Параллельная реализация метода расщепления для системы из нескольких GPU с применением в задачах аэрогидродинамики*

С.Б. Березин[1], И.С. Каргапольцев[1], Н.Д. Марковский[2], Н.А. Сахарных[1,2]

МГУ им. М.В. Ломоносова[1], NVIDIA Ltd.[2]

Метод прямого численного моделирования турбулентных течений требует сильно детализированных сеток и огромных вычислительных ресурсов. Современные GPU имеют высокую пропускную способность памяти и вычислений с плавающей точкой, поэтому эти устройства очень хорошо подходят для решения задач аэрогидродинамики. Однако, из-за ограничения на доступный объем памяти, для расчета больших сеток необходимо эффективное распределение работы между отдельными GPU в системе. В этой статье будет предложен эффективный параллельный алгоритм метода расщепления для нескольких GPU и рассмотрены варианты балансировки работы для системы с несколькими узлами. Также будет проведен анализ производительности метода на различных входных данных.

1. Введение

Быстрый рост производительности графических процессоров (GPU) в течение последнего десятилетия открыл новые возможности для использования GPU в реальных приложениях, требующих больших вычислительных мощностей. Несколько лет назад к решению подобных задач можно было прийти только путем использования суперкомпьютеров и кластеров с большим числом процессоров. Сейчас в нашем распоряжении находятся гораздо более энерго-эффективные и компактные устройства GPU с массивно-параллельной архитектурой, которые могут использоваться как со-процессоры, значительно ускоряя расчеты.

Относительно недавно появились удобные инструменты для разработки приложений общего назначения для GPU - такие как CUDA, OpenCL, Parallel Nsight [1]. Эти средства значительно упростили разработку сложных приложений, эффективно использующих параллельную архитектуру GPU.

Аэрогидродинамика является, пожалуй, одной из самых сложных областей моделирования явлений природы, требующих огромного числа вычислительных ресурсов. Численное моделирование течений жидкостей и газов требует применения самых современных суперкомпьютеров для получения результата достаточной точности за приемлемое время..

2. Решение уравнений Навье-Стокса на GPU

Течения жидкостей и газов в естественной среде (движение воздуха в земной атмосфере, воды в реках и морях, газа в атмосферах звезд и т.п.) и инженерных сооружениях (в трубах, каналах, струях, пограничных слоях около движущихся в жидкости или газе твердых тел, следах за такими телами и т.п.) могут быть описаны системой уравнений Навье-Стокса. Полная форма уравнений Навье-Стокса в трёхмерной декартовой системе координат для идеального газа с постоянной плотностью имеет вид:

$$\nabla \cdot \vec{\mathbf{u}} = 0, \tag{1}$$

$$\rho \left[\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right] = -\nabla T + \frac{1}{Re} \nabla^2 \mathbf{u}, \tag{2}$$

$$\rho \left[\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T \right] = \frac{1}{Re \cdot Pr} \Delta T + \frac{\gamma - 1}{\gamma \cdot Re} \Phi, \tag{3}$$

*Работа выполнена при финансовой поддержке РФФИ, проекты № 10-01-00288-а, № 09-07-00424-а.

$$p = \rho RT, \tag{4}$$

где $\mathbf{u} = (u, v, w)$ – компоненты вектора скорости, p – давление, ρ – плотность. Здесь уравнение неразрывности 1 выражает постоянство объема, 2 – закон сохранения энергии, 3 – уравнение теплопроводности, и замыкает систему уравнение состояния 4. Число Рейнольса $Re = \frac{UL}{\nu}$ характеризует свойства течения: $\nu = \frac{\mu}{\rho}$ – кинематическая вязкость, μ – вязкость, а L и U – средняя скорость и размер области в конкретной задаче. В уравнениях также присутствуют число Прандтля Pr, газовая постоянная R и удельная теплоемкость γ. Φ – это диссипативная функция, характеризующая работу вязких сил:

