Использование графического ускорителя вычислений для моделирования динамики жидкости методом *Lattice-Boltzmann*

А. С. Мордвинцев

4 декабря 2007 г.

Содержание

Введение

В данной работе рассматривается подход к моделированию поведения жидкости, известный в англоязычной литературе как метод *Lattice-Boltzmann* [1-3]. Этот активно развивающийся с 90-х годов 20-го века подход основан на применении клеточных автоматов для моделирования некоего подобия микродинамики жидкой среды. Оказалось, что этот метод вполне жизнеспособен, а в некоторых ситуациях даже превосходит традиционные подходы.

Здесь представлены две реализации этого метода. Первая использует для вычислений только центральный процессор компьютера (*Central Processing Unit*, *CPU*), а вторая переносит большой объем вычислений на графический ускоритель (*Graphics Processing Unit*, *GPU*). Выполнено сравнение производительности этих реализаций.

Для проверки адекватности модели производится симуляция возникновения некоторых известных природных феноменов.

1. Теоретические основы

1.1. Жидкая среда

Поведение жидкости может принимать множество разнообразных форм — от достаточно простых (например, стационарный поток в трубе) до очень сложных и непредсказуемых (турбулентный поток). На практике для изучения нетривиальных течений используют разнообразные экспериментальные установки, аэродинамические трубы, а также компьютерное моделирование. Для того, чтобы получить представление о том, насколько сложным и интересным может быть поведение жидкости, можно ознакомиться с книгой [4].

Известным примером такого нетривиального поведения жидкости является так называемая вихревая дорожка *Кармана* (рис. 1), возникающая при определенных условиях при обтекании вязким потоком цилиндрического препятствия. Явление заключается в формировании за препятствием цепочки вихрей, которые плывут вниз по течению. Оси вращения вихрей параллельны оси цилиндра, а направление вращения чередуется.

Примером ситуации, для которой существует простое аналитическое описание потока, является так называемое течение *Пуазейля*. Это стационарное ламинарное течение вязкой жидкости по цилиндрической трубе. Показано, что в этом случае возникает параболическое распределение

Рис. 1. Вихревая дорожка Кармана

скоростей по сечению трубы. Однако, это решение применимо только к достаточно маленьким скоростям потока, узким трубам или очень вязким жидкостям. При невыполнении требуемых условий течение становится турбулентным, возникают очень сложные и непредсказуемые возмущения.

1.2. Уравнение *Навье-Стокса*. Классический подход к численному моделированию жидкости

Движение текучих сред описывается уравнениями *Навье-Стокса* [5-7]. В случае вязкой несжимаемой жидкости уравнение имеет вид

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{\nabla p}{\rho} + \nu \nabla^2 \mathbf{u}, \tag{1.1}$$

$$\nabla \cdot \mathbf{u} = 0, \tag{1.2}$$

где $\mathbf{u}$ — скорость, p — давление, ρ — плотность, а ν — кинематическую вязкость среды. Во многих случаях, этой модели достаточно для адекватного описания поведения жидкости. Предположение о несжимаемости среды часто можно использовать и при описании «сжимаемых» текучих сред (например, воздуха при комнатной температуре), если скорости движения в моделируемой ситуации значительно меньше скорости звука.

Нелинейность уравнения *Навье-Стокса* делает невозможным его аналитическое решение в большинстве случаев. Поэтому для изучения реальных течений используют экспериментальные установки и компьютерное моделирование.

Наибольшее распространение получили методы [8], которые позволяют строить в моделируемом пространстве регулярную или нерегулярную сетку, а затем численно решать дискретную форму уравнений *Навье-Стокса* на ней.

1.3. Статистический подход к описанию жидкости. Уравнение *Больцмана*

На микроскопическом уровне жидкость можно представить множеством частиц, взаимодействующих между собой по законам классической или квантовой механики. Для компьютерного моделирования даже небольшого объема жидкости этот подход неприемлем из-за невыполнимых требований к вычислительным ресурсам.

Другой подход к описанию текучих сред предлагает статистическая физика. Можно рассматривать не поведение каждой частицы в отдельности, а вероятностное распределение $f(\mathbf{r}, \mathbf{c}, t)$, определенное таким образом, что $f(\mathbf{r}, \mathbf{c}, t)d\mathbf{r}d\mathbf{c}$ обозначает количество частиц в момент времени t, находящихся в параллелепипеде, ограниченном точками $\mathbf{r}$ и $\mathbf{r} + d\mathbf{r}$, скорости которых лежат интервале между $\mathbf{c}$ и $\mathbf{c} + d\mathbf{c}$. Предположим, что на все частицы действует сила $\mathbf{f}$. Тогда, если частицы вещества не сталкиваются между собой, то изменение распределения f со временем должно происходить на основании уравнения:

$$f(\mathbf{r} + \mathbf{c}dt, \mathbf{c} + \frac{\mathbf{f}}{m}dt, t + dt)d\mathbf{r}d\mathbf{c} - f(\mathbf{r}, \mathbf{c}, t)d\mathbf{r}d\mathbf{c} = 0, \tag{1.3}$$

где m — масса частицы, а dt — малый интервал времени.

