Методы динамической настройки DVMH-программ на кластеры с ускорителями*

В.А. Бахтин[1,2], А.С. Колганов[1,2], В.А. Крюков[1,2], Н.В. Поддерюгина[1], М.Н. Притула[1]

Институт Прикладной Математики им. М.В. Келдыша Российской академии наук[1],
Московский государственный университет имени М.В. Ломоносова[2]

DVM-система предназначена для разработки параллельных программ научно-технических расчетов на языках C-DVMH и Fortran-DVMH. Эти языки используют единую модель параллельного программирования (DVMH-модель) и являются расширением стандартных языков Си и Фортран спецификациями параллелизма, оформленными в виде директив компилятору. DVMH-модель позволяет создавать эффективные параллельные программы для гетерогенных вычислительных кластеров с ускорителями. При использовании DVMH-модели программист не использует явные операции копирования данных, расположенных в памяти центрального процессора(ЦПУ) или ускорителей. Для фрагментов программы (регионов), которые могут выполняться на ускорителях, он указывает входные и выходные данные, а также те данные, которые изменяются или используются вне регионов. Это позволяет динамически выбирать устройства, на которых регион будет выполняться, распределять работу между устройствами с учетом их производительности, многократно выполнять регионы для подбора оптимальной конфигурации. В статье демонстрируется влияние перечисленных методов на эффективность выполнения некоторых тестов (из пакета NAS NPB) и реальных приложений.

1. Введение

В последнее время все большее распространение получают вычислительные кластеры, в узлах которых установлены ускорители различной архитектуры [1]. Данная тенденция заметно усложняет процесс программирования, так как требует освоения на достаточном уровне сразу нескольких технологий программирования (MPI, Pthreads, CUDA или OpenCL). С целью упрощения использования ускорителей, были предложены модели программирования и соответствующие им директивные расширения языков такие, как OpenACC [2], OpenMP 4.x [3] и DVMH [4].

Модель программирования DVMH позволяет разрабатывать параллельные программы для кластеров, в узлах которых помимо универсальных многоядерных процессоров установлены ускорители компании NVidia (далее ГПУ) и сопроцессоры Intel Xeon Phi. Модель поддерживает использование всех перечисленных архитектур, как по отдельности, так и одновременно в рамках одной программы.

Одним из важных аспектов функционирования такой программной модели, как DVMH, является вопрос эффективного 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.

В статье рассматриваются некоторые оптимизации, которые применяются во время выполнения DVMH-программы на кластерах с ускорителями.

2. Модель DVMH. Схема выполнения DVMH-программы

Под DVMH-программой будем понимать программу на языке Fortran-DVMH [5] или программу на языке C-DVMH.

* Исследование выполнено при финансовой поддержке грантов РФФИ № 13-07-00580, 14-01-00109.

При написании DVMH-программы программист представляет целевую ЭВМ в виде многомерной решетки виртуальных процессоров, на которые будут распределяться данные и вычисления. От размерности этой решетки (количества ее измерений) зависит количество распределяемых измерений массивов. Конкретные размеры каждого измерения решетки могут не указываться в программе, а задаваться при ее запуске. В качестве виртуальных процессоров служат MPI-процессы. При этом на один узел кластера может быть отображен один или несколько MPI-процессов, каждый из которых, в свою очередь, использует многоядерные процессоры и ускорители узла.

В программе сначала требуется определить массивы, которые должны быть распределены между виртуальными процессорами (распределенные данные). При этом распределение может производиться блоками разного размера, учитывая различия времен при вычислении разных элементов массива, а также различия в производительности виртуальных процессоров. Остальные переменные (распределяемые по умолчанию) отображаются по одному экземпляру на каждый виртуальный процессор (размноженные данные).

Распределение вычислений производится посредством их отображения на распределенные массивы.

Модель позволяет задать несколько уровней параллелизма:

- параллельное выполнение вычислений (фрагментов программы, оформленных в виде параллельных задач) на разных секциях решетки виртуальных процессоров. Например, это используется при реализации многоблочных методов;
- параллельное выполнение распределенных циклов - выполнение витков тесно-гнездового цикла виртуальными процессорами задачи. Правила распределения витков циклов между ними задаются при объявлении распределенного цикла;
- параллельное выполнение циклов на общей памяти - выполнение витков тесно-гнездового цикла на вычислительных устройствах, используемых MPI-процессом.

Параллельное выполнение циклов возможно и при наличии зависимости между витками цикла. Это может быть регулярная зависимость по элементам массива, зависимость по редукционным переменным, зависимость по приватизируемым переменным. Все такие зависимости должны быть специфицированы.

Виртуальный процессор может быть гетерогенным - состоять из мультипроцессора и ускорителей. При этом каждый ускоритель имеет свою оперативную память. На ускорителях могут выполняться только указанные программистом регионы - специальным образом оформленные фрагменты кода с одним входом и одним выходом, состоящие из последовательных участков и параллельных циклов.