$$\Phi = \Phi_x + \Phi_y + \Phi_z$$

$$\begin{aligned}
\Phi_x &= 2(\frac{\partial u}{\partial x})^2 + (\frac{\partial v}{\partial x})^2 + (\frac{\partial w}{\partial x})^2 + \frac{\partial u}{\partial y}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial z}\frac{\partial w}{\partial x}, \\
\Phi_y &= (\frac{\partial u}{\partial y})^2 + 2(\frac{\partial v}{\partial y})^2 + (\frac{\partial w}{\partial y})^2 + \frac{\partial v}{\partial x}\frac{\partial u}{\partial y} + \frac{\partial v}{\partial z}\frac{\partial w}{\partial y}, \\
\Phi_z &= (\frac{\partial u}{\partial z})^2 + (\frac{\partial v}{\partial z})^2 + 2(\frac{\partial w}{\partial z})^2 + \frac{\partial w}{\partial x}\frac{\partial u}{\partial z} + \frac{\partial w}{\partial y}\frac{\partial v}{\partial z}.
\end{aligned} \tag{5}$$

Подробный вывод уравнений, их описание и применение можно найти в [2].

Во многих случаях течение становится турбулентным – скорость, температура и давления начинают хаотично изменяться во времени и пространстве. В случае турбулентных течений численное решение системы (1) - (4) требует явного описания всего диапазона пространственных и временных масштабов, что возможно при числе узлов сетки пропорциональном $Re^{9/4}$ [3]. Обычно турбулентность наступает при больших числах Re, порядка нескольких тысяч или десятков тысяч. Таким образом, практическое применение численного моделирования течений на основе системы уравнений Навье-Стокса сильно ограничено объемом памяти и производительностью существующих вычислительных систем. Существуют модели LES (large eddy simulation), основанные на воспроизведении наиболее значимых вихрей и параметризации оставшейся части спектра [3, 4], что позволяет снизить требования к разрешению сетки и вычислительным ресурсам, но требует выдвижения некоторых априорных утверждений о природе моделируемого течения, что не всегда представляеся возможными. В данной работе будет рассмотрен метод прямого численного решения системы уравнений Навье-Стокса без дополнотельных упрощений.

3. Метод покоординатного расщепления и соответствующий разностный метод первого порядка

Parallel reduction for 3D Navier-Stokes equation and its 1-st order discretization

Метод покоординатного расщепления применяется для решения параболических или эллиптических уравнений в частных производных. Идея метода заключается в расщеплении уравнений на несколько более простых – по уравнению вдоль каждой из осей координат. Эта операция проводится таким образом, что производные вдоль соответствующего направления берутся неявно, а остальные координаты считаются постоянными.

Например, уравнение для x-компоненты 2

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = -\frac{\partial T}{\partial x} + \frac{1}{Re}(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}) \tag{6}$$

расщепляется на 3 уравнения 7-9. Применение к ним конечно-разностной схемы первого порядка приводит к набору независимых трехдиагональных систем.

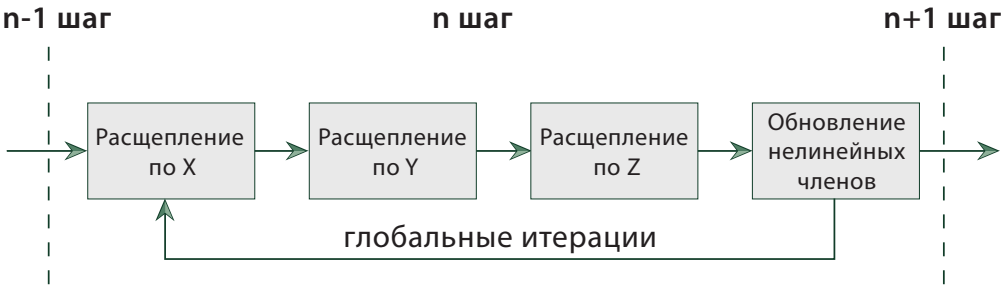


Рис. 1. Целый шаг метода покоординатного расщепления.