Однако в жидкостях и газах частицы сталкиваются между собой. Для учета изменения распределения частиц, создаваемого этими столкновениями, в правую часть уравнения (1.3) добавляют член $\Omega(f)d\mathbf{r}d\mathbf{c}dt$. Поделив обе части полученного уравнения на $d\mathbf{r}d\mathbf{c}dt$ и устремив dt к нулю, получим уравнение *Больцмана*:

$$\frac{\partial f}{\partial t} + \frac{\partial f}{\partial \mathbf{r}} \cdot \mathbf{c} + \frac{\partial f}{\partial \mathbf{c}} \cdot \frac{\mathbf{f}}{m} = \Omega(f). \tag{1.4}$$

Локальные плотность, скорость и плотность внутренней энергии можно найти непосредственно из функции распределения при помощи

следующих интегралов:

$$\rho(\mathbf{r},t) = \int mf(\mathbf{r},\mathbf{c},t)d\mathbf{c}, \tag{1.5}$$

$$\rho(\mathbf{r},t)\cdot\mathbf{u}(\mathbf{r},t) = \int m\mathbf{c}f(\mathbf{r},\mathbf{c},t)d\mathbf{c}, \tag{1.6}$$

$$\rho(\mathbf{r},t)\cdot e(\mathbf{r},t) = \frac{1}{2}\int m\|\mathbf{c}-\mathbf{u}\|^2 f(\mathbf{r},\mathbf{c},t)d\mathbf{c}. \tag{1.7}$$

Из термодинамики известно, что внутренняя энергия связана с температурой соотношением:

$$e = \frac{3}{2m}k_B T, \tag{1.8}$$

где k_B — постоянная Больцмана, T — температура.

Оператор столкновений должен быть определен так, чтобы выполнялись законы сохранения массы, импульса и энергии.

Одним из используемых на практике операторов столкновений является оператор *BGK*, названый по именам создателей (*Бхатнагар — Гросс — Крук*). Он имеет следующий вид:

$$\Omega = -\frac{1}{\tau}(f(\mathbf{r},\mathbf{c},t) - \overline{f}(\mathbf{r},\mathbf{c},t)), \tag{1.9}$$

где τ обозначает скорость релаксации и определяет время между столкновениями частиц, а $\overline{f}$ — равновесное распределение, соответствующее распределению Максвелла-Больцмана:

$$\overline{f} = \frac{\rho}{m}\left(\frac{m}{2\pi k_B T}\right)^{3/2}\exp\left[\frac{-m(\mathbf{c}-\mathbf{u})^2}{2k_B T}\right]. \tag{1.10}$$

При помощи метода *Chapman-Enskog expansion*, описанного в работе [2], можно показать, что поведение среды, описанной уравнением *Больцмана* с соответствующим оператором столкновений ведет себя в соответствии с уравнениями *Навье-Стокса*.

1.4. Клеточные автоматы. Решетчатый газ

Концепция клеточных автоматов появилась в середине 20-го века. Джон фон Нейман пытался построить математическую модель механизма, способного к саморазмножению. Его интересовали фундаментальные требования, накладываемые природой на такие системы. Один из его коллег, Станислав Улам, порекомендовал ему обратить внимание на модель, которую он использовал на тот момент в своем исследовании роста

кристаллов. Основываясь этих идеях, Нейман создал среду, представлявшую сбой двумерную решетку, каждая ячейка которой могла принимать одно из 29 состояний. За такт глобального времени все ячейки изменяли состояние, согласно сложному набору правил. При этом новое состояние ячейки определялось только ее состоянием и состояниями соседних с ней ячеек на предыдущей итерации. В этом мире Нейману удалось построить конфигурацию, состоящую приблизительно из двухсот тысяч клеток, способную с саморепликации. Ее описание можно найти в статье [9].

Самым известным клеточным автоматом является игра *Жизнь*, придуманная Джоном Конвеем в 1970 году. Она получила популярность после статьи [10] Мартина Гарднера в журнале Scientific American. Конвей пытался упростить автомат Неймана, сохранив при этом возможность построения самокопирующихся структур, и преуспел в этом. Эта игра, несмотря на простоту правил, открывает большой простор для построения конфигураций со сложным и интересным поведением. Благодаря этому игра получила очень большую популярность, особенно среди людей с техническим образованием.

Клеточные автоматы хорошо зарекомендовали себя при построении моделей физических процессов, таких, как формирование кристаллов, течение жидкости, распространение волн, перенос тепла и многих других. Некоторые из этих моделей описаны в работе [11].

Одной из первых успешных попыток имитации поведения текучих сред при помощи клеточных автоматов стала модель *решетчатого газа* [1], получившая название *FHP*, по первым буквам имен придумавших ее ученых (*Frisch-Hasslacher-Pomeau*). Эта модель имитирует поведение системы частиц одинаковой массы, передвигающихся с фиксированными скоростями по гексагональной решетке (рис. 2). Когда в одной ячейке оказывается более одной частицы, используется оператор столкновений для определения нового состояния ячейки. Рассмотрим подробнее вариант модели *FHP*, имеющий самый простой оператор столкновений. Он обозначается как *FHP-I*.

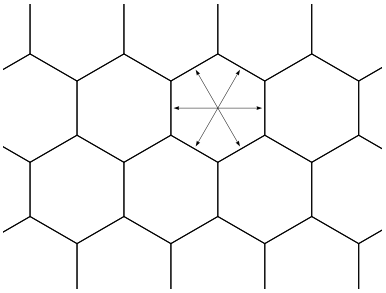

Рис. 2. Решетка и скоростные каналы модели *FHP-I*

Модель *FHP-I* имеет следующие свойства:

- Частицы перемещаются по однородной гексагональной решетке, в которой каждая ячейка связана с шестью соседними.
- Векторы, соединяющие центры соседних ячеек, называются скоростными векторами, и могут быть описаны следующим образом:

$$\mathbf{c}_i = \left(\cos\left(\frac{\pi}{3}i\right), \sin\left(\frac{\pi}{3}i\right)\right), i = 0 \ldots 5. \tag{1.11}$$

- Каждая ячейка содержит шесть скоростных каналов, каждому из которых соответствует скоростной вектор.
- В каждом скоростном канале каждой ячейки может находиться не более одной частицы.
- Все частицы обладают одинаковой массой (для простоты равной единице) и неразличимы.
- Эволюция состояния решетки во времени происходит путем чередования двух фаз: *столкновения* и *переноса*.
- В столкновении участвуют только частицы, находящиеся в одной ячейке.