Выполнение DVMH-программы начинается с выполнения начальной задачи сразу на всех виртуальных процессорах. Эта задача может запустить группу дочерних подзадач, которые будут выполняться на выделенных для них секциях решетки виртуальных процессоров. После завершения выполнения всех запущенных дочерних задач начальная задача продолжает свое выполнение и может запустить ту же самую или другую группу дочерних задач. При выполнении любой задачи могут быть запущены регионы, которые начинают выполняться на выделенном им подмножестве ускорителей или на мультипроцессоре. Выполнение задачи до входа в регион и после выхода из него производится на мультипроцессорах. Витки распределенных циклов задачи распределяются между выделенными для этой задачи виртуальными процессорами. 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. Для каждого устройства выбирается метод обработки части цикла, называемый обработчиком, а также оптимизационные параметры для выбранного обработчика. Осуществляется динамическая настройка оптимизационных параметров, включая количество нитей для обработчиков на мультипроцессоре, 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 уст-

ройстве. 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 выполнении на ускорителях.

Такая схема выполнения DVMH-программы становится возможной, т.к. в программе отсутствуют явные операции копирования данных из памяти ЦПУ в память ускорителей (и обратно). При использовании DVMH-модели программист для каждого региона указывает, какие данные являются входными и выходными, а также определяет данные, которые изменяются или используются вне регионов. Это позволяет динамически изменять устройства, на которых выполняется регион, распределять работу между устройствами с учетом их производительности и многократно выполнять регионы для подбора оптимальной конфигурации. Это важное свойство DVMH-модели, которое выгодно отличает ее от моделей OpenACC и OpenMP 4.x.

3. Методы динамической настройки DVMH-программ

Одним из достоинств DVM-системы является то, что получаемые параллельные программы могут динамически настраиваться при запуске на выделенные для их выполнения ресурсы (количество узлов кластера, ядер, ускорителей и их производительность).

Далее рассматриваются основные методы динамической настройки DVMH-программ, которые используются в DVM-системе.

3.1 Отображение подзадач на узлы кластера с учетом их производительности

В модели DVM [6-7] для каждой подзадачи пользователь явно определял множество виртуальных процессоров, на которых она должна выполняться (**Рис. 1**).

```
! описание массива виртуальных процессоров
!DVM$ PROCESSORS P(NUMBER_OF_PROCESSORS( ))
! описание массива задач
!DVM$ TASK MB (2)
! определение числа виртуальных процессоров
NP = NUMBER_OF_PROCESSORS( ) / 2
! явное отображение задач по виртуальным процессорам
!DVM$ MAP MB( 1 ) ONTO P( 1 : NP )
!DVM$ MAP MB( 2 ) ONTO P( NP+1 : 2*NP )
```

Рис. 1. Параллельные задачи в модели DVM

В результате такого отображения каждая из двух подзадач будет выполняться на секции, содержащей половину всех узлов кластера.

Таким образом, при использовании DVM-модели распределение подзадач производилось вручную, и оно зачастую было затруднено вследствие:

- большого количества подзадач;
- заметного разброса их сложности;
- необходимости осуществлять распределение на различное количество узлов.

Модель DVMH лишена данного недостатка. В системе поддержки выполнения DVMH-программ реализован алгоритм автоматического распределения виртуальных процессоров между подзадачами.

В языки C-DVMH и Fortran-DVMH были добавлены конструкции для поддержки автоматического распределения подзадач. От программиста требуется указать относительные сложности подзадач, минимальное и максимальное число виртуальных процессоров для каждой подзадачи, а также параметры зависимости времени выполнения каждой подзадачи от числа выделенных ей виртуальных процессоров (доли параллельных вычислений в общем объеме вычислений для задачи).

При запуске многоблочной DVMH-программы 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.

3.2 Отображение массивов и циклов на узлы кластера с учетом их производительности

Одним из возможных вариантов использования кластеров с сопроцессорами Intel Xeon Phi является т.н. «симметричный режим». В этом режиме центральный процессор и сопроцессор можно рассматривать как отдельные вычислительные узлы кластера. Одна и та же программа компилируется отдельно для ЦПУ и отдельно для сопроцессора. Скомпилированные программы одновременно запускаются на сопроцессоре и ЦПУ и могут синхронизироваться с помощью технологии MPI.

При использовании данного режима необходимо сбалансировать нагрузку MPI-процессов, которые выполняются на различных устройствах. Для MPI-программ, как правило, эта проблема решается за счет реализации еще одного уровня параллелизма с использованием нитей (Pthreads или OpenMP). На ЦПУ и сопроцессоре запускается разное число MPI-процессов, массивы распределяются между MPI-процессами равными порциями, но за счет использования различного числа нитей удается сбалансировать их нагрузку.

Такой механизм балансировки можно использовать и для DVMH-программ. Существуют переменные окружения, которые позволяют задать число MPI-процессов, запускаемых на различных узлах кластера и число нитей для разных MPI-процессов:

```
export DVMH_PPN='2,1,1' # Number of processes per node
export DVMH_NUM_THREADS='8,8,240,240' # Number of CPU threads per process
```

Кроме того, в DVM-системе реализованы возможности для задания производительностей процессоров (или их автоматического определения при запуске программы) и их учета при распределении данных и вычислений между узлами кластера. Эта возможность была реализована еще в 2002 году и позволяла DVM-программам эффективно выполняться на неоднородных кластерах [8].

