УДК 533.7+519.688

АДАПТАЦИЯ МЕТОДА ПРЯМОГО СТАТИСТИЧЕСКОГО МОДЕЛИРОВАНИЯ ДЛЯ ВЫЧИСЛЕНИЙ НА ГПУ

А. В. Кашковский[1], **А. А. Шершнёв**[1], **М. С. Иванов**[1]

Представлены технология и алгоритмы программной реализации метода прямого статистического моделирования течений разреженного газа для параллельных вычислений с использованием процессоров видеокарт. Показаны особенности использования видеопроцессоров и даны рекомендации по составлению ГПУ-программ.

Ключевые слова: CUDA, метод прямого статистического моделирования, численные алгоритмы, графические процессорные устройства.

Введение. Метод прямого статистического моделирования Монте-Карло (ПСМ) в настоящее время является основным инструментом исследования течений разреженного газа и широко применяется, в частности, в вычислительной аэродинамике космических аппаратов (КА). Однако метод ПСМ чрезвычайно требователен к размерам памяти и быстродействию. Для моделирования течений газа вокруг КА на высотах менее 100 км требуются параллельные вычисления с использованием 500 и более процессоров. Такие вычисления можно проводить только на больших вычислительных системах. Фактически, каждый такой расчет становится уникальным, что мешает массовости использования метода ПСМ. В связи с этим любые алгоритмы ускорения или повышения эффективности параллелизации всегда остаются актуальными.

Один из наиболее перспективных подходов к решению данного вопроса заключается в использовании графических процессоров видеокарт (Graphics Processor Unit, или ГПУ). Первые видеокарты представляли собой узкоспециализированные устройства, предназначенные для ускорения визуализации дву-, а затем и трехмерных изображений. В процессе своего развития графические процессоры превратились в высокопроизводительные вычислительные устройства, позволяющие проводить крупномасштабные параллельные расчеты. Однако их использование требует существенной модификации программ.

В настоящей статье представлена методика переноса на ГПУ программного обеспечения, изначально предназначенного для вычислений на кластерных системах с использованием библиотеки MPI (Massage Passing Interface).

1. Метод ПСМ. Метод ПСМ — это стохастический численный метод решения нелинейного кинетического уравнения Больцмана, традиционно рассматривающийся как метод компьютерного моделирования движения большого количества частиц, представляющих газовое течение [5, 6].

Моделирование течения производится в вычислительной области, построенной вокруг исследуемой модели. В простейшем случае эта область представляет собой прямоугольник или параллелепипед (для дву- или трехмерных задач соответственно), разбитый на достаточно малые ячейки равномерной прямоугольной сеткой. Эти ячейки используются для организации межмолекулярных столкновений и сбора статистической информации. Моделирование ведется по времени дискретными интервалами Δt, на каждом из которых выполняется:

1) перемещение каждой частицы в соответствии с ее текущей скоростью за время Δt, которое изменяет координаты частиц;

2) моделирование бинарных столкновений частиц, которые изменяют скорости частиц; сталкиваться могут только частицы, расположенные в одной ячейке; вероятность столкновения зависит от числа частиц в ячейке и их относительных скоростей [7].

Вычисления методом ПСМ проще всего начинать с равномерного поля течения, которое, взаимодействуя с телом, изменяет свою структуру и постепенно приходит к стационарному течению. Например, на рис. 1 показана эволюция частиц по времени при обтекании цилиндра на скорости 7500 м/с и высоте 90 км. После выхода на стационарное состояние можно накапливать статистическую информацию. Перенос каждой частицы происходит независимо от других частиц. Точно так же, межмолекулярные

[1] Институт теоретической и прикладной механики им. С. А. Христиановича СО РАН, ул. Институтская, 4/1, 630090, Новосибирск; А. В. Кашковский, науч. сотр., e-mail: sasa@itam.nsc.ru; А. А. Шершнёв, мл. науч. сотр., e-mail: antony@itam.nsc.ru; М. С. Иванов, зав. лабораторией, e-mail: ivanov@itam.nsc.ru

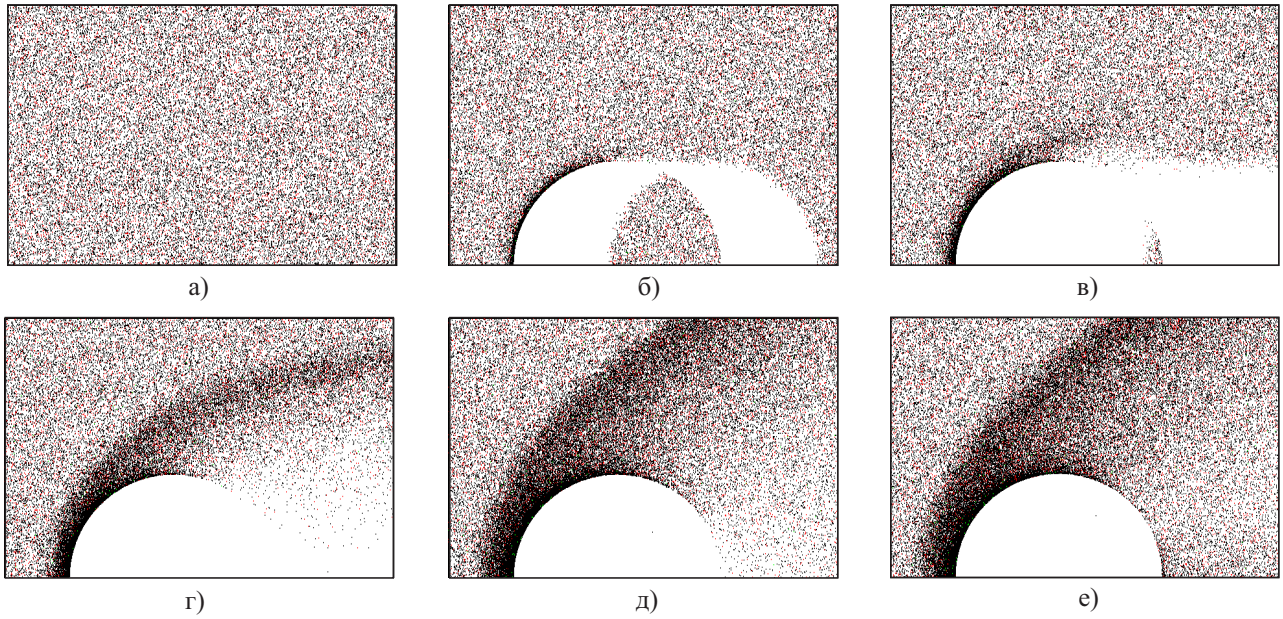

Рис. 1. Эволюция частиц к стационарному состоянию: а) $t = 0.0000$; б) $t = 0.0001$; в) $t = 0.0002$; г) $t = 0.0005$; д) $t = 0.0020$; е) $t = 0.0200$