Во время фазы переноса все частицы перемещаются по скоростным каналам в соответствующие соседние ячейки. Направление движения частиц при этом сохраняется. В фазу столкновения частицы в ячейках перераспределяются по скоростным каналам в соответствии с правилами, представленными на рис. 3. Если распределение частиц по каналам ячейки отличается от показанного на рисунке, состояние ячейки не изменяется.

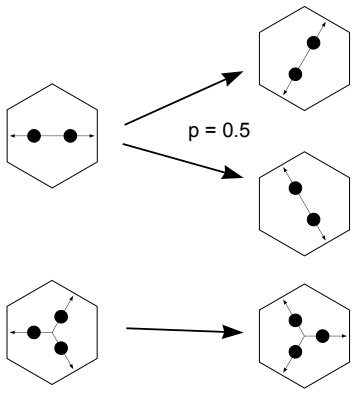

Рис. 3. Столкновения в модели *FHP-I*

Заметим, что исход лобового столкновения двух частиц является недетерминированным. Однако на практике можно использовать различные детерминированные варианты правил для разных ячеек решетки, или выбирать вариант результата столкновения в зависимости от номера итерации. В таком случае алгоритм становится детерминированным и даже обратимым.

Формально, оператор столкновений можно записать в виде булевой формулы, выражающей новое состояние скоростного канала ячейки через состояние всех ее каналов на предыдущей итерации. Основным требованием к оператору столкновений является сохранение массы и импульса.

Для задания граничных условий можно использовать другие операторы столкновений. Например, для создания стенок часто используют правило, которое разворачивает все залетевшие в ячейку во время фазы переноса частицы.

Существуют расширения модели *FHP-I*, известные как модели *FHP-II* и *FHP-III*. В модель *FHP-II* вводится дополнительный скоростной канал, соответствующий частицам, находящимся в покое и несколько новых правил, а в модели *FHP-III* (рис. 4) количество различных столкновений доведено до максимума. Эти модели обладают несколько лучшими свойствами по сравнению с *FHP-I*.

Несмотря на грубость микродинамики, используемой в моделях, основанных на решетчатом газе, они позволяют воспроизводить многие явления, возникающие в реальных средах. Вот что, по словам коллег, говорил об этом факте Ричард Фейнман:

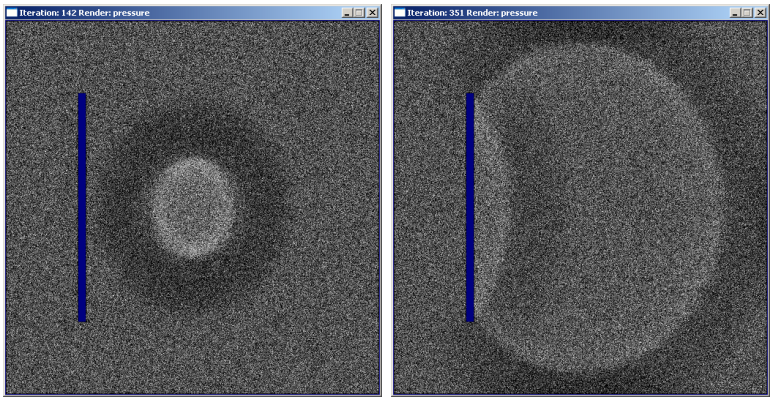

Рис. 4. Распространение звуковой волны в модели *FHP-III*

> Мы заметили, что поведение текучих сред в природе очень слабо зависит от свойств составляющих их частиц. Например, течение песка очень похоже на течение воды или на течение кучи шариков. Мы воспользовались этим фактом для того, чтобы создать воображаемую частицу, поведение которой нам очень легко симулировать. Эта частица является идеальным шариком, который может двигаться с одинаковой скоростью в одном из шести направлений. При достаточно больших масштабах, поток таких шариков очень похож на поток обыкновенной жидкости.[1]

Однако, модель *FHP* обладает несколькими серьезными недостатками, описанными в работе [2]:

- «Шумные» результаты. Для получения гладких значений скорости приходится усреднять данные по большому количеству ячеек. Для этого необходимы очень большие размеры решетки, что требует большого объема памяти и вычислительных ресурсов.

[1] В оригинале на `http://www.longnow.org/views/essays/articles/ArtFeynman.php`

> We have noticed in nature that the behavior of a fluid depends very little on the nature of the individual particles in that fluid. For example, the flow of sand is very similar to the flow of water or the flow of a pile of ball bearings. We have therefore taken advantage of this fact to invent a type of imaginary particle that is especially simple for us to simulate. This particle is a perfect ball bearing that can move at a single speed in one of six directions. The flow of these particles on a large enough scale is very similar to the flow of natural fluids.

- Некорректное поведение. Анализ показывает, что поведение данной модели в макроскопическом переделе хоть и похоже на среду, описываемую уравнением *Навье-Стокса*, все же имеет серьезные отклонения.

- Трудности моделирования трехмерной среды. Построение модели решетчатого газа для трехмерной среды достаточно нетривиально, а ее реализация требует больших вычислительных ресурсов.

2. Метод *Lattice-Boltzmann*

Метод *Lattice-Boltzmann* [1-3], появившийся в конце 80-х – начале 90-х годов 20-го века является развитием идеи использования клеточных автоматов для моделирования упрощенной микродинамики текучих сред. Оказалось, что этот подход вполне способен конкурировать с традиционными методами, развивавшимися уже не один десяток лет.

