Российская академия наук
Институт прикладной математики им. М.В. Келдыша РАН

На правах рукописи

Притула Михаил Николаевич

ОТОБРАЖЕНИЕ DVMH-ПРОГРАММ НА КЛАСТЕРЫ С ГРАФИЧЕСКИМИ ПРОЦЕССОРАМИ

Специальность 05.13.11 – математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени
кандидата физико-математических наук

Научный руководитель –
доктор физико-математических наук
Крюков Виктор Алексеевич

Москва – 2013

Оглавление

Введение

Актуальность темы

В настоящее время большое количество параллельных программ для кластеров разрабатываются с использованием низкоуровневых средств передачи сообщений (MPI [1]). MPI-программы трудно разрабатывать, сопровождать и повторно использовать при создании новых программ. Данная проблема осложняется тем, что в последнее время появляется много вычислительных кластеров с установленными в их узлах ускорителями. В основном, это графические процессоры. Программисту требуется теперь освоение на достаточном уровне сразу нескольких моделей и языков программирования. Традиционным подходом можно назвать использование технологии MPI для разделения работы между узлами кластера, а затем технологий OpenMP [2] и CUDA [3] или OpenCL [4] для загрузки всех ядер центрального и графического процессоров.

Поэтому прикладной программист хотел бы получить инструмент, автоматически преобразующий его последовательную программу в параллельную программу для кластера с ускорителями, либо высокоуровневый язык параллельного программирования, обеспечивающий эффективное использование современных параллельных систем. Проведенные в 90-х годах активные исследования убедительно показали, что полностью автоматическое распараллеливание для кластерных ЭВМ реальных производственных программ возможно только в очень редких случаях.

Сначала очень высокие требования к эффективности выполнения параллельных программ, а затем и стремительные изменения в архитектуре параллельных ЭВМ привели к тому, что до настоящего времени так и нет общепризнанного высокоуровневого языка параллельного программирования, позволяющего эффективно использовать возможности

современных ЭВМ, а также в достаточной степени абстрагироваться от конкретной конфигурации вычислительной системы с целью обеспечения эффективной переносимости программы.

Цели работы

Целями данной диссертационной работы являлись:

- исследование возможностей отображения DVM-программ [5] на кластер с ускорителями, обеспечивающих распределение вычислений между универсальными многоядерными процессорами (ЦПУ) и несколькими графическими процессорами (ГПУ);
- разработка алгоритмов распределения вычислений между ЦПУ и ГПУ и алгоритмов управления перемещением данных между памятью ЦПУ и памятью ГПУ;
- разработка алгоритма распределения подзадач между узлами кластера.

Постановка задачи

Необходимо предложить принципы и алгоритмы отображения DVM-программ на кластер с ускорителями, позволящие использовать DVM-языки с минимальным их расширением. При этом исходить из того, что в узлах кластера будут размещаться ускорители разной архитектуры.

Так как целевой машиной выбран кластер с многоядерными процессорами и ускорителями, то ставится задача разработать программные средства, обеспечивающие использование для вычислений одновременно как ЦПУ, так и ГПУ. В используемых низкоуровневых средствах программирования для ГПУ (и в новых высокоуровневых средствах тоже) есть сложности для прикладного программиста, связанные с копированием данных на ускоритель и обратно, поэтому ставится задача по возможности автоматизировать данный процесс в разрабатываемом расширении модели DVM.

Необходимо разработать и внедрить в систему поддержки выполнения DVM-программ алгоритм распределения подзадач между узлами кластера, обеспечивающий балансировку загрузки вычислительных ресурсов. Алгоритм должен исходить из наличия заданного времени выполнения каждой подзадачи на различном числе процессоров.

Разработать средства отладки, позволяющие находить ошибки программиста, допущенные при распараллеливании программы на кластер с ускорителями, а также корректность вычислений, произведенных на графическом процессоре.

Научная новизна работы

1. Разработаны алгоритмы распределения подзадач между узлами кластера, позволяющие балансировать загрузку вычислительных узлов кластера при выполнении многоблочных программ.
2. Разработаны алгоритмы распределения витков параллельных циклов внутри узлов между ядрами ЦПУ и несколькими ГПУ.
3. Разработаны алгоритмы автоматического перемещения требуемых актуальных данных между памятью ЦПУ и памятями нескольких ГПУ.
4. Созданы средства сравнительной отладки DVMH-программ, базирующиеся на сопоставлении результатов одновременного выполнения на ЦПУ и на ГПУ одних и тех же фрагментов программы.

Проведенные эксперименты с тестами и реальными приложениями показали, что для класса задач, при решении которых используются разностные методы на статических структурных сетках, можно писать программы на языке Фортран-DVMH, которые эффективно выполняются на кластерах с ускорителями.

Практическая значимость

С использованием разработанных алгоритмов создана система поддержки выполнения DVMH-программ, являющаяся неотъемлемой частью компиляторов DVMH-программ. Разработка компилятора с языка Фортран-DVMH существенно упростила создание эффективных программ для кластеров с ускорителями, способных автоматически настраиваться на целевую конфигурацию вычислительной системы. Компилятор с языка Фортран-DVMH, включающий в себя систему поддержки выполнения DVMH-программ, входит в состав DVM-системы и используется на факультете ВМК МГУ при проведении практикума по технологиям параллельного программирования. С использованием этого компилятора был распараллелен на кластер с ускорителями ряд прикладных вычислительных задач.

Апробация работы и публикации

Основные результаты диссертации были доложены на российских и международных научных конференциях и семинарах:

1. Международной научной конференции “Научный сервис в сети Интернет: суперкомпьютерные центры и задачи”, сентябрь 2010 г., г. Новороссийск.
2. Международной научной конференции “Научный сервис в сети Интернет: экзафлопсное будущее”, сентябрь 2011 г., г. Новороссийск.
3. XIII международном семинаре “Супервычисления и математическое моделирование”, октябрь 2011 г., г. Саров.
4. International Research Conference on Information Technology. Seventh International PhD&DLA Symposium, октябрь 2011 г., г. Pecs, Hungary.
5. Международной суперкомпьютерной конференции “Научный сервис в сети Интернет: поиск новых решений”, сентябрь 2012 г., г. Новороссийск.

6. APOS/HOPSA Workshop on Exploiting Heterogeneous HPC Platforms, январь 2013 г., г. Берлин.
7. Международной конференции "Параллельные вычислительные технологии (ПаВТ'2013)", апрель 2013 г., г. Челябинск.
8. Международной суперкомпьютерной конференции “Научный сервис в сети Интернет: все грани параллелизма”, сентябрь 2013 г., г. Новороссийск.

Имеется 12 публикаций [6-17], из которых три [10,12,16] – в журналах из списка ВАК.

Краткое содержание работы

В первой главе приводится обзор языков и технологий программирования для кластеров и графических процессоров.

Вторая глава посвящена описанию схемы построения компилятора с языка Фортран-DVMH, функций системы поддержки выполнения DVMH-программ.

В третьей главе описывается набор алгоритмов учета актуального состояния переменных. Эти алгоритмы описывают механизмы обработки ситуаций в программе, потенциально приводящих к перемещениям данных.

Четвертая глава посвящена описанию режимов распределения данных и вычислений между вычислительными устройствами.

В пятой главе описывается алгоритм распределения подзадач между узлами кластера.

Шестая глава содержит описание дополнительных возможностей по функциональной отладке и отладке производительности.

Седьмая глава содержит описание приложений и тестов, разработанных с использованием языка Фортран-DVMH.

Автор выражает благодарность всем разработчикам DVM-системы.

Работа выполнена под руководством доктора физико-математических наук, заведующего отделом Института прикладной математики им. М.В. Келдыша Крюкова Виктора Алексеевича, которому автор выражает искреннюю признательность.

Глава 1. Обзор языков и технологий программирования для кластеров и графических процессоров

Появление многопроцессорных ЭВМ с распределенной памятью значительно расширило возможности решения больших задач, однако резко усложнило разработку программ. Программист должен распределить данные между процессорами, а программу представить в виде совокупности процессов, взаимодействующих между собой посредством обмена сообщениями.

В настоящее время большинство параллельных программ для кластеров разрабатывается с использованием низкоуровневой модели передачи сообщений MPI [1], при использовании которой программист пишет программу, выполняющуюся параллельными процессами. Для разделения работы между ними, для передачи информации от одного процесса другому, а также для синхронизации параллельных процессов предоставляются библиотечные вызовы.

Также используется модель односторонних коммуникаций, представителем которой является библиотека SHMEM [18].

Имеются и основанные на директивах высокоуровневые расширения стандартных языков программирования для программирования кластеров, языки Фортран-DVM [5, 19] и Си-DVM [5, 20]. Подобные расширения не являются широко используемыми, однако, представляют интерес как средство упрощения программирования кластеров.

С развитием графических процессоров стало возможным использовать их и для вычислений общего назначения, а в последние годы их стали устанавливать в суперкомпьютеры для ускорения вычислений.

Для программирования графических процессоров производители аппаратного обеспечения предложили две довольно низкоуровневые технологии – CUDA и OpenCL. Относительная сложность и необычность (в сравнении с работой на ЦПУ) использования графических процессоров для вычислений общего назначения удерживает их от повсеместного использования.

В то же время успех OpenMP, предназначенного для систем с общей памятью, давал прикладным программистам надежду на появление подобных средств для использования графических процессоров, а исследовательским группам – направление для разработок.

Однако распространить стандарт OpenMP на графические процессоры не получалось по следующим причинам:

Во-первых, наличие отдельной памяти графического процессора и необходимости управлять перемещениями данных на ускоритель и с ускорителя, что еще усложняется при подключении сразу нескольких ускорителей к одной ЭВМ.

Во-вторых, более сложная, иерархическая модель параллелизма; наличие нескольких принципиально различных видов памяти на ускорителе.

В-третьих, ограниченные аппаратные возможности синхронизации и использования готовых библиотек с ускорителя.

1.1 Разработанное расширение DVM-модели

В рамках данного исследования было предложено расширение DVM-модели, позволяющее разрабатывать программы для кластеров с графическими процессорами. Эта расширенная модель названа DVMH (DVM for Heterogeneous systems).

Модель DVMH является расширением модели DVM конструкциями, предназначенными для двух задач:

1. Организация вычислений, спецификации потоков данных.

2. Управление отгружаемыми данными, актуальностью.

Далее вид всех директив приводится для языка Фортран-DVMH.

1.1.1 Организация вычислений, спецификации потоков данных

Вычислительный регион выделяет фрагмент программы с одним входом и одним выходом для возможного выполнения на одном или нескольких вычислителях. Регион может быть исполнен на одном или сразу нескольких ускорителях и/или на многоядерном процессоре, при этом на ЦПУ может быть исполнен любой регион, а на возможность использования каждого типа ускорителей накладываются свои дополнительные ограничения на содержание региона, при этом содержание хост-секции (специального вида секции региона, которая всегда исполняется на ЦПУ) не влияет на определение возможности выполнения региона на том или ином устройстве.

Например, с использованием CUDA-устройства может быть исполнен любой регион без использования операций ввода/вывода, вызовов внешних процедур, рекурсивных вызовов.

Для управления тем, на каких вычислителях регион может исполняться можно использовать указание **targets** (см. ниже).

Регион описывается следующим образом:

!DVM$ REGION [*clause* {**,** *clause*}]

region_inner

!DVM$ END REGION

Вложенные (статически или динамически) регионы не допускаются.

region_inner – это нуль или более следующих друг за другом конструкций, каждая из которых является одним из нижеследующего:

- Параллельный DVM-цикл. Пространство витков параллельного цикла разделяется в соответствии с заданным в директиве

параллельного цикла правилом отображения между вычислителями, выбранными для данного региона.

- Последовательная группа операторов. Каждый оператор последовательной группы операторов исполняется на всех вычислителях, выбранных для исполнения региона, кроме случая модификации в нем распределенных данных – тогда действует правило собственных вычислений.
- Хост-секция. Хост-секция ограничивается специальными директивами:

  !DVM$ HOSTSECTION

  hostsection_inner

  !DVM$ END HOSTSECTION

  Объявляет специального вида секцию исполнения на ЦПУ.

  hostsection_inner – это часть программы с одним входом и одним выходом, эта часть будет исполняться на ЦПУ в последовательном режиме. Разрешено использование внутри таких секций директив **get_actual**. Директивы **actual** запрещены. Всякие изменения переменных в этой секции могут пропасть. Такие секции предлагается использовать в отладочных целях для промежуточного контроля значений переменных по ходу исполнения региона. Операции вывода разрешены, вызовы внешних процедур разрешены.

В качестве *clause* для вычислительных регионов может быть задано:

1. **in**(*subarray_or_scalar* {**,** *subarray_or_scalar*}),
   out(*subarray_or_scalar* {, *subarray_or_scalar*}),
   inout(*subarray_or_scalar* {**,** *subarray_or_scalar*}),
   local(*subarray_or_scalar* {**,** *subarray_or_scalar*}),
   inlocal(*subarray_or_scalar* {**,** *subarray_or_scalar*})

Указание направления использования подмассивов и скаляров в регионе. **in** – по входу в регион нужны актуальные данные. **out** – в регионе значение переменной изменяется, причем это изменение может быть использовано далее. **inout(***a***)** – сокращенная запись одновременно двух указаний: **in(***a***)**, **out(***a***).** **local** — в регионе значение переменной изменяется, но это изменение не будет использовано далее. **inlocal(***a***)** — сокращенная запись одновременно двух указаний: **in(***a***)**, **local(***a***)**. Если для переменной указано **in**, и не указано **out** или **local**, то считается, что в такую переменную в регионе вообще нет записей и она не меняется в процессе его исполнения.

2. **targets(***target_name* {**,** *target_name*}**)**

   где *target_name* это **CUDA** или **HOST**

   Указание списка типов вычислителей, на которых предполагается исполнять регион. Такое указание может быть только одно в директиве региона. 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.

3. **async**

   Указание возможности асинхронного исполнения региона. При запуске региона в любом режиме (синхронный, асинхронный) ожидание завершения ранее запущенного региона возникает,

если указаниями **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).

Указание всех используемых переменных в регионе не обязательно. При этом используемые, но не указанные в директиве региона переменные включаются в регион в автоматическом режиме компилятором по правилам:

1. Все используемые массивы считаются используемыми полностью (не выделяются подмассивы).
2. Всякая переменная, которая используется на чтение получает атрибут **in**.
3. Всякая переменная, которая используется на запись получает атрибут **inout**.
4. Всякая используемая в регионе переменная, направление использования которой не поддается определению, получает атрибут **inout**.
5. Указания **local** и **out** в автоматическом режиме не проставляются.

1.1.2 Управление отгружаемыми данными, актуальностью

Перемещения данных между вычислительными регионами осуществляется в соответствии с указаниями в директиве региона. Управление перемещениями данных вне регионов осуществляется с помощью следующих исполняемых директив:

1. **get_actual**[**(***subarray_or_scalar* {**,** *subarray_or_scalar*}**)**]

   Делает все необходимые обновления для того, чтобы в основной памяти (памяти ЦПУ) были самые новые данные в указанном подмассиве или скаляре (при этом, возможно, ничего и не следует перемещать). В случае отсутствия у директивы параметров все имеющиеся новые данные с ускорителей переписываются в основную память.