столкновения в какой-либо ячейке не зависят от столкновений в других ячейках. Поэтому эти две операции и являются наиболее подходящими для параллелизации. Одним из наиболее распространенных способов параллелизации метода ПСМ с использованием технологии MPI [1] является "разделение области" (domain decomposition). Ячейки сетки определенным образом распределяются между процессорами, и в памяти этих процессоров хранится вся информация, связанная с этими ячейками и находящимися в этих ячейках частицами. Каждый процессор производит перенос принадлежащих ему частиц. Если после переноса частица оказалась приписанной к ячейке, принадлежащей другому процессору, то вся необходимая информация об этой частице пересылается на соответствующий вычислительный узел. После этого производятся столкновения между частицами и т.д. Ниже представлены этапы параллельного вычисления одного шага методом ПСМ (рис. 2).

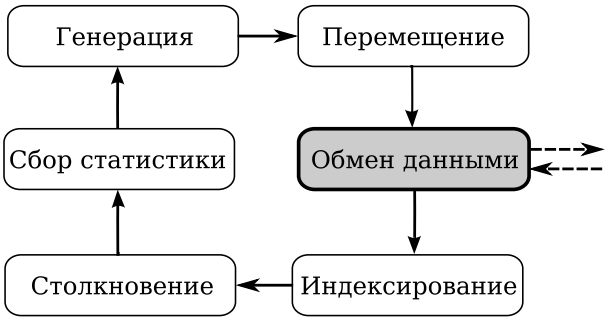

Рис. 2. Схема вычислений методом ПСМ

Генерация — создание новых частиц. Каждый процессор на границе расчетной области создает частицы в соответствии с заданной функцией распределения. Совсем не обязательно, чтобы вновь созданные частицы оказались в ячейках, принадлежащих данному процессору. Они будут переданы своему процессору после шага *Перемещение*. Новые частицы дописываются в конец списка частиц процессора. Число частиц, созданных на данном шаге, запоминается.

Перемещение — сдвиг частиц на заданный шаг по времени. Пусть $\boldsymbol{x}$ — координаты частицы, $\boldsymbol{v}$ — компоненты вектора скорости частицы, Δt — временной шаг. При отсутствии тела частица смещается в соответствии со своим вектором скорости: $\boldsymbol{x} \to \boldsymbol{x} + \Delta t \boldsymbol{v}$. При наличии тела необходимо найти точку пересечения траектории и поверхности тела. Если пересечение есть, то частица сначала переносится в точку пересечения (сдвигается на величину $\boldsymbol{v}t_c$, где t_c — время пересечения частицы с телом), потом производится вычисление скорости, с которой частица отражается от стенки, а затем частица смещается на величину оставшегося времени $\Delta t - t_c$, при этом также проверяется возможность пересечения с телом. Для только что созданных частиц шаг Δt берется уменьшенным пропорционально случайной величине, чтобы имитировать равномерный по времени вход частиц в область.

Программно перенос частиц осуществляется с последней частицы в списке. Если какая-либо частица вылетает из расчетной области, то она больше не нужна и уничтожается. На ее место копируется последняя частица списка, а число частиц уменьшается на единицу (рис. 3). Поскольку последняя частица в списке уже была передвинута, алгоритм переходит к следующей частице. Таким образом, частицы всегда

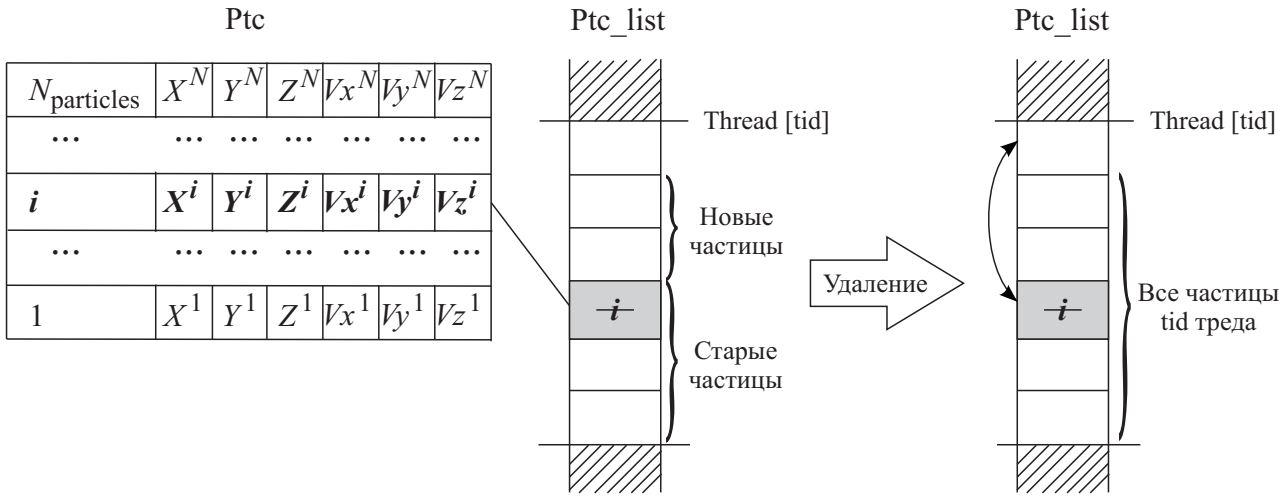

Рис. 3. Удаление частицы

хранятся в виде одного непрерывного блока.

По окончании перемещения частицы определяется, в какой ячейке она оказалась. Если эта ячейка принадлежит другому процессору, то частица копируется в буфер, предназначенный для передачи на этот процессор, а на текущем процессоре удаляется.

Обмен данными — пересылка частиц между процессорами. Обмен производится с помощью буферов, которые заполняются частицами на предыдущем этапе. Именно эта операция использует библиотеку MPI для пересылки данных и синхронизации.

