УДК 519.688

ПРИМЕНЕНИЕ ПЛАНИРОВЩИКА ДЛЯ ЭФФЕКТИВНОГО ОБМЕНА ДАННЫМИ НА СУПЕРКОМПЬЮТЕРАХ ГИБРИДНОЙ АРХИТЕКТУРЫ С МАССИВНО-ПАРАЛЛЕЛЬНЫМИ УСКОРИТЕЛЯМИ

П. Б. Богданов[1], А. А. Ефремов[1], А. В. Горобец[2], С. А. Суков[2]

Предложена оптимизация обменов данными в рамках многоуровневой параллельной модели на основе MPI, OpenMP и OpenCL, предназначенная для расчетов на различных типах современных суперкомпьютерных архитектур, включая гибридные системы на графических процессорах (GPU) и ускорителях Intel Xeon Phi. Представлен упрощающий гетерогенную реализацию планировщик задач, управляющий потоками вычислительных и коммуникационных заданий OpenCL и использующий представление вычислительной задачи в виде графа исполнения вычислительных подпрограмм, процедур передачи данных и связей между ними. С применением планировщика реализована коммуникационная схема для моделирования на гибридных суперкомпьютерах задач газовой динамики методом конечного объема на неструктурированных сетках. Описана реализация MPI и CPU–GPU-обменов с одновременным выполнением вычислений и передачи данных, приведены показатели полученной параллельной эффективности. Работа выполнена при поддержке РФФИ (коды проектов 12–01–33022 и 12–01–00486).

Ключевые слова: газовая динамика, планировщик, параллельные вычисления, GPU, OpenCL, MPI, OpenMP.

1. Введение. Современные гибридные (гетерогенные) суперкомпьютерные архитектуры основываются на применении массивно-параллельных ускорителей в качестве математических сопроцессоров в дополнение к центральным процессорам. К таким ускорителям, например, относятся графические процессоры GPU (Graphics Processing Unit) производства AMD и NVIDIA и ускорители Intel Xeon Phi существенно многоядерной архитектуры MIC (Many Integrated Core). Для вычислительных экспериментов на таких гибридных системах хорошо подходит многоуровневая параллельная модель, сочетающая различные типы параллелизма.

На первом уровне используется MPI (Message Passing Interface) для объединения множества вычислительных модулей (узлов) суперкомпьютера в рамках модели с распределенной памятью. Расчетная область, представленная неструктурированной гибридной сеткой, разбивается на подобласти, что соответствует традиционному геометрическому параллелизму.

На втором уровне для распределения работы между вычислительными ресурсам узла, процессорами и ускорителями может использоваться дальнейшая декомпозиция подобласти и тот же MPI (в этом случае обмены данными в подгруппах внутри узлов происходят через копирование в общей памяти узла, что значительно быстрее, чем обмены по сети между узлами). Кроме того, может использоваться OpenMP (Open MultiProcessing) в рамках модели с общей памятью.

На третьем уровне посредством OpenCL (Open Computing Language) задействуются массивно-параллельные ускорители узла. Кроме того, OpenCL может успешно использоваться и для вычислений на CPU (Central Processing Unit). Таким образом, второй уровень может быть сведен лишь к дальнейшей декомпозиции подобласти, а все ресурсы задействуются одним управляющим процессом, выполняющим вычислительные задания на всех доступных устройствах — процессорах и ускорителях.

Сложность такой вычислительной модели обусловлена двумя факторами. Во-первых, адаптация алгоритма к архитектуре ускорителя и оптимизация использования его ресурсов представляет собой достаточно сложную задачу. Во-вторых, эта многоуровневая модель требует более сложной коммуникационной схемы, включающей в себя двухуровневый обмен данными: между CPU и ускорителями и MPI-обмены между узлами суперкомпьютера.

[1] Научно-исследовательский институт системных исследований РАН (НИИСИ РАН), Нахимовский просп., д. 36, корп. 1, 117218, г. Москва; П. Б. Богданов, науч. сотр., e-mail: bogdanov@niisi.msk.ru; А. А. Ефремов, мл. науч. сотр., e-mail: antonyef@mail.ru

[2] Институт прикладной математики РАН им. М. В. Келдыша (ИПМ РАН), Миусская пл., 4а, 125047, Москва; А. В. Горобец, ст. науч. сотр., e-mail: cherepock@gmail.com; С. А. Суков, ст. науч. сотр., e-mail: ssoukov@gmail.com

В настоящее время существует несколько подходов, призванных скрыть низкоуровневые интерфейсы программных моделей типа CUDA или OpenCL, избавить разработчика от близкой к "железу" программной рутины и позволить сосредоточиться непосредственно на решении задачи. В этих подходах можно выделить два направления: модели, в основе которых (по аналогии с OpenMP) лежат специальные директивы, и продвинутые логические модели с концепцией исполняемого потока задач и трактов передачи данных. К первому типу можно отнести такие разработки, как OpenACC (Open ACCelerator) [1] и OpenHMPP (Open Hybrid Multicore Parallel Programming) [2], а ко второму типу — StarPU [3], CGCM [4], Distri-GPU [5].

В обоих подходах разработчику предоставлены механизмы управления асинхронной передачей данных. Следует отметить, что приведенные выше директивные подходы являются проприетарными и лоббируются группой поддерживающих стандарт компаний как новый стандарт гетерогенного программирования. С одной стороны, у директивных подходов есть один существенный плюс — старые коды можно потенциально ускорить за счет расстановки специальных директив без существенного изменения исходного кода. С другой стороны, все механизмы работы директивных программных моделей скрыты от разработчика, а получаемый выигрыш, как правило, невелик. Это ставит под вопрос итоговую эффективность получаемых кодов. Некоторые примеры использования директивных подходов приведены в [6–8]. Из результатов, представленных в этих работах, видно, что максимальное достигаемое ускорение на флагманском продукте компании nVidia Tesla K20X против пары мощных серверных процессоров составляет около трех раз. По нашим оценкам, этот показатель можно увеличить еще как минимум в полтора раза при использовании низкоуровневого API (Application Programming Interface). Тем не менее, такой результат "из коробки" при использовании директив весьма неплох, и в будущем, при существенной доработке средств компилирования, этот инструмент можно будет всерьез рассматривать как замену низкоуровневых моделей.

Модели типа "планировщик задач" выглядят более прозрачно и понятно, но при этом подразумевают декомпозицию исходной задачи на составляющие операции и представление задачи в виде графа, описывающего исполнение вычислительных подпрограмм, процедур передачи данных и связи между ними. Каждую вершину и ребро в графе программы можно оценить явно, что дает еще одну степень свободы для тонкой оптимизации исполнения алгоритма на конкретной целевой платформе.

Существенным недостатком такого рода моделей является их слабая применимость при ускорении старых кодов, ибо для использования планировщиков потребуется выделить в имеющемся коде, как правило устаревшем с точки зрения подходов программирования, основные вычислительноемкие процедуры и определить между ними связи по данным. Часто это равносильно проектированию исходного кода заново. С примерами использования концепции планировщиков при разработке программного обеспечения под гибридные вычислительные комплексы можно ознакомиться в работах [9–11]. Как видно из приведенных материалов, каждый разработчик представляет свою достаточно нетривиальную модель планировщика и сопутствующих абстракций. Тем не менее, это позволяет задействовать все вычислители на узле, обеспечивая при этом сокрытие передачи данных за вычислениями там, где это необходимо.

С учетом вышеизложенного, был выбран второй подход, для реализации которого было необходимо проработать общую логическую модель программирования узла произвольной аппаратной архитектуры, где каждое вычислительное устройство доступно в рамках программного интерфейса OpenCL, который позволяет получить доступ ко всем типам вычислителей в гибридной системе.

В настоящей статье представлена реализация программной инфраструктуры гетерогенных вычислений в рамках стандарта OpenCL. Реализованная инфраструктура планировщика представляет собой специальную службу системного уровня, которая обеспечивает балансировку нагрузки по имеющимся на узле вычислительным устройствам и сокрытие передачи данных за вычислениями. Стратегия одновременного выполнения передачи данных и вычислений на ускорителях, так называемое перекрытие обменов и вычислений (overlap), позволяет существенно повысить параллельную эффективность.

