спецификацией в директиве **parallel** (область действия – тело параллельного цикла).

В пределах нижестоящего оператора или параллельного цикла компилятор заменяет все вхождения удаленной ссылки ссылкой на буфер. Пересылка удаленных данных производится перед выполнением оператора или цикла.

Пример 7.4. Спецификация удаленных ссылок типа **remote**.

#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];
```

Первые две директивы **remote_access** специфицируют удаленные ссылки для отдельных операторов. Спецификация **remote_access** в параллельном цикле специфицирует удаленные данные (столбец матрицы) для всех процессоров, на которые распределен массив A.

Замечание. Директива **remote_access** для параллельных циклов и регионов пока не реализована.

8 Распределение вычислений. Вычислительный регион.

Вычислительный регион выделяет часть программы (с одним входом и одним выходом) для возможного выполнения на одном или нескольких вычислительных устройствах.

Регион может быть выполнен на одном или сразу нескольких ускорителях и/или ЦПУ. При этом на ЦПУ может быть выполнен любой регион. Для выполнения региона на различных ускорителях на регион могут накладываться различные дополнительные ограничения. Например, на CUDA-устройстве может быть выполнен любой регион, в котором нет операторов ввода/вывода, вызовов внешних процедур.

Вложенные (статически или динамически) регионы не допускаются.

Распределенные массивы распределяются между выбранными вычислителями (с учетом весов (см. раздел 14.2.1) и быстродействия вычислителей), нераспределенные данные размножаются. Витки параллельных циклов внутри региона делятся между выбранными для выполнения региона вычислителями в соответствии с правилом отображения параллельного цикла, заданного в директиве параллельного цикла.

8.1 Директива region.

Синтаксис.

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**

Спецификации *in-out-local-clause* предназначены для указания направления использования данных в регионе:

in - входные данные: в регионе должны быть самые последние значения этих данных;

out - выходные данные: значения указанных переменных в регионе изменяются и могут быть использованы далее;

local - локальные данные: значения указанных переменных в регионе изменяются, но эти изменения не будут использованы далее;

inout - сокращенная запись одновременно двух спецификаций **in** и **out;**

inlocal - сокращенная запись одновременно двух спецификаций **in** и **local.**

В спецификациях *in-out-local-clause* не обязательно указывать все используемые в регионе переменные. Для используемых в регионе, но не указанных в спецификациях переменных действуют следующие правила по умолчанию:

- все используемые массивы считаются используемыми полностью (подмассивы не выделяются);
- всякая переменная, которая используется на чтение, получает атрибут **in**;
- всякая переменная, которая используется на запись, получает атрибут **inout**;
- всякая переменная, направление использования которой не поддаётся определению, получает атрибут **inout**;
- атрибуты **local** и **out** не проставляются.

Если для переменной указано только направление **in** (не указано **out** или **local**), это означает, что в такую переменную в регионе вообще нет записей и она не меняется в процессе его выполнения.

В спецификациях *in-out-local-clause* для каждого измерения массива можно указать секцию (*subscript-range*), задав диапазон индексов.

Допускаются составные указания, например, **out**(s[1:5]), **out** (s[7:10]) или **in**(s[1:5]), **out**(s[6:10]) и пересекающиеся указания, например, **out**(s[1:6]), **out**(s[3:10]) или даже **out** (s[1:6]), **out**(s[3:5]).

Не допускаются конфликтующие указания, такие как **out**(v), **local**(v).

Спецификация **targets** предназначена для указания списка типов вычислителей, на которых предполагается выполнять регион.

Пока *target_name* в спецификации **targets** может принимать только два значения:

CUDA - регион предполагается выполнять на CUDA-устройстве;
HOST - регион предполагается выполнять на центральном процессоре.

В директиве **region** может быть только одна спецификация **targets**.

Данное указание ограничивает набор типов вычислительных устройств, для использования которых регион будет подготовлен компилятором. Действительное же выполнение региона может происходить только на доступных DVMH-программе ускорителях (или на ЦПУ), количество и типы которых указываются при ее запуске с помощью переменных окружения. Определение конкретных исполнителей региона (из числа доступных вычислителей, для которых были сгенерированы программы региона) производится во время выполнения программы.

Спецификация **async** предназначена для указания асинхронного исполнения региона. При запуске региона в любом режиме (синхронный, асинхронный) ожидание завершения ранее запущенного региона возникает, если спецификациями **in**, **out**, **local**, **inout**, **inlocal** задается необходимость изменить данные, используемые этим (ранее запущенным) регионом или необходимость использовать (запись или чтение) данные, изменяемые этим (ранее запущенным) регионом (**out**, **inout**, **local**, **inlocal**). Управление не перейдет на следующий за синхронным регионом оператор, пока текущий регион не закончит исполнение. Управление может перейти на следующий за асинхронным регионом оператор, не дожидаясь его завершения (или даже его старта).

Замечание. Спецификация **async** находится в процессе реализации.

8.2 Описание конструкций региона.

Вычислительный регион имеет вид:

#pragma dvm region [*clause-list*]

{

<region inner>

}

Содержимое региона - <region inner> - это последовательность следующих друг за другом конструкций - в произвольном порядке и в любом количестве:

- параллельный цикл;
- последовательная группа операторов;
- хост-секция.

Регион может быть пустым, в этом случае <region inner> не содержит ни одной конструкции.

8.2.1 Параллельный цикл

Параллельный цикл - важнейшая часть вычислительного региона.

В параллельном цикле, входящем в регион, может быть задана дополнительная спецификация *cuda-clause*.

Синтакис:

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

Спецификация **cuda_block** предназначена для указания размера блока нитей CUDA-устройства. Если указано целочисленное выражение - тогда блок полагается одномерным, может указываться два или три целочисленных выражения через запятую - соответственно, блок будет полагаться имеющим указанную размерность.

8.2.2 Последовательная группа операторов

Каждый оператор последовательной группы операторов исполняется на всех вычислителях, выбранных для выполнения региона, кроме случая модификации в нем распределенных данных - тогда действует правило собственных вычислений.

Замечание. Контроль собственных вычислений в последовательной группе операторов региона пока не реализован. Операторы, в которых модифицируются распределенные данные, должны быть вне региона. Можно прервать выполнение региона в этом месте программы или перенести операторы за пределы региона.

8.2.3 Хост-секция

Хост-секция - часть программы с одним входом и одним выходом внутри региона, которая всегда будет выполняться на ЦПУ.

Синтаксис:

hostsection-directive **::= host_section**

Хост-секция имеет вид:

#pragma dvm host_section

{

<hostsection inner>

}

Хост-секции предназначены для промежуточного контроля значений переменных по ходу выполнения региона.

Внутри хост-секций разрешены операции вывода и вызовы внешних процедур. Можно использовать директивы **get_actual** (раздел 9) **,** директивы **actual** запрещены.

9 Управление перемещением данных, актуальность. Директивы актуализации get_actual и actual.

Вне вычислительных регионов управление перемещением данных между оперативной памятью ЦПУ и памятями ускорителей задается при помощи специальных директив актуализации.

Синтаксис:

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*] **]**

Директива **get_actual** делает все необходимые обновления для того, чтобы в памяти ЦПУ были актуальные (т.е. самые новые) значения данных в указанных в списке подмассивах и скалярах.

Директива **actual** объявляет тот факт, что указанные в списке подмассивы и скаляры имеют самые новые значения в памяти ЦПУ. Значения указанных переменных и элементов массивов, находящиеся в памяти ускорителей, считаются устаревшими и перед использованием будут при необходимости обновлены.

10 Директива указания свойств объявленных функций.

Если фактический аргумент вызываемой функции является распределенным массивом, то соответствующий формальный аргумент должен иметь *наследуемое распределение*.

Наследуемое распределение означает, что формальный аргумент наследует распределение фактического аргумента при каждом вызове процедуры.

Наследуемое распределение описывается директивой **inherit**, которая вставляется перед объявлением функции и перед ее описанием.

Синтаксис:

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

Все переменные из списка должны присутствовать в аргументах функции. При этом те, которые имеют тип void * считаются шаблонами, а остальные – массивами. Если в списке указан шаблон, то формальный и фактический аргументы должны иметь тип void *.

Пример 10.1. Распределение локальных массивов и формальных параметров.