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = -\frac{\partial T}{\partial x} + \frac{1}{Re}\frac{\partial^2 u}{\partial x^2}, \tag{7}$$

$$\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial y} = \frac{1}{Re}\frac{\partial^2 u}{\partial y^2}, \tag{8}$$

$$\frac{\partial u}{\partial t} + w\frac{\partial u}{\partial z} = \frac{1}{Re}\frac{\partial^2 u}{\partial z^2}. \tag{9}$$

Остальные уравнения могут быть преобразованы аналогичным образом. В общем случае метод расщепления может быть применен к задаче любой размерности.

Поскольку уравнения Навье-Стокса являются нелинейными, необходимо также проводить итерации для обновления нелинейных коэффициентов. В данной реализации проводятся итерации двух типов: глобальные и локальные. Глобальные итерации применяются для целого временного шага для всей системы, локальные – для каждого дробного шага, соответствующего одному из направлений. Общая схема алгоритма приведена на рис. 1. На каждом целом временном шаге система расщепляется по трем направлениям, и соответствующие системы уравнений последовательно решаются в глобальных итерациях. Число глобальных итераций выбирается так, чтобы численная ошибка, рассчитываемая как невязка в уравнении неразрывности, после каждого шага не превышала установленную величину.

На каждом дробном шаге расщепления соответствующие уравнения решаются при использовании конечно-разностной схемы. На рис. 2 показана модель данных и расчетов для отдельного шага расщепления. Основное вычислительное ядро – это решатель наборов трехдиагональных систем. Полученные решения систем используются для обновления нелинейных коэффициентов. После этого проводится несколько локальных итераций для уменьшения численной ошибки.

Подобная схема была использована для решения двумерных уравнений Навье-Стокса в работе [5]. В дальнейшем на основе этой идеи был реализован численный метод для трехмерного случая и динамическая система визуализации [6].

4. Реализация метода прогонок на одном GPU

В CUDA-реализации прогонки каждая нить решает ровно одну трёхдиагональную систему. За счет массивной параллельности нитей на GPU и большого числа систем удается эффективно загрузить устройство. На рис. 3 показано распределение трехмерного массива на дробном шаге расщепления по отдельным нитям. Каждая нить обрабатывает отдельный столбец массива в определенном направлении. Код соответствующего ядра аналогичен

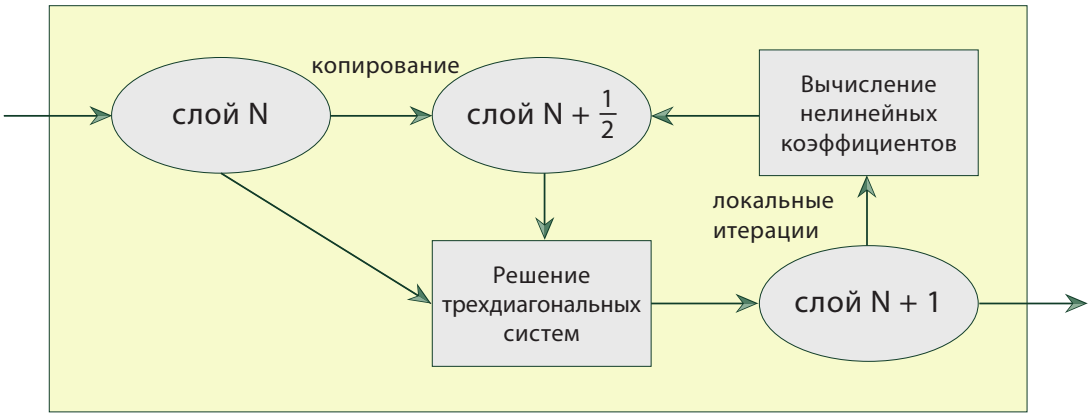


Рис. 2. Дробный шаг расщепления.