2. **actual**[**(***subarray_or_scalar* {**,** *subarray_or_scalar*}**)**]

   Объявляет тот факт, что указанный подмассив или скаляр самую новую версию имеет в основной памяти (памяти ЦПУ). При этом указанные переменные или элементы массивов, находящиеся в памяти ускорителей считаются устаревшими и перед использованием будут по необходимости обновлены. В случае отсутствия параметров все переменные в памяти ускорителей объявляются устаревшими.

Данные директивы служат для организации взаимодействия покрытой регионами части программы с остальной частью программы. Можно выделить две основных сферы применения для них:

1. Использование в процессе инкрементального распараллеливания.
2. Использование для поддержки работы с библиотеками внешних функций, в том числе библиотек ввода/вывода.

1.1.3 Пример программы на языке Фортран-DVMH

Рассмотрим программу для решения уравнения теплопроводности методом Якоби на языке Фортран-DVMH.

Пример программы на языке Фортран-DVMH

```
      PROGRAM JACH
      PARAMETER (L=8, ITMAX=20)
```

```
      REAL A(L,L), EPS, MAXEPS, B(L,L)
CDVM$ DISTRIBUTE (BLOCK, BLOCK) :: A
CDVM$ ALIGN B(I,J) WITH A(I,J)
      MAXEPS = 0.5E-7
CDVM$ REGION
CDVM$ PARALLEL(J,I) ON A(I, J)
      DO 1 J = 1, L
      DO 1 I = 1, L
        A(I, J) = 0.
       IF(I.EQ.1.OR.J.EQ.1.OR.I.EQ.L.OR.J.EQ.L)THEN
          B(I, J) = 0.
        ELSE
          B(I, J) = (1. + I + J)
        ENDIF
    1 CONTINUE
CDVM$ END REGION
      DO 2 IT = 1, ITMAX
        EPS = 0.
CDVM$   ACTUAL(EPS)
CDVM$   REGION
CDVM$   PARALLEL(J,I)ON A(I,J), REDUCTION(MAX(EPS))
        DO 21 J = 2, L-1
        DO 21 I = 2, L-1
          EPS = MAX(EPS, ABS(B(I, J) - A(I, J)))
```

```
            A(I, J) = B(I, J)
      21    CONTINUE
   CDVM$    PARALLEL(J, I) ON B(I, J), SHADOW_RENEW(A)
            DO 22 J = 2, L-1
            DO 22 I = 2, L-1
              B(I, J) = (A(I-1,J)+A(I,J-1)+A(I+1,J)+
        *                          A(I, J+1)) / 4
      22    CONTINUE
   CDVM$    END REGION
   CDVM$    GET_ACTUAL(EPS)
            PRINT 200,  IT, EPS
   200      FORMAT(' IT = ',I4, '   EPS = ', E14.7)
            IF (EPS .LT. MAXEPS) GOTO 3
       2 CONTINUE
       3 CONTINUE
   CDVM$ GET_ACTUAL(B)
         PRINT *, B
         END
```

Программа на языке Фортран-DVMH представляет из себя программу на языке Фортран, дополненную директивами языка Фортран-DVMH.

Директивы **DISTRIBUTE** и **ALIGN** определяют распределение данных, в данном случае равномерное блочное распределение, которое задается для массива **A**, а для массива **B** задается его выравнивание на массив A, что означает распределение массива **B** так же, как и массива **A**.

Параллельные циклы оформляются с помощью директивы **PARALLEL**, в которой также указываются редукционные операции, операции обновления теневых граней (теневые грани – это расширения локальной части массива для организации доступа к расположенным на соседних вычислительных устройствах элементам этого массива). Витки параллельных циклов разделяются между вычислительными устройствами в соответствии с правилом отображения витков, заданным в директиве **PARALLEL**.

Параллельные циклы расположены в вычислительных регионах (директивы **REGION** и **END REGION**), что приводит к подготовке компилятором выполнения этих циклов с использованием графических процессоров.

В показанном примере используются директивы актуализации (**GET_ACTUAL**) и объявления актуальности (**ACTUAL**) для сопряжения регионов с внерегионным пространством. Актуализация использована для переменных **EPS** и **B** перед выводом и использованием во внерегионном пространстве, а объявление актуальности использовано для переменной **EPS** после ее модификации во внерегионном пространстве.

Эта программа может быть выполнена на кластере с графическими процессорами, причем между ускорителями и многоядерными центральными процессорами будут распределены данные и вычисления.

1.2 Обзор высокоуровневых моделей программирования для ускорителей

В 2011 году была предложена модель DVMH для программирования кластеров с ускорителями с помощью директив.

В ноябре 2011 года был предложен стандарт OpenACC [21] для программирования графических процессоров с помощью директив, его вторая версия вышла в июне 2013 года.

В июле 2013 года опубликован стандарт OpenMP 4.0, в который вошли средства для работы с подключаемыми вычислительными устройствами, обладающие собственной памятью (сопроцессоры, ускорители).

Вызывает интерес сравнение этих трех предлагаемых подходов.

Главное отличие этих подходов заключается в том, что DVMH предназначен для написания программ для кластеров, в узлах которых установлены многоядерные процессоры и ускорители разных архитектур, отличающиеся как по типу памяти (общая или раздельная), так и по скорости обменов между памятью ускорителей и памятью ЦПУ. Стандарты OpenACC и OpenMP 4.0 предназначены для написания программ, выполняющихся в отдельных узлах такого кластера.

В спецификациях DVMH и OpenACC просматривается ориентированность на графические процессоры в части организации параллелизма, управляемого кеширования данных, отсутствия средств синхронизации и программного интерфейса времени выполнения, предназначенных для использования на ускорителе. Для OpenMP 4.0 это неверно и может являться определенным препятствием для его реализации для графических процессоров.

В стандартах OpenACC и OpenMP 4.0 очень похожая методика управления перемещением данных, основанная на указании каждого перемещения пользователем и определяемых пользователем интервалах (для OpenMP 4.0 – только статически определенных) существования экземпляра переменной на ускорителе. В DVMH-языках применена иная методика, основанная на указаниях входных и выходных данных для фрагментов кода, предназначенных для выполнения на ускорителях (такие фрагменты называются вычислительными регионами или просто регионами), и позволяющая автоматически определять необходимые перемещения данных в зависимости от того, на каком устройстве (ускорителе или многоядерном процессоре) был выполнен тот или иной вычислительный регион.

В стандартах OpenACC и OpenMP 4.0 управление тем, исполнять ли данный регион на ускорителе или нет, задается вычисляемыми во время выполнения выражениями. В DVMH такой возможности нет, однако имеется несколько режимов планирования, определяющих, на каких устройствах выполнять регионы.

Модели DVMH и OpenMP 4.0 предусматривают использование нескольких ускорителей для одной программы, но стоит отметить, что в DVMH это является следствием наличия указаний по распараллеливанию на кластер и не требует дополнительного управления со стороны пользователя, а OpenMP 4.0 предоставляет набор средств по использованию нескольких ускорителей с полностью ручным управлением каждым из них. В OpenACC одновременная работа с несколькими ускорителями не предусмотрена и предполагает дополнительное использование технологии распараллеливания для мультипроцессора (например, OpenMP).

Прямое перемещение данных между ускорителями в стандартах OpenMP 4.0 и OpenACC не предусмотрено, в то время как в DVMH этот функционал имеется благодаря выбранной методике управления перемещениями данных.

Все три модели предоставляют средства для асинхронного выполнения вычислительных регионов на ускорителе.

Все три модели предоставляют возможность не указывать полностью списки используемых или перемещаемых переменных, применяя консервативные умолчания в этом случае.

Для написания программ для кластера с ускорителями DVMH предоставляет все средства, а при использовании OpenACC или OpenMP 4.0 придется добавить использование коммуникационной библиотеки, например, MPI.

По части поддержки циклов с зависимостями DVMH имеет поддержку приватных переменных, редукционных зависимостей, регулярных

зависимостей по одному или нескольким измерениям сразу; OpenACC имеет поддержку только приватных переменных и редукционных зависимостей; OpenMP 4.0 имеет поддержку приватных переменных и редукционных зависимостей, а также с помощью встроенных средств синхронизации и программного интерфейса времени выполнения позволяет распараллеливать и циклы с другими типами зависимостей.

Глава 2. Схема построения компилятора с языка Фортран-DVMH

Компилятор с языка Фортран-DVMH является развитием компилятора с языка Фортран-DVM. Компилятор с языка Фортран-DVM является source-to-source компилятором. Это позволяет:

1. Абстрагироваться от целевой процессорной архитектуры.
2. Иметь возможность использовать предпочитаемый пользователем компилятор со стандартного языка Фортран, что позволяет иметь легкое использование имеющихся у пользователя библиотек.
3. Переложить тяжелую работу по оптимизации генерируемого объектного кода на хорошо развитые компиляторы со стандартного языка Фортран.

2.1 Схема работы компилятора с языка Фортран-DVMH

Компилятор с языка Фортран-DVMH состоит из блока анализа программы, блока конвертации, системы поддержки выполнения DVMH-программ (библиотека LibDVMH).

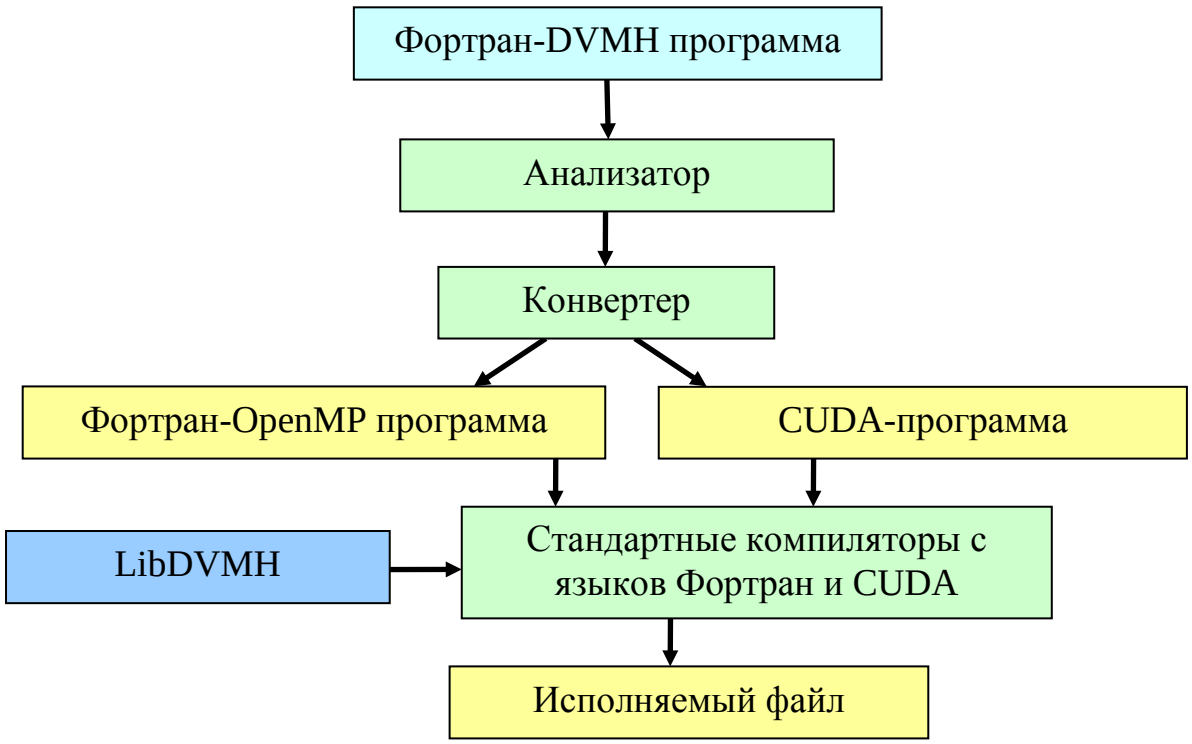

Рисунок 1. Схема работы компилятора с языка Фортран-DVMH

Анализатор строит структуру исходной программы и дополняет заданные программистом указания о режимах использования данных, для которых есть правила автоматического определения.

Конвертер преобразует исходную программу в эквивалентную ей пару программ, которые опираются на систему поддержки выполнения DVMH-программ и уже могут быть скомпилированы в машинный код стандартными компиляторами.

LibDVMH является библиотекой функций, которая обеспечивает выполнение DVMH-программ.

При обработке большинства директив конвертер генерирует один или несколько вызовов процедур LibDVMH с параметрами, отражающими указания пользователя в данной директиве.

При компиляции вычислительных регионов происходит их разделение на составные части – параллельные гнезда циклов и последовательные операторы. Эти части выделяются в отдельные подпрограммы с

использованием соответствующих технологий программирования: для ГПУ – CUDA, для ЦПУ – OpenMP. Использование и настройка таких подготовленных подпрограмм ведется напрямую из LibDVMH, что позволяет их гибко комбинировать, многократно запускать, откладывать их выполнение (асинхронный режим).

2.2 Основные функции системы поддержки выполнения DVMH-программ

Основными функциями системы поддержки выполнения DVMH-программ являются:

- отображение абстрактной параллельной машины (идеальной машины, наиболее подходящей для выполнения программы) на заданную при запуске решетку процессов и имеющийся в каждом узле набор вычислительных устройств;
- создание и уничтожение распределенного массива;
- отображение распределенного массива на абстрактную параллельную машину;
- отображение параллельного цикла и параллельных задач на абстрактную параллельную машину;
- создание и загрузка буферов для доступа к удаленным данным;
- подготовку и организацию параллельного выполнения участков программы на разнородных вычислительных устройствах;
- балансировку загрузки узлов кластера и вычислительных устройств каждого узла (статическую и динамическую);
- динамический подбор значений оптимизационных параметров;
- управление текущим состоянием и местонахождением данных, их перемещениями;

- выполнение редукционных операций;
- обновление теневых граней распределенных массивов (теневые грани – это расширения локальной части массива для организации доступа к расположенным на соседних вычислительных устройствах элементам этого массива);
- возможности для функциональной сравнительной отладки корректности работы на ускорителях;
- возможности для отладки производительности.

Глава 3. Алгоритмы учета актуального состояния переменных при выполнении DVMH-программ

При программировании графических процессоров применяется методика полного им управления со стороны ЦПУ. Это означает, что программист в первую очередь пишет программу для ЦПУ, из которой уже использует графический процессор.

Данная ситуация серьезно осложняется тем, что графический процессор не имеет прямого доступа в основную память ЦПУ, а располагает своей собственной быстродействующей широкополосной памятью. Это приводит к тому, что программисту необходимо перед запуском вычислений на ГПУ загрузить используемые данные на ГПУ, а после проведения вычислений выгрузить результаты работы с ГПУ.

Так как операции загрузки из памяти ЦПУ в память ГПУ и выгрузки из памяти ГПУ в память ЦПУ медленные (примерно в 100 раз медленнее доступа в память ГПУ с ГПУ), то программисту приходится применять иную схему работы с ГПУ, при которой данные по возможности не выгружаются, а остаются в памяти ГПУ до следующего их использования.

3.1 Способы управления перемещениями данных на ускоритель и обратно

Есть как минимум два способа управления перемещениями данных на ускоритель и обратно:

1. Ручное копирование.
2. Указание входных и выходных данных для фрагментов программы

3.1.1 Ручное копирование

При ручном копировании программист использует команды копирования, в которых указывает адреса в памяти ГПУ и памяти ЦПУ, количество байт, которое необходимо скопировать и направление копирования (на ГПУ или с ГПУ).