```
#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 Функции.

11.1 Вызов функции из параллельного цикла.

Функция, вызываемая из параллельного цикла, не должна иметь побочных эффектов и содержать обменов между процессорами (*прозрачная функция*). Как следствие этого, прозрачная функция не содержит операторов ввода-вывода и DVMH-директив.

11.2 Вызов функции вне параллельного цикла.

Если в качестве аргумента функции используется распределенный массив, то аргумент должен указывать на начало массива и размерности фактического и формального аргументов должны совпадать.

11.3 Формальные аргументы.

Формальные аргументы могут иметь только наследуемое распределение (раздел 10). Если формальным аргументом является шаблон, то он должен быть перечислен в списке директивы **inherit и** иметь тип void*.

Пример 11.1 Распределение формальных параметров.

```
#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 Локальные массивы.

Локальные массивы могут распределяться в функции (указаниями **distribute** и **align**, директивами **redistribute** и **realign**). Локальный массив может быть выровнен на формальный аргумент. Указание **distribute** распределяет локальный массив на ту подсистему процессоров, на которой была вызвана функция (*текущая подсистема*).

Пример 11.2. Распределение локальных массивов и формальных аргументов.

```
#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 Ввод/вывод.

В C-DVMH реализован стандарт C99 с некоторыми ограничениями и расширениями.

Разрешены следующие операции ввода/вывода:

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.

Ввод/вывод массива целиком для распределенных данных обеспечивается функциями fread/fwrite. Перед поэлементным вводом/выводом распределенных массивов должна стоять директива **remote_access** (см. раздел 7.3).

Для функций fopen и freopen расширен набор режимов открытия файла:

- При добавлении литеры “l” или “L” к режиму открытия файла каждый процессор открывает свой локальный файл и все операции производятся каждым процессором независимо от других. При открытии в этом режиме в имени файла можно использовать конструкцию “%d” для задания различных имен различными процессорами.

  Следующий оператор

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

приведет к открытию N файлов (где N – кол-во процессоров в текущей многопроцессорной системе) на запись с именами out_0000.txt, out_0001.txt, out_0002.txt, out_0003.txt, …

Удаление всей группы файлов осуществляется при помощи функции

int dvmh_remove_local(const char *filename);

- При добавлении литеры “p” или “P” к режиму открытия файла все процессоры осуществляют параллельный ввод/вывод в глобальный файл.

Ограничение.

- В функциях семейства **printf** формат “%” использовать нельзя.
- Запись в локальные файлы и чтение их локальных файлов осуществляется на одной и той же процессорной решетке и в массивы, имеющие одинаковое распределение.

13 Компилятор с языка C-DVMH.

Компилятор с языка C-DVMH преобразует исходную программу в параллельную программу на языке СИ с вызовами функций системы поддержки выполнения DVMH-программ (библиотека Lib-DVMH). Кроме того, компилятор создает для каждой исходной программы несколько дополнительных модулей для обеспечения выполнения регионов программы на ГПУ с использованием технологии CUDA.

Для каждого параллельного цикла из региона компилятор генерирует процедуру-обработчик и ядро для вычислений на ГПУ и процедуру-обработчик для выполнения этого параллельного цикла на ЦПУ.

Обработчик – это подпрограмма, осуществляющая обработку части параллельного цикла на конкретном вычислительном устройстве. Аргументами обработчика являются описатель устройства и часть параллельного цикла. Обработчик запрашивает порцию для исполнения (границы цикла и шаг), конфигурацию параллельной обработки (количество нитей), инициализацию редукционных переменных, а после выполнения порции передает результат частичной редукции в систему поддержки.

Так, для обработки частей цикла CUDA-обработчик вызывает специальным образом сгенерированное CUDA-ядро. CUDA-ядро выполняется на графическом процессоре, производя вычисления, составляющие тело цикла.

По умолчанию компилятор генерирует обработчики для всех типов вычислителей, для которых он не обнаружил противопоказаний, анализируя их содержимое. Посредством спецификации **targets** директивы **region** пользователь может указать, на вычислителях каких типов предполагается выполнять регион. В таком случае компилятор генерирует указанные виды обработчиков.