Вся нетривиальная работа по обслуживанию вычислений на гибридном вычислительном узле может обеспечиваться библиотекой планировщика автоматически, разработчику остается только формализовать свою задачу в виде графа исполнения, вершинами которого являются команды загрузки/выгрузки данных и выполнения счетных процедур, а ребра соответствуют зависимостям между этими командами.

Наша работа, в частности, сосредоточена на многоуровневой коммуникационной схеме с одновременным выполнением передачи данных и вычислений на ускорителях. Параллельная инфраструктура обмена данными демонстрируется на примере конечно-объемного алгоритма на неструктурированных гибридных сетках для моделирования сжимаемых течений. Этот иллюстративный алгоритм использует схему первого порядка аппроксимации по пространству, вследствие чего имеет крайне низкую вычислительную стои-

мость на шаг по времени, тем самым максимально усложняя задачу эффективного распараллеливания в рамках модели с распределенной памятью. Поскольку вес обменов данными в общем времени вычислений достаточно высок, этот алгоритм является хорошим примером для оптимизации схемы обменов данными. Конечно же, коммуникационный алгоритм предназначен для схем повышенного порядка аппроксимации как на основе полиномиальной реконструкции, так и на основе квазиодномерной реконструкции, где будет гарантированно иметь более высокую параллельную эффективность, чем в рассматриваемом модельном примере. Кроме того, представленный алгоритм обменов данными может применяться и для конечно-объемных и конечно-разностных методов на структурированных сетках.

Далее статья структурирована следующим образом. В разделе 2 подробно описана реализация и функционал инфраструктуры планировщика вычислительных заданий. Раздел 3 посвящен многоуровневому распараллеливанию с использованием представленного планировщика на примере газодинамического алгоритма на неструктурированных сетках. Там кратко описывается рассматриваемая математическая модель сжимаемого течения и численный алгоритм, представлено описание реализации обменов данными с перекрытием, приводятся полученные в тестах показатели параллельной эффективности и масштабируемости. В разделе 4 сформулированы основные результаты и выводы.

2. Инфраструктура планировщика. Концепция планировщика основывается на понятиях *регистр* — фрагмент в глобальной памяти ускорителя, *команда* — произвольное вычислительное задание и *программа* — множество команд, дополненное графом зависимостей между ними, на основе которого работает планировщик. Таким образом, реализуется некая виртуальная машина, которая выполняет команды над регистрами. Планировщик управляет разнородными вычислительными ресурсами узла, распределяет доступные ресурсы, позволяет разработчику контролировать регистры, выполнять передачу данных между ускорителем и CPU, а также запускать вычислительные задания.

Назовем *регистром* произвольный линейный участок в глобальной по иерархии OpenCL памяти устройства. Для устройства типа CPU глобальной является обычная оперативная память, тогда как для устройства типа GPU глобальной будет набортная память ускорителя. Регистр реализован на базе стандартного OpenCL-буфера (объект типа cl_mem), поэтому при создании регистра необходимо указать флаги доступа к этому буферу. По умолчанию тип устройства, на котором создается регистр, определяется автоматически, и в соответствии с ним устанавливаются флаги “чтение–запись” для ускорителей и “закрепленная память” (pinned memory) для центральных процессоров, что позволяет избегать дополнительного копирования памяти при работе на процессоре. Эти флаги можно задать и вручную. Программный интерфейс объекта регистр предельно прост, все вышеуказанные параметры определяются при создании регистра (указывается устройство, размер выделяемой памяти, флаги и очереди) и не меняются до его удаления.

При работе с регистром для обмена данными между этим регистром и хостом или другими регистрами используются две OpenCL-очереди (объект типа cl_command_queue): одна очередь на загрузку в регистр, другая — на выгрузку из него. Существует три возможных опции. Первая — когда регистр используется как локальная переменная на устройстве, доступ к которой осуществляется только из вычислительных ядер. В этом случае очереди для этого регистра не нужны. Вторая — при инициализации регистра автоматически создается одна отдельная очередь, используемая только в этом регистре и на чтение, и на запись (опция по умолчанию). Третья — можно задать обе очереди вручную. В случае правильной обработки OpenCL-событий (объект типа cl_event) и поддержки асинхронного исполнения команд (и то, и другое определяется OpenCL-драйвером устройства), в этой опции нет необходимости, достаточно использовать одну очередь для чтения и записи во все регистры на одном устройстве, достигая при этом оптимальной работы и сокрытия передачи данных за вычислениями. Однако, к сожалению, эти свойства поддерживаются не на всех OpenCL-устройствах, поэтому представлен гибкий механизм настройки очередей.

Второе определяющее понятие — *команда*. Команды можно условно разделить на три группы: *команды-инструкции (EXEC, execute)*, запускающие вычислительные ядра на устройстве; *команды загрузки (LD, load)* и *выгрузки (ST, store)* данных на устройство; команды для запуска не-OpenCL-кода на процессоре *(CPU-команды)*, в том числе команды синхронизации и др. Упрощенная схема работы планировщика с очередями загрузки, выгрузки и вычислительных заданий показана на рис. 1.

У каждой команды имеется свойство, общее для всех типов и определяющее зависимости по данным от других команд. Зависимости реализованы на базе OpenCL-событий. С каждой командой, поставленной в OpenCL-очередь, можно связать OpenCL-событие и таким образом отслеживать статус выполнения этой команды. С другой стороны, командам, поставленным в очередь, можно задать список OpenCL-событий, до завершения которых команда не может начать исполнение. На базе данного функционала реализована

модель зависимостей между командами. Каждая команда хранит список команд, от которых она зависит. Обработка зависимостей осуществляется посредством двух вызовов, позволяющих

1) проверить, готова ли команда к исполнению,

2) определить статус выполнения.

Проверка статуса выполнения сводится к обработке OpenCL-события. Проверка готовности к исполнению работает по простому алгоритму: если зависимостей нет, то команда готова; если есть, то проверяется статус зависимостей; если хотя бы одна не готова, то команда не готова.

Общий тип “Команда” реализован в виде C++ класса с виртуальным методом “Выполнить”. Каждая команда наследует все свойства и переопределяет этот метод в соответствии со своим функционалом. Такой подход позволяет задавать общие правила для обработки и запуска команд независимо от их типов. Общий список методов типа “Команда” имеет вид

— добавить зависимость (команда);

— проверить готовность к исполнению ();

— проверить окончание исполнения ();

— выполнить ().

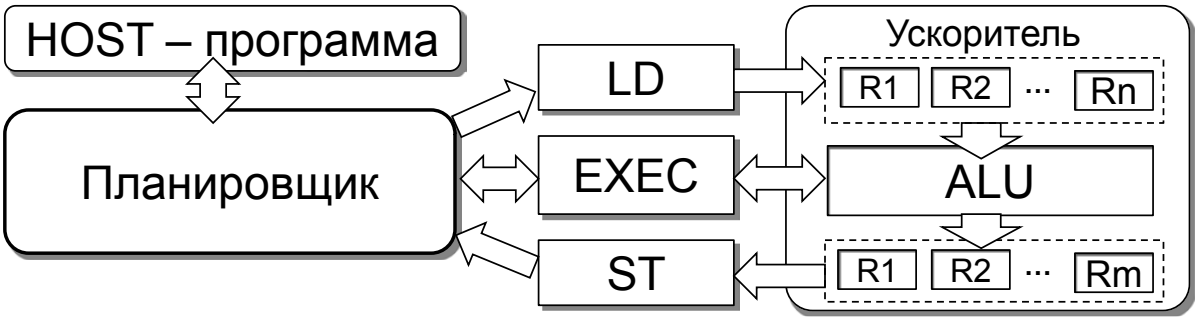

Рис. 1. Упрощенная схема работы планировщика

Резюмируя вышесказанное, получаем, что для каждого устройства можно ввести топологию регистров произвольных размеров и задать определенный набор команд-ядер, которые устройство может выполнить. Таким образом, в рамках предлагаемого подхода любые OpenCL-устройства можно понимать как виртуальные RISC-машины (Reduced Instruction Set Computer), исполняющие команды на регистрах. При этом стандарт OpenCL гарантирует, что каждое поддерживающее OpenCL устройство сможет исполнить любую команду (скорость исполнения и оптимизацию под конкретные вычислители в данный момент не рассматриваем). Команды реализованы в виде отдельных классов на C++, являющихся наследниками базового класса “Команда”, и построены по одинаковому принципу: аргументы передаются в конструктор, запоминаются, и затем используются в переопределенном методе “Выполнить”.