Такой метод имеется во всех низкоуровневых средствах программирования ГПУ, так как он является базовым для передачи данных на подключаемое устройство и получения данных обратно.

При использовании данного способа программист самостоятельно следит за тем, где какие изменения были произведены с тем, чтобы в нужные моменты времени производить копирования на ускоритель или обратно.

Данный способ дает максимальную гибкость при работе с ускорителями, но имеет и серьезные недостатки, о которых упоминается ниже.

3.1.2 Указание входных и выходных данных для фрагментов программы

При данном способе управления перемещениями данных программист указывает входные и выходные данные для фрагментов программы. Каждый такой фрагмент исполняется либо на ЦПУ, либо на ГПУ. При этом обновления входных данных перед выполнением каждого фрагмента производятся на основании информации о том, на каком устройстве был исполнен последний фрагмент, для которого эти данные были выходными.

Данный способ дает меньшую степень гибкости при работе с ускорителями, например, в части совмещения операций копирования с вычислениями.

3.2 Недостатки ручного копирования

Ручное управление программистом всеми перемещениями данных имеет ряд недостатков:

- Программисту приходится быть четко ориентированным на то, сколько и каких ускорителей используется в программе, а также предполагается ли одновременное использование ЦПУ для дополнительного разделения вычислений. Это может приводить к утере ее универсальности в плане используемой целевой ЭВМ, или же значительному усложнению в целях поддержки различных конфигураций оборудования.
- Для программ с достаточно большой степенью разветвленности, а также программ, имеющих многократно выполняемые части, причем при различных входных данных и различных путях выполнения, становится серьезной проблемой оптимизация перемещений данных, что может приводить как к излишним перемещениям (в случае если программист не приложил достаточно усилий для оптимизации или нарочно пошел на такие издержки для большей устойчивости программы к дальнейшим модификациям), так и к сложно находимым ошибкам, проявляющимся только при некоторых наборах входных данных.
- Если программист пишет программу, способную выполняться на разном числе узлов кластера (а именно такими программами являются DVM-программы), то обеспечить оптимальное отображение вычислений на многоядерные процессоры и ускорители внутри каждого узла – эта задача практически непосильна для программиста и должна решаться на системном уровне.

Использование метода управления перемещениями данных, основанного на задании входных и выходных данных регионов, и, как следствие, полное информирование системы поддержки выполнения программ о выполняемых модификациях переменных, позволяет избавиться от недостатков полностью ручного управления перемещениями данных, а также добавляет полезные возможности такие, как:

- сравнительная функциональная отладка (выполнение одного и того же региона с одними и теми же исходными данными с использованием ЦПУ и ускорителей, а затем сравнение значений полученных выходных данных);
- многократное выполнение регионов с одними и теми же исходными данными с целью поиска значений оптимизационных параметров (параметров, не влияющих на корректность исполнения участка программы, но влияющих на время его исполнения).

3.3 Определения терминов и базовых операций

Разработанные алгоритмы основываются на следующих понятиях:

- переменная – скаляр или массив с заданным количеством измерений, начальным и конечным индексом по каждому измерению, размером (в байтах) одного элемента;
- представитель некоторой переменной на некотором устройстве – экземпляр (возможно, неполный) переменной, размещенный в памяти устройства;
- состояние актуальности представителя – задание для всех элементов представителя, являются ли они (элементы) актуальными, т.е. имеют ли они последнее присвоенное этому элементу значение;

Для манипуляций с состояниями актуальности представителей переменных вводится особый вид функций – PCS_n – как отображение из Z^n в N U {0,+inf}. Ниже перечислены базовые операции над ними, где I(a) – индикаторная функция, принимающая значение 1 при истинном выражении, данном в качестве аргумента a; 0 – в противном случае:

- Объединение двух PCS. Объединением двух PCS p1 и p2 называется PCS p3 такое, что p3(x) = max(p1(x), p2(x))

- Разность двух PCS. Разностью двух PCS p1 и p2 называется PCS p3 такое, что p3(x) = I(p1(x)>p2(x))p1(x)
- Понижение одного PCS на уровень другого PCS. Понижением одного PCS p1 на уровень другого PCS p2 называется PCS p3 такое, что p3(x) = I(p1(x)<+inf или p2(x)=0)p1(x) + I(p1(x)=+inf и p2(x)>0)p2(x)
- Горизонтальная разность двух PCS. Горизонтальной разностью PCS p1 и p2 называется PCS p3 такое, что p3(x) = I(p1(x)!=p2(x))p1(x)
- Пересечение двух PCS. Пересечением двух PCS p1 и p2 называется PCS p3 такое, что p3(x) = min(p1(x), p2(x))
- Пересечение двух PCS с повышением. Пересечением двух PCS p1 и p2 с повышением называется PCS p3 такое, что p3(x) = I(0<p1(x)<=p2(x))p2(x)

В процессе работы DVMH-программы система поддержки выполнения DVMH-программ следит за состоянием актуальности представителей переменных на всех используемых устройствах, модифицирует его в соответствии с указаниями направления использования данных в вычислительных регионах, происходящими теневыми обменами, указаниями директив актуализации и другими происходящими событиями, затрагивающими данные пользователя.

Каждому представителю массива приписывается его состояние актуальности в виде PCS_n, где n – количество измерений массива, значения которого трактуются как степень актуальности элементов массива в представителе, где нуль означает неактуальное значение, +inf – актуальное значение, а натуральные значения – некоторое промежуточное состояние, которое будет трактоваться как актуальное или неактуальное в зависимости от контекста. Также элемент массива, имеющий степень актуальности +inf называется абсолютно актуальным.

С учетом схемы работы с теневыми гранями, при которой чтение из них может происходить в любой точке программы, а обновление происходит только по специальным указаниям пользователя, для полной поддержки динамического режима разделения работы между вычислительными устройствами узла, при котором локальная для данного устройства часть массива может изменяться от региона к региону, введена дифференцированная степень актуальности для элементов массива.

Для каждого массива заводится так называемый профиль его теневых граней. Профиль представляет собой PCS_n, для которого верно, что:

- он равен нулю вне множества $[-L_1:H_1]x[-L_2:H_2]x...x[-L_n:H_n]$, где L_i – ширина нижней теневой грани по i-ому измерению, а H_i – ширина верхней теневой грани по i-ому измерению;
- он равен +inf в точке (0,0,…,0) и только в ней;

Профиль задает требования на актуальность элементов теневых граней этого массива, принимаемые им значения будем называть требуемой степенью актуальности.

Для каждого массива ведется счетчик максимальной требуемой степени актуальности для элементов теневых граней. Во время работы программы значение этого счетчика равно максимуму профиля теневых граней за пределами точки (0,0,…,0).

3.4 Алгоритмы учета состояния актуальности переменных

В начале программы локальная часть массива и его теневые грани в памяти ЦПУ считаются абсолютно актуальными (степень актуальности равна +inf). Представители на всех остальных устройствах считаются полностью неактуальными (степень актуальности равна 0). Счетчик текущей максимальной требуемой степени актуальности элементов теневых граней

для всех переменных устанавливается в единицу. Профили теневых граней принимают значение 1 для всех элементов теневых граней.

Всякое выполнение указаний SHADOW_RENEW и SHADOW_COMPUTE помимо прочего увеличивает на единицу счетчик текущей максимальной степени актуальности указанных в директиве переменных-массивов и модифицирует их профиль теневых граней, меняя его в части указанных в директиве теневых граней путем задания им требуемой степени актуальности в текущее значение максимальной степени актуальности. Также они приводят к модификации состояния актуальности представителей, так как производится объявление актуальности: для SHADOW_RENEW – в памяти ЦПУ, а для SHADOW_COMPUTE – на том устройстве, на котором выполняется соответствующий цикл. Если SHADOW_RENEW происходит в вычислительном регионе, то также производится пометка обновленного в памяти ЦПУ массива для его последующего обновления на устройствах региона перед выполнением следующего параллельного цикла (или последовательного участка) данного региона. При этом обновлены на устройствах региона будут только изменившиеся теневые грани (как следствие изменившегося состояния актуальности и требований к степени актуальности теневых элементов).

Всякая актуализация для представителя переменной делается с учетом ее профиля теневых граней. Это означает, что запрашиваемые требования актуальности предварительно пересекаются с профилем теневых граней, растянутым на текущую локальную часть. При этом если проведение актуализации производится в ходе обработки вычислительного региона, то локальная часть определяется в соответствии с распределением данных, произведенном при входе в вычислительный регион. Для актуализаций за пределами вычислительных регионов (а такая актуализация возможна только в память ЦПУ) используется локальная часть текущего процесса целиком.

Под растягиванием профиля теневых граней подразумевается формирование требований актуальности D на основе профиля теневых граней P по правилу: $D(i_1,i_2,\ldots,i_n)=P(I(i_1<B_1)(i_1-B_1)+I(i_1>U_1)(i_1-U_1), I(i_2<B_2)(i_2-B_2)+I(i_2>U_2)(i_2-U_2), \ldots, I(i_n<B_n)(i_n-B_n)+I(i_n>U_n)(i_n-U_n))$, где B_i и U_i являются нижними и верхними границами локальной части соответственно (включительно).

Далее приводятся алгоритмы учета актуального состояния данных при различных ситуациях.

3.4.1 Обработка входа в вычислительный регион

При входе в вычислительный регион системе поддержки выполнения сообщается о каждой переменной направления ее использования для ее подмассивов. Это делается или как результат указаний пользователя, или как результат анализа компилятором. Для каждого массива задается отображение из множества индексов массива во множество {0, IN, OUT, INOUT, LOCAL, INLOCAL}, где: 0 означает, что данный элемент не используется в регионе; IN – региону нужно актуальное значение и данный элемент может использоваться только на чтение; OUT – региону не нужно актуальное значение, данный элемент может использоваться как на чтение, так и на запись, причем новое значение может быть использовано далее; INOUT – региону нужно актуальное значение, данный элемент может использоваться как на чтение, так и на запись, причем новое значение может быть использовано далее; LOCAL – региону не нужно актуальное значение, данный элемент может использоваться как на чтение, так и на запись, причем новое значение не может быть использовано далее; INLOCAL – региону нужно актуальное значение, данный элемент может использоваться как на чтение, так и на запись, причем новое значение не может быть использовано далее.

После определения распределения данных по устройствам (определения локальных частей каждого массива на каждом устройстве),

система поддержки выполнения для каждой используемой в регионе переменной выполняет следующие действия:

1. Для каждого используемого устройства:
   1.1. Определение части переменной, которая должна находиться на устройстве, как локальная часть + теневые грани.
   1.2. Если на нем не находится необходимая часть переменной:
      1.2.1. Выделение необходимого количества памяти.
      1.2.2. Перепись актуальных значений из предыдущего месторасположения.
      1.2.3. Освобождение памяти, выделенной под ранее находившуюся на устройстве часть переменной.
   1.3. Получение запрошенных (IN, INOUT, INLOCAL части переменной) актуальных данных с других устройств, причем подлежат актуализации в том числе и теневые грани с учетом применения профиля теневых граней.
2. Понижение актуальности для OUT, INOUT, LOCAL, INLOCAL частей переменной на всех устройствах.
3. Для каждого используемого устройства:
   3.1. Объявление актуальности для OUT, INOUT, LOCAL, INLOCAL частей переменной на данном устройстве, причем подлежат объявлению актуальности только локальные для данного вычислительного региона части представителей.

3.4.2 Обработка запроса актуализации

Запрос актуализации – это одна из основополагающих операций, которая в обобщенном виде лежит в основе почти всех манипуляций с данными. Она для заданных элементов переменной и для заданного устройства обновляет те элементы, которые не являются в требуемой степени

актуальными, находя в достаточной степени актуальные элементы переменной на других устройствах. Алгоритм можно представить следующим образом (p – запрашиваемая степень актуальности для элементов, является PCS; u – устройство, на которое производится актуализация):

1. Присвоить в p2 разность p и состояния актуальности представителя на устройстве u.
2. Для каждого устройства d, отличного от u:
   2.1. Присвоить в p1 пересечение с повышением p2 и состояния актуальности представителя на устройстве d.
   2.2. Скопировать с устройства d на устройство u все элементы x, для которых p1(x)>0.
   2.3. Вычесть из p2 p1.
   2.4. Обновить состояние актуальности на устройстве u, сделав объединение его с p1.

3.4.3 Обработка указания SHADOW_RENEW

Указание SHADOW_RENEW указывается программистом с целью обновления теневых граней массива. При этом сначала следует актуализировать пересылаемые с данного процесса элементы массива, а затем учесть обновление полученных от соседних процессов элементов. Ниже приводится алгоритм действий, производимых непосредственно до посылки и приема данных между процессами:

1. Актуализировать в память ЦПУ граничные элементы локальной части процесса.
2. Объявить актуальность для теневых элементов (относительно локальной части процесса) в памяти ЦПУ.

3. Обновить профиль теневых граней, сделав объединение старого профиля теневых граней с PCS, являющимся характеристической функцией обновленных теневых элементов, умноженной на увеличенный на единицу счетчик текущей максимальной требуемой степени актуальности для теневых элементов переменной-массива.
4. Если данные действия производятся в ходе работы региона, то пометить массив для последующего обновления перед следующим циклом или последовательным участком. При этом будут также обновлены теневые элементы, находящиеся на другом устройстве текущего процесса.

3.4.4 Обработка указания REMOTE_ACCESS

Указание REMOTE_ACCESS указывается программистом с целью получения эффективного доступа к удаленным элементам, не являющимися в общем случае элементами теневых граней. При указании REMOTE_ACCESS происходит:

1. Актуализация в память ЦПУ данных текущего процесса, к которым будет осуществляться удаленный доступ.
2. Создание специального буфера удаленных элементов.
3. Сбор со всех процессов в него запрошенных данных.
4. Если данные действия производятся в ходе работы региона, то:
   4.1. Создание представителей буфера удаленных элементов на ускорителях, использующихся данным регионом.
   4.2. Актуализация всех вновь созданных представителей на устройствах региона.
   4.3. После окончания работы с буфером удаленных элементов уничтожение представителей на ускорителях.

3.4.5 Обработка указания CONSISTENT

Указание CONSISTENT указывается программистом с целью объединения записей в размноженный массив, сделанных в параллельном цикле. При обработке указания CONSISTENT в ходе выполнения региона происходит:

1. Определения той части размноженного массива, которая была записана текущим процессом.
2. Актуализация в память ЦПУ записанной части размноженного массива.
3. Объявление актуальности в памяти ЦПУ всего размноженного массива целиком. Данное действие производится с учетом того, что остальные части приводимого в консистентное состояние массива будут получены с других процессов.

Глава 4. 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 исполнения всех поддерживаемых языком конструкций на разнородных вычислительных устройствах, балансировка нагрузки между вычислительными устройствами, а также выбор оптимального способа исполнения каждого участка кода на том или ином устройстве.

Параллелизм в DVMH-программах проявляется на нескольких уровнях:

- Распределение данных и вычислений по MPI-процессам. Этот уровень задается специальными директивами распределения или перераспределения данных и спецификациями параллельных подзадач 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 органично отобразить программу на кластер с многоядерными процессорами и ускорителями в узлах.

4.1 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 для вычислительных регионов возможно их исполнение на разнородных вычислительных устройствах. Внутри, равно как и вне регионов могут быть как параллельные циклы, так и последовательные участки программы.