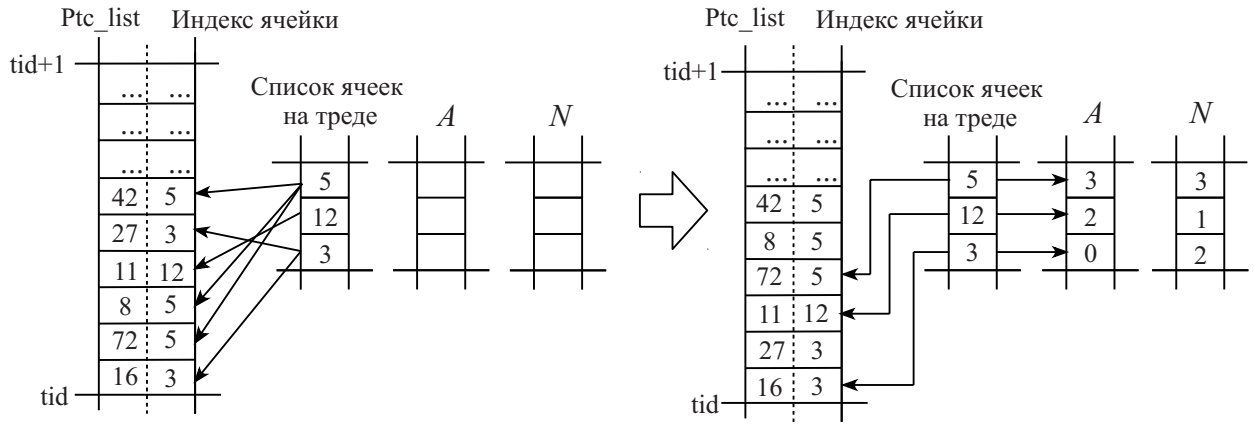

Рис. 4. Индексация частиц

Индексирование — для каждой ячейки определяется, сколько и какие частицы в ней находятся. Это необходимо для проведения столкновений. Номера частиц, упорядоченные по ячейкам, заносятся в индексный массив, а для каждой ячейки вычисляется число частиц и адрес, с которого начинаются принадлежащие ей частицы в индексном массиве (рис. 4).

Индексация проводится в три этапа.

1. Перебираются все частицы процессора и определяется число частиц N_i в каждой ячейке (i — индекс ячейки).

2. Расставляются адреса A_i, с которых будут начинаться частицы каждой ячейки. Адрес первой ячейки равен началу массива, адрес каждой следующей сдвигается на величину N_i: $A_{i+1} = A_i + N_i$. На этом же этапе число частиц в ячейке обнуляется, потому что на следующем этапе оно временно будет использоваться как счетчик.

3. Частицы перебираются вторично и заносятся в индексный массив по адресу $A = A_i + N_i$, после чего число частиц в ячейке N_i увеличивается на единицу. Таким образом, после перебора всех частиц значение переменной N_i опять будет равно числу частиц в данной ячейке.

Столкновения — моделирование бинарных столкновений частиц в каждой ячейке. Каждый процессор перебирает свои ячейки и производит столкновения принадлежащих ему частиц. Не вдаваясь в физиче-

ские детали процесса столкновения, можно отметить, что число столкновений (а соответственно, и объем вычислений) пропорционально числу частиц в ячейке и обратно пропорционально средней длине свободного пробега моделируемого течения. Для каждого столкновения необходимо случайным образом выбрать пару частиц, принадлежащих данной ячейке (именно для этого проводится индексация частиц). После этого проверяется возможность столкновения, и если оно произошло — меняются скорости частиц.

Сбор статистики — в счетчиках каждой ячейки суммируется число частиц, скорости и квадраты скоростей. Статистически обрабатывая эти счетчики, можно вычислить все параметры течения (плотность, скорость, температуру, давление, число Маха и т.д.).

В параллельных вычислениях методом ПСМ очень важно обеспечить оптимальность загрузки процессоров. В численных методах вычислительной аэродинамики, использующих разностные схемы, объем вычислений в каждой ячейке одинаков и для оптимальной загрузки процессоров достаточно равномерно распределить ячейки между процессорами. В методе ПСМ объем вычислений в ячейке зависит от числа частиц (причем зависимость нелинейная), при этом в процессе вычислений число частиц может изменяться в десятки раз. Выделение одинакового числа ячеек каждому процессору не является достаточным условием оптимальной загрузки. Существует множество алгоритмов оптимизации, которые позволяют сбалансировать нагрузку на вычислительные узлы и увеличить эффективность параллелизации. Они могут быть как динамическими (изменяющими разбиение области в процессе вычислений), так и статическими, в которых область разбивается на части один раз до вычислений и в процессе расчета не меняется. Наиболее простой в реализации статический алгоритм оптимизации — случайное распределение ячеек по процессорам. При этом на каждый процессор попадают ячейки из разных областей течения, в результате чего суммарный объем вычислений на каждом процессоре примерно одинаков. Однако данный алгоритм имеет существенный недостаток, связанный с большим объемом данных, участвующих в межпроцессорном обмене. Если передача достаточно медленная (например, на кластерах, где процессоры соединены локальной вычислительной сетью), то общая эффективность данного алгоритма заметно снижается.

2. Технология CUDA. Программно-аппаратная архитектура CUDA (Compute Unified Device Architecture, унифицированная архитектура вычислительных устройств) [2] создана компанией NVIDIA для проведения вычислений общего назначения на видеокартах производства NVIDIA. Данная архитектура предоставляет программисту высокоуровневый интерфейс к большому классу видеоустройств, скрывая от него низкоуровневые драйверы к реальным видеопроцессорам. Это позволяет работать с унифицированным виртуальным устройством и управлять вычислениями посредством широко распространенных языков (C/C++, Fortran), что существенно облегчает разработку программ.

Вычислительная часть CUDA основана на концепции "одна команда — множество данных" (ОКМД), которая подразумевает, что одна и та же инструкция одновременно применяется к множеству данных. Процедура, выполняемая на устройстве, называется *ядром* (*kernel*). Ядро выполняется большим числом вычислительных *потоков* (*threads*), каждый из которых обрабатывает свою порцию данных. Важно, что число потоков может быть больше реального числа вычислительных микропроцессоров — в этом случае потоки выполняются группами. Все потоки объединяются в одномерные, двумерные или трехмерные *блоки* (*blocks*), которые, в свою очередь, формируют одномерные, двумерные или трехмерные *сетки* (*grids*). Выбор числа потоков, размерности блоков и сеток предоставлен программисту для удобства представления данных задачи.