LD-команда позволяет загружать данные из памяти хоста в регистр. Существует два основных сценария загрузки: загрузка линейного участка памяти и загрузка прямоугольного участка (под прямоугольным участком понимается подматрица в плотной матрице, по сути, надстройка для работы с двумерными массивами). Главные критерии при реализации (в порядке важности) — сокрытие передачи данных за вычислительной нагрузкой и достижение оптимальной скорости передачи данных между устройством и хостом.

Аргументами LD-команды являются регистр, параметры участка хост-памяти, указатель на его начало и, возможно, определяющие алгоритм флаги. Для передачи данных используются OpenCL-очереди, указанные в регистре. OpenCL предоставляет два способа обеспечить передачу данных между памятью хоста и буфером: использовать стандартные вызовы OpenCL для линейного (clEnqueueWriteBuffer) и прямоугольного (clEnqueueWriteBufferRect) регионов или же отобразить буфер на адресное пространство хоста (clEnqueueMapBuffer) и копировать данные посредством прямой записи, возможно, с использованием стандартных функций языка C — memcpy() и других.

К сожалению, стандартные вызовы для копирования прямоугольных регионов памяти в зависимости от драйвера OpenCL-устройства могут работать неудовлетворительно, поэтому реализован универсальный вариант передачи прямоугольного региона: с помощью memcpy() происходит промежуточное копирование данных в линейный участок памяти на хосте, который затем передается посредством стандартного вызова OpenCL. В случае, когда устройство подключено через узкий канал (например, шина PCIe 2.0 16x), можно оптимизировать передачу прямоугольника, организовав двойную буферизацию. Данные че-

рез шину передаются блоками фиксированного размера, например по 16Мбайт; размер подбирается для достижения оптимальной скорости передачи, в это время параллельно заполняется другой блок.

Также можно осуществлять многопоточный вызов memcpy(), что позволяет добиться ускорения передачи при работе на системе с несколькими каналами памяти хоста. В планировщике реализованы все эти варианты передачи, переключаться между ними можно, указывая соответствующие флаги при создании команд. Кроме того, при передаче прямоугольника с помощью нарезки в линейный участок, автоматически, в случае необходимости, осуществляется "padding" — дополнение строк нулями до указанной ширины.

Интерфейс LD-команды, передающей данные с хоста в регистр, имеет два конструктора (линейная передача или передача прямоугольника), в которых происходит установка параметров и добавление зависимостей по регистру (см. ниже), и переопределенный метод "Выполнить", в зависимости от флагов вызывающий тот или иной метод передачи. ST-команда реализует передачу данных из регистра на хост (clEnqueueWriteBuffer, clEnqueueWriteBufferRect). Алгоритмы и устройство класса, в целом, аналогичны LD-команде, кроме следующего отличия: в общем случае, ST-команда раскладывает прямоугольный регион из памяти устройства (возможно, совершая обратную к "padding" операцию — удаляя добавленные LD-командой нули) в прямоугольную область памяти на хосте.

EXEC-команды предназначены для запуска скомпилированного кода вычислительных ядер OpenCL (объект типа cl_kernel). С точки зрения запускающей функции, ядра имеют следующие параметры: размер запускаемой многомерной группы нитей в каждом измерении, количество и тип аргументов ядра. Для создания единого универсального интерфейса команды, способной запустить любое ядро, был применен интерфейс с произвольным числом аргументов (см. список аргументов va_list из stdarg.h), аналогичный функции printf() языка C — сначала в строке передаются типы параметров, а затем сами параметры. Эти параметры задаются при создании EXEC-команды. Таким образом, интерфейс для работы с ней имеет следующий вид:

1) создать (программа, имя ядра, параметры сетки нитей, строка с типами аргументов, ...); на месте многоточия — аргументы, которые затем приводятся к указанным типам;

2) выполнить ().

Кроме того, планировщик по схожему принципу умеет запускать произвольный хост-код, написанный на языке C/C++, на процессоре (в обычном смысле, не как на OpenCL-устройстве).

Планировщик ставит команды в специальные очереди выполнения. Все очереди устроены одинаково и имеют несколько параметров: можно указать характер исполнения команд (последовательный или асинхронный), максимальное количество асинхронно выполняющихся команд и "глубину буфера команд". Команды в очереди всегда запускаются в том порядке, в котором они были в эту очередь поставлены. При последовательном исполнении следующая команда в очереди не может начать выполнение, пока не завершилась предыдущая. В случае асинхронного исполнения следующая команда может начать исполнение до завершения предыдущей, при этом порядок завершения их работы не гарантируется. Команда в очереди может быть блокирующей (например, выполнение CPU-кода, выполнение LD-команды посредством серии memcpy() и т.д.), поэтому выполнение команды может быть делегировано отдельной нити, которая вызывает метод "Выполнить" данной команды и принимает блокировку на себя. Чтобы ограничить одновременное число используемых нитей, используется параметр ограничения на количество одновременно выполняющихся задач.

Логические очереди планировщика реализованы на базе OpenCL-очередей, которые имеют некоторые накладные расходы при отправке очередной команды на устройство; чтобы их минимизировать, можно отправлять команды не по одной, а сразу группами. Параметр "глубина буфера команд" отвечает за размер такой группы команд.

По умолчанию, EXEC- и CPU-очереди работают последовательно, а LD- и ST-очереди — асинхронно. Синхронизация между очередями осуществляется посредством расстановки зависимостей между командами из этих очередей. Зависимости можно расставлять явно при создании команд и добавлении их в очереди планировщика либо неявно после формирования всех очередей. Логические очереди реализованы на базе STL-массивов, и имеется возможность доступа по индексу. Можно изменять рабочие параметры этих четырех очередей, а также увеличивать общее количество очередей. Реализована поддержка автоматической установки зависимости команд по регистрам, которая работает следующим образом: когда команда принимает регистр в качестве аргумента, она принудительно добавляет в свой список зависимостей последнюю команду, использовавшую этот регистр. Это позволяет, не влияя на общую производительность, упростить код и уменьшить вероятность возникновения ситуаций с неопределенным поведением в случае ошибки программирования. Хотя определяемый планировщиком базис из трех типов команд

(LD/ST, EXEC, CPU) достаточен для решения прикладных задач, реализована возможность создавать свои команды и ставить их в любые очереди.

Итоговая программа записывается в виде графа зависимостей в базисе предлагаемых четырех типов команд. Задача написания вычислительных ядер для устройств является, с одной стороны, основополагающей, а с другой — в том или ином виде присутствующей в любой парадигме программирования (поскольку ядра непосредственно решают задачу). Вообще говоря, в идеальной ситуации написание и оптимизация ядер должны занимать основную часть времени при решении задачи. Один планировщик может выполнять как весь граф зависимостей (всю программу), так и его часть на одном или нескольких устройствах сразу. Наиболее типовое применение таково: один планировщик работает на одном устройстве и выполняет соответствующую этому устройству часть графа зависимостей. Построение программы для планировщика состоит в последовательном создании необходимых команд, указанию зависимостей между ними и постановке команд в очереди этого планировщика.