3.3 Отображение массивов и циклов на устройства узла с учетом их производительности

В настоящее время в DVMH-программах данные распределяются блочно: каждое распределённое измерение делится несколькими точками на отрезки. При этом есть возможность по каждому измерению распределённого массива задавать или равномерное блочное распределение, или распределение взвешенными блоками, т.е. с учётом 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. Статический режим с использованием подобранной схемы распределения.

Рассмотрим подробнее эти режимы распределения.

В простом статическом режиме в каждом регионе распределение производится одинаково. При помощи переменных окружения пользователь задаёт вектор весов вычислительных устройств, имеющихся в каждом узле кластера (или они могут быть грубо определены автоматически), затем эти веса накладываются на параметры внешнего распределения данных. В таком режиме сводятся к минимуму перемещения данных в связи с их перераспределением, но не

учитывается различное соотношение производительности вычислительных устройств на разных участках кода.

В основе режимов подбора лежит предположение об итерационности программы, т.е. повторении одних и тех же вычислений (в плане производимых операций, а не вычисляемых значений) достаточно большое количество раз. В динамическом режиме с подбором схемы распределения в каждом регионе распределение выбирается на основе постоянно пополняющейся истории запусков данного региона и его соседей.

В процессе работы DVMH-программы ведут учет времени обработки всех параллельных циклов на используемых устройствах, поддерживая в актуальном состоянии табличную функцию зависимости производительности (в витках в секунду) от объема вычислений (в витках). После каждого выполнения цикла эта зависимость уточняется - новое измерение заменяет старые, находящиеся в заданной окрестности. Так, при достаточно плавном (во времени) изменении веса витка, функция останется актуальной, по крайней мере, в окрестности точек, соответствующих текущим запускам цикла.

Наличие и накопление этой информации позволяет оптимизировать веса распределения.

Формально задача ставится так:

Есть набор вычислительных устройств D1,D2,D3,...,DN.

Есть набор циклов L1,L2,L3,...,LM.

Для каждого цикла известно типичное количество витков, количество вхождений, зависимость времени выполнения на конкретном устройстве для заданного количества витков цикла.

Требуется найти вектор весов W1,W2,...,WN, sum{Wi}=1.0, такой, что он минимизирует функционал времени T=sum{Li.execCount * max{Li.Time(Dj, Li.typicalPart * Wj); 1 <= j <= N}; 1 <= i <= M}, где execCount - количество выполнений данного цикла; typicalPart - усредненное количество витков цикла; Time - фунция времени от вычислительного устройства и количества витков.

Данная задача решается приближенно модификацией алгоритма градиентного спуска.

Построенная схема распределения может быть сохранена в файл и использована при последующих запусках программы. В качестве начального приближения может быть использован вектор весов вычислительных устройств, как и для простого статического режима.

Из данного режима возможен переход в третий режим в любой точке выполнения программы - статический режим с использованием подобранной схемы распределения. В этом режиме в каждом регионе распределение выбирается на основе предоставленной схемы распределения, построенной при работе программы во втором режиме, причем есть возможность, как перейти в этот режим непосредственно из второго, так и использовать схему распределения из файла, полученного в результате работы программы во втором режиме. При использовании схемы распределения из файла не гарантируется ее корректное применение 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).

3.4 Трансформация массивов

Одним из преимуществ DVMH-программ является то, что обращения к массивам в параллельных циклах, исполняемых на ГПУ, программируются в обобщенном виде, позволяющем во время выполнения иметь иной порядок измерений в массиве, нежели было задано в исходной программе. Для циклов с регулярными зависимостями по данным также поддерживается более сложная реорганизация уже на уровне элементов массива - диагонализация.

Данная особенность дает еще одну степень свободы для оптимизации во время выполнения - выбор представления массива при выполнении конкретного цикла.

Для реализации этих возможностей система поддержки ведет учет текущего представления массива на всех вычислительных устройствах в виде перестановки измерений и наличия и способа диагонализации. Диагонализация бывает двух видов - параллельно побочной диагонали или параллельно главной диагонали.

Перед выполнением каждого цикла анализируется его правило отображения и сопоставляется с правилами выравнивания использующихся в нем массивов. Эта информация позволяет

связать измерения цикла с измерениями массива и вывести оптимальный для данного цикла вид представления массива в памяти вычислительного устройства. После того, как оптимальный вид определен, происходит переход из текущего представления в новое по правилам:

1. Если текущий вид не тот же, что и новый, то:

1.1. Если текущий вид диагонализирован, то раздиагонализировать его => обновленный текущий вид.

1.2. Если текущий вид имеет иную перестановку измерений, чем требуется, то вычислить частное перестановок (разность) и применить к текущему представлению => обновленный текущий вид.

1.3. Если новый вид требует диагонализацию, то применить требуемую диагонализацию к текущему представлению => обновленный текущий вид.