Центральное процессорное устройство (central processing unit, ЦПУ) изначально приспособлено для решения задач общего плана и работает с произвольно адресуемой памятью. Программы на ЦПУ могут обращаться напрямую к любым ячейкам линейной однородной памяти. В CUDA, в отличие от ЦПУ, имеется шесть типов памяти, которые перечислены в табл. 1. В этой же таблице приведен размер памяти для видеокарты GeForce GTX 460 SE. Как видно из таблицы, читать можно из любой ячейки, доступной физически, но не во все записывать с ГПУ. Это ограничение связано с конструкцией видеокарт и является компромиссом между увеличением скорости работы определенных алгоритмов и снижением стоимости оборудования. Быстрый доступ к памяти реализуется посредством кэширования. В частности, текстурная память (фактически — это одно-, дву- или трехмерный массив чисел) выделяется в глобальной памяти, но обращение к ней идет через кэш и поэтому эта память быстрее. Однако из-за особенностей кэширования в нее нельзя записывать с ГПУ. Размер этой памяти задается пользователем при создании текстуры, причем ее размер не может превышать размер глобальной памяти.

3. Методика адаптации MPI ПСМ метода к технологии CUDA. Перенос любого программного обеспечения на совершенно другую, а тем более новую, программно-аппаратную платформу всегда сложен. Желательно написать несколько программ, набраться опыта, выяснить особенности данной платформы, а потом уже переносить большие вычислительные системы. С другой стороны, имеется ра-

Таблица 1

Типы памяти в CUDA

Тип памяти	Доступ	Уровень выделения	Скорость работы	Размер GeForce GTX 460 SE
регистры	чтение/запись	на поток	высокая	32 КиБ
локальная	чтение/запись	на поток	низкая	512 КиБ
разделяемая	чтение/запись	на блок	высокая	48 КиБ
глобальная	чтение/запись	на ГПУ	низкая	1023.19 МиБ
константная	только чтение	на ГПУ	высокая	64 КиБ
текстурная	только чтение	на ГПУ	высокая	—

ботоспособная реализация какого-либо вычислительного метода, за долгие годы отработаны алгоритмы, приемы, и очень тяжело написать аналогичную программу "с чистого листа" — ведь все эти алгоритмы придется разрабатывать и тестировать заново. Кроме того, хочется быстро получить уже работающую (пусть даже не очень эффективно) программу на новой платформе, а не через несколько лет планомерного изучения этой платформы. Именно поэтому ищутся пути, как быстро и с минимальными затратами и опытом адаптировать программу к новой платформе (идеальный вариант — опция в компиляторе, но в данном случае это не реально).

Высказанные соображения приводят к следующей идее: *трактовать каждый поток видеокарты как независимый процессор* и использовать уже наработанную идеологию параллелизации для MPI. Как указывалось выше, все вычисления каждым процессором производятся независимо, в своей области памяти, поэтому вся память видеокарты делится между заданным числом потоков (процессоров). Каждый поток в параллельном режиме производит вычисления с объектами, расположенными в выделенной ему памяти, не взаимодействуя с другими потоками. Блок обмена частиц полностью заменяется, однако идеология работы всех остальных блоков полностью сохраняется, а исходный код требует лишь небольших модификаций для учета специфики программирования на CUDA. Подобная методика, как ожидается, позволит достаточно быстро получить работоспособный (возможно, не самый эффективный) программный код, увидеть узкие места, проблемы и т.п., что и даст опыт адаптации именно данного метода и станет основой для написания более эффективного программного кода.

Безусловно, даже такая методика адаптации требует некоторых базовых знаний об устройстве CUDA и принципов программирования на ней [3, 4] и изначального учета данных рекомендаций.

1. Важно заметить, что ЦПУ не имеет прямого доступа к памяти видеокарты. Можно только осуществлять операцию копирования данных посредством CUDA-функций. Это сильно затрудняет отладку программ, потому что прежде, чем посмотреть какие-либо значения переменных, их нужно передать на ЦПУ.

Существует несколько аппаратно-архитектурных версий CUDA, зависящих от возможностей видеокарты. Настоятельно рекомендуется использовать версию не ниже 2.0, которая поддерживает вычисления с двойной точностью, а также позволяет напрямую выводить информацию на стандартное устройство вывода непосредственно с видеокарты, что существенно облегчает отладку.

2. Исполняемый код следует разделять на несколько небольших ядер, которые вызываются последовательно в соответствии с логикой программы. Это позволяет более равномерно загрузить видеопроцессор и упростить отладку ядер.

3. В силу особенностей ОКМД-архитектуры (одиночный поток команд, множественный поток данных), псевдослучайные числа лучше вычислять параллельно и заносить в специальный массив, из которого они извлекаются по мере надобности. Периодически они перевычисляются для обеспечения неповторяемости последовательности чисел.

Для вычисления последовательности псевдослучайных чисел используется алгоритм вихрь Мерсенна (Mersenne Twister) [8]. Исходный код был взят из демонстрационных примеров NVIDIA CUDA SDK [9] и модифицирован для использования в методе ПСМ.

4. Необходимо минимизировать копирование данных между процессором и видеокартой, так как скорость копирования не очень большая. В идеальном случае в начале расчета в памяти видеокарты создаются массивы, в них копируется все необходимые начальные данные, затем производятся вычисления, а по их окончании результаты копируются на ЦПУ.

4. Обмен модельными частицами между потоками. Как указывалось выше, межпроцессорное взаимодействие происходит на этапе обмена, и на CUDA и MPI оно, естественно, будет отличаться. Поэтому процесс обмена будет рассмотрен более подробно. Обмен частицами между потоками осуществляется для того, чтобы на каждом потоке были собраны только частицы, расположенные в принадлежащих ему ячейках. Важно организовать процесс таким образом, чтобы не возникло коллизий — попыток разных потоков одновременно писать в одну и ту же память. Поскольку информация о частицах находится в глобальной памяти, доступной всем потокам, ее копировать не надо, а достаточно передать ее адрес (индекс), хранящийся в списке частиц потока. Поэтому, говоря о передаче частиц, мы подразумеваем передачу ее адреса (индекса).

Очевидно, что в ячейке на входной границе области поток создает частицы, перемещает их и передает другим потокам, поэтому список его частиц кончится и он не сможет создавать новые. В то же время, в ячейках на выходной границе будут скапливаться пустые "места" под частицы, потому что частицы в эту ячейку передаются и уничтожаются. Чтобы избежать дисбаланса в числе доступных потоку частиц, каждый поток, получив частицу, взамен возвращает свою неиспользуемую частицу. Таким образом, число доступных частиц всегда сохраняется одинаковым на всех потоках.