При формировании очередей удобно использовать циклы, условные операторы и др. При использовании параллелизма по данным для ускорения на нескольких устройствах удобно писать общие функции-построители программ для планировщика. В качестве аргументов они получают планировщик и параметры задачи и строят программу по одинаковому алгоритму. После окончания формирования программы в планировщике последний отправляет программу на исполнение и дожидается окончания ее работы.

3. Реализация многоуровневого распараллеливания.

3.1. Математическая модель и численная реализация. Рассматривается проблема численного моделирования сжимаемых течений, описываемых следующей системой уравнений Эйлера:

$$\frac{\partial \boldsymbol{Q}}{\partial t} + \frac{\partial \boldsymbol{F}_1(\boldsymbol{Q})}{\partial x} + \frac{\partial \boldsymbol{F}_2(\boldsymbol{Q})}{\partial y} + \frac{\partial \boldsymbol{F}_3(\boldsymbol{Q})}{\partial z} = 0. \tag{1}$$

Здесь $\boldsymbol{Q} = (\rho, \rho u, \rho v, \rho w, E)^{\mathrm{T}}$ — вектор из пяти консервативных переменных, а конвективные потоки определяются так: $\boldsymbol{F}_1(\boldsymbol{Q}) = \big(\rho u, \rho u^2 + p, \rho uv, \rho uw, u(E+p)\big)^{\mathrm{T}}$, $\boldsymbol{F}_2(\boldsymbol{Q}) = \big(\rho v, \rho uv, \rho v^2 + p, \rho vw, v(E+p)\big)^{\mathrm{T}}$, $\boldsymbol{F}_3(\boldsymbol{Q}) = \big(\rho w, \rho uw, \rho vw, \rho w^2 + p, w(E+p)\big)^{\mathrm{T}}$.

Полная энергия определяется формулой $E = \dfrac{1}{2}\,\rho\big(u^2 + v^2 + w^2\big) + \rho\varepsilon$. Уравнения (1) замыкаются уравнением состояния идеального газа $p = \rho\varepsilon(\gamma - 1)$.

Для дискретизации по пространству используется схема для неструктурированных гибридных сеток с определением переменных в центрах сеточных ячеек, которыми могут быть тетраэдры, гексаэдры, треугольные призмы и четырехугольные пирамиды. В каждой ячейке определены среднеинтегральные значения консервативных переменных: $f_{ID} = \dfrac{1}{V_{ID}} \displaystyle\int\limits_{V_{ID}} f(x, y, z)\, dV$, где индекс ID обозначает идентификационный номер ячейки, V_{ID} — объем ячейки, а f_{ID} обозначает одну из пяти консервативных переменных. Более подробную информацию о подобном типе дискретизации можно найти, например, в [12].

Пусть τ — величина шага по времени. С помощью схемы Рунге–Кутты определяются значения на следующем временно́м слое: $\boldsymbol{Q}_{ID}^{n+1} = \boldsymbol{Q}_{ID}^{n} + \dfrac{\Phi_{ID}\tau}{V_{ID}}$. Конвективный поток Φ_{ID} в ID-й ячейке вычисляется как сумма потоков через грани ячейки: $\Phi_{ID} = \displaystyle\sum_{k=0}^{NS_{ID}-1} \Phi S_{ID/I_k^{ID}}$, где NS_{ID} — число граней ячейки. Поток $\Phi S_{ID/I_k^{ID}}$ через грань, имеющую идентификационный номер ID/I_k^{ID} (т.е. грань, разделяющую ID-ю и I_k^{ID}-ю ячейки), вычисляется по схеме Роу [13]: $\Phi S_{ID/I_k^{ID}} = F_{ROE}\left(QL_{ID/I_k^{ID}}, QR_{ID/I_k^{ID}}, \bar{n}_{ID/I_k^{ID}}\right)$. Этот поток зависит от вектора нормали $\bar{n}_{ID/I_k^{ID}}$ и от значений консервативных переменных “слева” и “справа” от грани в позиции центра масс грани $QL_{ID/I_k^{ID}}$ и $QR_{ID/I_k^{ID}}$ соответственно.

Далее, в схемах повышенного порядка, чтобы получить эти значения слева и справа от центра грани, могут использоваться различные подходы. В [14] представлен вычислительный алгоритм и реализация для массивно-параллельных ускорителей для схемы повышенного порядка на основе полиномиальной реконструкции. Аналогично, повышение порядка базовой численной схемы может осуществляться посредством квазиодномерной реконструкции [15], пример реализации численного алгоритма на основе подобной схемы представлен, например, в [16].

3.2. Декомпозиция расчетной области. Расчетная область, представленная неструктурированной сеткой, разделяется на подобласти для распределения работы между множеством узлов суперкомпьютера. Аналогичным образом подобласти разделяются далее на части для распределения нагрузки по различным вычислительным устройствам узла — подобласти 2-го уровня. Обмен данными между ускорителями

осуществляется по интерфейсным элементам между подобластями 2-го уровня, а MPI-обмен данными между узлами суперкомпьютера — по интерфейсам между подобластями 1-го уровня.

Для введения основных обозначений рассмотрим для простоты одноуровневое разбиение (для второго уровня распараллеливание реализовано идентичным образом), считая, что число подобластей равно числу задействованных ускорителей. Назовем *собственными* элементами те ячейки сетки, которые составляют подобласть ускорителя. В случае схемы 1-го порядка для вычисления конвективных потоков через грани ячеек необходимо знать значения из двух ячеек, которые эта грань разделяет. Таким образом, для вычисления потоков через грани, у которых одна ячейка собственная, а другая принадлежит соседней подобласти, необходимо получить данные от соседа.

Набор чужих ячеек, необходимых для расчета потоков через грани собственных ячеек, будем называть *гало*. Чужие ячейки, непосредственно граничащие с собственными ячейками, — это гало 1-го уровня. В случае схемы повышенного порядка может увеличиваться необходимое число уровней соседства. Гало-элементы 2-го уровня — чужие ячейки, соседние с гало-ячейками 1-го уровня, и т.д.

Назовем *расширенной подобластью* объединение множества собственных ячеек и гало-ячеек (до необходимого пространственному шаблону численной схемы уровня). Структуры данных на вычислительном устройстве, ответственном за подобласть, создаются для расширенной подобласти. *Интерфейсные* ячейки — это собственные ячейки, для расчета по которым необходимы данные из соседних подобластей. Соответственно, интерфейсные ячейки формируют гало для соседних подобластей, и данные из них нужны для расчета по ячейкам из соседних подобластей. *Внутренние* ячейки — это собственные ячейки, связанные только с собственными ячейками. Все типы элементов показаны на рис. 2.

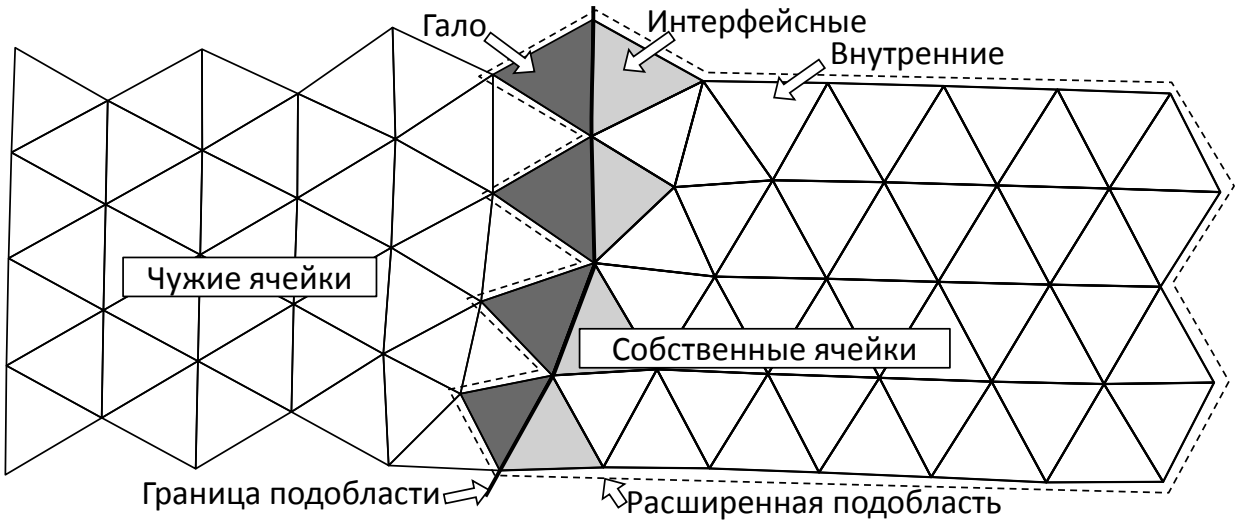

Рис. 2. Представление подобласти расчетной области

Аналогичным образом определяются и наборы граней. Интерфейсные грани — грани, разделяющие собственную и чужую ячейки, внутренние грани — разделяющие собственные ячейки.