Основная работа по реализации модели выполнения параллельной программы (например, распределение данных и вычислений) осуществляется динамически. Это позволяет обеспечить динамическую настройку DVMH-программ при запуске (без перекомпиляции) на конфигурацию параллельного компьютера (количество процессов, ускорителей, их производительность и тип, а также латентность и пропускную способность коммуникационных каналов). Тем самым программист получает возможность иметь один вариант программы для выполнения на последовательных и параллельных ЭВМ различной конфигурации.

14 Режимы распределения данных и вычислений между вычислительными устройствами.

Одним из важных аспектов функционирования такой программной модели, как DVMH, является вопрос отображения исходной программы на все уровни параллелизма и разнородные вычислительные устройства. Важными задачами механизма отображения является обеспечение корректного выполнения всех поддерживаемых языком конструкций на разнородных вычислительных устройствах, балансировка нагрузки между вычислительными устройствами, а также выбор оптимального способа выполнения каждого участка кода на том или ином устройстве.

14.1 Схема выполнения DVMH-программы.

Выполнение DVMH-программы можно представить, как выполнение последовательности вычислительных регионов и участков между ними, которые будем называть внерегионным пространством. Код во внерегионном пространстве выполняется на центральном процессоре, тогда как вычислительные регионы могут выполняться на разнородных вычислительных устройствах. Как внутри, так и вне регионов могут быть как параллельные циклы, так и последовательные участки программы.

Выполнение DVMH-программы начинается синхронно всеми запущенными процессорами. На каждый процессор выделяется одна основная последовательная нить выполнения.

При входе в вычислительный регион каждый процессор независимо выполняет дополнительное распределение данных, используемых этим вычислительным регионом, по вычислительным устройствам, выбранным для выполнения региона. На этом этапе производится динамическое планирование с целью балансировки нагрузки и минимизации временных затрат на перемещения данных, связанных с перераспределением данных.

При входе в параллельный цикл внутри региона каждый процессор разделяет работу в соответствии с распределением данных по вычислительным устройствам. Затем он выбирает для каждого конкретного устройства обработчик, а также оптимизационные параметры для выбранного обработчика. На этом этапе производится динамическая настройка оптимизационных параметров, включая

количество ЦПУ-нитей для обработчиков на ЦПУ, а также размер и форму блока нитей для CUDA-обработчиков с целью минимизации времени исполнения на каждом отдельном устройстве.

Параллельный цикл внутри региона при исполнении распадается на несколько частей, каждая из которых обрабатывается некоторым обработчиком на некотором вычислительном устройстве. На этом этапе собирается основная информация для отладки эффективности DVMH-программы.

При выходе из вычислительного региона собирается дополнительная информация для целей динамического планирования выполнения региона, а также может быть осуществлена сравнительная отладка с целью контроля корректности результатов, полученных при счете на ускорителях.

14.2 Режимы распределения данных и вычислений.

В DVMH-программах все распределяемые данные распределяются блочно: каждое распределенное измерение делится на отрезки. По каждому измерению распределенного массива можно задавать или равномерное блочное распределение, или распределение взвешенными блоками, т.е. с учетом заданного вектора весов. Эти указания непосредственно выполняются при распределении данных между процессорами. Вложенные распределения, появляющиеся при входе в вычислительный регион, строятся с использованием этой информации, но, в силу разнородности вычислительных устройств, могут иметь отличающуюся от внешней схему распределения.

Системой поддержки выполнения DVMH-программ поддерживаются четыре режима распределения данных и вычислений по вычислительным устройствам в точках входа в регионы:

1. Простой статический режим.
2. Простой динамический режим (пока нет описания).
3. Динамический режим с подбором схемы распределения.
4. Динамический режим с использованием подобранной схемы распределения.

Рассмотрим подробнее эти режимы распределения.

14.2.1 Простой статический режим.

В простом статическом режиме в каждом регионе распределение производится одинаково. Пользователем задается вектор относительных производительностей вычислительных устройств, имеющихся в каждом узле кластера (переменная окружения **DVMH_CUDAS_PERF,** см. Приложение 2 . Также они могут быть грубо определены автоматически при старте программы. Затем эти производительности накладываются на параметры межпроцессного распределения данных, которое по каждому распределенному измерению может быть распределено

как равномерно, так и с заданным вектором весов. В этом режиме сводятся к минимуму перемещения данных, связанные с их перераспределением, но не учитывается различное соотношение производительности вычислительных устройств на разных фрагментах программы.

В этом режиме используются обработчики по умолчанию, а оптимизационные параметры получают следующие значения:

- Количество используемых ЦПУ-нитей – максимально доступное текущему процессору.
- Метод планирования расписания ЦПУ-нитей – автоматический (в терминах OpenMP).
- Размеры и форма CUDA-блока нитей для каждого цикла выбираются исходя из количества измерений параллельного цикла, количества и месторасположения измерений с зависимостями, количества используемых аппаратных ресурсов на одну CUDA-нить, количества имеющихся аппаратных ресурсов графического процессора и способа их распределения по CUDA-нитям.
- Способ обработки редукционных операций в CUDA-обработчике выбирается динамически для каждого цикла, исходя из имеющегося объема памяти графического процессора – или более быстрый и более требовательный по объему памяти, или менее быстрый и менее требовательный по объему памяти.

14.2.2 Динамический режим с подбором распределения.

В этом режиме в каждом вычислительном регионе распределение данных и вычислений выбирается на основе постоянно пополняющейся истории запусков данного региона и его соседей, определяющихся динамически.

Для каждого варианта запуска региона определяются:

- Динамически предшествующие варианты запуска региона, являющиеся поставщиками актуальных (самых новых значений) входных данных, причем учитываются и ранее закончившиеся регионы, которые использовали актуализируемые данные только на чтение. Поставщиками актуальных данных также являются директивы **actual** и **get_actual**, приравниваемые к регионам, исполняемым исключительно на ЦПУ и имеющим пустое тело.
- Зависимость времени работы от распределения данных, причем учитываются все варианты запуска одного и того же вычислительного региона.
- Количество вхождений в регион.

Зависимость времени работы от распределения данных строится в виде табличной функции времени от распределений распределенных массивов, которая является суммой таких же табличных функций, построенных для параллельных циклов, последовательных участков и хост-секций, содержащихся внутри данного варианта запуска региона.

Время обработки последовательных участков считается постоянным.