2. Конец. Текущий вид приведен к новому требуемому.

Значок «=>» означает, что в результате данного действия текущий вид представления массива изменяется, и следующие шаги уже применяются к нему в качестве текущего вида.

Столь гибкий механизм автоматической реорганизации массивов в памяти вычислительных устройств позволяет значительно ускорить некоторые циклы. Например, при реализации метода переменных направлений, в которой массив в памяти ГПУ располагается на протяжении всего времени счета одинаково (неважно с каким порядком измерений), один из циклов будет работать примерно в 10 раз дольше остальных из-за невозможности объединения обращений к глобальной памяти, запрошенных нитями одного варпа. Если же массив во время выполнения сможет изменять свое представление в памяти, то удастся все циклы такой программы сделать одинаково быстрыми.

3.5 Использование динамической компиляции

Данный механизм позволяет компилировать CUDA-обработчики, сгенерированные компиляторами с языков Fortran-DVMH или C-DVMH во время выполнения программы.

Для реализации такой возможности была использована новая библиотека NVidia Run Time Compilation (NVRTC) [9], которая позволяет компилировать код для ГПУ на узлах кластера в момент выполнения программы. Библиотека доступна в CUDA Toolkit 7.0.

NVRTC - это библиотека динамической компиляции для CUDA-C++ программ. Она принимает исходный CUDA-код в виде строки символов и создает специальные объекты, которые в дальнейшем используются для создания PTX-кода и загрузки полученного кода на ГПУ.

Данный подход может способствовать увеличению производительности, так как все происходит в момент выполнения программы, когда многие данные уже известны, что является невозможным при статической компиляции.

CUDA-программистам для использования данного подхода требуется преобразовать CUDA-ядра в символьные строки, дописать компиляцию и вызов ядер с помощью низкоуровневого CUDA Driver API [10], что существенно усложняет и так не простую отладку ГПУ кода, ухудшает удобство его разработки и дальнейшее сопровождение.

В DVMH-компиляторе данная возможность реализована в системе поддержки выполнения DVMH-программ. Для включения данной возможности в DVMH-компиляторы была добавлена оптимизационная опция -rtc. При этом сохраняется возможность использовать как старый вариант программы (без динамической компиляции), так и новый вариант программы (с динамической компиляцией), меняя опции компиляции.

Можно отметить следующие преимущества использования динамической компиляции.

1) В момент выполнения программы становятся известны значения скалярных переменных (типов int, double, long и др.). Тем самым, перед компиляцией CUDA-ядра можно выполнить преобразование кода следующим образом: вместо передаваемого параметра, например, int param_1, объявить в теле CUDA-ядра переменную const int param_1 = dyn_value, где dyn_value - известное в момент компиляции значение данной переменной. В результате происходит сокращение используемых ГПУ ресурсов (например, регистров), что может привести к увеличению производительности данного ядра.

2) Во время статической компиляции необходимо указывать конкретную архитектуру ГПУ (например, Fermi, Kepler, Maxwell). Это может негативно сказываться на производительности при использовании разных поколений ГПУ в одном узле, так как старые поколения не поддерживают новые возможности современных ГПУ. При использовании динамической компиляции в DVMH-компиляторе такой проблемы нет, так как компиляция происходит под целевое ГПУ (учитывая архитектурные особенности) в момент выполнения.
3) DVMH-компилятор генерирует несколько CUDA-ядер и CUDA-обработчиков для выполнения параллельного цикла. Во время статической компиляции необходимо скомпилировать все CUDA-ядра вне зависимости от того, будут они выполняться или нет. Во время динамической компиляции будут скомпилированы только те из них, которые необходимо выполнить на ГПУ. Тем самым, может сократиться время компиляции программы.

4. Использование методов динамической настройки DVMH-программ

Для оценки влияния различных оптимизаций, применяемых во время выполнения DVMH-программ на кластерах с ускорителями, использовались DVMH-версии тестов NAS (тесты BT, EP, SP и LU) из пакета NPB 3.3.1 [11] и реальные приложения пользователей (программы Композит, Каверна и Контейнер).

Программа Композит предназначена для моделирования процесса распределения тепла в композитном материале. Программы Каверна и Контейнер [12] решают задачу о течении несжимаемой жидкости или слабосжимаемого газа около прямоугольной выемки, в которой в качестве исходной математической модели используется гиперболический вариант квазигазодинамической системы (двумерная и трехмерная постановка задачи).

Тестирование производилось на сервере MIC с установленными на нем 6-ядерным (12-поточным) процессором Intel Xeon E5-1660v2 с 48Гб оперативной памяти типа DDR3, сопроцессором Intel Xeon Phi 5110P с 8Гб оперативной памяти типа GDDR5 и ГПУ NVidia GTX Titan с 6Гб оперативной памяти типа GDDR5, а также гибридном вычислительном комплексе K-100 [13].