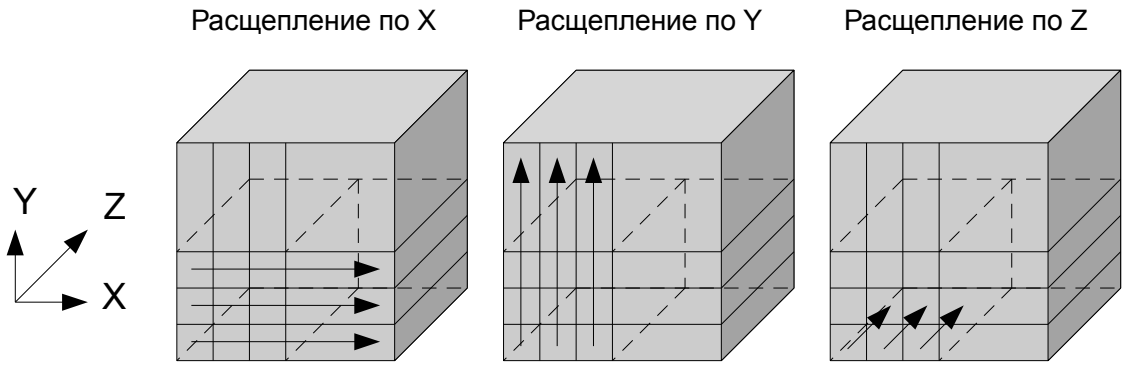


Рис. 3. Прогонки вдоль каждого направления расщепления.

CPU-версии (см. Рис. 4).

```
__device__ void solve_tridiagonal(
        FTYPE *a, FTYPE *b, FTYPE *c, FTYPE *d, FTYPE *x,
        int num, int id, int num_seg, int max_n )
{
    get(c,num-1) = 0.0;

    get(c,0) = get(c,0) / get(b,0);
    get(d,0) = get(d,0) / get(b,0);

    // прямой ход прогонки
    for (int i = 1; i < num; i++)
    {
      get(c,i) = get(c,i) / (get(b,i) - get(a,i) * get(c,i-1));
      get(d,i) = (get(d,i) - get(d,i-1) * get(a,i)) /
                 (get(b,i) - get(a,i) * get(c,i-1));
    }

    get(x,num-1) = get(d,num-1);

    // обратный ход прогонки
    for (int i = num-2; i >= 0; i--)
      get(x,i) = get(d,i) - get(c,i) * get(x,i+1);
}
```

Рис. 4. Реализация алгоритма прогонки на CUDA. Одна нить решает ровно одну систему.

Часто для достижения максимальной эффективности на GPU приходится менять структуру данных, а иногда и сам метод.

Анализ показал, что подобные прогонки вдоль оси Z работают медленнее, чем по X и Y. Это объясняется тем, что прогонки вдоль Z приводят к чтению **каждой нитью** последовательных значений в памяти. В результате, запросы нитей варпа к памяти не объединяются (uncoalesced, см. [1]). Напротив, в прогонках вдоль X и Y, **соседние нити** читают последовательные элементы в памяти, и все запросы объединяются. Оптимизировать прогонки по Z можно одним из двух способов: либо менять метод, либо менять шаблон доступа к данным. В данном случае реализован второй способ: входной массив данных транспонируется, а затем вместо Z прогонки запускается Y-прогонка. Эффективное транспонирование возможно за счет использования разделяемой памяти. В таблице 1 показано насколько быстрее работает улучшенный вариант с переупорядочиванием данных в одинарной и двойной точности.

Реализация метода расщепления тестировалась на двух задачах. В первой задаче моделировалось течение внутри прямоугольного канала со стенками параллельными осям координат (box_pipe). Такой вид границ позволяет равномерно загрузить вычислительное устройство. Во второй задаче рассматривалась геометрия акватории Белого моря (white_sea). В этом случае можно оценить насколько хорошо метод работает на реальных задачах со сложными границами и неравномерной загрузкой устройства. Тестирование производительности проводилось на одном GPU (Tesla C1060 или Tesla C2050), а также на одном многоядерном CPU (Intel Core i7, 4 ядра, 2.8GHz). На CPU метод был распараллен с помощью директив OpenMP и использовал все доступные ядра. Ниже приведены таблицы результатов по каждому направлению расщепления и для всего метода в целом.