Время выполнения хост-секций вычисляется как сумма времени выполнения операторов хост-секции (постоянная величина) и времени выполнения директив **get_actual**, содержащихся в данной хост-секции.

В этом режиме периодически включается построение схемы распределения данных во всех вариантах запуска вычислительных регионов на основе накопленных сведений в целом для программы на основе анализа последовательности вариантов запуска регионов и характеристик их выполнения. При построении таких схем учитываются как внутренние показатели вариантов запуска регионов в виде зависимости времени работы от распределения данных, так и последовательность выполнения вариантов запуска вычислительных регионов с целью минимизации в том числе и временных затрат на перераспределение данных. После того, как схема построена, она применяется, и происходит дальнейшее накопление характеристик и информации о вычислительных регионах и параллельных циклах.

Построенные схемы распределения могут быть записаны в файл для использования в последующих запусках программы как в этом режиме, так и в третьем режиме распределения данных и вычислений. В качестве начального приближения может быть использован вектор производительностей вычислительных устройств в том же виде, как и для простого статического режима.

Из данного режима возможен переход в третий режим в любой точке выполнения программы, что обеспечивает упрощенную схему подбора и использования схемы распределения без необходимости сохранения параметров в файле и повторного запуска программы.

14.2.3 Динамический режим с использованием подобранной схемы распределения.

В динамическом режиме с использованием подобранной схемы распределения в каждом вычислительном регионе распределение выбирается на основе предоставленной схемы распределения, построенной при работе программы во втором режиме, причем есть возможность как перейти в этот режим непосредственно из второго, так и использовать схему распределения из файла, полученного в результате работы программы во втором режиме.

При использовании схемы распределения из файла не гарантируется ее корректное использование в случае, если параметры программы были изменены. В особенности это касается тех параметров, которые влияют на путь выполнения программы (выбор другого метода расчета, отключение или включение этапов расчета).

14.3 Механизм динамического переупорядочивания массивов.

Для оптимизации доступа к глобальной памяти ГПУ в систему поддержки выполнения DVMH-программ был внедрен механизм динамического переупорядочивания массивов. Данный механизм перед каждым циклом использует информацию о взаимном выравнивании цикла и массива, которая уже имеется в DVMH-программе для отображения на кластер и распределения вычислений. Он устанавливает соответствие измерений цикла и измерений массива, после чего переупорядочивает массив таким образом, чтобы при отображении на архитектуру CUDA доступ к элементам осуществлялся наилучшим образом — соседние нити блока работали с соседними ячейками памяти.

Данный механизм осуществляет любую необходимую перестановку измерений массива, а также так называемую подиагональную трансформацию, в результате которой соседние элементы на диагоналях располагаются в соседних ячейках памяти, что позволяет применять технику выполнения цикла с зависимостями по гиперплоскостям без значительной потери производительности на операциях доступа к памяти.

15 Компиляция, выполнение и отладка CDVMH-программ.

Параллельная программа представляет собой обычную последовательную программу, в которую вставлены DVMH-директивы, определяющие ее параллельное выполнение.

CDVMH-программа – это один или несколько файлов с исходными текстами на языке C-DVMH, имеющих расширение **.cdv, .c, .h.**

Для компиляции и запуска на выполнение CDVMH-программы в рабочую директорию, в которой она находится, необходимо скопировать файл запуска dvm-команд (**dvm**) из директории **dvm_sys/user** DVM-системы или выполнить команду **dvminit.**

В файле **dvm** определены переменные окружения, которые могут быть изменены пользователем. В Приложении 2 приведено описание переменных окружения для DVMH-программ.

Для поддержания работы программы как одновременно последовательной и параллельной введено макроопределение **_DVMH**.

Оно определено при конвертации DVMH-программы, а также при последующей компиляции сконвертированного текста. Значение данного макроопределения является целым числом, большим 0.

Также при конвертации и/или компиляции DVMH-программы, делается неявное включение файла

#include <dvmh_runtime_api.h>,

в котором содержатся заголовки API (см. Приложение 4).

Пример 14.1. Использование _DVMH.

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

15.1 Компиляция. Получение готовой программы.

Команда компиляции и получения готовой CDVMH-программы имеет вид:

dvm c <опции компиляции> <имя CDVMH-программы>

Если программа имеет расширение **.cdv,** то имя программы можно указывать без расширения.

Если программа состоит из нескольких файлов, то для объединения файлов можно использовать makefile или перечислить все файлы в строке компиляции:

dvm c <опции компиляции> -o <имя выполняемого файла CDVMH-программы> < список файлов >

где:

<**список файлов**> - список имен файлов программы с расширениями **.cdv, .c, .h**.

Результат работы: готовая программа (выполняемый файл **<имя CDVMH-программы**> или с именем в командной строке) в текущем каталоге. Кроме того, компилятор создает для каждой исходной программы несколько дополнительных файлов. Описание этих модулей представлено в Приложении 3.

Если компилятор обнаружил ошибки, то выполняемый файл не создается.

Для CDVMH-программ предусмотрены новые режимы компиляции, которые задаются при помощи следующих опций:

-noH	-	режим игнорирования директив, обеспечивающих выполнение программы на кластерах с графическими процессорами
-oneThread	-	режим последовательного выполнения всех циклов на ГПУ
-autoTfm	-	режим динамического переупорядочивания массива (режим перестановки измерений массива для оптимизации доступа к глобальной памяти)
-noCuda	-	управление процессом компиляции – компилятор не готовит

выполнение регионов на CUDA-устройствах.

При компиляции CDVMH-программы с опцией –noH игнорируются следующие директивы и спецификации:

- директивы задания вычислительных регионов;
- некоторые спецификации параллельных циклов внутри региона: указание размера CUDA-блока для CUDA-устройства;
- директивы управления перемещениями данных вне регионов;
- директивы задания фрагментов программы внутри региона, которые надо выполнить на центральном процессоре.

При выполнении такой программы вычисления распределяются только по процессорам. Опцию целесообразно использовать для отладки программы.

Использование опций оптимизации (–autoTfm) может способствовать повышению производительности программ (раздел 14.4).

Остальные опции предназначены для отладки DVMH-системы.