4.1 Отображение подзадач на узлы кластера с учетом их производительности

Для демонстрации данной возможности использовалась программа Композит, разработанная на языке Fortran-DVMH. В данном приложении расчетная физическая область разбита на 36 блоков, для каждого из которых введена прямоугольная сетка различной подробности, так как требуемая детализация расчетов различна в разных блоках расчетной области. Вычислительная сложность обработки одного блока имеет разброс более чем в 30 раз. Эффективность распараллеливания данного приложения была исследована на кластере K-100. Использовалось 1, 2, 4, 8, 12, 24, 36, 48, 60, 72, 84, 96, 108, 120, 132, 144 процессоров.

Исследование эффективности распараллеливания показало:

1. При каждом шаге увеличения числа процессоров приложение ускоряется, в том числе на большем по сравнению с количеством подзадач числе процессоров.
2. При использовании 4-х процессоров происходит ускорение всего лишь в 2.4 раза. Эффективность распараллеливания на 4 процессора - 60%.
3. Начиная с 8 процессоров эффективность распараллеливания (по отношению к результату на 8 процессорах) не падает ниже 87%. По сравнению с последовательным вариантом - не ниже 47%.
4. Максимальное достигнутое ускорение - 68 раз по сравнению с последовательной программой.

Полученные результаты говорят о том, что механизм распределения подзадач, реализованный в DVM-системе, справляется с задачей ускорения приложения даже при количестве процессоров, в четыре раза превосходящем число подзадач.

4.2 Отображение массивов и циклов на узлы кластера с учетом их производительности

На **Рис. 2** показано ускорение теста EP на сервере MIC по сравнению с последовательной версией данной программы, выполненной на одном ядре ЦПУ. Данный тест выполнялся на разных архитектурах по отдельности, а также в следующих комбинациях: ЦПУ + ГПУ, ЦПУ + сопроцессор и сопроцессор + ЦПУ + ГПУ. Красным и сиреневым цветом показаны случаи, когда дополнительно использовалась балансировка нагрузки путем задания соотношения весов всех ядер ЦПУ и ГПУ и соотношения весов MPI-процессов, отображаемых на ЦПУ и сопроцессор.

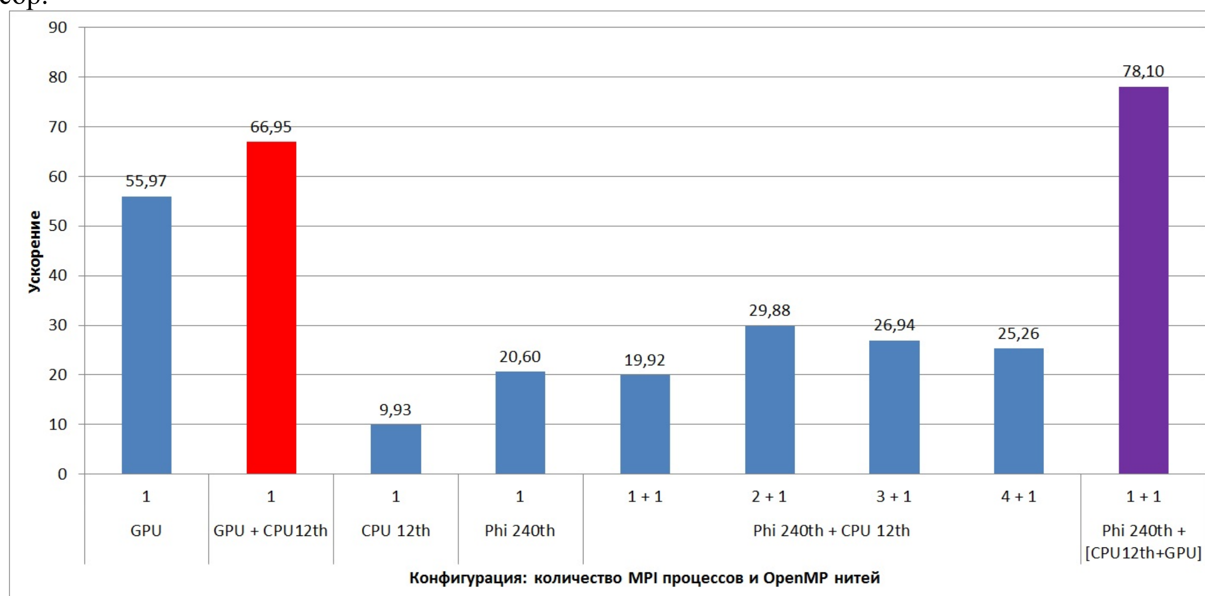

Рис. 2. Использование балансировки нагрузки в DVMH-программе на примере теста EP класса C

В итоге, на данном тесте при одновременном использовании ГПУ и ЦПУ удалось достичь производительности, которая на 17% больше производительности выполнения теста на одном ГПУ.

При совместном использовании сопроцессора и ЦПУ удалось достичь производительности, которая на 30% больше производительности выполнения теста на одном сопроцессоре. Наиболее выгодное соотношение MPI-процессов следующее: два MPI-процесса на Xeon Phi и один MPI-процесс на Xeon E5, при этом MPI-процессы имеют одинаковый вес, либо по одному MPI-процессу на ЦПУ и сопроцессор, и веса процессов соотносятся как 1 : 2 соответственно.