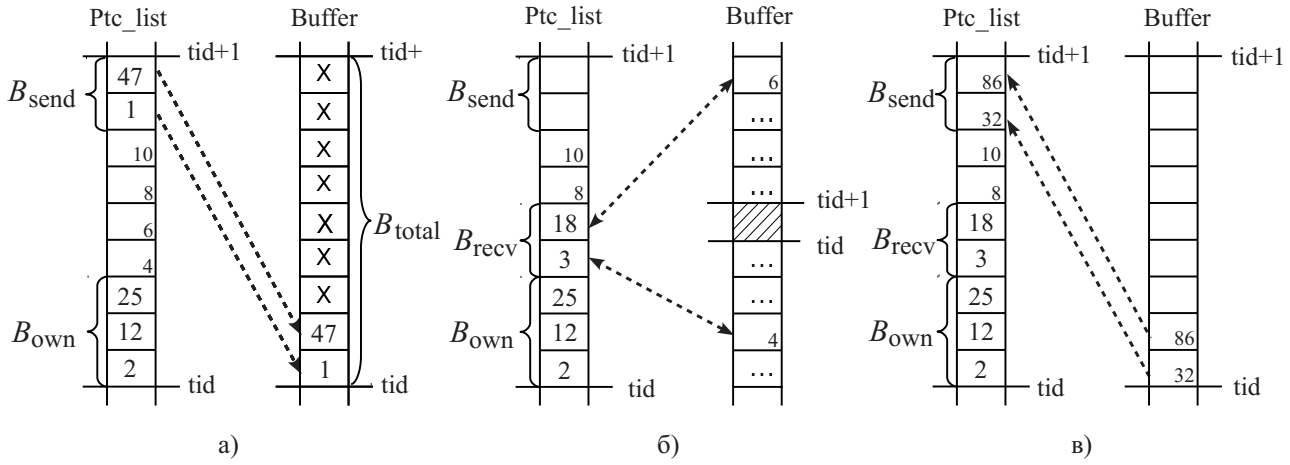

Рис. 5. Обмен частицами

Обмен осуществляется за три операции через буфер обмена, который у каждого потока свой.

1. Поток переносит в свой буфер обмена индексы частиц, которые должны быть переданы на другие потоки, при этом производится переупорядочивание частиц, в результате которого оставшиеся частицы группируются одним блоком (B_{own}) в начале списка частиц, а переданные — другим блоком (B_{send}), расположенным в конце списка (рис. 5а). Поскольку каждый поток пишет в свой буфер, коллизии не возникают. Необходимо отметить, что после завершения данного этапа блок B_{send} не содержит ссылок на частицы (так как он их передал) и не может их использовать.

2. Каждый поток просматривает буферы обмена всех других потоков и выбирает из них свои частицы. Каждый раз, когда поток находит принадлежащую ему частицу, он копирует ее в свой список частиц, а взамен в это же место буфера записывается ссылка на неиспользуемую частицу, которая будет возвращена потоку, приславшему частицу (рис. 5б). Из принятых частиц формируется третий блок — B_{recv}. Поскольку все потоки читают все буферы, возвращаемая взамен частица помечается, что она "пустая" и ее не надо анализировать при обмене с другими потоками. Поскольку каждый поток также работает со своим списком частиц, а находящаяся в буфере обмена частица может принадлежать только одному потоку, то коллизий также не возникает.

3. Каждый поток забирает из буфера обмена индексы возвращенных ему частиц и записывает их в свой B_{send} (рис. 5в). Теперь B_{send} содержит ссылки на частицы, которые можно использовать. Поскольку поток работает со своим буфером и своим списком, коллизий также не возникает.

Объединение блоков B_{own} и B_{recv} представляет собой список "реальных" частиц данного потока, а блок B_{send} с оставшимся списком — свободные (неиспользуемые) частицы. Очевидно, что максимально допустимое число частиц на потоке должно на каждом шаге быть больше, чем сумма рассмотренных блоков: $B_{\text{total}} \geqslant B_{\text{own}} + B_{\text{recv}} + B_{\text{send}}$. В противном случае произойдет перекрытие блоков и неизбежная путаница в адресах. В предельном случае, когда поток передает все свои частицы и столько же получает (равновесие по обмену), длина списка частиц на потоке должна быть в два раза больше среднего числа

частиц. Однако поскольку существуют флуктуации числа частиц, а в ячейках на наветренной стороне тела число частиц может возрастать в десятки раз, оказалось, что для моделирования одного миллиона частиц на 4096 потоках необходимо резервировать место для 4–5 миллионов частиц. Это является достаточно серьезным ограничением по памяти, которое не позволяет провести исследования больших задач (более трех миллионов частиц на видеокарте с объемом памяти 1 Гб).