15.2 Выполнение CDVMH-программы

dvm run [N1 [N2 [N3[N4]]]] <имя выполняемого файла CDVMH-программы >

где:

N1, N2, N3, N4 – размеры матрицы виртуальных процессоров (по умолчанию – **1 1 1 1**)

При запуске CDVMH-программы размеры матрицы виртуальных процессоров определяют число процессов, которые будут выполняться параллельно (N1*N2*N3*N4).

15.3 Сравнительная отладка.

Для CDVMH-программ реализована возможность сравнительной отладки. Это специальный режим работы DVMH-программы, при котором все вычисления в регионах одновременно выполняются на ЦПУ и ГПУ.

Сравнение данных на входе в регион позволяет обнаружить отсутствие спецификации **out** в ранее выполненных регионах или директивы актуализации **actual**.

Сравнение выходных данных, полученными в регионе при выполнении на ГПУ, с данными, полученными в регионе при выполнении на ЦПУ, позволяет выявить и локализовать ошибки, проявляющиеся при работе на ускорителях.

В сравнение включаются все выходные данные вычислительного региона. При этом целочисленные данные сравниваются на совпадение, а вещественные числа сравниваются с заданной точностью по абсолютной и относительной погрешности. В случае нахождения расхождений выдается информация о найденных расхождениях. Далее в программе используется та версия данных, которая была получена при выполнении на ЦПУ.

Ошибки при выполнении региона на ускорителе могут возникать по нескольким причинам:

1. Программистом произведено некорректное распараллеливание, не подходящее для массивно-параллельного выполнения в общей памяти.
2. Программист некорректно указал приватные или редукционные переменные в параллельном цикле.
3. Арифметические операции или математические функции на ускорителе отработали с иным по сравнению с работой на ЦПУ результатом. Это может происходить из-за различий в системе команд, приводящих к различным результатам (в пределах точности округлений).
4. Программист указал неверные директивы актуализации данных **get_actual** и **actual**, вследствие чего обрабатываемые данные на ЦПУ и ускорителе оказались разными.

Включение и использование режима сравнительной отладки не требует от программиста вносить какие-либо изменения в программу, а также заново ее компилировать.

Для включения режима сравнительной отладки необходимо в файле запуска dvm-команд установить значение переменной окружения **DVMH_COMPARE_DEBUG** равным 1, либо для запуска на выполнение использовать команду:

dvm cmph [N1 [N2 [N3[N4]]]] <имя выполняемого файла CDVMH-программы >

где:

N1, N2, N3, N4 – размеры матрицы виртуальных процессоров (по умолчанию – 1 1 1 1)

В случае обнаружения ошибок информация о них выдается в стандартный поток вывода ошибок или в файл. Имя этого файла можно указать в переменнной окружения **DVMH_LOGFILE.**

Точность сравнения переменных можно изменить, указав значения переменных окружения **DVMH_COMPARE_FLOATS_EPS, DVMH_COMPARE_DOUBLES_EPS, DVMH_COMPARE_LONGDOUBLES_EPS**.

Для выполнения сравнительной отладки значение переменной окружения **DVMH_LOGLEVEL**, задающей уровень детализации, должно быть не меньше 1. В этом случае выводится информация о наличии ошибок и множество индексов с

расхождениями в компактной форме. При наличии ошибок может быть полезной информация, которая выдается при задании уровня детализации 4 или 5.

15.4 Отладка производительности.

При задании уровня детализации **DVMH_LOGLEVEL** равного 4 или больше в выходной файл выдается информация о производительности каждого вычислительного устройства для каждого параллельного цикла. Имя этого файла можно задать в переменнной окружения **DVMH_LOGFILE.** Если имя файла не задано, то информация выдается в стандартный поток вывода ошибок.

Повышение производительности программ может дать использование опций оптимизации:

-autoTfm - режим динамического переупорядочивания массива (режим перестановки измерений массива для оптимизации доступа к глобальной памяти)

Опция –autoTfm может оказаться наиболее эффективной при наличии в программе циклов с регулярными зависимостями по данным.

Выбор режима распределения данных и вычислений также может повлиять на повышение производительности программы. Выбор режима осуществляется при помощи переменных окружения:

DVMH_SCHED_TECH - режим планирования. Существуют несколько режимов. Задавать можно как числом, так и словом, при этом регистр букв не учитывается. По умолчанию значение равно 1. Можно задать следующие режимы:

0	-	single	-	режим "одно устройство". Если процессу назначен хотя бы один ускоритель, то будет использоваться только он. Если назначено более одного ускорителя, то выбирается только один из них. Если у процесса нет в распоряжении ни одного ускорителя, то выполнение отдается на ЦПУ.
1	-	static	-	режим "простой статический". Распределение данных (а точнее шаблонов - корней деревьев выравниваний) будет производиться в каждом регионе по устройствам в заданной с помощью DVMH_CPU_PERF и DVMH_CUDAS_PERF пропорции.
2	-	dynamic1	-	режим "простой динамический". Распределение данных (а точнее шаблонов - корней деревьев выравниваний) будет подбираться в процессе работы программы так, чтобы найти такие значения DVMH_CPU_PERF и DVMH_CUDAS_PERF, которые минимизируют общее время выполнения программы в простом статическом режиме при их задании.

3	-	dynamic2	-	режим "динамический с подбором". Распределение данных (а точнее шаблонов - корней деревьев выравниваний) будет подбираться в процессе работы программы так, чтобы найти веса устройств для каждого из регионов, которые минимизируют общее время выполнения программы в режиме с использованием схемы распределения.
4	-	scheme	-	статический режим с использованием схемы, полученной в динамическом режиме с подбором. Используется файл, задающий веса и значения оптимизационных параметров для каждого региона и параллельного цикла.

DVMH_SCHED_FILE - имя файла с сохраненной схемой распределения для режима планирования 4. По умолчанию используется Scheme.dvmh.

На эффективность выполнения программы в определенном режиме может оказывать влияние также задание следующих переменных окружения:

DVMH_CPU_PERF - закольцованный список неотрицательных вещественных чисел, задающий условную производительность ЦПУ-устройства (всех рабочих ЦПУ-нитей вместе) в каждом из процессов. Индексируется номером процесса. По умолчанию значение равно 1.