Для того чтобы рационально разбить исходную расчетную область на подобласти, строится дуальный граф связей контрольных объемов, в котором вершинам соответствуют контрольные объемы (ячейки), а ребрам — грани, разделяющие контрольные объемы. Для представления дуального графа используется стандартный построчно-разреженный формат CSR (Compressed Storage Format) для хранения разреженных матриц. Для получения сбалансированного разбиения с минимизацией числа интерфейсных элементов используется внешняя библиотека декомпозиции METIS [17]. Подавая на вход процедуре METIS_PartGraphKway дуальный граф сетки в CSR-формате и желаемое количество подобластей, на выходе получим линейный массив индексов принадлежности каждого элемента той или иной подобласти. Следует отметить, что на вход процедуре METIS_PartGraphKway в качестве параметра подается тип разбиения: METIS_OBJTYPE_CUT — разбиение с минимизацией по количеству интерфейсных граней и METIS_OBJTYPE_VOL — с минимизацией по суммарному объему обмена данными. Выбор подходящего типа разбиения может существенно влиять на параллельную эффективность. В нашем случае был выбран параметр METIS_OBJTYPE_VOL. В случае декомпозиции 2-го уровня для распределения нагрузки между CPU и GPU на вход процедуре разбиения подается также массив весов, содержащий для каждого элемента сетки некоторый заданный вес. Вес ячеек, принадлежащих CPU- и GPU-подобластям,

выбирается эмпирически, пропорционально фактической производительности CPU и GPU; таким образом, обеспечивается балансировка загрузки устройств.

Применимость последовательной библиотеки METIS ограничена памятью и производительностью вычислительного модуля, на котором выполняется разбиение. Для больших сеток с числом элементов более 100 миллионов применяется библиотека ParMETIS [18] и реализованные на ее базе параллельные средства декомпозиции неструктурированных сеток. Кроме того, используется библиотека [19], работающая по геометрическому принципу, т.е. учитывающая координаты сеточных элементов. Во многих случаях второе средство работает быстрее и дает более сбалансированное разбиение.

3.3. Построение схемы обмена данными. После определения принадлежности сеточных элементов для корректного представления расширенной подобласти вводятся три типа нумерации. Исходная нумерация элементов всей расчетной области является глобальной нумерацией элементов. Элементы сетки в расширенной подобласти переупорядочиваются следующим образом: сначала идут внутренние элементы, затем интерфейсные, а затем гало. Кроме того, гало-элементы группируются по принадлежности к подобластям. Локальная нумерация и расширенная локальная нумерация идентифицируют элементы в подобласти и в расширенной подобласти соответственно. Для отображения идентификаторов из одной индексации в другую формируются соответствующие индексные массивы.

Далее, необходимо сформировать схему обменов данными, т.е. для каждого элемента определить, какие подобласти его используют. Для каждого интерфейсного элемента формируется список, начинающийся с номера подобласти-владельца и содержащий номера подобластей, в гало которых он входит. В случае схемы первого порядка применяется простейший алгоритм обхода вершин дуального графа связей контрольных объемов, в котором для каждой вершины выполняется обход по соседним вершинам с проверкой номера владельцев.

Для схемы повышенного порядка требуется столько проходов данного алгоритма маркировки, сколько уровней гало необходимо для шаблона численной схемы. Чтобы иметь возможность быстро обрабатывать сетки с сотнями миллионов и миллиардами элементов, средство для построения схемы обменов реализовано в виде отдельной программы в составе [16], работающей в параллельном режиме с распределенной памятью (MPI) с произвольным числом процессов, не обязательно совпадающим с числом подобластей.

После того как полностью определены интерфейсы между подобластями, строятся буферные массивы для обменов данными: для каждой соседней подобласти создаются массивы отправки и приема сообщений, размер которых определяется числом затронутых интерфейсных и гало-элементов соответственно. На этом этапе оптимизация заключается в переупорядочивании сеточных элементов таким образом, чтобы они располагались в памяти наиболее компактно.

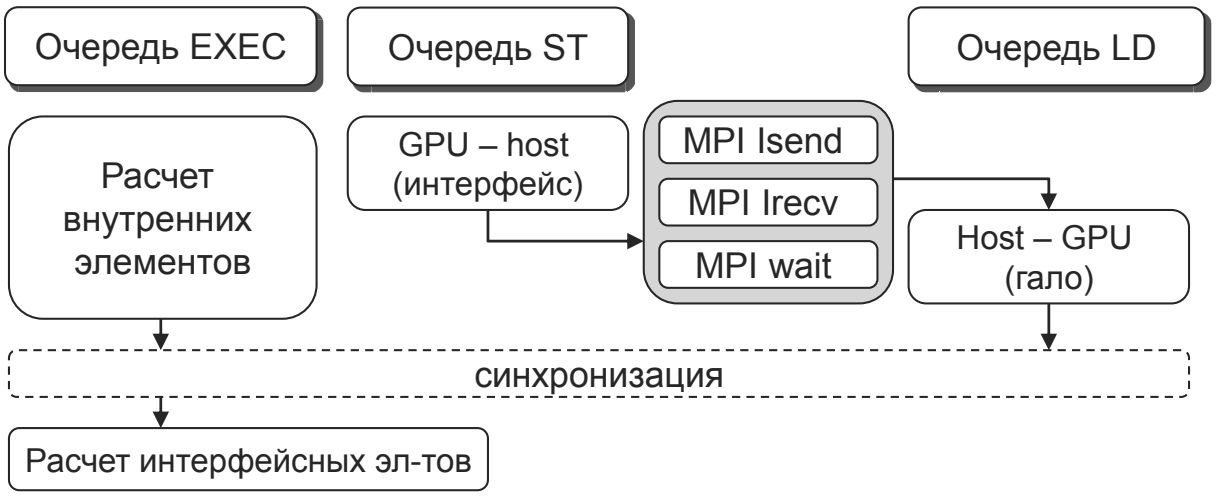

Рис. 3. Схема одновременного обмена данными и вычислений на ускорителе (overlap)

3.4. Обмен данными одновременно с вычислениями (overlap). Обмен данными одновременно с вычислениями, т.е. *overlap*, основан на декомпозиции вычислительной операции над элементами подобласти на два этапа:

1) расчет внутренних элементов, для которых не требуются значения из чужих элементов,

2) расчет интерфейсных элементов.

В этом случае первый этап, существенно более вычислительно емкий, может выполняться одновременно с обменом данными, необходимым для второго этапа. Общая схема такого обмена изображена на рис. 3.

Подготовив все служебные объекты, можно приступить к формализации алгоритма обмена гало-узлов. В [20] описан модельный алгоритм 1-го порядка, на примере которого здесь демонстрируется реализация обмена данными. Один временной шаг расчета можно представить в виде нескольких последовательных этапов: расчет потоков через внутренние грани контрольных объемов, суммирование потоков с граней в центры ячеек, расчет граничных условий — потоков через граничные грани расчетной области, шаг интегрирования Рунге–Кутты. При декомпозиции задачи на подобласти добавляется еще один этап — обновление гало. Структура такого алгоритма показана на рис. 4 слева.