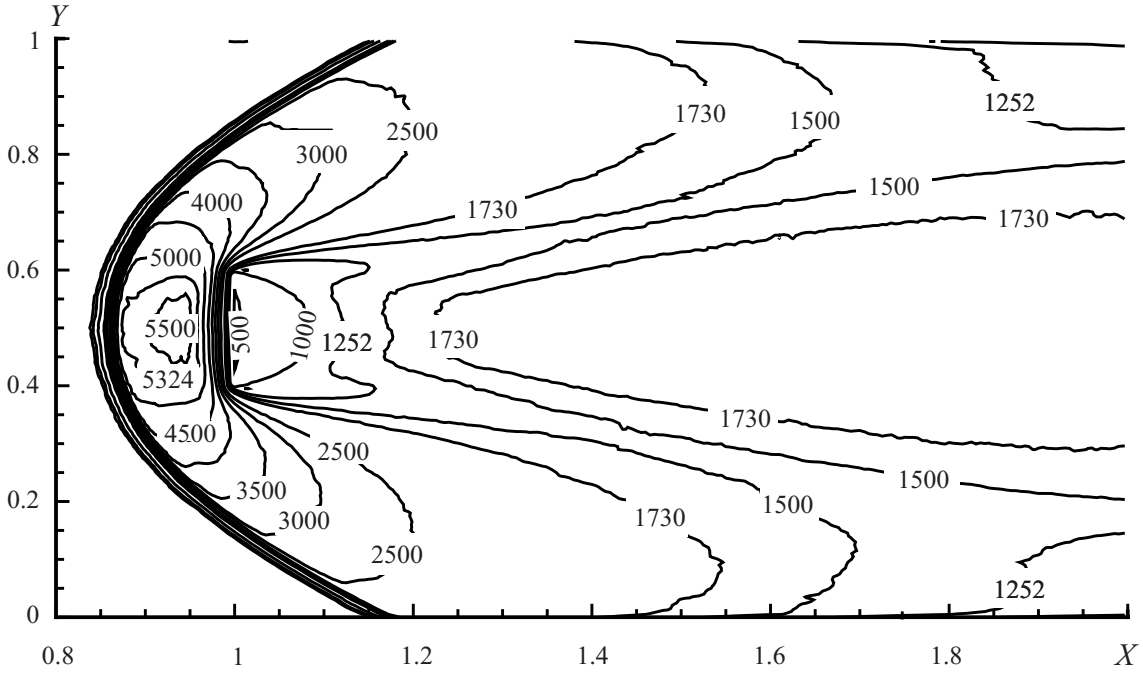

Рис. 6. Поле температуры [K]

5. Модификации алгоритма обмена и оценка их эффективности. Для оценки эффективности предложенного алгоритма были произведены замеры времени выполнения каждого этапа за один шаг по времени. В качестве тестовой задачи рассматривалось обтекание пластины длиной 0.2 м, расположенной поперек потока в центре расчетной области размером 2×1 м. Скорость невозмущенного потока составляла 7500 м/с, температура газа равнялась 300 K, плотность — 10^{-7} кг/м^3 (число Кнудсена Kn, вычисленное по длине пластины, равно 1.43). Температура пластины также равнялась 300 K, на пластине накладывались граничные условия диффузного отражения с полной аккомодацией энергии. Вычислительная сетка содержала 20 000 ячеек, временной шаг составлял 10^{-5} с. Поле температуры, полученное в расчете, представленно на рис. 6.

Таблица 2

Время вычислений одного шага, мс

Блок	$t_{\text{ЦПУ}}$	t_{CUDA}	$t_{\text{ЦПУ}}/t_{\text{CUDA}}$
Генерация	5.1	0.30	17.00
Перемещение	15.8	2.13	7.41
Индексация	28.6	3.54	8.08
Столкновения	5.9	0.84	7.02
Суммирование	19.0	3.06	6.20
Random	—	7.00	0.00
Обмен	—	380.27	0.00
Сумма	74.4	397.14	1/5.3

Вычисления проводились на видеокарте NVIDIA GeForce GTX 460 SE. Эти же вычисления были проведены с использованием однопроцессорной ЦПУ-программы на процессоре Intel Pentium E5300 @ 2.60ГГц. Полученные значения времени и ускорения представлены в табл. 2.

Вычисление последовательности псевдослучайных чисел (блок Random) на CUDA вынесено в отдельную подпрограмму, которая вызывается периодически; ее время работы может быть измерено таймером. На ЦПУ случайные числа вычисляются в момент вызова и время их вычисления просто не может быть вычислено из-за быстротечности процесса, поэтому на ЦПУ время вычисления случайных чисел равномерно “размазано” по всем другим блокам.

Из табл. 2 видно, что основные блоки вычисляются на CUDA в 5–10 раз быстрее чем на ЦПУ. Однако время пересылки частиц между потоками оказалось практически в 200 (!) раз больше, чем все другие вычисления вместе взятые, поэтому суммарная производительность оказалась почти в 6 раз хуже, чем на ЦПУ. Такая очевидная разница во времени выполнения вычислительных этапов и достаточно простого этапа обмена явно указывает, что этот блок написан крайне неэффективно с точки зрения CUDA и требует

существенного пересмотра алгоритма.

Как указывалось в разделе 4, обмен производится в три этапа:

1) занесение в буфер обмена частиц, передаваемых на другие потоки;

2) выборка каждым потоком из буфера обмена частиц, принадлежащих потоку, и занесение в буфер возвращаемых "мест" под частицы;

3) выборка из буфера возвращенных "мест" под частицы.

Проведенные дополнительные замеры времени показали, что первый этап производится за 1.43 мс, второй — 377.82 мс, третий — 1.02 мс. Очевидно, что проблема в алгоритме второго этапа. Листинг этого алгоритма приведен ниже.

```
 __device__ void TakeFromBufParticles (unsigned int *d_Map, Mem *d_mem, int  max_ptc){
// Вычисление номера текущего потока
  const int tid = blockDim.x*blockIdx.x+threadIdx.x;

  Ptc p;
  NewPart[tid] = 0;
  int num, leng, th, num2;
  int N_i, N_j;
  unsigned int ns = 20;
  leng = BufLength/NumThreads;

  // цикл по числу всех потоков
  for (int i=0; i<NumThreads; i++){
     // цикл по числу частиц в буфере i-го потока.
     for (int j=0; j<NumPtcOnBuf[i];j++){
       // номер очередной частицы в буфере
       num2 = Buffer[i*leng+j];
       // отрицательный номер указывает,
       // что частица уже взята другим потоком
       if (num2<0) continue;
       // Получаем координаты частицы и вычисляем номер ячейки
       p = d_mem->ptc[num2];
       num = d_mem->grid->IdxD(p); // индекс ячейки
       // из Map определяется номер потока,
       // которому принадлежит частица
       th = d_Map[num]>>ns;

       if (tid==th){
         // частица принадлежит данному потоку, и забирается им
         N_i = NumPtcOnTh[tid] + NumPtcOnBuf[tid];
         N_j = d_ptc_th[tid*LengthOnTh+N_i];
         d_ptc_th[tid*LengthOnTh+N_i] = num2; //Buffer[i*leng+j];
         // на то же место в буфере заносится индекс пустого места
         Buffer[i*leng+j] = N_j-(max_ptc*2);
        (NumPtcOnTh[tid])++;
        (NewPart[tid])++;
       }
     }
   }
 }
```

Поскольку алгоритм не содержит никаких сложных вычислительных элементов, становится понятно, что основная проблема в том, что все потоки одновременно читают из одной и той же области глобальной памяти (строка `num2 = Buffer[i*leng+j];`), а так как кэширования глобальной памяти нет, то, похоже, они вынуждены делать это по очереди, что и обусловливает задержку.

Как указывалось выше, в CUDA есть разделяемая (shared) память, которую видят все потоки, входящие в блок. Эта память кэшируемая, а следовательно, быстрая. Поэтому основная идея первой модификации — сначала считать содержимое буфера в разделяемую память, а потом уже читать из нее.