DVMH_CUDAS_PERF - закольцованный список неотрицательных вещественных чисел, задающий условную производительность CUDA-устройств узла. Индексируется номером CUDA-устройства по нумерации CUDA Runtime на каждом узле независимо. По умолчанию значение равно 1.

DVMH_CUDA_PREFER_SHARED - признак задания для запуска ядер режима предпочтения разделяемой памяти. Задается булевым значением. По умолчанию значение равно 0.

Значения переменных окружения **DVMH_NUM_CUDAS** и **DVMH_NUM_THREADS** могут тоже оказать влияние на эффективность выполнения.

Литература.

1. DVM-система [Электронный ресурс] - : [web-сайт] - Режим доступа: http://dvm-system.org/ - 28.05.2016
2. Н.А.Коновалов, В.А.Крюков, Ю.Л.Сазанов. C-DVM - язык разработки мобильных параллельных программ. "Программирование", № 1, 1999, стр 54-65
3. В.А. Бахтин, М.С. Клинов, В.А. Крюков, Н.В. Поддерюгина, М.Н. Притула, Ю.Л. Сазанов. Расширение DVM-модели параллельного программирования для кластеров с гетерогенными узлами // Вестник Южно-Уральского университета, Серия "Математическое

моделирование и программирование", Челябинск: Издательский центр ЮУрГУ, 2012, № №18 (277), Выпуск 12, С. 82-92

16 Примеры программ.

Небольшие программы из научной области приводятся для иллюстрации свойств языка C-DVMH. Они предназначены для решения систем линейных уравнений:

$$\mathbf{A\,x = b}$$

где
A – матрица коэффициентов,
b – вектор свободных членов,
x – вектор неизвестных.

Для решения этой системы используются следующие основные методы.

Прямые методы. Хорошо известный метод исключения Гаусса является наиболее широко используемым алгоритмом этого класса. Основная идея алгоритма заключается в преобразовании матрицы *A* в верхнетреугольную матрицу и использовании затем обратной подстановки, чтобы привести ее к диагональной форме.

Явные итерационные методы. Наиболее известным алгоритмом этого класса является метод релаксации Якоби. Алгоритм выполняет следующие итерационные вычисления

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

Неявные итерационные методы. К этому классу относится метод последовательной верхней релаксации. Итерационное приближение вычисляется по формуле

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

При использовании "красно-черного" упорядочивания переменных каждый итерационный шаг разделяется на два полушага Якоби. На одном полушаге вычисляются значения "красных" переменных, на другом – "черных" переменных. "Красно-черное" упорядочивание позволяет совместить вычисления и обмены данными.

16.1 Пример 1. Алгоритм Якоби.

```
/* 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;
}
```

В результате выполнения директивы

#pragma dvm array distribute[block][block]

массив A будет распределен между вычислителями. Количество и тип используемых вычислителей задается при запуске программы с помощью переменных окружения и параметров командной строки.

Директива

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

задает совместное распределение двух массивов A и B. Элементы массива B будут распределены на тот же вычислитель, где будут размещены соответствующие элементы массива A.

Директива

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

задает распределение вычислений. Витки цикла будут выполняться на том вычислителе, где распределены соответствующие элементы массива A.

Спецификация **reduction(max(eps))** организует эффективное выполнение редукционной операции - глобальной операции с расположенными на различных вычислителях данных (нахождение максимального значения).

Спецификация **shadow_renew(A)** указывает на необходимость подкачки удаленных данных (теневых граней) с других вычислителей перед выполнением цикла.

Спецификация **cuda_block** указывает размер блока нитей CUDA-устройства.

Поскольку никакие дополнительные спецификации в директивах **region** не заданы, компилятор определяет направления использования переменных автоматически - **inout**(A,B,eps).

При выполнении первого вычислительного региона (цикла инициализации) для распределенных частей массивов A и B на ускорителях будет выделена необходимая память.

При входе во второй вычислительный регион (в итерационном цикле) осуществляется проверка, присутствуют ли актуальные представители для массивов A и B на вычислителе. Поскольку такие представители уже присутствуют, то никакие дополнительные операции копирования актуальных данных на вычислители не выполняются.

При выходе из вычислительного региона обновление данных в памяти хоста не производится. Перед тем, как сравнивать eps и MAXEPS, необходимо выполнить директиву **get_actual(eps),** а перед выводом массива B в файл, требуется скопировать последние изменения массива из памяти вычислителя при помощи директивы **get_actual(B)**.

16.2 Пример 2. Алгоритм метода исключения Гаусса.

Коэффициенты матрицы A хранятся в секции массива A[0:N-1][0:N-1], а вектор B – в секции A[0:N-1][N] того же массива.

```
/* 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 Пример 3. "Красно-черная" последовательная верхняя релаксация.

Точки обсчитываются в шахматном порядке: сначала «красные», потом "черные".