Гибкий механизм балансировки, реализованый в DVM-системе, позволил получить при при одновременном использовании всех устройств узла производительности, которая на 28% больше производительности выполнения теста EP на ГПУ и в 3.7 раз превышает производительность выполнения этого теста на сопроцессоре.

4.3 Отображение массивов и циклов на устройства узла с учетом их производительности

Как уже отмечалось, в системе поддержки выполнения DVMH-программ реализовано несколько режимов распределения данных и вычислений по вычислительным устройствам. Выбор того или иного режима осуществляется при помощи переменной окружения DVMH_SCHED_TECH. Наличие такой возможности позволяет автоматически находить наилучшую схему отображения и существенно ускорить выполнение DVMH-программ за счет использования всех ресурсов узла. В таблице 1 показаны времена выполнения программ Каверна (1600x1600, 200 итераций) и Контейнер (150x150x150, 100 итераций) на сервере MIC, вычислительном комплексе К-100 и тестовом сервере, на котором используется «бюджетная» графиче-

ская карта. Использовался динамический режим с подбором схемы распределения (DVMH_SCHED_TECH=2).

Таблица 1. Автоматическое распределение работы между устройствами узла

Программа	Время выполнения, сек		Совмещение работы (ГПУ+ЦПУ)			
	ГПУ	ЦПУ	Вес ЦПУ	Вес ГПУ	Время1, с	Время2, с
Сервер MIC	GTX Titan	E5-1660 (6 нитей)				
Каверна	13.37	41.91	0.256	0.743	12.81	12.43
Контейнер	27.44	60.12	0.268	0.732	23.30	22.72
K-100	C2050	X5670 (12 нитей)				
Каверна	16.87	9.76	0.350	0.649	15.21	14.72
Контейнер	30.12	56.24	0.291	0.708	28.31	24.42
Тестовый сервер	GTX 550Ti	i7-980X (12 нитей)				
Каверна	45.59	55.87	0.532	0.468	31.21	30.85
Контейнер	72.25	78.61	0.462	0.537	45.23	43.88

Во время работы DVMH-программы для всех параллельных циклов на всех используемых устройствах строится табличная функция зависимости производительности (в витках в секунду) от объема вычислений (в витках). Фрагмент данной таблицы приведен на **Рис. 3**.

```
    Performance statistics for parallel loop at cavity.fdv(601):
        Device #0:
            Handler #0:
                Best parameters: threads=11
                Table function (iterations => performance):
                          894400 => 2.21928e+08
                           1e+06 => 2.20961e+08
                        2.56e+06 => 2.22738e+08
        Device #1:
            Handler #0:
                Best parameters: thread-block=(32,8,1)
                Table function (iterations => performance):
                        1.56e+06 => 6.3996e+08
                      1.6656e+06 => 6.44597e+08
                        2.56e+06 => 6.74815e+08
    Simple dynamic run performances:
    DVMH_CPU_PERF='0.35016'
    DVMH_CUDAS_PERF='0.64984'
```

Рис. 3. Зависимость производительности от объема вычислений

Наличие и накопление такой информации позволяет найти оптимимальные веса распределения. Веса ЦПУ и ГПУ, указанные в таблице 2, были найдены автоматически. Выбор оптимальной схемы распределения потребовал для данных программ выполнения 3-7 итераций. Время1 – это время работы программы на узле при одновременном использовании всех ядер ЦПУ и одного ГПУ в динамическом режиме подбора. Время2 – это время работы программы, когда автоматически найденные веса распределения использовались при повторном запуске программы с самого начала.

Использование ядер ЦПУ позволило ускорить выполнение программы Контейнер в 1.2 раза (в 1.6 раза на старой ГПУ) по сравнению с запуском только на ГПУ.

4.4 Трансформация массивов

На **Рис. 4** показано ускорение выполнения тестов BT, SP и LU на 1 ГПУ при использовании механизма автоматической реорганизации массивов в памяти вычислительных устройств. Ис-

пользование опции компиляции –autoTfm, включающей данный режим, позволяет существенно ускорить выполнение программ (например, для теста LU класса C на архитектуре Kepler было получено ускорение более чем в 2.7 раза, чем при выполнении программы без такой реорганизации). Следует отметить, что для выполнения трансформации массивов может потребоваться дополнительная память на ГПУ, а если такая реорганизация выполняется очень часто, то накладные расходы на ее выполнение могут превысить эффект от данной оптимизации, поэтому данный механизм не используется по умолчанию.