Обмен данными логически делится на два этапа: обмены внутри вычислительного узла между хостом и устройствами и MPI-обмены между узлами. Полная последовательность действий этапа обмена для обновления гало такова:

— выгрузить с ускорителей на хост данные из интерфейсных узлов с текущего расчетного шага;

— сформировать посылки и отправить их внешним получателям посредством MPI;

— получить данные от внешних отправителей посредством MPI;

— загрузить обновленные гало на ускорители.

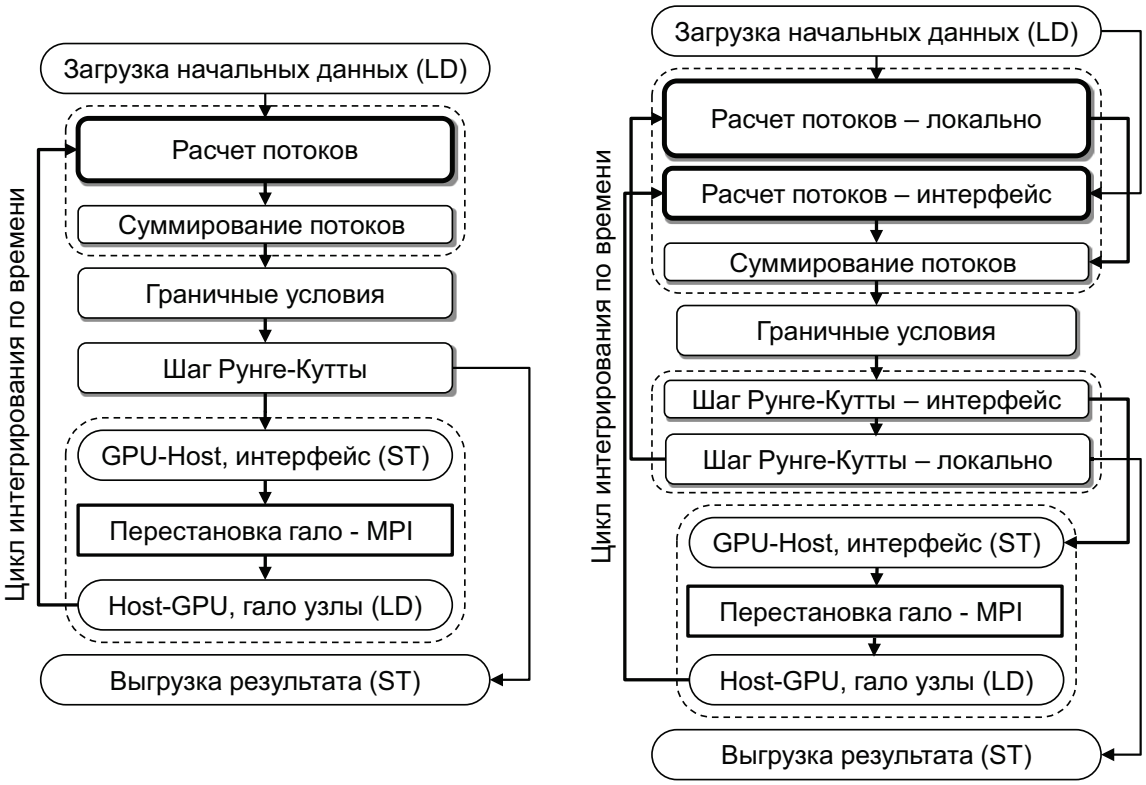

Рис. 4. Граф алгоритма для последовательного выполнения вычислений и обменов, справа — обмены одновременно с вычислениями (overlap)

В исходном варианте все стадии шага интегрирования выполняются последовательно. Это неоптимальный подход с точки зрения быстродействия, поскольку обмены можно скрыть на фоне этапа расчета потоков и этапа интегрирования Рунге–Кутты. Для этого в этих стадиях вычисления разделяются на две части — для внутренних и интерфейсных элементов.

На стадии интегрирования Рунге–Кутты сначала выполняется интегрирование для интерфейсных элементов, после этого полный цикл вычислений на текущем шаге для них закончен и их можно выгрузить на хост. Выгрузка начинается параллельно с интегрированием для внутренних элементов, а обмен данными продолжается параллельно с этапом расчета потоков для внутренних элементов на следующем

временно́м слое, поэтому обмены полностью скрываются и не отражаются на общем времени счета. Это дает возможность на фоне вычислений потоков для внутренних элементов произвести перестановку гало с предыдущего шага и раздачу их потребителям. Тем самым выгрузка интерфейсов, перестановка гало и раздача гало потребителям полностью скрываются за основными вычислениями. На рис. 4 справа показана общая схема расчетного шага с сокрытием этапа обмена гало на фоне вычислительной нагрузки. Этот граф исполнения передается планировщику, который автоматически обеспечивает параллельное выполнение независимых операций.

Для всех вычислительных узлов в распределенной системе схема на рис. 4 будет повторяться, так как MPI-обмены явно в вычислениях не фигурируют и полностью инкапсулированы в стадии перестановки гало.

Следует отметить, что существенное повышение производительности дало выставление MPI-барьеров (MPI_Barrier(MPI_COMM_WORLD)) в процедуре перестановки и рассылки гало. Это можно трактовать как принудительную синхронизацию в распределенной системе в случае неравномерных локальных и MPI-обменов. Эффект от добавления барьера показан на рис. 5.

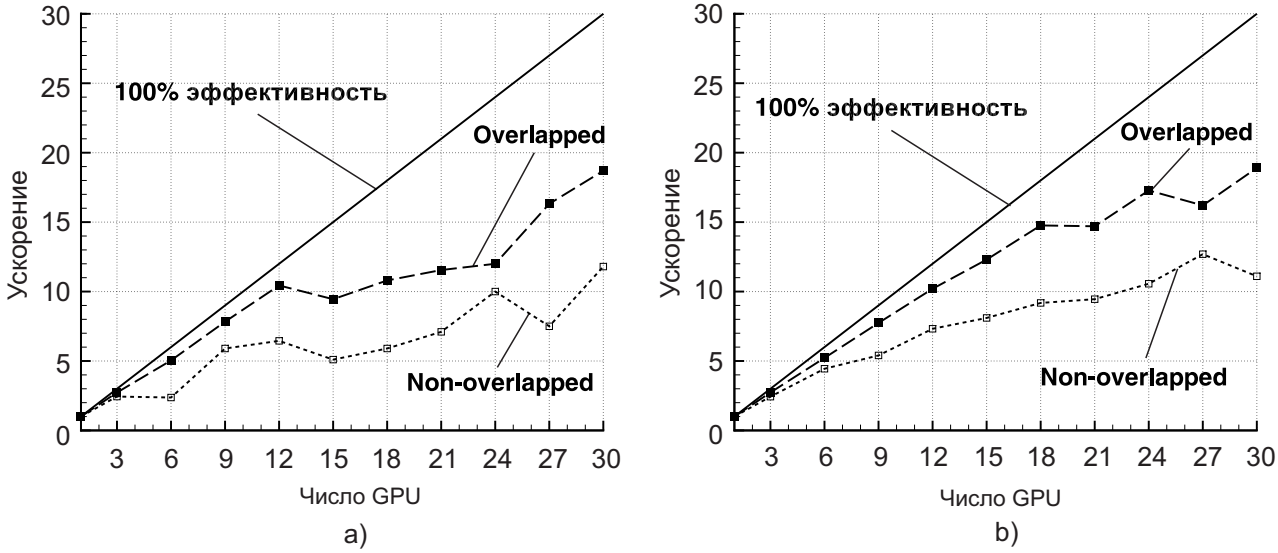

Рис. 5. Ускорение на K100 для исходного (a) и дополненного барьером (b) алгоритма обмена данными в режимах с overlap и без, сетка 4 миллиона ячеек

3.5. Параллельная эффективность и производительность. Производительность вычислений на одиночном ускорителе для данного алгоритма подробно рассмотрена в работе [20]. В среднем полученное ускорение относительно одного ядра CPU Intel Xeon X5670 2.9 ГГц составляет примерно 20 раз для GPU NVIDIA C2050 и около 50 раз для GPU AMD Radeon 7970. Производительность вычислений для схемы повышенного порядка на одиночном ускорителе представлена в [14], где демонстрируется достижение фактической производительности на одном GPU до 160 GFLOPS. Здесь же исследуется параллельная эффективность реализации вычислений на множестве ускорителей. Рассматриваются два типа систем: гибридный суперкомпьютер K100 ИПМ им. М.В. Келдыша РАН и экспериментальный вычислительный стенд с общей памятью НИИСИ РАН.