```
/* 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;
}
```

Приложение 1. Синтаксис директив C-DVMH.

Синтаксис C-DVMH-директив описывается расширенной БНФ. Используются следующие обозначения:

::=	есть по определению,
x | *y*	альтернативы,
[ ]	необязательная конструкция,
[ ]…	повторение конструкции 0 или более раз,
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* **)**

Приложение 2. Переменные окружения для CDVMH-программ.

Все переменные окружения, введенные в DVM-системе, имеют префикс DVMH_.

В файле запуска dvm-команд определены переменные окружения, которые могут быть изменены пользователем.

Все переменные с префиксом DVMH_ рассылаются средствами RTS на все процессы с нулевого, при этом приоритет имеет уже заданное процессу значение переменной.

Формат значения каждой переменной зависит от типа параметра, ею задаваемой:

Для строковых параметров берется неизмененное содержание переменной.

Для числовых параметров значение переменной передается в функцию sscanf для конвертации к соответствующему типу. В качестве десятичного разделителя нужно использовать точку.

Для булевых параметров значение переменной может быть (регистр букв игнорируется):

1, on, yes, enable – истина;

0, off, no, disable – ложь.

Для параметров-перечислений как правило допустимы как числовые значения, так и словесные. Конкретные случаи разобраны отдельно ниже.

Для параметров-списков их элементы отделяются запятыми или пробелами или их любой комбинацией как удобно пользователю, при этом пустыми элементы списка быть не могут (в частности, несколько подряд пробелов эквивалентны одному).

DVMH_PPN - количество процессов на узел (положительное целое число или список неотрицательных целых чисел). Эта переменная отличается от всех остальных тем, что используется только драйвером DVM-системы. Если на вычислительном кластере имеются узлы разного типа, то значение может быть задано списком целых чисел. По умолчанию значение равно 1.

DVMH_LOGLEVEL - уровень детализации журнала. Задавать можно как числом, так и словом, при этом регистр букв не учитывается. По умолчанию значение равно 1. Можно задать следующие уровни:

0	-	fatal	-	выдаются только фатальные ошибки,
1	-	error	-	ошибки, не являющиеся причиной аварийного завершения
2	-	warning	-	предупреждения. Обычно указывают на потенциальные ошибки, допущенные при распараллеливании.
3	-	info	-	информационные сообщения. Выводится некоторая справочная информация о версии DVM-системы, способе запуска программы, используемых вычислительных устройствах.
4	-	debug	-	отладочные сообщения. Информация для разработчиков DVM-системы.
5	-	trace	-	трассировочные сообщения. Информация для разработчиков DVM-системы. Возможен очень большой объем выводимой информации.

DVMH_FLUSHLOGLEVEL - уровень сообщений, выдача которых приводит к сбросу буфера ввода/вывода из памяти пользовательского процесса. Задается в том же формате, что и DVMH_LOGLEVEL. По умолчанию значение равно 0.

DVMH_LOGFILE - имя файла для выдачи журнала. Значение переменной строковое. При задании имени файла можно использовать конструкцию %d (например,'dvmh_%d.log'), в этом случае журнал каждого процесса будет в отдельном файле. Если переменная не задана или имя файла не задано, то используется стандартный поток вывода ошибок.

DVMH_FATAL_TO_STDERR - булево значение, указывающее нужно ли дублировать фатальные сообщения в стандартный поток вывода ошибок, если для журнала был задан файл. По умолчанию значение равно 1.

DVMH_NUM_THREADS - закольцованный список неотрицательных целых чисел, задающий количество рабочих ЦПУ-нитей в каждом из процессов. Индексируется номером процесса. Если переменная не задана, то ее оптимальное значение определяет система поддержки. Система поддержки обеспечивает эффективное использование всех ресурсов узла. Оптимальное значение зависит от количества устройств на узле, количества запущенных процессов на узле и количества CUDA-устройств для использования одним процессом (**DVMH_NUM_CUDAS)**.

DVMH_NUM_CUDAS - закольцованный список неотрицательных целых чисел, задающий количество виртуальных CUDA-ускорителей в каждом из процессов. Индексируется номером процесса. Если переменная не задана, то ее оптимальное значение определяет система поддержки. Система поддержки обеспечивает эффективное использование всех ресурсов узла. Оптимальное значение зависит от количества устройств на узле и количества запущенных процессов на узле.

DVMH_CPU_PERF - закольцованный список неотрицательных вещественных чисел, задающий условную производительность ЦПУ-устройства (всех рабочих ЦПУ-нитей вместе) в каждом из процессов. Индексируется номером процесса. По умолчанию значение равно 1.

DVMH_CUDAS_PERF - закольцованный список неотрицательных вещественных чисел, задающий условную производительность CUDA-устройств узла. Индексируется номером CUDA-устройства по нумерации CUDA Runtime на каждом узле независимо. По умолчанию значение равно 1.

DVMH_CUDA_PREFER_SHARED - признак задания для запуска ядер режима предпочтения разделяемой памяти. Задается булевым значением.

DVMH_SCHED_TECH - режим планирования. Существуют несколько режимов. Задавать можно как числом, так и словом, при этом регистр букв не учитывается. По умолчанию значение равно 1. Можно задать следующие режимы:

0	-	single	-	режим "одно устройство". Если процессу назначен хотя бы один ускоритель, то будет использоваться только он. Если назначено более одного ускорителя, то выбирается только один из них. Если у процесса нет в распоряжении ни одного ускорителя, то выполнение отдается на ЦПУ.
1	-	static	-	режим "простой статический". Распределение данных (а точнее шаблонов - корней деревьев выравниваний) будет производиться в каждом регионе по устройствам в заданной с помощью DVMH_CPU_PERF и DVMH_CUDAS_PERF пропорции.
2	-	dynamic1	-	режим "простой динамический". Распределение данных (а точнее шаблонов - корней деревьев выравниваний) будет подбираться в процессе работы программы так, чтобы найти такие значения DVMH_CPU_PERF и DVMH_CUDAS_PERF, которые минимизируют общее время выполнения программы в простом статическом режиме при их задании.
3	-	dynamic2	-	режим "динамический с подбором". Распределение данных (а точнее шаблонов - корней деревьев выравниваний) будет подбираться в процессе работы программы так, чтобы найти веса устройств для каждого из регионов, которые минимизируют общее время выполнения программы в режиме с использованием схемы распределения.
4	-	scheme	-	статический режим с использованием схемы, полученной в динамическом режиме с подбором. Используется файл, задающий веса и значения оптимизационных параметров для каждого региона и параллельного цикла.

DVMH_SCHED_FILE - имя файла с сохраненной схемой распределения для режима планирования 4. По умолчанию используется Scheme.dvmh.

DVMH_REDUCE_DEPS - булево значение, указывающее считать ли измерения параллельного цикла длины 1 измерениями с зависимостями, если измерения являются зависимыми исходя из указаний ACROSS. По умолчанию значение равно 0 (не считать).