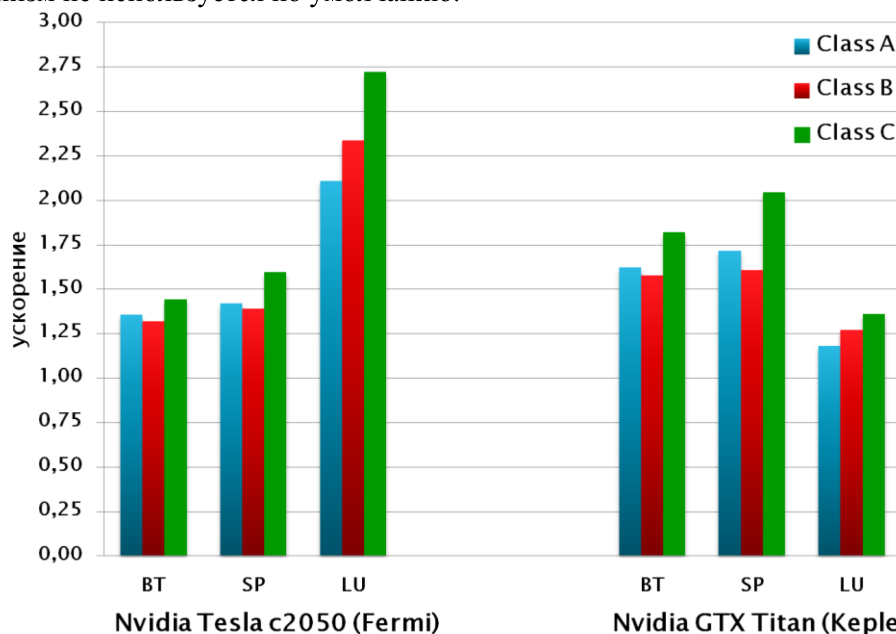

Рис. 4. Ускорение выполнения тестов NAS в результате автоматической реорганизации массивов в памяти

4.5 Использование динамической компиляции

Преимущества динамической компиляции можно предомонстрировать с помощью тестов SP и LU. Использовать данную оптимизацию выгодно в итерационных методах, когда один и тот же параллельный цикл может быть выполнен много раз. При первом выполнении цикла, в системе поддержки выполнения DVMH-программ происходит настройка и компиляция соответствующего ему CUDA-ядра. При повторном выполнении данного цикла, в случае совпадения параметров CUDA-ядра, компиляция не производится.

Таблица 2. Характеристики запуска теста SP

	Количество регистров			Время выполнения, ms		
	Было	Стало	Сокращение	Было	Стало	Ускорение
compute_rhs	125	82	52%	74	66.5	11%
x_solve	153	112	37%	64.5	42.9	50%
y_solve	132	107	23%	64.4	42.6	51%
z_xolve	128	107	20%	53	48	10%

Таблица 3. Характеристики запуска теста LU

	Количество регистров			Время выполнения, ms		
	Было	Стало	Сокращение	Было	Стало	Ускорение
Loop ACROSS+	148	110	35%	0.22	0.14	57%
Loop ACROSS–	141	110	28%	0.22	0.14	57%
Loop	120	89	35%	125.4	126.5	-1%

В таблицах 2 и 3 показаны результаты работы процедур без применения динамической компиляции и с её применением. Программы компилировались с использованием CUDA

Toolkit 7.0 для архитектуры Kepler. Для теста SP использовалась расчетная сетка 250*250*250, а для LU - 270*270*270, которые занимали примерно по 4ГБ памяти ГПУ каждая.

Видно, что в результате сокращения количества используемых регистров можно получить прирост производительности до 50%. В отдельных случаях отмечается и замедление программы. Это может быть связано с тем, что текущая версия библиотеки NVRTC является экспериментальной. Стоит отметить, что для теста LU первые два цикла имеют зависимости по данным и для выполнения одного параллельного цикла происходит многократный запуск одного и того же обработчика. В таблице 3 указано среднее время выполнения такого обработчика (на 20 итераций было сделано примерно 20800 запусков данного цикла).

В результате данной оптимизации общая производительность всей программы SP увеличилась на 28%, а программы LU - на 36%.

Выводы

Система автоматизации разработки параллельных программ (DVM-система) существенно упрощает процесс разработки параллельных программ для гибридных вычислительных кластеров. Получаемые DVMH-программы без каких-либо изменений могут эффективно выполняться на кластерах различной архитектуры, использующих многоядерные универсальные процессоры, графические ускорители и сопроцессоры Intel Xeon Phi. Это достигается как за счет различных оптимизаций, которые выполняются статически, при компиляции DVMH-программ [14-16], так и за счет динамических оптимизаций, которые были рассмотрены в данной статье.

Параллельные программы могут динамически настраиваться при запуске на выделенные для их выполнения ресурсы (количество узлов кластера, ядер, ускорителей и их производительность). Такое свойство программ позволяет запускать их на произвольной конфигурации (число процессоров, нитей, ускорителей), компоновать сложные программы из имеющихся простых программ, а также повысить эффективность использования параллельных систем коллективного пользования за счет более гибкого распределения ресурсов между отдельными программами.