В литературе часто рассматривают метод *Lattice-Boltzmann* с двух позиций. Исторически, это усовершенствование метода решетчатого газа, призванное избавиться от характерного для него шума. Для этого в скоростных каналах ячеек вместо бита, определяющего есть ли там частица или нет, хранится действительное число, определяющее вероятность нахождения частицы в канале. Возник вопрос, каким должен быть оператор столкновений для такой модели. Сначала использовался оператор, построенный на основе булевой формулы, используемой в модели *FHP*. Затем его последовательно упрощали, и, в итоге, пришли к оператору, основанному на модели столкновений *BGK*. Этот вариант оператора столкновений и стал классическим, хотя известны работы, посвященные другим вариантам столкновений, например, [15].

Одним из достоинств метода *Lattice-Boltzmann* является возможность выбора оператора столкновений, отражающего особенности микродинамики исследуемой среды. Таким образом, удается моделировать явления, с которыми плохо справляются традиционные методы, основанные непосредственно на имитации макроскопических свойств жидкости. К таким явлениям относятся течения через пористые среды, разделение несмешиваемых жидкостей [14] и т.д.

С другой стороны можно рассматривать данный метод как дискретизацию уравнения *Больцмана* и численное решение дискретного его варианта. В этом случае можно интерпретировать численное значение, содержавшееся в скоростном канале не как вероятность, а как удельную плотность частиц, движущихся в заданном направлении в данной точке.

Как оказалось, при дискретизации в методе *Lattice-Boltzmann* можно использовать не только гексагональную решетку, необходимую в модели *FHP*, но и более привычную прямоугольную. Так же появляется свобода при выборе характеристик решетки для трехмерных симуляций. Для решеток часто используют обозначения вида $DxQy$, где x – размерность решетки, y – количество скоростных каналов в каждой ячейке (количество соседних клеток, влияющих на ее новое состояние на следующей итерации).

Примеры решеток:

- *D2Q7* – гексагональная двухмерная решетка, где седьмой скоростной канал соответствует частице, находящейся в покое;
- *D2Q9* – прямоугольная двухмерная решетка, четыре частицы двигаются параллельно осям координат , четыре по диагонали и одна находится в покое;
- *D3Q19* – одна из решеток, используемых в трехмерных моделях.

В данной работе используется решетка *D2Q9*. Для описания работы алгоритма введем некоторые обозначения. Пусть вектор $\mathbf{r}$ обозначает центр какой-либо ячейки квадратной решетки. Введем вектора $\mathbf{c}_i$:

$$\begin{aligned} \mathbf{c}_0 &= \mathbf{0}, \\ \mathbf{c}_{1,5} &= (\pm 1, 0), \\ \mathbf{c}_{3,7} &= (0, \pm 1), \\ \mathbf{c}_{2,4,6,8} &= (\pm 1, \pm 1), \end{aligned}$$

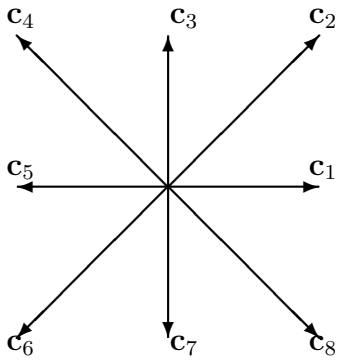

Рис. 5. Скоростные каналы для решетки *D2Q9*

Каждая ячейка используемой модели содержит девять скоростных каналов. Канал с индексом $i = 0 \ldots 8$ соответствует частице, движущейся со скоростью $\mathbf{c}_i$. Пусть $f_i(\mathbf{r}, t)$ обозначает плотность частиц в скоростном канале i ячейки $\mathbf{r}$ в момент времени t.

Локальную плотность и скорость среды для ячейки можно записать как

$$\rho = \sum_i f_i(\mathbf{r}, t), \tag{2.1}$$

$$\mathbf{u} = \frac{1}{\rho} \sum_i \mathbf{c}_i f_i((r), t). \tag{2.2}$$

Заметим, что эти формулы схожи с формулами (1.5) и (1.6), выражающими сходные характеристики через функцию распределения в уравнении *Больцмана*.

Итерацию метода *Lattice-Boltzmann* можно записать следующим образом:

$$f_i(\mathbf{r} + \mathbf{c}_i, t + 1) - f_i(\mathbf{r}, t) = \Omega_i(f(\mathbf{r}, t)), \tag{2.3}$$

являющимся дискретным аналогом уравнения *Больцмана*.

Член Ω_i обозначает оператор столкновения. При использовании модели столкновений BGK он записывается ввиде:

$$\Omega_i = -\frac{1}{\tau} \left(f_i(\mathbf{r}, t) - \overline{f_i}(\mathbf{r}, t) \right). \tag{2.4}$$

В этой формуле τ обозначает скорость релаксации. В работе [3] показано, что в описываемой модели эта величина позволяет настраивать вязкость жидкости, причем между τ и вязкостью имеется следующая зависимость: $\nu = (\tau - 0.5)/3$. Функция $\overline{f_i}$ обозначает равновесное распределение и зависит от локальных скорости и плотности среды. В данной работе для его вычисления используется следующая формула:

$$\overline{f_i} = W_i \rho \left(1 + 3 \cdot (\mathbf{c}_i \cdot \mathbf{u}) + \frac{9}{2} (\mathbf{c}_i \cdot \mathbf{u})^2 - \frac{3}{2} \mathbf{u}^2 \right). \tag{2.5}$$

Коэффициенты W_i равняются следующим величинам:

$$W_0 = \frac{4}{9}, \quad W_{1,3,5,7} = \frac{1}{9}, \quad W_{2,4,6,8} = \frac{1}{36}. \tag{2.6}$$

В принципе, этих формул уже достаточно для написания программы для моделирования жидкости. Остается только задать граничные условия. Для создания препятствий для потока в этой работе используется самый простой вариант — прямая аналогия с моделью *FHP*. Частицы просто разворачиваются в обратном направлении в ячейках, находящихся на границах препятствий. Этот метод имеет определенные недостатки, на которые указаны в статье [2], но он показал себя неплохо. Для создания потока, состояние ячеек за границами решетки предполагается

постоянным и равным равновесному распределению, соответствующему требуемой скорости потока и плотности.