Поскольку, опять же, разделяемая память видна всем потокам блока, чтение производится параллельно, когда каждый очередной поток блока читает число с очередным порядковым номером, что гораздо быстрее, чем будет читать один поток (такая методика в CUDA называется “объединение” (coalescing)). Листинг фрагмента первой модификации приводится ниже:

```
...
__shared__ int shared_buf[512];
...
  for (int i=0; i<NumThreads; i++){
    // синхронизация выполнения потоков
    __syncthreads ();
    nbuf = NumPtcOnBuf[i]; // число частиц в буфере i-го потока

    // чтение из буфера в shared память.
    for (ti=threadIdx.x; ti<nbuf; ti+=blockDim.x){
      num2 = Buffer[i*leng+ti];
      shared_buf[ti] = num2;
    }

    __syncthreads (); // синхронизация выполнения потоков

    // буфер потока считан в shared память

   // цикл по числу частиц в буфере i-го потока.
   for (int j=0; j<nbuf; j++){
     // номер очередной частицы в буфере
     num2 = shared_buf[j];
  }
...
```

После такой модификации время вычисления этой функции снизилось до 107.85 мс, т.е в стало 3.5 раза меньше! Хотя удалось достигнуть значительного ускорения, все же время выполнения осталось достаточно большим.

Во второй модификации помимо номера частицы в разделяемую память заносился и поток, на который должна была передаваться частица. Это также существенно уменьшает число обращений к памяти и вычислений:

```
...
  __shared__ int shared_buf[512];
  __shared__ int shared_th[512];
...

   // чтение из буфера в shared память.
   for (ti=threadIdx.x; ti<nbuf; ti+=blockDim.x){
     num2=Buffer[i*leng+ti];
     shared_buf[ti] = num2;
     if (num2>=0){
       p = d_mem->ptc[num2];
       // индекс ячейки
       num = d_mem->grid->IdxD(p);
       // вычислить номер потока по карте
       th = d_Map[num]>>ns;
       shared_th[ti] = th;
     }
...

// цикл по числу частиц в буфере i-го потока.
   for (int j=0; j<nbuf; j++){
```

```
    // номер очередной частицы в буфере
    num2 = shared_buf[j];
    if (num2 < 0) continue;
    th = shared_th[j];
...
```

Тем не менее, время выполнения функции снизилось незначительно — до 74.30 мс. Попытки улучшить это время мелкими модификациями (например, использовать память регистров для счетчиков) существенного ускорения не принесло — порядка нескольких процентов. Таким образом, основное время выполнения тратится на то, что все потоки прямо или косвенно полностью перечитывают весь буфер.

В этой связи в следующей модификации алгоритм был перестроен для принципиального снижения операций чтения из буфера. Для этого буфер, выделенный потоку, был равномерно поделен на разделы, в каждый из которых должны заноситься частицы для определенных номеров потоков. Поскольку число пересылаемых частиц в любом случае остается неизменным, суммарный размер буфера потока остается тем же. Например, в первый раздел будут заноситься частицы для потоков с 1 по 32, во второй — с 33 по 64 и т.д. Таким образом, при чтении буфера потоки просматривают только свой раздел и нет необходимости просматривать все остальные разделы. Если, например, число разделов равно 32, то число читаемой информации должно уменьшиться в 32 раза, что должно привести к существенному ускорению. Число разделов в буфере (32) принято равным числу блоков в расчете — это упростило организацию работы с буфером, в частности, использование разделяемой памяти. Процесс чтения из разделов буфера в разделяемую память приведен на листинге ниже.

```
...
// номер текущего блока потоков, он же номер раздела в буфере
  const int blk = blockIdx.x;
...
// размер окна, в которое читаются данные из буфера
#define WIN 1024
  __shared__ int shared_buf[WIN];
  __shared__ int shared_th[WIN];
...
  // адрес начала блока blk в буфере
  int adrbuf = blk*BufLength/gridDim.x
...
   // чтение из буфера в shared память.
  for (ii=0; ii<nbuf; ii+=WIN){
     for(ti=threadIdx.x; ti<WIN; ti+=blockDim.x){
        i = ii+ti;
        if (i < nbuf){
          num2 = Buffer[adrbuf+i];
          shared_buf[ti] = num2;
...
         shared_th[ti] = th;
...
```

В этом алгоритме появился дополнительный нулевой этап, в котором сначала проводится разметка разделов в буфере. Далее все этапы те же самые. Как и ожидалось, объем читаемой информации снизился и время работы функции обмена существенно уменьшилось — до 6.98 мс! Таким образом, в результате данной модификации без изменения самого принципа обмена частиц, а только за счет изменения алгоритма, учитывающего особенности CUDA, удалось ускорить обмен почти в 55 раз!

В литературе по использованию CUDA настоятельно рекомендуют сократить использование обращений к глобальной памяти, потому что оно *медленное*. Представленный пример алгоритма обмена приведен для того, чтобы показать что используемый критерий *медленное* соответствует замедлению выполнения программы в десятки и сотни раз, и поэтому надо тщательно планировать работу с глобальной памятью. Отметим, что даже чтение из одного и того же фрагмента памяти приводит к существенным задержкам.

Время выполнения отдельных этапов различных модификаций показано в табл. 3.

В третьей модификации несколько увеличился этап занесения частиц в буфер, так как их теперь надо распределять по нескольким разделам. В этой модификации время выполнения второго этапа по сравнению со второй уменьшилось, как и ожидалось, примерно в 32 раза. По непонятной причине уменьшилось и время третьего этапа. Возможно, что из-за разделения буферов уменьшилось число столкновений потоков чтения.

Таблица 3
Время вычислений операции обмена на одном шаге, мс

Этап	Исходный	Мод. 1	Мод. 2	Мод. 3
0	—	—	—	1.86
1	1.43	1.47	1.53	2.38
2	377.82	107.85	74.30	2.18
3	1.02	1.02	0.94	0.56
Сумма	380.27	110.34	76.77	6.98

С учетом сделанной модификации, итоговое сравнение времени выполнения одного шага на ГПУ и ЦПУ показано в табл. 4.