Суперкомпьютер K100 состоит из 64 вычислительных модулей, содержащих два 6-ядерных CPU Intel Xeon X5670 2.9 ГГц и три GPU NVIDIA C2050, объединенных коммуникационной средой InfiniBand. На данной системе тестировалась эффективность многоуровневой коммуникационной схемы с MPI и host-GPU обменами, скрываемыми за вычислениями на GPU. Тест на сетке 4 миллиона ячеек демонстрирует ускорение для MPI-распараллеливания. Используется схема 1-го порядка, имеющая минимальную вычислительную стоимость, что максимально усложняет достижение высокой параллельной эффективности. На рис. 5 показано сравнение ускорения относительно вычислений на одном GPU при обмене данными в последовательном режиме и одновременно с вычислениями (overlap). Графики приводятся для исходной реализации обменов данными и реализации, дополненной вызовом барьера для принудительной синхронизации процессов.

На рис. 6a для сетки из 16 миллионов ячеек показано ускорение относительно одного узла суперкомпьютера, имеющего три GPU, для обменов в последовательном режиме и параллельно с вычислительной нагрузкой.

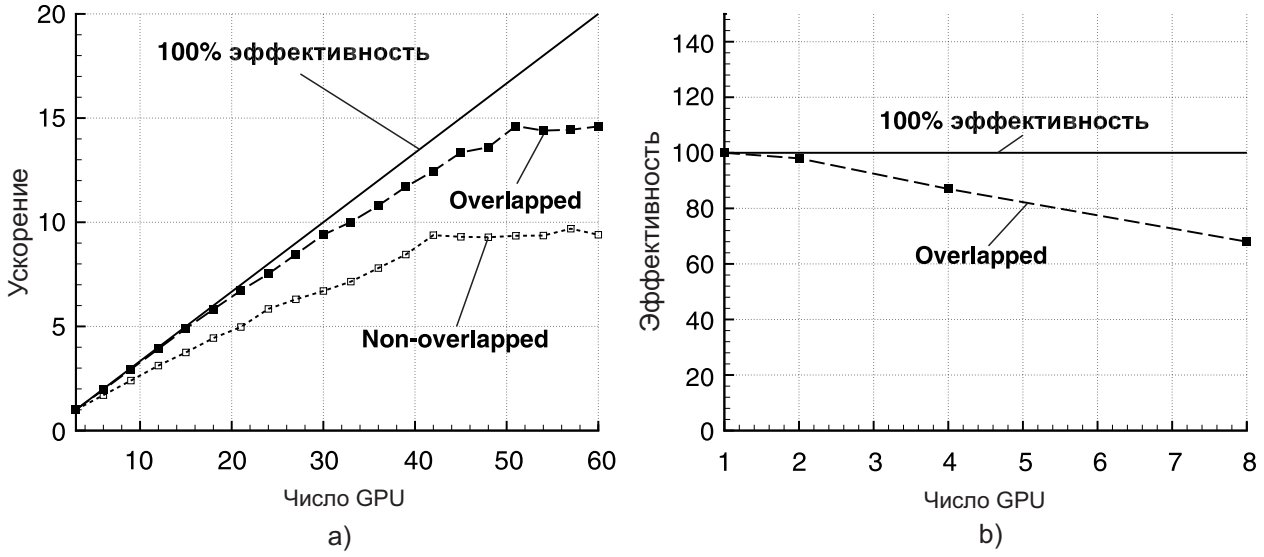

Рис. 6. Ускорение на K100 относительно одного узла с тремя GPU на сетке из 16 миллионов ячеек (a). Эффективность масштабирования при загрузке 4 миллионов ячеек на GPU на экспериментальной системе с общей памятью (b)

Из результатов видно, что благодаря сокрытию обменов за вычислениями даже для схемы 1-го порядка приемлемая параллельная эффективность обеспечивается при минимальной загрузке порядка 150–200 тысяч сеточных ячеек на одно GPU-устройство, что, как правило, на порядок меньше загрузки в реальных расчетах.

Для полноты картины рассмотрим систему другого типа, работающую в рамках общей CPU-памяти. Вычислительный стенд представляет собой одиночный модуль, содержащий два 12-ядерных CPU Opteron 6176 SE и восемь GPU AMD 7970 (рис. 7), объединенных системой PCI-Express расширения Cyclone Microsystems.

На данной системе имитируется вычислительный модуль возможной перспективной архитектуры суперкомпьютера с узлами высокой вычислительной плотности (так называемые “fat node”). Поскольку ускорители имеют весьма ограниченный объем памяти, важно обеспечивать хорошее масштабирование, т.е. возможность эффективного увеличения числа сеточных ячеек пропорционально числу задействованных ускорителей.

Тесты на этой системе демонстрируют масштабируемость на вычислительном модуле с большим числом ускорителей и общей памятью хоста. Загрузка на один ускоритель составляет 4 миллиона узлов, размер сетки увеличивается пропорционально числу ускорителей. Результаты по достигнутой эффективности показаны на рис. 6b. Более подробно данные приведены в таблице, где также представлено ускорение относительно последовательного расчета на том же узле на одном ядре CPU.

Масштабирование на одиночном узле с большим числом GPU

Число GPU	Число ячеек сетки, млн	Ускорение относительно 1 CPU ядра	Эффективность масштабирования
1	4	60	100%
2	8	118	98%
4	16	210	87%
8	32	325	68%

4. Заключение. Предложен упрощающий гетерогенную реализацию планировщик, управляющий потоками вычислительных и коммуникационных заданий OpenCL и использующий представление вычислительной задачи в виде графа исполнения вычислительных подпрограмм, процедур передачи данных и связей между ними.

Рассмотрен численный алгоритм для моделирования внешнего обтекания невязким сжимаемым газом на основе явной конечно-объемной схемы первого порядка с определением переменных в центрах сеточных ячеек для неструктурированных гибридных сеток.

Данный алгоритм имеет минимальную вычислительную стоимость, что максимально усложняет достижение высокой параллельной эффективности. Несмотря на это, реализованная посредством планиров-