3. Реализация на графическом ускорителе

За последние несколько лет резко выросла производительность графических ускорителей (далее *GPU*, *Graphics Processing Unit*) современных компьютеров. Также значительно увеличилась их функциональность. Ранее ускорители могли использовать ограниченное, жестко заданное производителем и стандартами множество методов преобразования координат, расчета освещения, наложения текстур и т.д. Теперь же стало возможным программирование различных этапов построения изображения. Благодаря этому появилось много новых графических эффектов, которые мы видим в современных компьютерных играх. Это новые методы построения теней, процедурная генерация геометрии и текстур, волны на поверхности воды, отражения, преломление лучей и т.д. Ещё одним заметным усовершенствованием *GPU* стала поддержка чисел с плавающей точкой для представления изображений. В графике эта возможность используется в основном для построения *HDR*[2]-изображений и сохранении промежуточных результатов в многопроходных алгоритмах.

GPU изначально были ориентированы на хорошо распараллеленные высокопроизводительные вычисления. К тому же они хорошо распространены и относительно недороги. Из-за этого на них обратили внимание инженеры и ученые, которым необходимо выполнять объемные расчеты для обработки результатов экспериментов, моделирования физических процессов. Оказалось, что дешевые *GPU* иногда даже превосходят по эффективности дорогие специализированные вычислители, заточенные под конкретную задачу. Так же нетрадиционным применением графических ускорителей заинтересовались разработчики игр, применившие их не только для графических эффектов, но и для расчетов игровой физики или искусственного интеллекта в реальном времени.

Так появился термин *GPGPU* (*General-Purpose computation on GPU*), обозначающий «непрофильное» использование *GPU* для решения слабо связанных традиционной интерактивной графикой задач. Ускоритель в данном случае используется в качестве потокового вычислителя с большим количеством параллельно работающих процессоров.

Приложение может использовать вычислительные мощности *GPU* различными способами. Традиционный способ, когда при реализации алгоритма программист оперирует непосредственно с графическим *API* и описывает отдельные шаги алгоритма виде небольших программных шейдеров, описан в разд. 3. Однако, этот подход не очень удобен из-за того, что программисту нужно изучать особенности графических *API*

[2] *High dynamic range* (Расширенный динамический диапазон)

(*OpenGL* или *Direct3D*) и специальные языки, которые в них используются для написания кода, выполняющегося на видеокарте (*GLSL*, *Cg*, *HLSL*).

Для того, чтобы упростить разработку *GPGPU*-приложений, было создано несколько библиотек, которые скрывали от пользователя тонкости работы с *GPU*, выставляя наружу интерфейсы для выполнения различных операций над массивами данных, например, операций линейной алгебры, таких как умножение матриц или решение систем уравнений. Примерами таких библиотек являются *BrookGPU*[3] и *RapidMind*[4].

Производители видеокарт, заметив интерес к необычным применениям своей продукции, создали свои средства для создания *GPGPU*-приложений. Разработка *ATI* называется *CTM*[5], а *NVIDIA* — *CUDA*[6]. Использование этих средств позволяет снять некоторые принципиальные ограничения, накладываемые графическими *API* и более гибко и эффективно использовать ресурсы видеокарты.

Несмотря на преимущества, на момент написания этой работы все упомянутые методы имели один серьезный недостаток, сужающий область их применения. При вычислениях можно использовать только действительные числа с одинарной точностью (*IEEE 754 Single Precision Floating Point*). Использование двойной точности пока не доступно, однако *NVIDIA* утверждает, что в скором времени будут выпущены ускорители с поддержкой двойной точности вычислений.

Основной задачей данной работы была реализация вычислительного алгоритма на центральном процессоре (*CPU*) и на *GPU* для сравнения производительности этих реализаций. В качестве «подопытного кролика» избран метод *Lattice-Boltzmann* для моделирования динамики жидкой среды. Этот, базирующийся на клеточных автоматах алгоритм, распараллеливается, что делает его идеальным кандидатом для реализации на потоковом процессоре, в качестве которого используется *GPU*.

В качестве демонстрационного примера к статье была разработана программа `GPUflow` (рис. 6), позволяющая настраивать параметры симуляции потока в интерактивном режиме, изменять конфигурацию препятствий и запускать как *CPU*, так и *GPU*-версию алгоритма для сравнения производительности.

Это приложение написано на языке *C++* под платформу *win32*. Для его работы необходима видеокарта *NVIDIA* не ниже шестой серии. Работоспособность проверена на картах: *GeForce 6600*, *GeForce Go*

[3] `http://graphics.stanford.edu/projects/brookgpu/`

[4] `http://www.rapidmind.net`

[5] *Close To Metal*

[6] *Compute Unified Device Architecture*

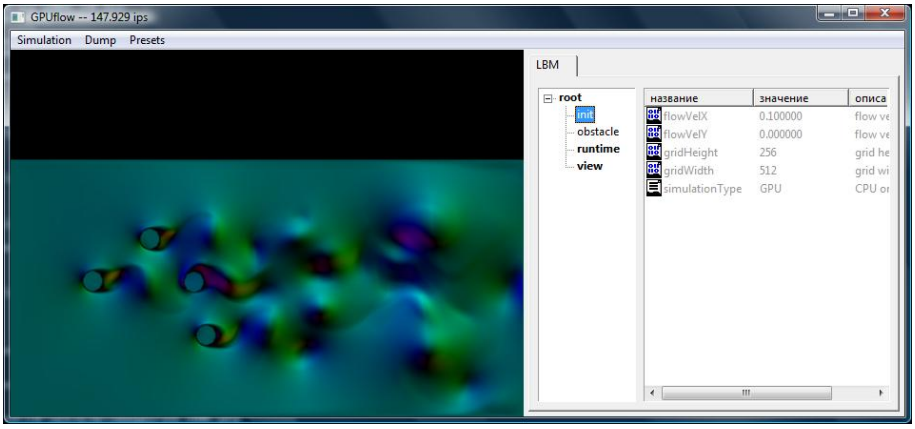

Рис. 6. Окно демонстрационного приложения

7600, *GeForce 8800*. В приложении А можно найти прокомментированные фрагменты кода *CPU*-реализации алгоритма. *GPU*-реализация будет рассмотрена в разд. 3. Примеры использования программы будут приведены в главе 4.