5. Реализации метода прогонок на нескольких GPU

Схема работы метода прогонок на нескольких GPU, подключенных к одному хосту может состоять из следующих трёх стадий:

- Разделение данных между GPU
- Распараллеливание кода

Таблица 1. Влияние транспонирования на производительность (среднее время работы ядра в миллисекундах), NVIDIA Tesla C2050.

точность	одинарная			двойная		
ядро	оригинал	с трансп.	ускорение	оригинал	с трансп.	ускорение
X dir	523	528	**1.0x**	717	709	**1.0x**
Y dir	525	531	**1.0x**	694	686	**1.0x**
Z dir	1681	533	**2.4x**	1901	693	**2.7x**
трансп.		164			190	
всего	2729	1756	**1.6x**	3312	2278	**1.5x**

Таблица 2. Результаты тестов производительности на модели box_pipe (систем в секунду)

направление	CPU (4 ядра)	Tesla C1060	Tesla C2050	Tesla C2050 vs CPU
X	1.55	5.61	17.73	**11.4x**
Y	2.17	5.36	18.11	**8.3x**
Z	2.34	5.64	18.12	**7.8x**
все	1.96	5.30	16.09	**8.2x**

Таблица 3. Результаты тестов производительности на модели white_sea (систем в секунду)

направление	CPU (4 ядра)	Tesla C1060	Tesla C2050	Tesla C2050 vs CPU
X	3.65	15.40	37.95	**10.4x**
Y	5.12	16.89	44.91	**8.8x**
Z	2.25	5.65	13.38	**5.9x**
все	3.82	11.88	27.40	**7.2x**

- Синхронизация результатов между GPU

Скорость обмена данными между GPU существенно ниже скорости глобальной памяти, поэтому необходимо минимизировать количество пересылаемой информации. Разделение данных между GPU лучше всего провести вдоль направления X, поскольку все трехмерные массивы хранятся по строкам в виде одномерного вектора $(\mathrm{i}, \mathrm{j}, \mathrm{k}) \rightarrow [\mathrm{dimY} \cdot \mathrm{dimZ} \cdot \mathrm{i} + \mathrm{dimZ} \cdot \mathrm{j} + \mathrm{k}]$, и сетка естественным образом разбивается на блоки размера, кратного $\mathrm{dimY} \cdot \mathrm{dimZ}$. В этом случае распараллеливание прогонок вдоль Y и Z тривиально и сводится к одновременному запуску ядер на нескольких GPU, работающих над соответствующими часятми сетки (см. Рис. 5).

```
for (int i = 0; i < nGPU; i++)
{
    cudaSetDevice(i); // Переключение устройства
    kernel<<<...>>>(devArray[i], ...); // Расчет своей области сетки
}
```

Рис. 5. Одновременный запуск вычислительных ядер на нескольких GPU.

Прогонка вдоль направления X разделяется между всеми GPU и выполняется последовательно, по цепочке: каждый GPU сначала ожидает данные, затем вычисляет часть результатов и отправляет их следующему GPU, используя cudaMemcpyPeer (рис. 6).

Таким образом, с ростом числа GPU, прироста производительности вдоль оси X не будет, и, согласно закону Амдала, это послужит основным лимитирующим фактором производительности при дальнейшем масштабировании. Тем не менее, другие части шага по X – вычисление производных и диссипативной функции – можно эффективно распределить между GPU, за счёт чего достигается хороший скейлинг на нескольких GPU, подключённых к одному хосту (рис. 7).

В силу того, что вычисление производных у границ требует доступа к граничным значениям частей сетки соседних GPU, необходима их эффективная синхронизация на каждом шаге метода. Синхронизацию такого рода можно реализовать с помощью функции прямого асинхронного копирования *cudaMemcpyPeerAsync* (или *cudaMemcpyAsync* при использовании UVA) (рис. 8). Также было проведено исследование влияния на производительность возможности прямого копирования данных между GPU в устройствах Tesla поколения Fermi минуя оперативную память процессора.

Доступна реализация метода для нескольких GPU с использованием технологии MPI. Код можно использовать на нескольких нодах с различным количеством GPU на каждой из нод. Полный исходный код модели, использующей данную реализацию метода можно найти на сайте проекта [7].