Литература

1. Top500 List - November 2014. URL: http://top500.org/list/2014/11/ (дата обращения 12.06.2015).
2. OpenACC. URL: http://www.openacc-standard.org/ (дата обращения 12.06.2015).
3. OpenMP 4.0 Specifications. URL: http://openmp.org/wp/openmp-specifications/ (дата обращения 12.06.2015).
4. Бахтин, В.А. Расширение DVM-модели параллельного программирования для кластеров с гетерогенными узлами / В.А. Бахтин, М.С. Клинов, В.А. Крюков, Н.В. Поддерюгина, М.Н. Притула, Ю.Л. Сазанов // Вестник Южно-Уральского государственного университета, серия: «Математическое моделирование и программирование».- 2012. - №18(277). - С. 82-92.
5. Описание языка Fortran-DVMH. URL: http://dvm-system.org/static_data/tutorial/FDVMH-tutorial.pdf (дата обращения 12.06.2015).
6. Коновалов, Н.А. Fortran DVM - язык разработки мобильных параллельных программ / Н.А. Коновалов, В.А. Крюков, С.Н. Михайлов, А.А. Погребцов // Программирование. - 1995. - № 1. - С. 49-54.
7. Коновалов, Н.А. C-DVM - язык разработки мобильных параллельных программ / Н.А. Коновалов, В.А. Крюков, Ю.Л. Сазанов // Программирование. - 1999. - № 1. - С. 20-28.
8. Крюков, В.А. Разработка параллельных программ для вычислительных кластеров и сетей / В.А. Крюков // Информационные технологии и вычислительные системы. – 2003. - № 1-2. -

С. 42-61. URL: ftp://ftp.keldysh.ru/dvm-distr/journ1-2_page42_61.pdf (дата обращения 12.06.2015).

9. NVIDIA CUDA Runtime Compilation Library. URL: http://docs.nvidia.com/cuda/nvrtc/index.html (дата обращения 12.06.2015).
10. NVIDIA CUDA Driver API. URL: http://docs.nvidia.com/cuda/cuda-driver-api/index.html (дата обращения 12.06.2015).
11. NAS Parallel Benchmarks. URL: http://www.nas.nasa.gov/publications/npb.html (дата обращения 12.06.2015).
12. Бахтин В.А. Расширение DVM-модели паралельного программирования для кластеров с гетерогенными узлами / В. А. Бахтин, В.А Крюков, Б.Н. Четверушкин, Е.В. Шильников // Доклады Академии наук. - 2011. - Т. 441, № 6, декабрь. - С. 734-736.
13. Гибридный вычислительный кластер К-100. URL: http://www.kiam.ru/MVS/resourses/k100.html (дата обращения 12.06.2015).
14. Бахтин, В.А. Отображение на кластеры с графическими процессорами DVMH-программ с регулярными зависимостями по данным / В.А. Бахтин, А.С. Колганов, В.А. Крюков, Н.В. Поддерюгина, М.Н. Притула // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. 2013. Т. 2. No 4. С. 44-56.
15. Алексахин, В.Ф. Распараллеливание на графические процессоры тестов NAS NPB3.3.1 на языке Fortran DVMH / В.Ф. Алексахин, В.А. Бахтин, О.Ф. Жукова, А.С. Колганов, В.А. Крюков, Н.В. Поддерюгина, М.Н. Притула, О.А. Савицкая, А.В. Шуберт // Вестник Уфимского государственного авиационного технического университета.- 2015. - Т. 19, №1(67). - С. 240-250.
16. Алексахин, В.Ф. Распараллеливание на языке Fortran-DVMH для сопроцессора Intel Xeon Phi тестов NAS NPB3.3.1 / В.Ф. Алексахин, В.А. Бахтин, О.Ф. Жукова, А.С. Колганов, В.А. Крюков, И.П. Островская, Н.В. Поддерюгина, М.Н. Притула, О.А. Савицкая // Сборник трудов Международной научной конференции «Параллельные вычислительные технологии 2015», Челябинск: Издательский центр ЮУрГУ, 2015, С. 19-30.

Dynamic tuning methods of DVMH-programs for clusters with accelerators

Vladimir Bakhtin, Alexander Kolganov, Victor Krukov, Natalya Podderugina and Mikhail Pritula

Keywords: DVMH, high-level programming language, accelerator, coprocessor, GPU, MIC, NAS Parallel Benchmarks

DVM system is being developed at KIAM RAS with active participation of students and postgraduate students of CMC department of Lomonosov MSU. DVM system is designed for computational parallel program development in C-DVMH and Fortran-DVMH languages. These languages are directive-based extensions of standard C and Fortran languages and use a common parallel programming model (DVMH model).

DVMH model enables creation of effective parallel programs (DVMH programs) for heterogeneous computational clusters exploiting multicore CPUs as well as various accelerators (GPUs and/or Intel Xeon Phi coprocessors).

When using DVMH model, the programmer does not use explicit copy operation of data located in CPU or accelerators memory. For fragments of programs that can be executed on accelerators (regions), the programmer specifies the input and output data, and the data that is modified or used outside the region. This allows to dynamically select the devices that can be used for region execution, distribute the work between the devices in accordance with their performance, repeatedly perform the regions for the selection of the optimal configuration. The article demonstrates the effect of these methods on the effectiveness of some tests (NAS NPB Benchmarks), and real applications.