В этом разделе кратко описана суть GPGPU-реализации алгоритма. В этой работе взаимодействие с видеокартой происходит при помощи *API OpenGL*. Для написания шейдеров (программ, выполняющихся на видеокарте) используется язык *Cg* [13], разработанный компанией *NVIDIA*.

Итак, для хранения состояния решетки клеточного автомата в данной реализации используются три текстуры с внутренним форматом `GL_FLOAT_RGB32_NV`. Размеры каждой из этих текстур совпадают с размерами решетки. Девять значений, находящихся в скоростных каналах ячейки с координатами (x, y), упакованы в тексели этих текстур способом, показанным на рис. 7. Всего присутствует два таких набора текстур. В одном хранится предыдущее состояние решетки, а в другой на этапе переноса или столкновения записывается обновленное. Для записи в текстуры используется расширение *OpenGL* `GL_EXT_framebuffer_object`. К `framebuffer`'у подключено сразу три текстуры, что позволяет, используя расширение `GL_ARB_draw_buffers`, обновлять все три текстуры за один проход. В отличие от *CPU*-реализации, фазы переноса и столкновения здесь разнесены на два отдельных прохода. Исходный код соответствующих этим фазам шейдеров можно найти в приложении В. Препятствия в *GPU*-реализации создаются на фазе столкновений при помощи выполнения в заданных участках решетки специального шейдера, разворачивающего влетающие в ячейки частицы.

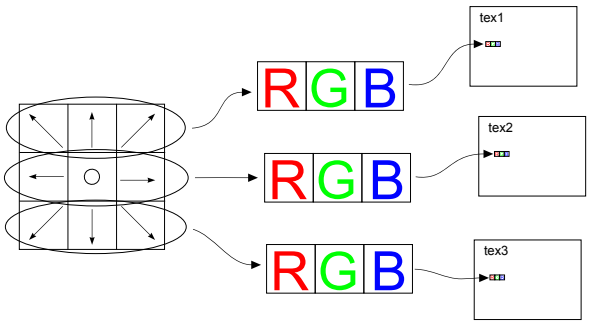

Рис. 7. Представление состояния решетки набором текстур

Обратим внимание на метод визуализации поля скоростей, используемый в данной работе. Вектор скорости преобразуется в цвет из цветового пространства *HSL*, который затем преобразуется в формат *RGB*, используемый в *OpenGL* для представления цветов. В статье [12] описана процедура перехода от *HSL* к *RGB*. В пространстве цветов *HSL* цвет представляется тремя величинами: оттенком (hue), насыщенностью (saturation) и яркостью (lightness). Обозначим их как h, s и l. Тогда h будет равняться углу между осью абсцисс и вектором скорости, пройденным в направлении против часовой стрелки, l равняется декартовой норме вектора скорости, умноженной на яркость, а h – единица. Яркость изображения регулируется через панель настроек демонстрационного приложения. На рис. 8 показана диаграмма, отображающая соответствие между вектором скорости и цветом визуализации.

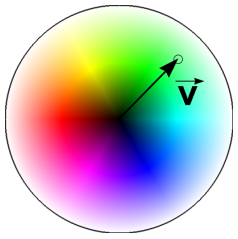

Рис. 8. Диаграмма соответствия между направлением вектора скорости и цветом визуализации

4. Вычислительные эксперименты

4.1. Ламинарный поток в трубе

Качество той или иной физической модели можно оценить, проверив, проявляются ли в этой модели явления, свойственные исследуемой системе. Для жидких сред в качестве одного из таких явлений можно использовать течение ламинарного потока в трубе (известное также как *поток Пуазейля*).

Рассмотрим двухмерный поток жидкости между двумя бесконечными плоскостями (рис. 9). В случае, когда течение стационарно и направление движения жидкости параллельно граничным плоскостям, существует несложное аналитическое решение уравнения *Навье-Стокса*. В соответствии с этим уравнением, на границах скорость потока будет равняться нулю, а при удалении от них скорость будет возрастать, достигая максимума на середине. Профиль скорости вдоль оси **y** будет образовывать параболу, а вдоль оси **x** скорость будет постоянной.

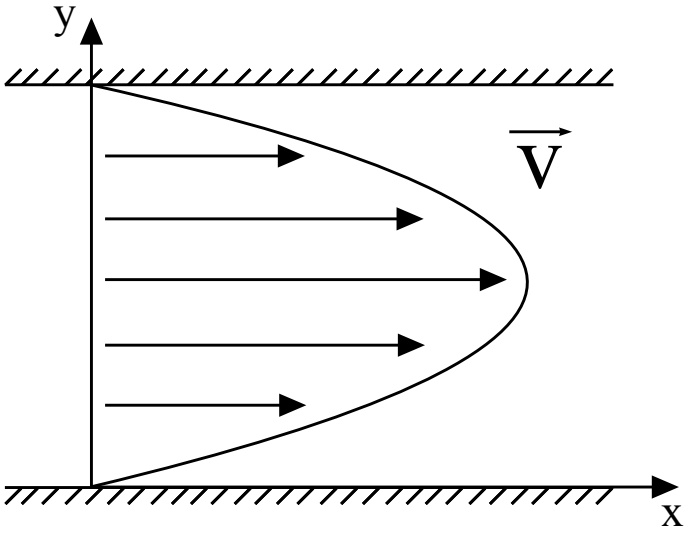

Рис. 9. Поток *Пуазейля*

Была сделана попытка получить параболическое распределение скоростей в симуляции методом *Lattice-Boltzmann*. В эксперименте использовалась решетка размером 1024x100. Ячейки, лежащие на верхней и нижней границе решетки, были помечены как препятствия. Правая и левая границы были установлены в состояние, соответствующее равновесному распределению, вычисленному по формуле (2.5), где $\rho = 1$,