Исполнение DVMH-программы начинается синхронно всеми запущенными процессами в модели SPMD. На каждый процесс выделяется одна основная последовательная нить исполнения.

При входе в вычислительный регион каждый процесс независимо выполняет дополнительное к межпроцессному 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 процесс разделяет работу в соответствии с распределением данных по вычислительным устройствам. Затем он выбирает для каждого устройства метод обработки части цикла на конкретном устройстве, называемый обработчиком, 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.

4.2 Режимы распределения данных и вычислений

В DVMH-программах все распределяемые данные распределяются блочно: каждое распределенное измерение делится несколькими точками на отрезки. При этом есть возможность по каждому измерению распределенного массива задавать или равномерное блочное распределение, или распределение взвешенными блоками, т.е. с учетом заданного вектора весов. Эти указания непосредственно выполняются при распределении данных между MPI-процессами. 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 три режима распределения данных и вычислений по вычислительным устройствам в точках входа в регионы:

1. Простой статический режим.
2. Динамический режим с подбором схемы распределения.
3. Динамический режим с использованием подобранной схемы распределения.

Рассмотрим подробнее эти режимы распределения.

4.2.1 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 (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 таком 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 (при наличии нескольких) обработчики по-умолчанию, а оптимизационные параметры получают следующие значения:

- Количество используемых ЦПУ-нитей – максимально доступное текущему процессу.
- 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 глобальной памяти графического процессора – или более быстрый и более требовательный по объему памяти, или менее быстрый и менее требовательный по объему памяти.

4.2.2 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.

Каждый вычислительный регион в данном режиме рассматривается в виде нескольких родственных вариантов запуска, каждый из которых определяется парой (вычислительный регион, соответствие данных), где под соответствием данных понимается соответствие используемых в исходном коде вычислительного региона локальных переменных реальным переменным (массивам или скалярам).

Для каждого варианта запуска региона определяются:

- Динамически предшествующие варианты запуска региона, являющиеся поставщиками актуальных входных данных, причем учитываются и ранее закончившиеся регионы, которые использовали актуализируемые данные только на чтение. В качестве поставщиков актуальных данных также могут быть директивы ACTUAL и GET_ACTUAL, приравниваемые к регионам, исполняемым исключительно на ЦПУ и имеющим пустое тело.
- Зависимость времени работы от распределения данных, причем принимаются во внимание все варианты запуска одного и того же вычислительного региона с учетом их разных соответствий данных.
- Количество вхождений.

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.

Так как каждый параллельный цикл распределяется только по одной абстрактной машине (распределенному массиву), то функция зависимости времени работы параллельного цикла на устройстве строится как функция одного вещественного аргумента – объема вычислений (с учетом весов при распределении), отданных конкретному устройству. Так как для обработки параллельного цикла даже для одного устройства допустимо наличие нескольких обработчиков, каждый со своими оптимизационными параметрами, то для параллельных циклов имеется семейство табличных функций времени от данного объема вычислений, а итоговая функция зависимости времени от объема вычислений, отданных устройству для параллельного цикла строится, как минимум среди соответствующего семейства.

Время обработки последовательных участков считается постоянным. Время выполнения хост-секций строится как сумма постоянной величины, затраченной на исполнение операторов хост-секции с временными затратами, потраченными на GET_ACTUAL, имеющиеся в данной хост-секции.

Для интерполяции и экстраполяции значений построенных функций применяется модель логарифмически-увеличивающейся производительности устройства в зависимости от отданного ему объема вычислений.

В этом режиме периодически включается построение субоптимальной схемы распределения данных во всех вариантах запуска вычислительных регионов на основе накопленных сведений в целом для программы, анализируя целиком последовательность вариантов запуска регионов и их характеристики выполнения. При построении таких схем учитываются как внутренние показатели вариантов запуска регионов в виде зависимости времени работы от распределения данных, так и последовательность исполнения вариантов запуска вычислительных регионов с целью минимизации в том числе и временных затрат на перераспределение данных. После построения такой схемы, она применяется и происходит дальнейшее

накопление характеристик и информации о вычислительных регионах и параллельных циклах.

В процессе работы программы в данном режиме решаются следующие задачи:

1. Идентификация варианта запуска региона.
2. Сбор информации о потоке данных между вариантами запуска регионов, а также между вариантами запуска регионов и внерегионным пространством.
3. Сбор информации о быстродействии вычислительных устройств на отдельных параллельных циклах.
4. Построение глобальной схемы распределения данных и вычислений по вычислителям, в которой для каждого варианта запуска региона выбран вычислитель.

Рассмотри подробнее каждую из этих задач.

Идентификация вариантов запуска региона происходит при входе в вычислительный регион по следующей схеме:

1. Если предыдущий (динамически предшествующий) вариант запуска данного региона был уникальным и вместе с тем часть из используемых им переменных ныне не существует, то происходит обновление информации об этом варианте запуска (пометка ныне не существующих данных, как локальных), а также отождествление с совпадающим по соответствию данных вариантом запуска (если таковой имеется).
2. Фиксируется порядок регистрации используемых в нем переменных, т.е. формируется упорядоченный список ссылок на фактические переменные вместе с указанными вырезками для массивов.

3. По семейству родственных вариантов запуска, т.е. вариантов запуска одного региона, производится поиск по полному совпадению соответствия данных. Если совпадение найдено, то текущее выполнение региона признается принадлежащим найденному варианту запуска. Иначе создается новый вариант запуска региона и текущее выполнение региона признается принадлежащей вновь созданному уникальному варианту запуска.

Сбор информации о потоке данных между вариантами запуска вычислительных регионов происходит при входе в вычислительный регион сразу после идентификации текущего его варианта запуска следующим образом:

1. По каждой части (в случае массива) используемой на чтение переменной определяется источник ее актуального значения – вариант запуска региона, в котором данная часть переменной была выходной (или операция объявления актуальности).
2. По каждой части всех используемых на чтение переменных ведется подсчет – сколько раз имел место тот или иной источник ее актуального значения.

Сбор информации о быстродействии вычислителей для каждого варианта запуска региона производится следующим образом:

1. Регион запускается на том вычислителе, на котором текущий вариант запуска региона еще не был запущен, если такой имеется.
2. Если такого не имеется, регион запускается на том вычислителе, на котором текущий вариант запуска исполняется наиболее эффективно.

3. Для каждого параллельного цикла осуществляется замер быстродействия выбранного вычислителя на нем, а также запоминается информация об отображении каждого параллельного цикла.
4. После завершения выполнения региона происходит запись полученных быстродействий с детализацией по каждому параллельному циклу и последовательному участку.
5. Считается количество выполнений варианта запуска региона, а также среднее значение и среднеквадратическое отклонение для значений эффективности по каждому вычислителю.

Построение глобальной схемы распределения вычислений по вычислителям, в которой для каждого варианта запуска региона выбран вычислитель производится в два этапа:

1. Построение вспомогательного графа.
2. Выбор вычислителей для вариантов запуска регионов по вспомогательному графу жадным алгоритмом.

Построенный граф состоит из вершин, которые при начальном построении соответствуют вариантам запуска регионов и ребер, которые соответствуют связям по данным между вариантами запуска регионов. Каждая вершина содержит информацию о суммарном времени выполнения этой вершины на каждом из устройств, включающее известные на данный момент временные затраты на перемещение данных для обеспечения работы данной вершины и известные на данный момент временные затраты на перемещение выходных данных данной вершины.

Каждое ребро содержит информацию о суммарных временных затратах на перемещения данных вдоль данного ребра для каждой пары устройств при условии, что эта пара устройств выбрана для выполнения инцидентных данному ребру вершин.

Выбор вычислителей для вариантов запуска регионов по вспомогательному графу жадным алгоритмом состоит из циклического выполнения следующих шагов:

1. Определение веса каждой вершины, как максимальное возможное время ее выполнения.
2. Определение веса каждого ребра, как максимальные возможные временные затраты на перемещения данных вдоль данного ребра.
3. Нахождение среди всех вершин и ребер того объекта, чей вес максимален.
4. Если была найдена вершина, то данной вершине приписывается то устройство, на котором время ее выполнения минимально (соответственно это устройство выбирается для того набора вариантов запуска регионов, которому соответствует данная вершина), вершина изымается из графа. Те ребра, которые были инцидентны изъятой вершине сливаются с вершинами, инцидентными им и оставшимися в графе по правилу: новое время выполнения на устройстве принимается увеличенным на величину временных затрат на перемещение данных с/на то устройство, которое было выбрано для изъятой вершины.
5. Если было найдено ребро, то данное ребро изымается из графа, а инцидентные ему вершины сливаются вместе, при этом данной новой вершине соответствует объединение наборов вариантов запуска регионов для сливаемых вершин. При этом не допускается образование кратных ребер – они также сливаются вместе.

В результате работы данного алгоритма граф становится пустым, а каждому варианту запуска региона приписывается вычислитель. При этом на последнем шаге работы алгоритма получается оценка времени работы всей программы при выбранной схеме распределения.

Кроме распределения данных, подвергается подбору также выбор обработчика для каждого параллельного цикла для каждого устройства и оптимизационные параметры обработчиков такие, как количество ЦПУ-нитей для ЦПУ-обработчиков и размер и форма CUDA-блока нитей для CUDA-обработчиков.

Есть возможность записи построенных схем распределения в файл для использования в последующих запусках программы как в этом режиме, так и в третьем режиме. Также 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 файл и повторного запуска программы.

4.2.3 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.

В данном режиме работает тот же алгоритм идентификации варианта запуска региона, что и во втором режиме. Это влечет следующие последствия:

1. 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).

2. Первое выполнение варианта запуска региона при некоторых условиях может происходить медленнее, чем все остальные. Это связано с апостериорным отождествлением вариантов запуска регионов, работающих с локальными переменными.

Глава 5. Алгоритм распределения подзадач между узлами кластера

Одним из достоинств DVM-системы является то, что получаемые параллельные программы могут настраиваться при запуске на количество выделенных для них процессоров. Такое свойство программ позволяет запускать их на произвольном числе процессоров, компоновать сложные программы из имеющихся простых программ, повысить эффективность использования параллельных систем коллективного пользования за счет более гибкого распределения процессоров между отдельными программами. Этим свойством не обладают DVM-программы, использующие механизм подзадач.

Распределение подзадач по процессорам производится вручную и оно зачастую затруднено вследствие:

- большого количества подзадач,
- заметного разброса их сложности,
- необходимости осуществлять распределение на различное количество процессоров.

5.1 Формализация постановки задачи

За M обозначим количество процессоров, за N – количество подзадач. Для каждой подзадачи известно:

- минимальное количество процессоров, которое может быть использовано для счета подзадачи Kmin;
- максимальное количество процессоров, которое может быть использовано для счета подзадачи Kmax;

- функция зависимости времени исполнения от количества процессоров time(k) > 0 такая, что для всех k из диапазона [Kmin, Kmax - 1] выполнено: time(k) * k <= time(k + 1) * (k + 1)

Требуется составить оптимальное – с минимальным финишным временем – расписание прохождения подзадач, где для каждой подзадачи будет указано стартовое время, группа процессоров для ее счета. Подзадачи не могут считаться одновременно используя один и тот же процессор. Для всех процессоров из группы, назначенной для счета подзадачи, времена старта счета этой подзадачи совпадают, равно как и времена счета (продолжительность) подзадачи – синхронное выполнение одной подзадачи.

Как известно, задача составления многопроцессорного расписания NP-полна, а, значит, исследуемая задача NP-трудна, так как является обобщением NP-полной задачи.

Для более традиционной постановки, в которой для каждой подзадачи необходимо выделить только один процессор, имеется множество эвристических алгоритмов. Однако в литературе не удалось найти алгоритм автоматического распределения процессоров между подзадачами, требующими для своего выполнения не одного, а нескольких процессоров и допускающих выполнение на различном количестве процессоров.

5.1 Эвристический алгоритм распределения подзадач

Для описания алгоритма вводятся несколько дополнительных обозначений:

- Proc(x, d) – множество всех процессоров, свободных с момента времени x и, как минимум, до момента (x + d)
- Proc(x) – отображение такое, что Proc(x)(d) = Proc(x, d) для всех d и x
- Procs(x) – множество всех процессоров, свободных в момент времени x

- Tasks – множество всех подзадач
- Kmin(t) – минимальное количество процессоров, которое может быть использовано для счета подзадачи t
- Kmax(t) – максимальное количество процессоров, которое может быть использовано для счета подзадачи t
- time(t, k) – время счета подзадачи t с использованием k процессоров

Алгоритм распределения подзадач можно описать следующим образом:

1. Положим tRestMin – сумма по всем задачам t из Tasks величин time(t, Kmin(t)) * Kmin(t)
2. Упорядочить подзадачи по убыванию величины time(t, Kmin(t)) * Kmin(t) – список sortedTasks, где t принимает все значения из Tasks
3. Положим tMax и tOccupied равными нулю
4. Взять задачу t из начала списка sortedTasks
5. Удалить задачу t из списка sortedTasks
6. Уменьшить tRestMin на величину time(t, Kmin(t)) * Kmin(t)
7. Положим множество notExamined равным [Kmin(t), Kmax(t)]
8. Для каждого момента времени x, являющегося либо началом отсчета времени, либо моментом изменения множества Procs(x) в порядке возрастания выполнять:
   8.1. Для каждой длительности d, не меньшей time(t, k) для некоторого k, в порядке убывания выполнять:
      8.1.1. Для каждого количества процессоров k, принадлежащему notExamined и не большему, чем мощность Proc(x, d), а также такому, что time(t, k) <= d в порядке возрастания выполнять:

8.1.1.1. Положим tSuggested(x, k) равным максимуму из трех величин: tMax, x + time(t, k), x + (tOccupied + tRestMin) / k

8.1.2. Исключить из множества notExamined рассмотренные в цикле 8.1.1 количества процессоров

8.1.3. Если множество notExamined пусто, то перейти к пункту 9

9. Пусть x0 – такое минимальное, что минимизирует tSuggested(x0, k) для некоторого k. Пусть k0 – минимальное из таких, что минимизируют tSuggested(x0, k0)
10. Положим tMax максимуму из tMax и x0 + time(t, k0)
11. Увеличим tOccupied на величину time(t, k0) * k0
12. Зарезервировать группу из k0 процессоров на время [x0, x0 + time(t, k0)], выделив подмножество из Proc(x0, d) с таким максимальным d, что мощность Proc(x0, d) не менее k0
13. Если список sortedTasks не пуст, то перейти к пункту 4
14. Конец

В результате будет составлено полное расписание прохождения подзадач на многопроцессорной системе.

Алгоритм имеет алгоритмическую сложность асимптотически равную O(N * (N + M)), затраты по памяти асимптотически равны O(N + M).

5.2 Способы применения предложенного алгоритма

В язык Фортран-DVM были добавлены конструкции для поддержки автоматического распределения подзадач, однако от программиста требуется указать относительные сложности подзадач, минимальное и максимальное число процессоров для каждой подзадачи, а также параметры зависимости времени выполнения каждой подзадачи от числа выделенных ей процессоров.

При запуске многоблочной DVMH-программы с целью использования ускорителей возможны два режима распределения подзадач:

- логическим процессором для алгоритма распределения назначается один ускоритель и тем самым балансируется нагрузка на каждый ускоритель;
- логическим процессором для алгоритма распределения назначается целый узел кластера и тем самым балансируется нагрузка на каждый узел, а уже при выполнении подзадачи задействуются все вычислительные устройства узлов в соответствии с общими механизмами распределения данных и вычислений.