6. Заключение

В данной статье была представлена эффективная GPU-реализация метода покоординатного расщепления для решения полной системы уравнений Навье-Стокса в трехмерной области. Численный метод был протестирован на модельной задаче и продемонстрировал хорошие результаты по производительности.

Один из важных результатов работы - это успешная реализация на системах с разделенной памятью с несколькими GPU. Учитывая возрастающие потребности науки, очень важно иметь параллельную реализацию, использующую все доступные ресурсы. Одно устройство имеет относительно небольшой объем памяти и вычислительных ресурсов. Обобщение программы на несколько GPU устройств позволяет легко масштабировать расчеты и эффективно использовать вычислительные кластеры для моделирования сложных течений.

Среди ближайших планов является имплементация параллельного алгоритма без ограничения скейлинга вдоль направления X. Для этого элементы сетки вдоль направления

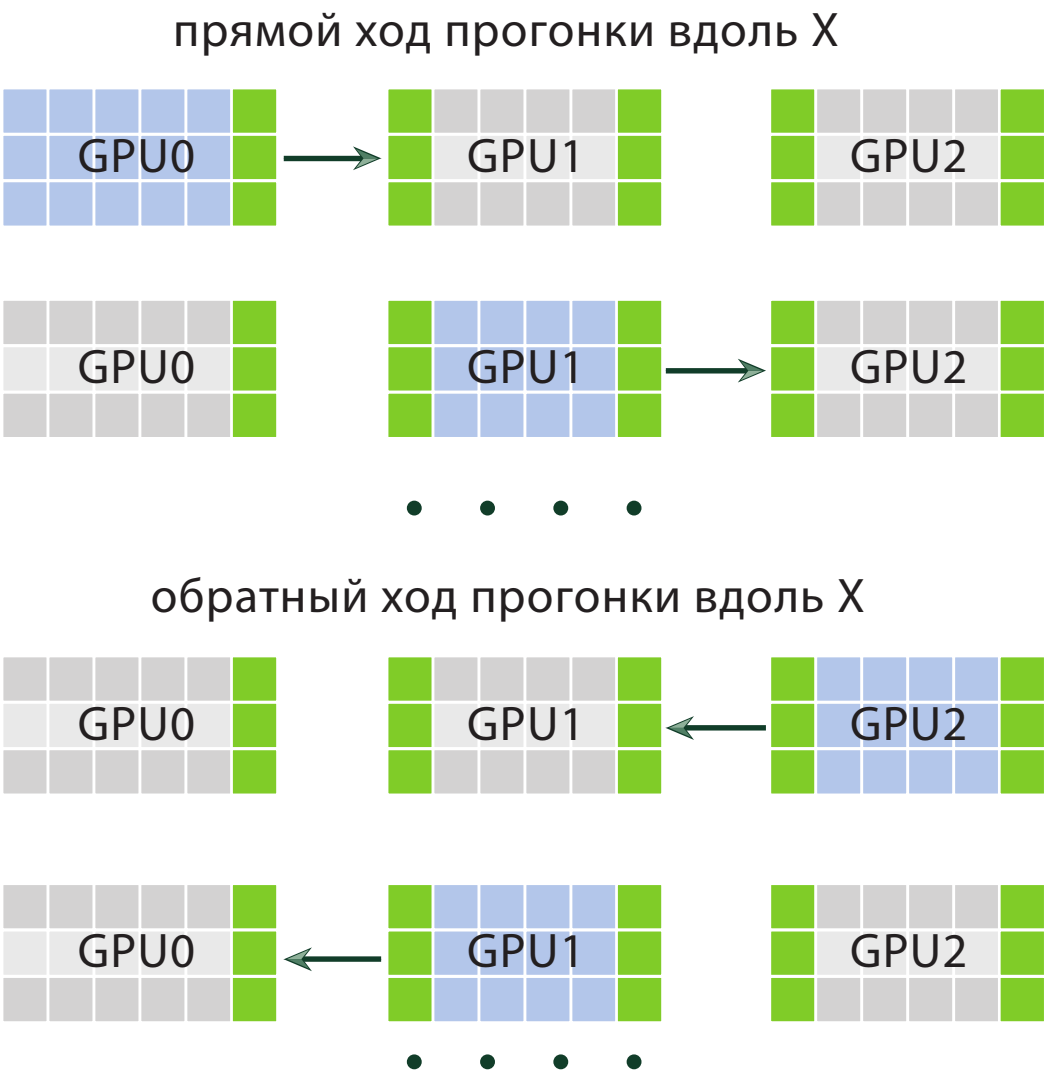
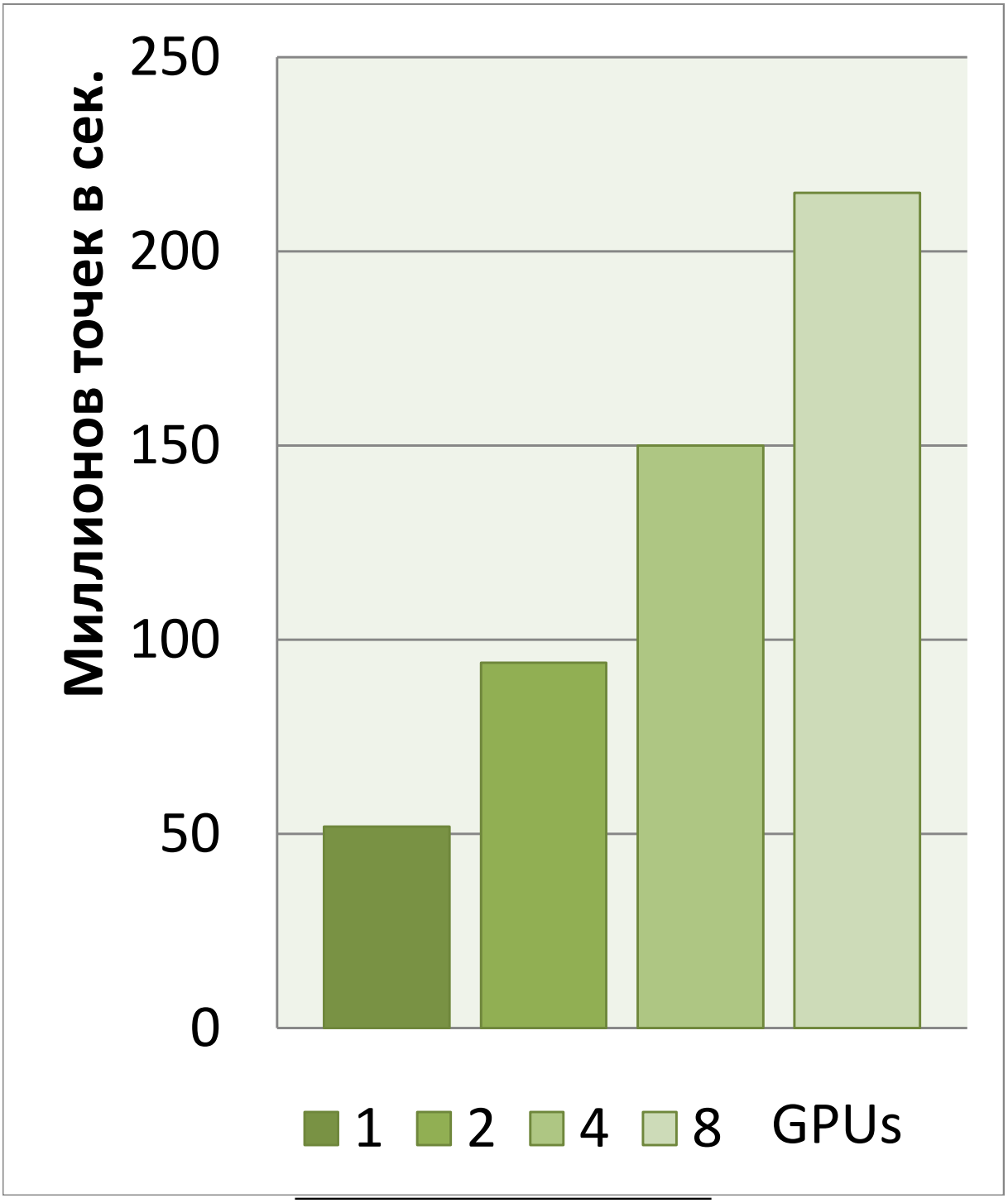


Рис. 6. В каждый момент времени работает только один GPU над своей частью прогонки.