$\mathbf{u} = (0.01, 0)$. Число *Рейнольдса* варьировалось от эксперимента к эксперименту изменением вязкости среды при помощи подстройки коэффициента релаксации в формуле (2.4). Перед началом эксперимента все свободные ячейки решетки инициализировались состоянием, идентичным состоянию правой и левой границ. После запуска симуляции система постепенно изменяла свое состояние, формируя неоднородное распределение скоростей между стенками. Когда поле скоростей стабилизировалось, выполнялся снимок распределения скоростей вдоль центрального столбца решетки. Распределения скоростей при различных числах *Рейнольдса* изображены на рис. 10.

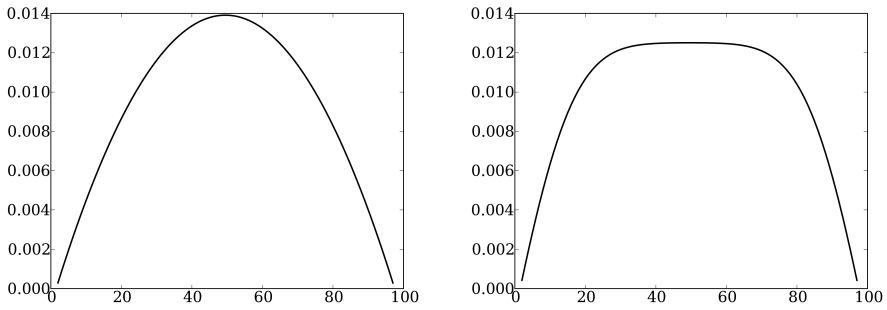

Рис. 10. Профили скоростей в симуляции течения *Пуазейля*. Слева $Re = 10$, справа $Re = 1000$

Число *Рейнольдса* является важной характеристикой потока жидкости, и вычисляется по формуле $Re = h \cdot u/\nu$, где h — характерное для исследуемой ситуации расстояние (в данном случае расстояние между плоскостями, равное 100), u — скорость набегающего потока, а ν — вязкость жидкости. Для небольших чисел *Рейнольдса* симуляция формирует параболическое распределение, однако максимальное значение скорости оказывается несколько меньше ожидаемого. Вероятно, это вызвано нереалистичными граничными условиями. Возможно, именно по этой причине с увеличением числа *Рейнольдса* распределение начинает отклоняться от параболического.

4.2. Вихревая дорожка *Кармана*

Более интересным явлением, возникающим в природе, является так называемая вихревая дорожка *Кармана* (рис. 11). Вихревая дорожка — это последовательность вихрей, формирующаяся в потоке за цилиндрическим препятствием. Она возникает при числах *Рейнольдса* больших 40-

50. Число *Рейнольдса* в этом случае обычно определяют как $Re = d \cdot u/\nu$, где d — диаметр препятствия, u — скорость потока жидкости (или скорость движения препятствия в ней), а ν – кинематическая вязкость.

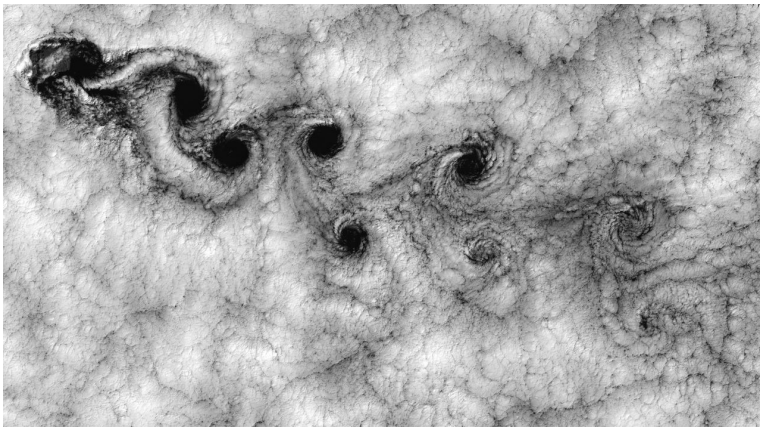

Рис. 11. Вихревая дорожка в атмосфере

Вычислительные эксперименты проводились на решетке размером 512x256. Скорость потока на границах решетки равнялась 0.05 ячейкам за итерацию. Диаметр препятствия равнялся 20 ячейкам. Для достижения требуемых чисел *Рейнольдса* изменялась вязкость среды.

Практика показала, что условия возникновения вихревой дорожки в симуляции близки к указанным в литературе. Так, при $Re = 30$ (рис. 12) цепочка вихрей ещё не формируется. Её можно создать, искусственно внося в поток возмущения, "разбалтывающие"его. В этом случае за цилиндром возникают небольшие периодические колебания, которые со временем затухают, и симуляция снова приходит к состоянию, показанному на рисунке. При $Re = 45$ (рис. 13) за препятствием уже формируется цепочка медленно вращающихся вихрей. При дальнейшем увеличении числа *Рейнольдса* вихри начинают вращаться быстрее и формируются чаще (рис. 14).