Разработанный алгоритм также может быть использован в планировщиках систем очередей для кластеров коллективного доступа, а также для планирования работы объединения кластеров, каждый из которых имеет заданное количество процессоров.

Глава 6. Дополнительные возможности по функциональной отладке и отладке производительности

Как упоминалось ранее, выполнение DVMH-программы можно представить как последовательность выполнения вычислительных регионов и участков внерегионного пространства.

Для регионов заданы все входные и выходные данные, а во внерегионном пространстве могут использоваться директивы актуализации и объявления актуальности.

Для DVMH-программ есть возможность сравнительной отладки. Это специальный режим работы DVMH-программы, при котором все вычисления в регионах одновременно выполняются на ЦПУ и других вычислительных устройствах.

Такой механизм позволяет выявлять и локализовывать ошибки, проявляющиеся при работе на ускорителях.

Включение и использование этого режима не требует от программиста менять программу или что-либо дополнительно сообщать о своей DVMH-программе. Реализация этого механизма основывается на следующих обстоятельствах:

1. Во-первых, вычислительный регион может быть выполнен одновременно независимо на нескольких вычислительных устройствах.
2. Во-вторых, системе поддержки выполнения DVMH-программ известен исчерпывающий набор входных и выходных данных региона, так как он ей необходим для управления перемещениями данных между устройствами.

3. В-третьих, системе поддержки выполнения DVMH-программ известно текущее состояние актуальности для всех представителей переменных.

В данном режиме сравниваются следующие значения переменных:

1. При завершении региона: равенство выходных данных, полученных в регионе при выполнении на ускорителях данным, полученным в регионе при выполнении на ЦПУ, неизменность входных данных.
2. При начале выполнения региона: равенство представителей на всех вычислительных устройствах для переменных, зарегистрированных с атрибутом **in** в регионе в актуальной их части.
3. Перед выполнением запроса актуализации: равенство представителей на всех вычислительных устройствах для актуализируемых переменных в актуальной их части.

6.1 Сравнение при завершении региона

При включенном режиме сравнительной отладки региона в момент принятия решения о распределении данных по устройствам сначала ЦПУ исключается из набора используемых устройств и производится обычное распределение, как если бы регион не мог быть исполнен на ЦПУ. Затем на ЦПУ отображаются все данные процесса целиком. Вследствие этого при отображении витков параллельных циклов одни и те же витки будут выполнены дважды – один раз на ЦПУ, другой – на ускорителе.

Также есть возможность подвергать сравнительной отладке не все, а только некоторые вычислительные регионы, задавая во время выполнения соответствующее указание для региона. Это свойство полезно для сокращения времени выполнения программы, так как выполнение региона на ЦПУ зачастую заметно медленнее выполнения на ускорителях, а также

имеются значительные временные затраты на частую перепись данных с ускорителей и их поэлементное сравнение.

При начале обработки региона создаются копии используемых только на чтение переменных.

Объявленные только как входные данные не имеют права быть измененными в регионе и для контроля этого производится сравнение их значений при выходе из региона с теми значениями, которые они имели при входе в регион.

Такое сравнение позволяет находить ошибки в указании направления использования переменных в директиве региона.

Если найдено расхождение, то пользователю сообщается о том, что зафиксирована необъявленная запись в переменные и данные переменные добавляются к сравнению выходных данных региона, так как фактически они являются не только входными, но и выходными данными региона.

После завершения обработки региона производится выгрузка с ускорителей в специальные буферы тех частей переменных, которые были зарегистрированы с атрибутом **out** в регионе, после чего производится поэлементное сравнение с теми значениями, которые получились на выходе региона при выполнении на ЦПУ.

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 пользователю выдается информация о них, и далее в программе используется та версия данных, которая была получена при счете на центральном процессоре.

Данный тип сравнения используется для контроля корректности выполнения региона на ускорителе, а также отсутствия указания необходимого **out** или **local** атрибута для переменных, которые изменяются в

регионе. 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 при 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).

6.2 Сравнение при входе в регион и при выполнении запроса актуализации

При включенном режиме сравнительной отладки в точке входа в регион, равно как и в точке выполнения директивы актуализации, происходит сравнение актуальных частей запрашиваемых данных, находящихся на различных устройствах.

При запросе актуализации сначала выясняется какая часть переменной по сведениям LibDVMH не нуждается в обновлении, а затем значения этой части сравниваются с представителями данной переменной на других устройствах в той части, которая на них является актуальной. При этом если найдено расхождение, то считается, что это вызвано отсутствием необходимого в таком случае использования директивы объявления актуальности. Соответственно, правильным значением считается то значение, которое имеется в памяти ЦПУ, а пользователю сообщается о наличии необъявленной с помощью директивы объявления актуальности модификации переменной во внерегионном пространстве.

При входе в регион зарегистрированные с атрибутом **in** переменные подвергаются такой же процедуре сравнения, как и для директивы актуализации.

Данный тип сравнения позволяет находить ошибки программиста, связанные с отсутствием указания директивы объявления актуальности для данных, измененных во внерегионном пространстве.

6.3 Возможности по отладке производительности

Возможности по отладке производительности позволяют узнать временные затраты на выполнение параллельных циклов, последовательных участков; временные затраты и объемы перемещений данных при обновлении теневых граней, выполнении редукционных операций, выполнении директив актуализации и перераспределении данных.

Для каждого параллельного цикла, включенного в регион, для каждого используемого вычислительного устройства производится построение табличной функции зависимости достигнутой производительности вычислительного устройства от объема отданных ему вычислений. Также подсчитывается общее время работы каждого параллельного цикла. Данная информация является основной для отладки производительности DVMH-программы.

Также немаловажными характеристиками являются время, потраченное на копирование данных на ускоритель и обратно, а также количество операций копирования и общий объем перемещенных данных между памятью ЦПУ и памятью ускорителя. По количеству операций копирования можно оценить, например, есть ли излишние копирования при работе программы, а по общему объему и временным затратам их вклад в общее время работы программы и соотношение времени, потраченного на копирования со временем, потраченным на произведение вычислений.

Из затрат на перемещения отдельно выделяются затраты при:

- обновлении теневых граней;

- организации доступа к удаленным элементам;
- восстановлении консистентности размноженных массивов;
- выполнении директив актуализации;
- выполнении межрегионного перераспределения данных;
- выполнении пользовательского (указанного директивами REDISTRIBUTE или REALIGN программистом) перераспределения данных;
- обновлении устаревших данных на ускорителях (как следствие изменения их в памяти ЦПУ).

Глава 7. Приложения и тесты, разработанные с использованием языка Фортран-DVMH

В данной главе рассматриваются вычислительные приложения из различных областей физики, демонстрирующие применимость языка Фортран-DVMH.

Также приводятся результаты сравнения производительности на тестах из пакета NASA NPB [22] программ на языке Фортран-DVMH и программ, написанных с использованием низкоуровневых технологий программирования специалистами из других исследовательских групп.

Все рассматриваемые в данной главе распараллеленные на языке Фортран-DVMH программы могут работать на кластере с использованием графических процессоров или без, а также с использованием общей памяти внутри узла (распараллеливание в модели SMP).

7.1 Приложения, демонстрирующие применимость языка Фортран-DVMH

В данном разделе рассматриваются приложения из области гидродинамики – одна двумерная задача и одна трехмерная. Также рассматриваются приложения из области квантовой механики и микроэлектроники.

Целью их рассмотрения является показать применимость языка Фортран-DVMH для написания программ для решения реальных задач, а также обозначить характеристики текстов полученных параллельных программ, указывающих на произведенные для распараллеливания изменения исходной последовательной программы.

7.1.1 Каверна

Программа «Каверна» предназначена для моделирования циркуляционного течения в плоской квадратной каверне с движущейся верхней крышкой в двумерной постановке в широком диапазоне как параметров задачи, так и параметров численного метода.

Последовательная версия программы занимает 496 строк.

В ходе разработки параллельной программы для данной задачи были преобразованы некоторые циклы; добавлены директивы языка Фортран-DVMH для распределения данных и вычислений (18 распределенных массивов, 7 вычислительных регионов, 28 параллельных циклов), организации доступа к удаленным данным (8 мест), актуализации (11 мест).

Текст параллельной программы занимает 613 строк.

В таблицах 1 и 2 приведены времена выполнения 200 итераций программы «Каверна» на сетке 3200x3200 на суперкомпьютере «Ломоносов» на разном числе процессорных ядер и ГПУ (в секундах).

Таблица 1

Время выполнения программы «Каверна» на сетке 3200x3200 на разном числе процессорных ядер

1	**2**	**4**	**8**	**16**	**64**	**128**	**256**	**400**	**512**	**1024**
1241,83	631,47	332,36	182,95	100,05	40,20	21,33	11,74	7,11	6,44	3,48

При использовании 1024 ядер программа «Каверна» ускорилась в 357 раз по сравнению с выполнением на 1 ядре. При использовании 1 ускорителя программа ускоряется в 17 раз по сравнению с выполнением программы на 1 ядре. Максимальное ускорение, полученное с использованием ускорителей – 390 раз по сравнению с выполнением программы на 1 ядре.

Таблица 2

Время выполнения программы «Каверна» на сетке 3200x3200 на разном числе ГПУ

1	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**400**
73,07	39,34	19,94	11,65	7,17	4,80	3,96	3,45	3,32	3,19

7.1.2 Контейнер

Программа «Контейнер» предназначена для численного моделирования течения вязкой тяжелой жидкости под действием силы тяжести в прямоугольном контейнере с открытой верхней стенкой и отверстием в одной из боковых стенок в трехмерной постановке в широком диапазоне как параметров задачи, так и параметров численного метода.

Последовательная версия программы занимает 828 строк.

В ходе разработки параллельной программы для данной задачи были преобразованы некоторые циклы; добавлены директивы языка Фортран-DVMH для распределения данных и вычислений (26 распределенных массивов, 5 вычислительных регионов, 21 параллельный цикл), организации доступа к удаленным данным (6 мест), актуализации (7 мест).

Текст параллельной программы занимает 942 строки.

В таблицах 3 и 4 приведены времена выполнения программы «Контейнер» на суперкомпьютере «Ломоносов» на разном числе процессорных ядер и ГПУ.

Таблица 3

Время выполнения программы «Контейнер» на разном числе ядер

Сетка, количество	**4**	**8**	**64**	**128**	**256**	**512**	**1024**	**2048**

итераций								
200x200x200 itmax=200	754,01	384,92	49,87	29,90	14,52	8,63	5,63	6,01
400x400x400 itmax=100	-	1202,32	164,02	85,68	43,10	22,53	13,54	7,66
800x800x800 itmax=50	-	-	576,16	318,75	151,78	79,68	41,26	21,91
1600x1600x1600 itmax=20	-	-	-	-	-	235,64	117,88	62,68

Таблица 4

Время выполнения программы «Контейнер» на разном числе ГПУ

Сетка, количество итераций	**1**	**2**	**4**	**32**	**128**	**256**	**512**	**1024**	**1280**
200x200x200 itmax=200	166,95	86,05	45,77	8,99	3,99	3,26	3,01	3,60	4,32
400x400x400 itmax=100	-	-	168,80	26,09	8,39	4,86	3,20	2,88	3,26
800x800x800 itmax=50	-	-	-	92,20	30,32	13,58	7,56	4,67	4,17
1600x1600x1600 itmax=20	-	-	-	-	-	37,38	20,14	10,74	8,95

При использовании 2048 ядер программа «Контейнер» ускорилась в 26,3 раза по сравнению с выполнением на 64 ядрах. При использовании 64 ускорителей программа ускоряется в 11 раз по сравнению с выполнением программы на 64 ядрах. При использовании 1280 ускорителей программа ускоряется в 138 раз по сравнению с выполнением программы на 64 ядрах.

7.1.3 Состояния кубитов

Программа «Состояния кубитов» предназначена для проведения трехмерных нестационарных расчетов состояния кубитов квантового компьютера на основе совместного решения трехмерного уравнения Пуассона и нестационарного уравнения Шредингера для двух электронов в квантовом приборе на основе кремниевой квантовой проволоки с учетом спина этих электронов.

Последовательная версия программы занимает 683 строк.

В ходе разработки параллельной программы для данной задачи были преобразованы некоторые циклы; добавлены директивы языка Фортран-DVMH для распределения данных и вычислений (23 распределенных массива, 5 вычислительных регионов, 61 параллельный цикл), организации доступа к удаленным данным (8 мест), актуализации (5 мест).

Текст параллельной программы занимает 1011 строк.

В таблице 5 приведены времена выполнения 192 итераций программы «Состояния кубитов» на сетке 121x121x241 на суперкомпьютере «К-100» на разном числе процессорных ядер и ГПУ (в секундах).

Таблица 5

Время выполнения программы «Состояния кубитов» на сетке 121x121x241 на разном числе процессорных ядер и ГПУ

12 ядер	**24 ядра**	**36 ядер**	**48 ядер**	**60 ядер**	**1 ГПУ**	**2 ГПУ**	**4 ГПУ**	**6 ГПУ**

190,38	104,20	74,57	56,71	39,96	102,06	59,66	32,97	22,04

При использовании 60 ядер программа «Состояния кубитов» ускорилась в 4,76 раз по сравнению с выполнением на 12 ядрах. При использовании 1 ускорителя программа ускоряется в 1,87 раз по сравнению с выполнением программы на 12 ядрах.

7.1.4 Кристаллизация 3D

Программа «Кристаллизация 3D» связана с задачей, в которой лазерный луч в начальный момент находится в точке (x,y), затем проплавляет материал в направлении оси z и смещается в направлении оси x. Когда луч смещается, расплавленный материал начинает кристаллизоваться. Положение луча в каждый момент времени задается некоторой функцией.

Последовательная версия программы занимает 530 строк.

В ходе разработки параллельной программы для данной задачи были преобразованы некоторые циклы; добавлены директивы языка Фортран-DVMH для распределения данных и вычислений (21 распределенный массив, 6 вычислительных регионов, 27 параллельных циклов), организации доступа к удаленным данным (5 мест), актуализации (5 мест).

Текст параллельной программы занимает 865 строк.

В таблице 6 приведены времена выполнения 100 000 итераций программы «Кристаллизация 3D» на сетке 103x5x103 на суперкомпьютере «К-100» на разном числе процессорных ядер и ГПУ (в секундах).

Таблица 6

Время выполнения программы «Кристаллизация 3D» на сетке 103x5x103 на разном числе процессорных ядер и ГПУ

1 ядро	**12 ядер**	**50 ядер**	**100 ядер**	**1 ГПУ**

514,4	256,3	89,09	79,97	396,3

При использовании 100 ядер программа «Кристаллизация 3D» ускорилась в 6,43 раз по сравнению с выполнением на 1 ядре. При использовании 1 ускорителя программа ускоряется в 1,3 раз по сравнению с выполнением программы на 1 ядре.

7.1.5 Спекание 2D

Программа «Спекание 2D» связана с задачей, в которой лазер плавит порошок, состоящий из твердой легкоплавкой компоненты. В порошке имеются поры, заполненные газом. В процессе плавления легкоплавкой компоненты порошка образуется жидкость.

Последовательная версия программы занимает 831 строку.