1 GPU	2 GPU	4 GPUs	8 GPUs
51.9	94.1	150.0	215.0

Рис. 7. Производительность системы на базе Tesla M2050, измеряемая в количестве точек обработанного объема в секунду для 1–8 GPUs.

```
// Массивы указателей с граничными полосами на каждом устройстве:
float **dev_halo_right; //  правая синхронизируемая область
float **dev_halo_left; //  левая синхронизируемая область
float **dev_data_right; // правая граничная область
float **dev_data_left; //  левая граничная область
float **dev_data; // область, которую не требуется синхронизировать

cudaStream_t *devStream; // один поток на устройство

// Выделение и инициализация массивов и потоков на каждом устройстве.
...

int haloSize = ...;

// Синхронизация правой области с левой на устройстве i+1.
for( int i = 0; i < numDev - 1; i++)
    cudaMemcpyPeerAsync(dev_halo_left[i+1], i+1, dev_data_right[i], i,
        sizeof(float) * haloSize, devStream[i]);

// Синхронизация левой области с правой на устройстве i-1.
for( int i = 1; i < numDev; i++)
    cudaMemcpyPeerAsync(dev_halo_right[i-1], i-1, dev_data_left[i], i,
        sizeof(float) * haloSize, devStream[i]);

// Ожидание окончания копирования:
for( int i = 0; i < numDev; i++)
{
    cudaSetDevice(i);
    cudaStreamSynchronize(devStream[i]);
}

// Очередной полный пересчёт узлов в отдельных частях сетки.
for( int i = 0; i < numDev; i++)
{
    // Переключение индекса текущего GPU.
    cudaSetDevice(i);
    kernel<<<grid, block, 0, devStream[i]>>>(dev_data[i], dev_data_left[i],
                  dev_data_right[i], dev_halo_left[i], dev_halo_right[i]);
}

...
```

Рис. 8. Пример синхронизации между устройствами путем асинхронного копирования

Z разбиваются на блоки XY, в которых выполнение прогонок вдоль X для одних блоков покрывается прогонками вдоль X и Y для других. (Наличие локальных итераций вдоль каждого направления будет компенсироваться большим количеством блоков). Ожидается, что эффективность данного алгоритма будет улучшаться с размером сетки.

Литература

1. NVIDIA Inc. NVIDIA CUDA programming guide, version 4.1. 2011.
2. Флетчер К. Вычислительные методы в динамике жидкостей. Том 2 // Мир, М., 1991.
3. Лапин Ю.В. Статистическая теория турбулентности // Научно-технические ведомости 2(36)/2004, Сп.Б., Изд-во Политехнического университета.
4. Pierre Sagaut Large eddy simulation for incompressible flows, 3rd edition // Springer, 2006
5. Пасконов В. М., Березин С. Б. Неклассические решения классической задачи о течении вязкой несжимаемой жидкости в плоском канале // Прикладная математика и информатика, №17: МАКС Пресс, Москва, 2004.

6. Пасконов В. М., Березин С. Б., Корухова Е. С. Динамическая система визуализации для многопроцессорных систем с общей памятью и ее применение для численного моделирования турбулентных течений вязких жидкостей // Вестник Московского университета, Серия 15: Вычислительная математика и кибернетика, 2007. С. 7–16.

7. Sakharnykh N., Berezin S., Paskonov V. Fluid solver based on Navier-Stokes equations using finite difference methods in areas with dynamic boundaries:
URL: http://code.google.com/p/cmc-fluid-solver/