DVMH_ALLOW_ASYNC - признак асинхронного выполнения регионов (регионов со спецификацией async). Задается булевым значением. По умолчанию значение равно 0. (Пока не реализовано.)

DVMH_COMPARE_DEBUG - признак включения режима сравнительной отладки выполнения программы на ускорителях. Более удобным способом запуска сравнительной отладки может служить команда cmph драйвера DVM-системы. Задается булевым значением. По умолчанию значение равно 0.

DVMH_COMPARE_FLOATS_EPS - неотрицательное вещественное число, задающее точность сравнения переменных с плавающей точкой одинарной точности по относительной и абсолютной погрешности при сравнительной отладке. Нуль задает требование полного совпадения. По умолчанию равна FLT_EPSILON*1000, где FLT_EPSILON – минимальное положительное x такое, что 1.0+x=1.0.

DVMH_COMPARE_DOUBLES_EPS - неотрицательное вещественное число, задающее точность сравнения переменных с плавающей точкой двойной точности по относительной и абсолютной погрешности при сравнительной отладке. Нуль задает требование полного совпадения. По умолчанию равна DBL_EPSILON*10000, где DBL_EPSILON – минимальное положительное x такое, что 1.0+x=1.0.

DVMH_COMPARE_LONGDOUBLES_EPS - неотрицательное вещественное число, задающее точность сравнения переменных с плавающей точкой длинной двойной точности по относительной и абсолютной погрешности при сравнительной отладке. Нуль задает требование полного совпадения. По умолчанию равна LDBL_EPSILON*100000, где LDBL_EPSILON – минимальное положительное x такое, что 1.0+x=1.0.

DVMH_SYNC_CUDA - признак синхронизации с CUDA-устройством после каждого запуска ядра. При включении этого параметра улучшается локализация ошибки выполнения, возникающей в CUDA-ядре, а при отключении меньше времени тратится на накладные расходы. Используется для поиска ошибок в системе. Задается булевым значением. По умолчанию значение равно 0.

DVMH_FORTRAN_NOTATION - признак использования Фортран-нотации при выведении в журнал секций массивов (например, после нахождения расхождений при сравнительной отладке). Задается булевым значением. Значение по-умолчанию зависит от того, на каком языке написана главная программная единица (PROGRAM в Фортране или main в C/C++).

DVMH_CACHE_CUDA_ALLOC - признак кеширования выделения и освобождения памяти на CUDA-устройстве. Кеширование может заметно ускорить работу программы, тогда как отказ от него сделает поведение программы более предсказуемым. Задается булевым значением. По умолчанию значение равно 1.

DVMH_USE_GENBLOCK - признак безусловного использования метода распределения GENBLOCK при выполнении распределения средствами системы поддержки. Задается булевым значением. По умолчанию значение равно 0.

DVMH_SET_AFFINITY - признак привязки нитей к процессорам средствами системы поддержки. Задается булевым значением. По умолчанию значение равно 1.

DVMH_OPT_PARAMS - признак выполнения подбора оптимизационных параметров, таких как количество нитей, используемых для каждого параллельного цикла или размер CUDA-блока нитей, используемого для каждого параллельного цикла. Задание этого признака приведет к варьированию оптимизационных параметров во время выполнения программы, а также к выдаче результатов подбора в файл схемы распределения (см. DVMH_SCHED_FILE), допустимо использовать независимо от режима распределения. Задается булевым значением. По умолчанию значение равно 0.

DVMH_NO_DIRECT_COPY - признак запрета использования прямой адресации ЦПУ-памяти из ГПУ с целью произведения актуализации данных. Стоит отметить, что отсутствие запрета само по себе не означает использование прямой адресации. Нужна поддержка со стороны аппаратуры, ОС, а также соответствие более строгим требованиям на форму и тип данных массивов. Задается булевым значением. По умолчанию значение равно 0.

DVMH_SPECIALIZE_RTC – признак разрешения специализации параметров при использовании runtime-компиляции CUDA-ядер. Задается булевым значением. По умолчанию значение равно 1.

DVMH_PREFER_CALL_SWITCH - признак вызова обработчика через переключатель по количеству параметров, если это возможно. В РТС предусмотрено 2 способа вызова обработчиков - через переключатель, имеющее ограничение на количество аргументов (на момент 13.05.2016 - 256 штук) и с использованием библиотеки libffi, которая такого ограничения не имеет, но доступна не на всех платформах. Задается булевым значением. По умолчанию значение равно 1.

DVMH_NUM_VARIANTS_FOR_VAR_RTC - положительное целое число, задающее максимальное количество вариантов специализации переменной перед отказом от ее специализации. По умолчанию значение равно 3.

DVMH_PARALLEL_IO_THRES - положительное целое число, задающее минимальный размер выводимых или вводимых данных (в байтах), который РТС будет пытаться вводить или выводить параллельно (только если все принципиальные ограничения соблюдены). По умолчанию значение равно 104857600, т.е. 100 Мбайт.

DVMH_IO_BUF_SIZE - положительное целое число, задающее максимальный размер буфера ввода/вывода при выполнении его через процессор ввода/вывода. По умолчанию значение равно 104857600, т.е. 100 Мбайт.

DVMH_IO_THREAD_COUNT - неотрицательное целое число, задающее кол-во нитей ввода/вывода, выполняющих асинхронные операции ввода/вывода. Указание нуля приведет к отключению асинхронного ввода/вывода. По умолчанию значение равно 5.

Приложение 3. Описание выходных файлов компилятора.

В результате компиляции DVMH-программы помимо файла, содержащего готовую CDVMH-программу, образуется еще несколько файлов. Рассмотрим их на примере программы prog.cdv.

1. Файл prog.DVMH.c - файл, содержащий код для всего внерегионного пространства, хост-секций, управляющий код для организации параллельного выполнения и распределения данных. Также в него помещаются обработчики параллельных циклов и последовательных участков регионов, предназначенные для выполнения на ЦПУ (мультипроцессоре). Язык - стандартный СИ + OpenMP (поддержка OpenMP от компилятора не требуется).
2. Файл prog.DVMH_cuda.cu - файл, содержащий обработчики параллельных циклов и последовательных участков региона, предназначенные для выполнения на ГПУ фирмы NVIDIA с использованием технологии CUDA. В нем же размещаются CUDA-ядра для этих обработчиков. Язык - CUDA C++.

Приложение 4. Пользовательское API.

1. Барьерная синхронизация в рамках текущей многопроцессорной системы.

   void dvmh_barrier();

2. Количество процессоров всего в текущей многопроцессорной системе.

   int dvmh_get_total_num_procs();

3. Эффективное количество измерений в текущей многопроцессорной системе.

   int dvmh_get_num_proc_axes();

4. Количество процессоров в текущей многопроцессорной системе на оси axis (нумерация с единицы).

   int dvmh_get_num_procs(int axis);

5. Запрос времени по настенным часам, без синхронизаций.

   double dvmh_wtime();

6. Функции локального ввода/вывода, не связанные с файловыми дескрипторами (для функций, связанных с файловыми дескрипторами, см. замечание про функции **fopen** и **freopen** в разделе 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);