В ходе разработки параллельной программы для данной задачи были преобразованы некоторые циклы; добавлены директивы языка Фортран-DVMH для распределения данных и вычислений (17 распределенных массивов, 8 вычислительных регионов, 27 параллельных циклов), организации доступа к удаленным данным (3 места), актуализации (19 мест).

Текст параллельной программы занимает 1167 строк.

В таблице 7 приведены времена выполнения 10 000 итераций программы «Спекание 2D» на сетке 1003x1003 на суперкомпьютере «К-100» на разном числе процессорных ядер и ГПУ (в секундах).

Таблица 7

Время выполнения программы «Спекание 2D» на сетке 1003x1003 на разном числе процессорных ядер и ГПУ

1 ядро	**12 ядер**	**64 ядра**	**128 ядер**	**1 ГПУ**	**2 ГПУ**	**3 ГПУ**	**6 ГПУ**
2173	308,3	69,93	41,19	168,8	162,8	147,9	110,4

При использовании 128 ядер программа «Спекание 2D» ускорилась в 52,76 раз по сравнению с выполнением на 1 ядре. При использовании 1 ускорителя программа ускоряется в 12,87 раз по сравнению с выполнением программы на 1 ядре.

7.1.6 Спекание 3D

Программа «Спекание 3D» связана с задачей, в которой лазер плавит порошок, состоящий из твердой легкоплавкой компоненты. В порошке имеются поры, заполненные газом. В процессе плавления легкоплавкой компоненты порошка образуется жидкость.

Последовательная версия программы занимает 677 строк.

В ходе разработки параллельной программы для данной задачи были преобразованы некоторые циклы; добавлены директивы языка Фортран-DVMH для распределения данных и вычислений (19 распределенных массивов, 4 вычислительных региона, 51 параллельный цикл), организации доступа к удаленным данным (3 места), актуализации (22 места).

Текст параллельной программы занимает 1236 строк.

В таблице 8 приведены времена выполнения 1 000 итераций программы «Спекание 3D» на сетке 103x103x103 на суперкомпьютере «К-100» на разном числе процессорных ядер и ГПУ (в секундах).

Таблица 8

Время выполнения программы «Спекание 3D» на сетке 103x103x103 на разном числе процессорных ядер и ГПУ

1 ядро	**12 ядер**	**64 ядра**	**200 ядер**	**1 ГПУ**	**6 ГПУ**	**12 ГПУ**	**24 ГПУ**
751	192,9	40,23	30,46	50,88	18,47	14,85	12,64

При использовании 200 ядер программа «Спекание 3D» ускорилась в 24,66 раз по сравнению с выполнением на 1 ядре. При использовании 1

ускорителя программа ускоряется в 14,76 раз по сравнению с выполнением программы на 1 ядре.

7.2 Тесты на производительность из набора NASA NPB

Для сравнения программ на языке Фортран-DVMH с написанными вручную с использованием низкоуровневых технологий программирования программами по достигаемой эффективности приводятся ускорения тестов EP, BT, SP, LU из набора NASA NPB при использовании 1 ГПУ в сравнении со временем работы на 1 ядре ЦПУ.

Сравнение производилось с программами, написанными исследователями из Сеульского национального университета. Их работа «Performance Characterization of the NAS Parallel Benchmarks in OpenCL» опубликована в сборнике трудов международной конференции «2011 IEEE International Symposium on Workload Characterization» [23], а исходные тексты программ находятся в свободном доступе.

В таблице 9 приведены достигнутые ускорения для тестов EP, BT, SP, LU различных классов на 1 ГПУ по отношению ко времени работы на 1 ядре ЦПУ для программ на языке Фортран-DVMH и программ на языке Си с использованием технологии OpenCL. Запуски производились на кластере «К-100».

Таблица 9

Ускорения тестов NASA NPB для программ на языке Фортран-DVMH и программ на языке Си с использованием технологии OpenCL на 1 ГПУ по отношению ко времени работы на 1 ядре ЦПУ

Тест	**EP**	**BT**		**SP**			**LU**		
Класс	**C**	**A**	**B**	**A**	**B**	**C**	**A**	**B**	**C**
DVMH	46,24	2,4	2,22	5,88	6,37	7,72	6,47	9,06	11,1

OpenCL	44,63	0,89	1,04	3,05	3,36	2,87	5,43	6,72	6,4

DVMH-версии тестов по эффективности не уступают OpenCL-версиям, а на тестах, имеющих циклы с зависимостями, превосходят их благодаря имеющимся в компиляторе с языка Фортран-DVMH оптимизациям для обработки таких циклов.

Заключение

Основные результаты работы

1. Разработаны принципы отображения DVMH-программ на кластеры с ускорителями, обеспечивающие динамическое распределение вычислений между универсальными многоядерными процессорами (ЦПУ) и несколькими графическими процессорами (ГПУ).
2. Разработаны и реализованы в системе поддержки выполнения DVMH-программ следующие алгоритмы:
   - алгоритмы распределения независимых подзадач между узлами кластера, обеспечивающие балансировку загрузки узлов;
   - алгоритмы распределения витков параллельных циклов внутри узлов – между ядрами ЦПУ и несколькими ГПУ;
   - алгоритмы автоматического перемещения требуемых актуальных данных между памятью ЦПУ и памятями нескольких ГПУ.
3. Созданы средства сравнительной отладки DVMH-программ, базирующиеся на сопоставлении результатов одновременного выполнения на ЦПУ и на ГПУ одних и тех же фрагментов программы.

Проведенные эксперименты с тестами и реальными приложениями показали, что для класса задач, при решении которых используются разностные методы на статических структурных сетках, можно писать программы на языке Фортран-DVMH, которые эффективно выполняются на кластерах с ускорителями.

Литература

1. MPI: The Message-Passing Interface Standard. URL: http://www.mpi-forum.org/docs/mpi-1.1/mpi1-report.pdf (дата обращения 08.11.2013).
2. OpenMP Application Program Interface. URL: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf (дата обращения 08.11.2013).
3. Боресков А.А., Харламов А.А. Основы работы с технологией CUDA. — М.: ДМК-Пресс, 2010. — 232 стр.
4. The OpenCL Specification. URL: http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf (дата обращения 08.11.2013).
5. Коновалов Н.А., Крюков В.А., Погребцов А.А., Поддерюгина Н.В., Сазанов Ю.Л. Параллельное программирование в системе DVM. Языки Fortran-DVM и C-DVM. // Труды Международной конференции «Параллельные вычисления и задачи управления» (PACO’2001). — Москва, 2001. — с. 140–154.
6. Кривов М.А., Притула М.Н. Конвейерная модель представления параллельных программ. // Материалы Девятой международной конференции-семинара “Высокопроизводительные параллельные вычисления на кластерных системах”, ноябрь 2009 г., г. Владимир. – Владимир: Издательство ВлГУ, 2009, стр. 255-258.
7. Притула М.Н. Эффективный алгоритм планирования задач, допускающих параллельный счет на разном числе процессоров. Его использование в системе DVM. // Труды Международной научной конференции “Научный сервис в сети Интернет: суперкомпьютерные центры и задачи”, сентябрь 2010 г., г. Новороссийск. – М.: Изд-во МГУ, 2010, с. 679-681.

8. 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. // Супервычисления и математическое моделирование. Труды XIII Международного семинара / Под ред. Р.М. Шагалиева. – Саров: ФГУП “РФЯЦ-ВНИИЭФ”, 2012, с. 84-91.

9. Bakhtin V.A., Krukov V.A., Pritula M.N. Extension of DVM parallel programming model for clusters with heterogeneous nodes. // Research conference on information technology. Honoring volume on Pollack Mihaly faculty of engineering and information technology. Seventh international PhD & DLA symposium, октябрь 2011 г., г. Pecs, Hungary. – Komlo: Rotari Press, 2011, с. C17.

10. 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 государственного университета, серия "Математическое моделирование и программирование", №18 (277), выпуск 12 – Челябинск: Издательский центр ЮУрГУ, 2012, с. 82-92.

11. В.А. Бахтин, Н.А. Катаев, М.С. Клинов, В.А. Крюков, Н.В. Поддерюгина, М.Н. Притула. Автоматическое распараллеливание Фортран-программ на кластер с графическими ускорителями. // Параллельные вычислительные технологии (ПаВТ'2012): труды международной научной конференции (Новосибирск, 26 марта - 30 марта 2012 г.). – Челябинск: Издательский центр ЮУрГУ, 2012. с. 373-379.

12. М.А. Кривов, М.Н. Притула, С.А. Гризан, П.С. Иванов. Оптимизация приложений для гетерогенных архитектур. Проблемы и варианты решения. Информационные технологии и вычислительные системы, № 2012/03. ISSN 2071-8632. http://www.jitcs.ru/

13. В.А. Бахтин, И.Г. Бородич, Н.А. Катаев, М.С. Клинов, В.А. Крюков, Н.В. Поддерюгина, М.Н. Притула, Ю.Л. Сазанов. Распараллеливание с помощью DVM-системы некоторых приложений гидродинамики для кластеров с графическими процессорами. // Труды Международной суперкомпьютерной конференции “Научный сервис в сети Интернет: поиск новых решений”, сентябрь 2012 г., г. Новороссийск. – М.: Изд-во МГУ, 2012, с. 444-450.

14. В.А. Бахтин, М.С. Клинов, В.А. Крюков, Н.В. Поддерюгина, М.Н. Притула, Ю.Л. Сазанов. Использование языка Fortran DVMH для решения задач гидродинамики на высокопроизводительных гибридных вычислительных системах. // Параллельные вычислительные технологии (ПаВТ'2013): труды международной научной конференции (г. Челябинск, 1-5 апреля 2013 г.). – Челябинск: Издательский центр ЮУрГУ, 2013, с. 58-67.

15. М.Н. Притула. Отображение DVMH-программ на кластеры с ускорителями. // Параллельные вычислительные технологии (ПаВТ'2013): труды международной научной конференции (г. Челябинск, 1-5 апреля 2013 г.). – Челябинск: Издательский центр ЮУрГУ, 2013, с. 515-520.

16. В.А. Бахтин, М.С. Клинов, В.А. Крюков, Н.В. Поддерюгина, М.Н. Притула, А.А. Смирнов. Использование языка Fortran DVMH для решения задач гидродинамики на высокопроизводительных гибридных вычислительных системах. Вестник Южно-Уральского государственного университета, серия "Вычислительная математика и информатика", том №2, выпуск №3 – Челябинск: Издательский центр ЮУрГУ, 2013, с 106-120.

17. В.А. Бахтин, А.С. Колганов, В.А. Крюков, Н.В. Поддерюгина, М.Н. Притула. Отображение на кластеры с графическими процессорами циклов с зависимостями по данным в DVMH-программах. // Труды

Международной суперкомпьютерной конференции “Научный сервис в сети Интернет: все грани параллелизма”, сентябрь 2013 г., г. Новороссийск. – М.: Изд-во МГУ, 2013, с. 250-257.

18. Shmem Programming Manual. URL: http://staff.psc.edu/oneal/compaq/ShmemMan.pdf (дата обращения 08.11.2013).
19. Коновалов Н.А., Крюков В.А., Михайлов С.Н., Погребцов А.А. Fortran DVM – язык разработки мобильных параллельных программ // Программирование. — 1995. — № 1. — с. 49–54.
20. Коновалов Н.А., Крюков В.А., Сазанов Ю.Л. C-DVM – язык разработки мобильных параллельных программ // Программирование. — 1999. — № 1. — с. 54–65.
21. The OpenACC Application Programming Interface. URL: http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf (дата обращения 08.11.2013).
22. NAS Parallel Benchmarks. сайт.- URL: http://www.nas.nasa.gov/publications/npb.html (дата обращения 08.11.2013)
23. Seo S., Jo G., Lee J. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. // 2011 IEEE International Symposium on Workload Characterization (IISWC). — 2011. — p. 137–148.
24. Diamos G., Kerr A., Yalamanchili S., Clark N. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. // Proceedings of The 19th International Conference on Parallel Architectures and Compilation Techniques, Vienna, Austria, 11-15 September, 2010.
25. Jablin T.B., Prabhu P., Jablin J.A., Johnson N.P., Beard S.R., August D.I.. Automatic CPU-GPU communication management and optimization. // Proceedings of The International Conference for Programming Language Design and Implementation, San Jose, California, USA, 4-8 June, 2011.

26. Lee S., Eigenmann R. OpenMPC: Extended OpenMP programming and tuning for GPUs. // Proceedings of The International Conference for High Performance Computing, Networking, Storage, and Analysis, New Orleans, Louisiana, USA, 13-19 November, 2010.

27. Meng J., Skadron K. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. // Proceedings of The 23rd International Conference in Supercomputing, Yorktown Heights, NY, USA, 8-12 June, 2009.

28. Pienaar J.A., Raghunathan A., Chakradhar S. MDR: performance model driven runtime for heterogeneous parallel platforms. // Proceedings of The 25th International Conference in Supercomputing, Tucson, Arizona, USA, 31 May – 4 June, 2011.

29. Sim J., Dasgupta A., Kim H., Vuduc R. GPUPerf: A performance analysis framework for identifying potential benefits in GPGPU applications. // Proceedings of The 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New Orleans, Louisiana, USA, 25-29 February, 2012.

30. Wolfe M. Implementing the PGI accelerator model. // Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, Pittsburgh, Pennsylvania, USA, 14 March, 2010.

31. Dave C., Bae H., Min S.-J., Lee S., Eigenmann R., Midkiff S. Cetus: A source-to-source compiler infrastructure for multicores. // IEEE Computer, p. 36–42, 2009.

32. Ren M., Park J.Y., Houston M., Aiken A., Dally W.J. A tuning framework for software-managed memory hierarchies. // Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, New York, NY, USA: ACM, 2008, p. 280–291.

33. Lee J., Samadi M., Park Y., Mahlke S. Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems // Proceedings of The 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, UK, 9-11 September, 2013, p. 245–255.

34. Che S., Sheaffer J.W., Skadron K. Dymaxion: optimizing memory access patterns for heterogeneous systems. // Proceedings of The International Conference for High Performance Computing, Networking, Storage, and Analysis, Seattle, WA, USA, 14-17 November, 2011.

35. Dubach C., Cheng P., Rabbah R., Bacon D., Fink S. Compiling a high-level language for GPUs (via language support for architectures and compilers). // Proceedings of The International Conference for Programming Language Design and Implementation, Beijing, China, 11-16 June, 2012.

36. Han T.D., Abdelrahman T.S. hiCUDA: a high-level directive-based language for GPU programming. // Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, New York, NY, USA, 2009.

37. Zhang Y., Mueller F. Hidp: A hierarchical data parallel language. // 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Shenzhen, China, 23-27 February, 2013.

38. Augonnet C., Thibault S., Namyst R., Wacrenier P.-A. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. // Proceedings of the 15th International Euro-Par Conference on Parallel Processing, Euro-Par'09, Berlin, Heidelberg, Germany: Springer-Verlag, 2009, p. 863-874.

39. Ayguadé E., Badia R.M., Igual F.D., Labarta J., Mayo R., Quintana-Ortí E.S. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. // Proceedings of the 15th International Euro-Par Conference on Parallel Processing, Euro-Par'09, Berlin, Heidelberg, Germany: Springer-Verlag, 2009, p. 851-862.