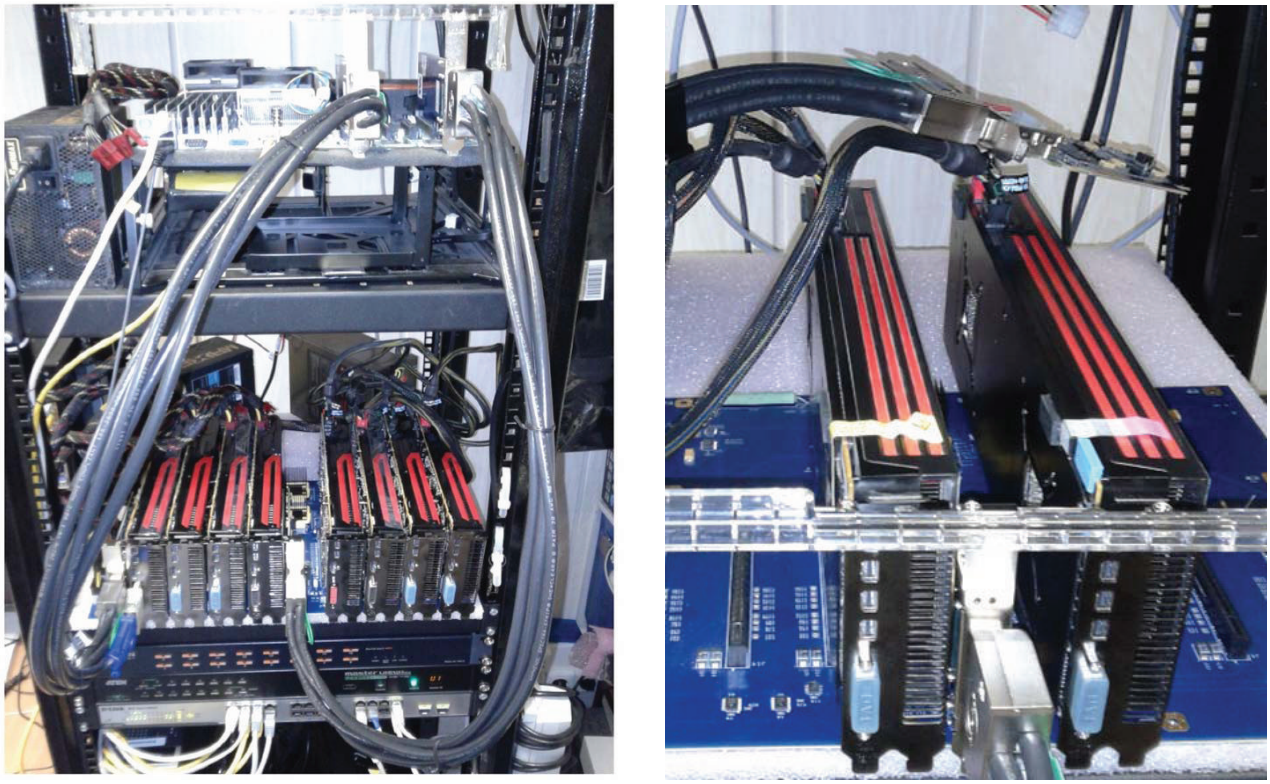

Рис. 7. Экспериментальный вычислительный стенд с системой PCI-Express расширения Cyclone, содержащий два 12-ядерных CPU Opteron 6176 SE и восемь GPU AMD 7970

щика коммуникационная схема с режимом overlap, т.е. с одновременным выполнением обменов данными и вычислений, позволила получать высокую параллельную эффективность даже на очень небольших сетках с низкой вычислительной нагрузкой на ускорители, на порядок меньшей, чем в реальных расчетах.

Таким образом, показана принципиальная возможность применения программной инфраструктуры гетерогенных вычислений в вычислительных задачах такого типа. Высокая эффективность программной инфраструктуры продемонстрирована как на гибридном суперкомпьютере, так и на прототипе вычислительного узла высокой плотности.

Для расчетов использовался суперкомпьютер К100 ИПМ им. М.В. Келдыша РАН и экспериментальный вычислительный НИИСИ РАН. Авторы выражают благодарность этим организациям.

СПИСОК ЛИТЕРАТУРЫ

1. *Wolfe M.* The OpenACC application programming interface. Version 2.0, 2013 (http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf).
2. *Dolbeau R., Bihan S., Bodin F.* HMPP: A Hybrid Multi-core Parallel Programming environment, CAPS enterprise, 2013 (http://www.caps-entreprise.com/wp-content/uploads/2012/08/caps-hmpp-gpgpu-Boston-Workshop-Oct-2007.-pdf).
3. *Augonnet C., Thibault S., Namyst R., Wacrenier P.-A.* StarPU: a unified platform for task scheduling on heterogeneous multicore architectures // Concurrency and Computation: Practice and Experience. Special Issue: Euro-Par 2009. 2011. **23**. 187–198.
4. *Jablin T.B., Prabhu P., Jablin J.A., Johnson N.P., Beard S.R., August D.I.* Automatic CPU–GPU communication management and optimization // Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). June 2011 (http://liberty.princeton.edu/Publications/pldi11_cuda.pdf).
5. *Song F., Dongarra J.* A scalable framework for heterogeneous GPU-based clusters // SPAA'12 Proc. of the 24th ACM Symposium on Parallelism in Algorithms and Architectures. 2012. 91–100 (http://web.eecs.utk.edu/∼song/publication/spaa12.pdf).
6. *Milakov M., Messmer P., Bradley T.* Accelerating NEMO with OpenACC // GPU Technology Conference. San-Jose, California, USA. March 18–21, 2013 (http://on-demand.gputechconf.com/gtc/2013/presentations/S3209-Accelerating-NEMO-with-OpenACC.pdf).

7. *Sankaran R., Chen J., Grout R., Levesque J.* Hybridization of a direct numerical simulation software for massively parallel accelerator-based architectures // Extreme Scaling Workshop. Chicago, Illinois, USA. June 15–16, 2012 (https://www.xsede.org/documents/271087/369160/ExtScale-Sankaran.pdf).
8. *Fuhrer O., Osuna C., Lapillonne X., Gysi T., Bianco M., Schulthess T.* Towards GPU-accelerated operational weather forecasting // GPU technology Conference. San-Jose, California, USA. March 18–21, 2013 (http://on-demand.gputechconf.com/gtc/2013/presentations/S3417-GPU-Accelerated-Operational-Weather-Forecasting.pdf).
9. *Namyst R.* Programming heterogeneous, accelerator-based machines with StarPU // Parallel Processing and Applied Mathematics. Torun, Poland. Sept. 11–14, 2011 (http://www.ppam.pl/2011/phocadownload/StarPU.pdf).
10. *Kurzak J., Luszczek P., Faverge M., Dongarra J.* LU factorization with partial pivoting for a multicore system with accelerators // IEEE Transactions on Parallel and Distributed systems. 2013. **24**, N 8. 1613–1621.
11. *Schor L., Tretter A., Scherer T., Thiele L.* Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of OpenCL // IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia). Montreal, Canada. October 3–4, 2013 (ftp://ftp.tik.ee.ethz.ch/pub/people/lschor/stst2013a.pdf).
12. *Barth T.* Numerical methods for conservation laws on structured and unstructured meshes. VKI for Fluid Dynamics. Lectures Series. Vol. 5. Rhode-Saint-Genèse: Von Kármán Inst., 2003.
13. *Roe P.L.* Approximate Riemann solvers, parameter vectors and difference schemes // J. Comput. Phys. 1981. **43**, N 2. 357–372.
14. *Суков С.А., Горобец А.В., Богданов П.Б.* Адаптация и оптимизация базовых операций газодинамического алгоритма на неструктурированных сетках для расчетов на массивно-параллельных ускорителях // Журн. вычислит. матем. и матем. физики. 2013. **53**, № 8. 1360–1373.
15. *Абалакин И.В., Козубская Т.К.* Схема на основе реберно-ориентированной квазиодномерной реконструкции переменных для решения задач аэродинамики и аэроакустики на неструктурированных сетках // Математическое моделирование. 2013. **25**, № 8. 109–136.
16. *Абалакин И.В., Бахвалов П.А., Горобец А.В., Дубень А.П., Козубская Т.К.* Параллельный программный комплекс NOISETTE для крупномасштабных расчетов задач аэродинамики и аэроакустики // Вычислительные методы и программирование. 2012. **13**. 110–125.
17. *Karypis G., Kumar V.* Parallel multilevel k-way partitioning scheme for irregular graphs // SIAM Review. 1999. **41**, N 2. 278–300.
18. *Schloegel K., Karypis G., Kumar V.* Parallel static and dynamic multi-constraint graph partitioning // Concurrency and Computation: Practice and Experience. 2002. **14**, N 3. 219–240.
19. *Головченко Е.Н.* Параллельный пакет декомпозиции больших сеток // Математическое моделирование. 2011. **23**, № 10. 3–18.
20. *Горобец А.В., Суков С.А., Железняков А.О., Богданов П.Б., Четверушкин Б.Н.* Применение GPU в рамках гибридного двухуровневого распараллеливания MPI+OpenMP на гетерогенных вычислительных системах // Вестник ЮУрГУ. 2011. **242**, № 25. 76–86.

Поступила в редакцию
31.10.2013