Таблица 4
Время вычислений одного шага [мс] и эквивалентное число процессоров

Этап	Тест 1			Тест 2			
	$t_{\text{ЦПУ}}$	t_{CUDA}	$t_{\text{ЦПУ}}/t_{\text{CUDA}}$	$t_{\text{ЦПУ}}$	t_{CUDA}	$t_{\text{ЦПУ}}/t_{\text{CUDA}}$	$t_{\text{DSMC++}}$
Генерация	5.1	0.30	17.0	3.	0.26	11.5	4.2
Перемещение	15.8	2.13	7.4	32.	20.55	1.6	80.0
Индексация	28.6	3.54	8.1	154.	21.38	7.2	34.3
Столкновения	5.9	0.84	7.0	112.	48.14	2.3	19.4
Суммирование	19.0	3.06	6.2	84.	22.24	3.8	11.7
Random		7.00			3.85		
Обмен		6.98			41.43		
Сумма	74.4	23.85	3.11	385.	157.85	2.4	149.6

Значения в колонке $t_{\text{ЦПУ}}/t_{\text{CUDA}}$ фактически равны числу процессоров ЦПУ, необходимых, чтобы посчитать данный этап за такое же время, как и на CUDA. В этой же таблице приведены также время вычисления более сложной задачи (Тест 2), в которой по сравнению с Тест 1 плотность течения ($\rho = 10^{-6}$ кг/м^3) и число частиц (10^6) увеличено в 10 раз, а число ячеек 500 тысяч. Из этой таблицы видно, что большинство вычислительных этапов на CUDA выполняются в 7-8 раз быстрее чем на ЦПУ, и небольшое суммарное ускорение (2-3 раза от ЦПУ) связано с операцией обмена и неравномерностью загрузки потоков ГПУ.

Сравнение с однопроцессорными вычислениями не совсем корректно, так как это подразумевает стопроцентную эффективность параллелизации, когда полностью отсутствует потеря времени на синхронизацию и межпроцессорный обмен. Поэтому Тест 2 был посчитан также с использованием параллельной MPI-реализации метода ПСМ — программой DSMC++ [10–13] на разном числе процессоров, чтобы по суммарному времени вычислений подобрать эквивалентное число процессоров. Вычисления производились на blade-сервере HP BL2x220c G7 вычислительного центра Новосибирского государственного университета. Искомый эквивалент составил приблизительно 5 процессоров Intel Xeon X5670 с тактовой частотой 3 ГГц. Времена вычислений различных этапов данного расчета представлены в колонке $t_{\text{DSMC++}}$. Необходимо отметить, что в этап “Перемещение” также входит межпроцессорный обмен, который невозможно выделить в силу особенностей реализации обмена в программе DSMC++. Кроме того, этот код рассчитан на вычисления разнообразных физических явлений, которые в данной задаче отсутствуют, но проверка на их возможность в коде выполняется, что замедляет некоторые этапы.

Конечно, приведенное ускорение в 4–6 раз нельзя назвать большим. Однако рассмотренная методика адаптации программ позволила достаточно быстро создать работоспособную программу для CUDA и получить большой опыт по использованию CUDA. В процессе разработки CUDA-программы появились идеи по полной переработке идеологии процессов, которые обещают существенное ускорение и уменьшение необходимой для вычислений памяти. Эти идеи не могли появиться без опыта, который был получен в результате представленной работы.

Заключение. Задачей данного исследования была проверка методики, которая позволила бы адаптировать уже имеющиеся наработки параллельных вычислений (в частности, на MPI-технологии) к использованию на CUDA. Данная задача выполнена на примере адаптации метода ПСМ. Фактически,

полностью сохранена вся идеология MPI-параллелизации. Большинство логических блоков претерпели незначительные изменения (касающиеся расположения данных в памяти). Все специфические особенности архитектуры CUDA оказались сосредоточены в одном единственном блоке, детальная проработка которого и позволила получить опыт использования CUDA, поэтому данная методика может быть рекомендована для адаптации на CUDA и для других вычислительных методов.

Данная работа была выполнена при поддержке междисциплинарных интеграционных проектов СО РАН № 39, 47 и 130 (2012 г.). Выражаем благодарность Информационно-вычислительному центру Новосибирского государственного университета за предоставление вычислительных ресурсов.

СПИСОК ЛИТЕРАТУРЫ

1. MPI: A message passing interface standard. Knoxville: University of Tennessee, 1994.
2. NVIDIA GPU Computing Documentation (http://developer.nvidia.com/nvidia-GPU-computing-documentation).
3. *Фролов В.* Введение в технологию CUDA // Компьютерная графика и мультимедиа. **6**, № 1. 2008 (http://cgm.computergraphics.ru/issues/issue16/cuda).
4. *Боресков А.* Основы CUDA (http://www.steps3d.narod.ru/tutorials/cuda-tutorial.html).
5. *Бёрд Г.* Молекулярная газовая динамика. М.: Мир, 1981.
6. *Bird G.A.* Molecular gas dynamics and the direct simulation of gas flows. Oxford: Clarendon Press, 1994.
7. *Ivanov M.S., Rogazinsky S.V.* Analysis of the numerical techniques of the direct simulation Monte-Carlo method in the rarefied gas dynamics // Soviet J. Numer. Anal. Math. Modelling. 1988. **3**, N 6. 453–465.
8. *Matsumoto M., Nishimura T.* Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator // ACM Trans. on Modeling and Computer Simulations. 1998. **8**, № 1. 3–30.
9. CUDA C/C++ SDK CODE Samples (http://developer.nvidia.com/cuda-cc-sdk-code-samples#MersenneTwister).
10. *Kashkovsky A.V., Markelov G.N., Ivanov M.S.* An object-oriented software design for the direct simulation Monte Carlo method // AIAA Paper N 2001–2895. 2001.
11. *Kashkovsky A.V., Bondar Ye.A., Zhukova G.A., Ivanov M.S., Gimelshein S.F.* Object-oriented software design of real gas effects for the DSMC method // Proc. of the 24th Int. Symp. on Rarefied Gas Dynamics. Vol. 762. New York: AIP Press, 2005. 583–588.
12. *Kashkovsky A.V., Vashchenkov P.V., Ivanov M.S.* Object-oriented software design for the three-dimensional direct simulation Monte Carlo method // Proc. of the 25th Int. Symp. on Rarefied Gas Dynamics. St. Petersburg, 2006. 456–461.
13. *Kashkovsky A.V., Vashchenkov P.V., Ivanov M.S.* Object-oriented software design approach for multidimensional application of direct simulation Monte Carlo method // Proc. of the 13th Int. Conf. on Methods of Aerophysical Research. 5–10 February, 2007. Part 4. Novosibirsk, 2007. 49–54.

Поступила в редакцию
23.03.2012