Приложение 1. Описание программного интерфейса системы поддержки выполнения DVMH-программ

Система поддержки выполнения DVMH-программ является библиотекой функций, написанной на языках Си, Си++, CUDA. Она предоставляет интерфейс, пригодный для использования из программ на языках Фортран, Си, Си++ и прочих, для которых допустимо использование функций из статических библиотек.

Инициализация системы поддержки и завершение с ней работы

```
void dvmh_init_();
```

Функция **dvmh_init_** производит инициализацию структур, управляющих функционированием системы поддержки, в соответствии с заданными в файлах и переменных окружения параметрами, определяющими наборы вычислительных устройств для использования, режимы планирования выполнения, накопления статистики и трассировки и т.д. Также она инициализирует используемые вычислительные устройства.

```
void dvmh_exit_();
```

Функция **dvmh_exit_** осуществляет корректное завершение функционирования системы поддержки (возврат занятой в процессе работы памяти, запись в файлы накопленной статистической и трассировочной информации и т.д.). После данного вызова работа с LibDVMH возможна только после вызова **dvmh_init_**.

Запросы актуализации данных в основной памяти

Актуализация заключается в следующем:

1. Выяснить, какая часть переменной (из запрашиваемой части) не является актуальной в основной памяти.
2. Скопировать эту часть в основную память с тех устройств, на которых будет найдены необходимые части в актуальном состоянии (возможно, с нескольких по “крупицам”)

```
void dvmh_get_actual_subvariable_(void *addr,
DvmType lowIndex[], DvmType highIndex[]);
```

addr – адрес начала расположения в основной памяти переменной, актуализация которой запрашивается.

lowIndex – массив, **i**-ый элемент которого указывает нижнюю границу запрашиваемой части переменной-массива по **(i+1)**-ому измерению. Может быть указана константа -2147483648, что означает открытую нижнюю границу.

highIndex – массив, **i**-ый элемент которого указывает верхнюю границу запрашиваемой части переменной-массива по **(i+1)**-ому измерению. Может быть указана константа -2147483648, что означает открытую верхнюю границу.

Открытая граница означает продолжение до конца имеющегося у данного процесса части пространства переменной-массива (локальная часть процесса + теневые грани).

Данная функция проводит актуализацию для части переменной-массива (подмассива).

```
void dvmh_get_actual_variable_(void *addr);
```

addr – адрес начала расположения переменной в основной памяти.

Данная функция проводит актуализацию для переменной целиком (локальная часть процесса + теневые грани).

`void dvmh_get_actual_subarray_(DvmType dvmDesc[], DvmType lowIndex[], DvmType highIndex[]);`

dvmDesc – заголовочный массив DVM-массива, актуализация которого запрашивается.

lowIndex – массив, **i**-ый элемент которого указывает нижнюю границу запрашиваемой части переменной-массива по **(i+1)**-ому измерению. Может быть указана константа -2147483648, что означает открытую нижнюю границу.

highIndex – массив, **i**-ый элемент которого указывает верхнюю границу запрашиваемой части переменной-массива по **(i+1)**-ому измерению. Может быть указана константа -2147483648, что означает открытую верхнюю границу.

Открытая граница означает продолжение до конца имеющегося у данного процесса части пространства переменной-массива (локальная часть процесса + теневые грани).

Данная функция проводит актуализацию для части переменной-массива (подмассива).

`void dvmh_get_actual_array_(DvmType dvmDesc[]);`

dvmDesc – заголовочный массив DVM-массива, актуализация которого запрашивается.

Данная функция проводит актуализацию для всего массива целиком (локальная часть процесса + теневые грани).

`void dvmh_get_actual_all_();`

Данная функция проводит актуализацию для всех данных, известных (когда-либо зарегистрированных в регионе) системе поддержки выполнения DVMH-программ.

Объявление актуальности данных в основной памяти

Операция объявления актуальности – это информирование системы поддержки выполнения о намерении изменить (или о факте произошедшего изменения) во внерегионном пространстве в основной памяти значение переменной. В случае отсутствия в программе асинхронных регионов, данные операции можно осуществлять постфактум.

```
void dvmh_actual_subvariable_(void *addr, DvmType lowIndex[], DvmType highIndex[]);
```

addr – адрес начала расположения в основной памяти переменной, объявление актуальности которой осуществляется.

lowIndex – массив, **i**-ый элемент которого указывает нижнюю границу части переменной-массива по **(i+1)**-ому измерению. Может быть указана константа -2147483648, что означает открытую нижнюю границу.

highIndex – массив, **i**-ый элемент которого указывает верхнюю границу части переменной-массива по **(i+1)**-ому измерению. Может быть указана константа -2147483648, что означает открытую верхнюю границу.

Открытая граница означает продолжение до конца локальной части пространства переменной-массива данного процесса.

Данная функция объявляет актуальность для части переменной-массива (подмассива).

```
void dvmh_actual_variable_(void *addr);
```

addr – адрес начала расположения переменной в основной памяти.

Данная функция объявляет актуальность для переменной целиком (локальная часть процесса).

```
void dvmh_actual_subarray_(DvmType dvmDesc[],
DvmType lowIndex[], DvmType highIndex[]);
```

dvmDesc – заголовочный массив DVM-массива, объявление актуальности которого осуществляется.

lowIndex – массив, **i**-ый элемент которого указывает нижнюю границу части переменной-массива по **(i+1)**-ому измерению. Может быть указана константа -2147483648, что означает открытую нижнюю границу.

highIndex – массив, **i**-ый элемент которого указывает верхнюю границу части переменной-массива по **(i+1)**-ому измерению. Может быть указана константа -2147483648, что означает открытую верхнюю границу.

Открытая граница означает продолжение до конца локальной части пространства переменной-массива данного процесса.

Данная функция объявляет актуальность для части переменной-массива (подмассива).

```
void dvmh_actual_array_(DvmType dvmDesc[]);
```

dvmDesc – заголовочный массив DVM-массива, объявление актуальности которого осуществляется.

Данная функция объявляет актуальность для всего массива целиком (локальная часть процесса).

```
void dvmh_actual_all_();
```

Данная функция объявляет актуальность для всех данных, известных (когда-либо зарегистрированных в регионе) системе поддержки выполнения DVMH-программ.

Следует заметить, что все неизвестные LibDVMH данные считаются актуальными в основной памяти.

Вспомогательные функции для поддержки DVM-директив

```
void dvmh_remote_access_(DvmType dvmDesc[]);
```

dvmDesc – заголовочный массив буфера удаленных элементов, заполненного одной из функций LibDVM: **crtrbl_**, **crtrba_**, **crtrbp_**.

Данная функция обеспечивает актуальное состояние для массива, к которому будет осуществляться удаленный доступ в той части, к которой будет осуществляться удаленный доступ. Данную функцию следует вызывать непосредственно перед функцией **loadrb_** из LibDVM после создания буфера удаленных элементов.

```
void dvmh_shadow_renew_(ShadowGroupRef *group);
```

***group** – теневая группа, созданная функцией **crtshg_**.

Данная функция обеспечивает актуальное состояние для граничных элементов входящих в теневую группу массивов, а также ведет учет актуальности состояния теневых элементов. Данную функцию следует вызывать непосредственно перед **strtsh_**. Для асинхронных обменов данная функция выполняется так же – перед **strtsh_**.

```
void dvmh_redistribute_(DvmType dvmDesc[], DvmType
*newValueFlagRef);
```

dvmDesc – заголовочный массив DVM-массива, который подвергается перераспределению.

***newValueFlagRef** – признак, говорящий о том, что значение элементов перераспределяемых распределенных массивов не нуждается в сохранении.

Данная функция обеспечивает сохранение актуальных данных в случае ***newValueFlagRef**==0, а также подготавливает систему поддержки к новым размерам локальных частей массивов. Затрагиваются все массивы, которые выровнены на то представление абстрактной машины, которое перераспределяется. В качестве аргумента можно указать любой распределенный массив, выровненный на перераспределяемое представление абстрактной машины.

Данную функцию следует вызывать непосредственно перед **redis_** или аналогами.

```
void dvmh_realign_(DvmType dvmDesc[], DvmType *newValueFlagRef);
```

dvmDesc – заголовочный массив DVM-массива, который подвергается перевыравниванию.

***newValueFlagRef** – признак, говорящий о том, что значение элементов массива не нуждается в сохранении.

Данная функция обеспечивает сохранение актуальных данных в случае ***newValueFlagRef**==0, а также подготавливает систему поддержки к новым размерам локальной части массива. Затрагивается только массив, указанный в параметре.

Данную функцию следует вызывать непосредственно перед **realn_** или аналогами.

Уничтожение переменных

```
void dvmh_destroy_variable_(void *addr);
```

addr – адрес начала расположения переменной в основной памяти.

Данная функция обеспечивает освобождение всех ресурсов, связанных с заданной переменной. Также такая переменная становится неизвестной для LibDVMH после данного вызова.

Вызывать этот метод следует непосредственно перед deallocate/free/delete для динамических переменных, а также по концу процедуры для локальных переменных.

```
void dvmh_destroy_array_(DvmType dvmDesc[]);
```

dvmDesc – заголовочный массив DVM-массива, над которым производится действие.

Данная функция обеспечивает освобождение всех ресурсов, связанных с заданной переменной. Также такая переменная становится неизвестной для LibDVMH после данного вызова.

Вызывать этот метод следует непосредственно перед deallocate/free/delete для динамических переменных, а также по концу процедуры для локальных переменных.

Сервисные функции для вызова из обработчиков

```
void *dvmh_get_device_addr(DvmType *deviceRef, void *variable);
```

***deviceRef** – номер используемого вычислительного устройства (см. **loop_get_device_num_**).

variable – адрес начала расположения переменной в основной памяти.

Данная функция возвращает адрес на устройстве для заданной переменной.

DvmType dvmh_calculate_offset_(DvmType *deviceRef, void *base, void *variable);

***deviceRef** – номер используемого вычислительного устройства (см. **loop_get_device_num_**).

base – адрес базы, от которой вычисляется смещение.

variable – адрес начала расположения переменной в основной памяти.

Данная функция возвращает смещение (в элементах) адреса представителя заданной скалярной переменной или обычного массива на заданном устройстве от заданной базы.

void *dvmh_get_natural_base(DvmType *deviceRef, DvmType dvmDesc[]);

***deviceRef** – номер используемого вычислительного устройства (см. **loop_get_device_num_**).

dvmDesc – заголовочный массив DVM-массива, для которого вычисляется естественная база.

Вычисляет естественную базу для DVM-массива. Это такая база, при использовании которой смещение равно нулю.

void dvmh_fill_header_(DvmType *deviceRef, void *base, DvmType dvmDesc[], DvmType dvmhDesc[]);

***deviceRef** – номер используемого вычислительного устройства (см. **loop_get_device_num_**).

base – адрес базы, от которой будет производиться индексация массива на устройстве.

dvmDesc – заголовочный массив DVM-массива, для которого запрашивается информация для индексирования представителя.

dvmhDesc – массив, который будет заполнен коэффициентами и смещением для индексирования представителя.

Заполняет урезанный DVM-заголовочный массив для представителя на заданном устройстве с использованием указанной базы.

```
void dvmh_fill_header_ex_(DvmType *deviceRef, void *base, DvmType dvmDesc[], DvmType dvmhDesc[], DvmType *outTypeOfTransformation, DvmType extendedParams[]);
```

***deviceRef** – номер используемого вычислительного устройства (см. **loop_get_device_num_**).

base – адрес базы, от которой будет производиться индексация массива на устройстве.

dvmDesc – заголовочный массив DVM-массива, для которого запрашивается информация для индексирования представителя.

dvmhDesc – массив, который будет заполнен коэффициентами и смещением для индексирования представителя.

***outTypeOfTransformation** – выходной параметр для извещения о текущем виде представителя массива на устройстве. 0 – нет трансформации, 1 – перестановка измерений, 2 – перестановка измерений и диагонализация.

extendedParams – выходной параметр, семиэлементный массив, в который записывается дополнительная информация, необходимая для индексирования представителя в диагонализированном виде. Заполняется только в случае диагонализированного вида представителя.

Заполняет урезанный DVM-заголовочный массив для представителя на заданном устройстве с использованием указанной базы.

`void dvmh_cuda_replicate_(void *addr, DvmType *recordSize, DvmType *quantity, void *devPtr);`

addr – адрес местонахождения эталонного значения в памяти ЦПУ.

***recordSize** – размер элемента массива в байтах.

***quantity** – количество заполняемых элементов массива в памяти ГПУ.

devPtr – адрес в памяти ГПУ, по которому располагается заполняемый массив.

Заполняет массив в памяти ГПУ значением элемента, заданным в памяти ЦПУ.

Функции работы с вычислительными регионами

`DvmhRegionRef region_create_(DvmType *flagsRef);`

***flagsRef** – параметры создания региона. Является побитовым объединением признака асинхронности и признака сравнительной отладки. Значение 1 – признак асинхронности, значение 2 – признак сравнительной отладки.

Создание региона.

`void region_register_subarray_(DvmhRegionRef *regionRef, DvmType *intentRef, DvmType dvmDesc[], DvmType lowIndex[], DvmType highIndex[]);`

***regionRef** – ссылка на регион, которую вернул вызов **region_create_**.

***intentRef** – регистрируемое направление использования переменной в регионе (1 – in, 2 – out, 4 – local, 3 – inout, 5 - inlocal).

dvmDesc – заголовочный массив DVM-массива, регистрируемого в регионе.

lowIndex – массив, **i**-ый элемент которого указывает нижнюю границу части переменной-массива по **(i+1)**-ому измерению. Может быть указана константа -2147483648, что означает открытую нижнюю границу.

highIndex – массив, **i**-ый элемент которого указывает верхнюю границу части переменной-массива по **(i+1)**-ому измерению. Может быть указана константа -2147483648, что означает открытую верхнюю границу.

Открытая граница означает продолжение до конца локальной части пространства переменной-массива данного процесса.

Регистрация подмассива в регионе с указанием атрибутов направления его использования. Регистрация необходима для создания представителей переменной на ускорителях, для проведения необходимых для выполнения региона перемещений данных, учета производимых регионом изменений над переменными.

```
void region_register_array_(DvmhRegionRef
*regionRef, DvmType *intentRef, DvmType dvmDesc[]);
```

***regionRef** – ссылка на регион, которую вернул вызов **region_create_**.

***intentRef** – регистрируемое направление использования переменной в регионе (1 – in, 2 – out, 4 – local, 3 – inout, 5 - inlocal).

dvmDesc – заголовочный массив DVM-массива, регистрируемого в регионе.

Регистрация массива целиком в регионе с указанием атрибутов направления его использования. Регистрация необходима для создания представителей переменной на ускорителях, для проведения необходимых для выполнения региона перемещений данных, учета производимых регионом изменений над переменными.

`void region_register_scalar_(DvmhRegionRef *regionRef, DvmType *intentRef, void *addr, DvmType *sizeRef, DvmType *varType);`

***regionRef** – ссылка на регион, которую вернул вызов **region_create_**.

***intentRef** – регистрируемое направление использования переменной в регионе (1 – in, 2 – out, 4 – local, 3 – inout, 5 - inlocal).

addr – адрес регистрируемой скалярной переменной в памяти ЦПУ.

***sizeRef** – размер скалярной переменной в байтах.

***varType** – тип скалярной переменной (-1 – неизвестный, 0 – char, 1 – int, 2 – long, 3 – float, 4 – double, 5 – float_complex, 6 – double_complex, 7 – logical)

Регистрация скалярной переменной в регионе с указанием атрибутов направления ее использования. Регистрация необходима для создания представителей переменной на ускорителях, для проведения необходимых для выполнения региона перемещений данных, учета производимых регионом изменений над переменными.

`void region_set_name_array_(DvmhRegionRef *regionRef, DvmType dvmDesc[], const char *name, int nameLength);`

***regionRef** – ссылка на регион, которую вернул вызов **region_create_**.

dvmDesc – заголовочный массив ранее зарегистрированного в регионе DVM-массива, для которого устанавливается имя.

name – символьная строка с именем переменной.

nameLength – длина в байтах символьной строки **name**.

Приписывание переменной имени в заданном регионе.

`void region_set_name_variable_(DvmhRegionRef *regionRef, void *addr, const char *name, int nameLength);`

***regionRef** – ссылка на регион, которую вернул вызов **region_create_**.

addr – адрес ранее зарегистрированной в регионе скалярной переменной, для которой устанавливается имя.

name – символьная строка с именем переменной.

nameLength – длина в байтах символьной строки **name**.

Приписывание переменной имени в заданном регионе.

`void region_prepared_for_devices_(DvmhRegionRef *regionRef, DvmType *devicesRef);`

***regionRef** – ссылка на регион, которую вернул вызов **region_create_**.

***devicesRef** – побитовое объединение типов устройств, где ЦПУ=1, CUDA=2.

Оповещение о наборе типов устройств, для которых подготовлен регион. В этом вызове происходит распределение данных по устройствам, выделение памяти для представителей переменных на устройствах, выполнение актуализации представителей для зарегистрированных с атрибутом **in** переменных.

`void region_handle_consistent_(DvmhRegionRef *regionRef, DAConsistGroupRef *group);`

***regionRef** – ссылка на регион, которую вернул вызов **region_create_**.

***group** – ссылка на группу приведения к консистентному состоянию.

Обработка восстановления консистентности. Данную функцию необходимо вызывать перед **strtcg_**, только внутри региона.

```
void region_after_waitrb_(DvmhRegionRef *regionRef,
DvmType dvmDesc[]);
```

***regionRef** – ссылка на регион, которую вернул вызов **region_create_**.

dvmDesc – заголовочный массив буфера удаленных элементов.

Создание представителей на устройствах региона для буфера удаленных элементов, а также заполнение их актуальными данными. Данную функцию следует вызывать после **waitrb_** перед каким-либо использованием буфера удаленных элементов в регионе.

```
void region_destroy_rb_(DvmhRegionRef *regionRef,
DvmType dvmDesc[]);
```

***regionRef** – ссылка на регион, которую вернул вызов **region_create_**.

dvmDesc – заголовочный массив буфера удаленных элементов.

Данная функция оповещает LibDVMH о завершении работы с буфером удаленных элементов.

```
void region_end_(DvmhRegionRef *regionRef);
```

***regionRef** – ссылка на регион, которую вернул вызов **region_create_**.

Данная функция завершает работу с регионом. После ее вызова ссылка на данный регион становится недействительной.

Функции работы с параллельными циклами и последовательными участками внутри регионов

`DvmhLoopRef loop_create_(DvmhRegionRef *regionRef, LoopRef *InDvmLoop);`

***regionRef** – ссылка на объемлющий данный цикл регион, которую вернул вызов **region_create_**.

***InDvmLoop** – ссылка на DVM-описатель параллельного цикла, которую вернул вызов **crtpl_**. В случае последовательного участка в качестве данного аргумента задается нуль.

Создает описатель параллельного DVMH-цикла, ссылка на который используется в дальнейших вызовах. Здесь и далее последовательный участок считается частным случаем параллельного цикла.

`void loop_insred_(DvmhLoopRef *InDvmhLoop, RedRef *InRedRefPtr);`

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которую вернул **loop_create_**.

***InRedRefPtr** – ссылка на описатель редукционной функции, включенной в данный параллельный цикл. Эту ссылку следует получить от вызова **crtrdf_**.

Функция для включения редукции в параллельный DVMH цикл.

`void loop_across_(DvmhLoopRef *InDvmhLoop, ShadowGroupRef *oldGroup, ShadowGroupRef *newGroup);`

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которую вернул **loop_create_**.

***oldGroup** – ссылка на так называемую старую теневую группу. Данная теневая группа обновляется перед выполнением цикла.

***newGroup** – ссылка на так называемую новую теневую группу. Данная теневая группа обновляется в ходе вычисления параллельного цикла.

Функция для оповещения LibDVMH о теневых группах, созданных для обработки указания **ACROSS** в директиве параллельного цикла. Т.к. вызов **across_** исходные теневые группы уничтожает, то в данный вызова следует передавать их копии.

```
void loop_set_cuda_block_(DvmhLoopRef *InDvmhLoop,
DvmType *InXRef, DvmType *InYRef, DvmType *InZRef);
```

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которую вернул **loop_create_**.

***InXRef** – размер CUDA-блока по координате X.

***InYRef** – размер CUDA-блока по координате Y.

***InZRef** – размер CUDA-блока по координате Z.

Функция для установки заданной программистом конфигурации блока CUDA-нитей.

```
void loop_shadow_compute_(DvmhLoopRef *InDvmhLoop,
DvmType dvmDesc[]);
```

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которую вернул **loop_create_**.

dvmDesc – заголовочный массив DVM-массива, который изменяется в данном цикле.

Функция для уведомления о том, какой массив изменяется при SHADOW_COMPUTE при данном параллельном цикле в регионе.

Вызывается для всех изменяемых в цикле массивов. Если в цикле изменяются все выходные и локальные массивы региона (т.е. все те, которые зарегистрированы с атрибутами **out** или **local**), то можно или сообщить о них всех, или вообще не сообщать – тогда примется консервативная позиция и будут считаться изменяемыми по SHADOW_COMPUTE все **out** и **local** массивы.

```
void loop_register_handler_(DvmhLoopRef
*InDvmhLoop, DvmType *deviceTypeRef, DvmType *flagsRef,
GenFunc f, DvmType *basesCount, DvmType *paramCount,
...);
```

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которую вернул **loop_create_**.

***deviceTypeRef** – тип устройства, для которого годится данный обработчик, где ЦПУ=1, CUDA=2.

***flagsRef** – битовое множество флагов. Флаг 1 означает, что обработчик является параллельным и для него на усмотрение системы поддержки будет выделено несколько (1 или более) слотов исполнения для устройства. Такой обработчик должен сам запросить у системы поддержки свое количество слотов (функция **loop_get_slot_count_**). Флаг 2 означает, что обработчик может исполняться только в основной нити выполнения программы (в частности это влечет за собой тот факт, что других его инстанций в параллель запущено не будет).

f – ссылка на функцию, являющуюся обработчиком для заданного цикла на устройствах заданного типа.

***basesCount** – количество передаваемых из LibDVMH баз.

***paramCount** – количество дополнительных аргументов для передачи в функцию-обработчик. Все эти дополнительные аргументы задаются в

конце списка фактических аргументов и должны передаваться исключительно по адресу.

Функция с переменным числом аргументов, предназначенная для регистрации обработчика для параллельного цикла. Для обработчиков задается тип устройства, для которого он может быть использован, его возможности и ограничения в виде флагов.

```
void loop_perform_(DvmhLoopRef *InDvmhLoop);
```

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которую вернул **loop_create_**.

Команда начала исполнения цикла. Если цикл находится в синхронном регионе, то в этом вызове будет происходить и ожидание конца цикла.

```
DvmType loop_get_device_num_(DvmhLoopRef
*InDvmhLoop);
```

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

Вызов для получения номера устройства, на котором исполняет текущий обработчик текущую порцию цикла.

```
DvmType loop_has_element_(DvmhLoopRef *InDvmhLoop,
DvmType dvmDesc[], DvmType indexArray[]);
```

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

dvmDesc – заголовочный массив DVM-массива, для которого проверяется принадлежность элемента локальной части.

indexArray – массив глобальных индексов элемента.

Данный вызов является аналогом **tstelm_** для вызова в обработчиках. Возвращает 1, если элемент принадлежит локальной части массива на данном устройстве в данном регионе. Возвращает 0 иначе. Индексы глобальные.

`void loop_fill_bounds_(DvmhLoopRef *InDvmhLoop, IndexType lowIndex[], IndexType highIndex[], IndexType stepIndex[]);`

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

lowIndex – массив, в который будут записаны начальные значения.

highIndex – массив, в который будут записаны конечные значения.

stepIndex – массив, в который будут записаны шаги.

Запрос на заполнение границ многомерного цикла (и шагов) для исполнения порции цикла обработчиком.

`void loop_fill_local_part_(DvmhLoopRef *InDvmhLoop, DvmType dvmDesc[], IndexType part[]);`

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

dvmDesc – заголовочный массив DVM-массива, для которого запрашиваются границы локальной части.

part – выходной массив, в который будут записаны начальные и конечные индексы локальной для текущего региона на текущем устройстве части.

Запрос на заполнение массива, описывающего локальную часть массива на текущем устройстве в текущем регионе.

`void loop_red_init_(DvmhLoopRef *InDvmhLoop, DvmType *InRedNumRef, void *arrayPtr, void *locPtr);`

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

***InRedNumRef** – номер редукционной переменной в порядке их регистрации через **loop_insred_**.

arrayPtr – ссылка на локальную переменную, редукционный массив.

locPtr – ссылка на локальную переменную, loc-массив.

Запрос на заполнение редукционного массива и сопутствующего ему loc-массива начальными данными для последующего накопления редукции.

`DvmType loop_get_slot_count_(DvmhLoopRef *InDvmhLoop);`

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

Запрос на получение количества слотов, выделенных для исполнения порции цикла в данном запуске данного обработчика

`DvmType loop_get_dependency_mask_(DvmhLoopRef *InDvmhLoop);`

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

Запрос маски зависимых измерений параллельного цикла. Младший разряд соответствует внутреннему циклу. 1 – есть зависимость, 0 – нет зависимости.

`void loop_cuda_register_red(DvmhLoopRef *InDvmhLoop, DvmType InRedNum, void **ArrayPtr, void **LocPtr);`

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

InRedNum – номер редукционной переменной в порядке их регистрации через **loop_insred_**.

arrayPtr – ссылка на локальную переменную для хранения ссылки на редукционный массив в памяти ГПУ.

locPtr – ссылка на локальную переменную для хранения ссылки на loc-массив в памяти ГПУ.

Функция для регистрации переменных, в которые будут помещены (в процессе итерирования цикла по **loop_cuda_do** или вызовом **loop_cuda_red_prepare_**) адреса для редукционной переменной и ее loc-массива в памяти ГПУ.

`void loop_cuda_red_init_(DvmhLoopRef *InDvmhLoop, DvmType InRedNum, void *arrayPtr, void *locPtr, void **devArrayPtr, void **devLocPtr);`

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

InRedNum – номер редукционной переменной в порядке их регистрации через **loop_insred_**.

arrayPtr – ссылка на локальную переменную, редукционный массив.

locPtr – ссылка на локальную переменную, loc-массив.

devArrayPtr – ссылка на локальную переменную для хранения ссылки на редукционный массив в памяти ГПУ.

devLocPtr – ссылка на локальную переменную для хранения ссылки на loc-массив в памяти ГПУ.

Альтернативная **loop_red_init_** функция для запроса заполнения редукционного массива и сопутствующего ему loc-массива начальными данными, которая в отличие от **loop_red_init_** кроме всего прочего создает массивы с начальными значениями в памяти ГПУ.

```
void loop_cuda_red_prepare_(DvmhLoopRef *InDvmhLoop, DvmType *InRedNumRef, DvmType *InCountRef, DvmType *InFillFlagRef);
```

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

***InRedNumRef** – номер редукционной переменной в порядке их регистрации через **loop_insred_**.

***InCountRef** – количество экземпляров редукционной переменной.

***InFillFlagRef** – признак, необходимо ли заполнение начальными значениями выделенных переменных.

Выделение памяти на заданное количество редукционных переменных (возможно, с инициализацией) для последующего проведения редукции.

```
CudaIndexType *loop_cuda_get_local_part(DvmhLoopRef *InDvmhLoop, DvmType dvmDesc[]);
```

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

dvmDesc – заголовочный массив DVM-массива, для которого запрашиваются границы локальной части.

Запрос на заполнение и выдачу массива, описывающего локальную часть массива на текущем устройстве в текущем регионе. Возвращает адрес массива в памяти ГПУ, в который записаны начальные и конечные индексы локальной для текущего региона на текущем устройстве части.

```
DvmType loop_cuda_autotransform_(DvmhLoopRef
*InDvmhLoop, DvmType dvmDesc[]);
```

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

dvmDesc – заголовочный массив DVM-массива, для которого запрашивается трансформация.

Выполнить наилучшую для данного цикла трансформацию для данного массива для графической платы. Возвращает тип произведенной трансформации: 0 – нет трансформации, 1 – перестановка измерений, 2 – подиагональная трансформация.

```
void loop_cuda_get_config_(DvmhLoopRef *InDvmhLoop,
DvmType *InSharedPerThread, DvmType *InRegsPerThread,
dim3 *InOutThreads, cudaStream_t *OutStream, DvmType
*OutSharedPerBlock);
```

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

***InSharedPerThread** – входной параметр, размер используемой разделяемой памяти в байтах в расчете на одну нить.

***InRegsPerThread** – входной параметр, количество используемых регистров в расчете на одну нить.

***InOutThreads** – выходной параметр, конфигурация CUDA-блока.

***OutStream** – выходной параметр, поток выполнения.

***OutSharedPerBlock** – выходной параметр, размер разделяемой памяти в расчете на блок.

Функция для запрашивания конфигурации для запуска ядер в данной порции цикла.

```
DvmType loop_cuda_do(DvmhLoopRef *InDvmhLoop, dim3
*OutBlocks, CudaIndexType **InOutBlocksInfo);
```

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

***OutBlocks** – выходной параметр, будет содержать конфигурацию решетки блоков для запуска ядра.

***InOutBlocksInfo** – выходной параметр, в него будет записан адрес на устройстве, где располагается информация по отображению цикла на блоки.

Вызов для итерирования вызовов ядер на ГПУ. Возвращает 1, если необходим запуск ядра после этого вызова и 0, если более ядро не запускать.

```
DvmType loop_cuda_get_overall_blocks_(DvmhLoopRef
*InDvmhLoop);
```

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

Возвращает общее количество CUDA-блоков для выполнения текущей порции цикла.

`void loop_red_finish_(DvmhLoopRef *InDvmhLoop, DvmType *InRedNumRef);`

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

***InRedNumRef** – номер редукционной переменной в порядке их регистрации через **loop_insred_**.

Запрос завершения редукции, которую ранее зарегистрировали и для которой был запрошен буфер с помощью **loop_cuda_red_prepare_**. После данного вызова не делать **loop_red_post_**, так как он выполнится изнутри данного вызова.

`void loop_red_post_(DvmhLoopRef *InDvmhLoop, DvmType *InRedNumRef, void *arrayPtr, void *locPtr);`

***InDvmhLoop** – ссылка на описатель параллельного DVMH-цикла, которая была передана в обработчик.

***InRedNumRef** – номер редукционной переменной в порядке их регистрации через **loop_insred_**.

arrayPtr – ссылка на локальную переменную, редукционный массив.

locPtr – ссылка на локальную переменную, loc-массив.

Вызов для возврата результатов частичной редукции в LibDVMH.