Если и дальше уменьшать вязкость, то при числах *Рейнольдса* больших тысячи возникают артефакты, вызванные, предположительно, вычислительными погрешностями. Однако можно увеличивать число *Рейнольдса* увеличением диаметра препятствия. При больших значениях Re цепочка вихрей теряет регулярность. При симуляции возникновения турбулентности с использованием описанной методики возникают проблемы. В некоторых точках скорость движения среды достигает критического порога, после которого симуляция становится нестабильной и "взрывается". С этим можно пытаться бороться уменьшением скорости

Рис. 12. Поток вокруг цилиндра, Re = 30

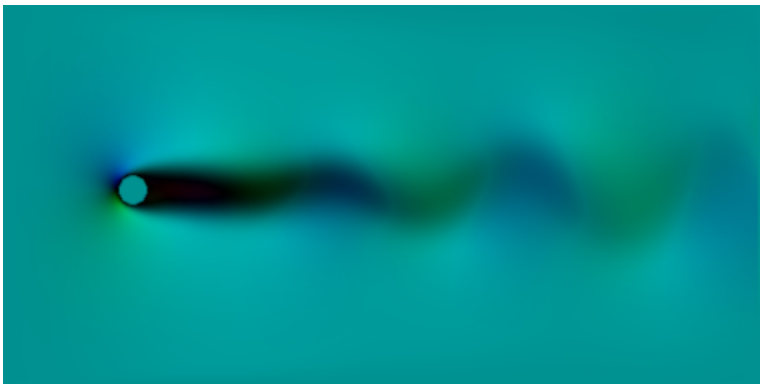

Рис. 13. Поток вокруг цилиндра, Re = 45

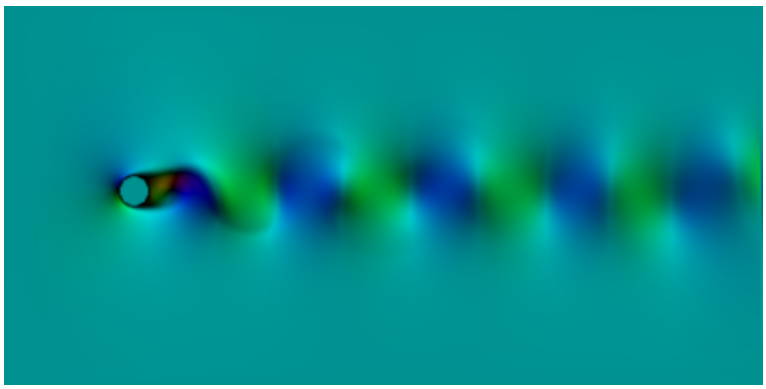

Рис. 14. Поток вокруг цилиндра, Re = 200

набегающего потока и соответствующим увеличением размеров препятствия, в этом случае потребуется увеличить размеры решетки, что приво-

дит к значительному увеличению вычислительных затрат. Существуют другие способы моделирования возникновения турбулентности при помощи метода *Lattice-Boltzmann* [15], которые здесь не рассматриваются.

4.3. Производительность

Важнейшей целью данной работы было продемонстрировать возможность увеличения скорости вычислений при помощи графического ускорителя. Для этого алгоритм был реализован на пиксельных шейдерах при помощи *OpenGL API* и языка *Cg* (разд. 3). Скорость работы измерялась в числе итераций клеточного автомата в секунду. В таблице приведено приблизительное число итераций для разных размеров решетки и вычислительных устройств.

Таблица 1. Сравнение производительности (число итераций клеточного автомата в секунду)

Процессор	256x128	512x256	1024x512
CPU Intel Core 2 Duo 1.83 GHz (1 поток)	108	27	7
GPU NVIDIA Geforce 7600 Go	445	160	47

Полученные результаты показывают значительное преимущество *GPU* при выполнении хорошо распараллеливаемых вычислений с плавающей точкой. Ещё более новые поколения графических ускорителей показывают лучшие результаты по сравнению с приведенными здесь. Однако на данный момент GPU обладают одним серьезным недостатком — они поддерживают операции только с 32-битными числами. Впрочем, производители обещают поддержку 64-битных чисел с плавающей точкой уже в ближайшее время, и тогда на *GPU* можно будет перенести практически все вычисления, возникающие при обработке сигналов, кодировании звука и видео, симуляции физических процессов, финансовые и инженерные расчеты и т.д., и получить при этом многократное увеличение производительности, а также разгрузить центральный процессор для управляющих функций. Более подробно с возможными применениями графических ускорителей можно ознакомиться на сайте `http://www.gpgpu.org`.

Приложение А. Реализация на центральном процессоре

Для начала опишем структуру данных, используемую для представления состояния ячейки решетки.

Listing 1. Структура данных для представления ячейки

```
// Будем использовать в вычислениях числа с одинарной
точностью
typedef float LBfloat;

// Для удобства дадим скоростным каналам буквенные имена
enum Site
{
  SITE_C,
  SITE_R, SITE_U, SITE_L, SITE_D,
  SITE_UR, SITE_UL, SITE_DL, SITE_DR,
  SITE_NUM
};

// Структура ячейки
struct D2Q9Cell
{
  // Веса W_i из уравнения (2.5)
  static const LBfloat weights[SITE_NUM];

  // Скоростные каналы (2.1)
  static const int c[SITE_NUM][2];

  // Этот массив содержит распределение частиц по скоростным
  // каналам (f_i), порядок отличается рис. 5
  // 0 - center; 1..4 - R, U, L, D; 5..8 - UR, UL, DL, DR
  LBfloat site[SITE_NUM];

  // Перегрузка оператора «квадратные скобки» для упрощения
доступа к значениям в скоростных каналах.
  LBfloat & operator[] (int i)
    { return site[i]; }
  const LBfloat & operator[] (int i) const
    { return site[i]; }
};
```

```
// Значения коэффициентов W_i
const LBfloat D2Q9Cell::weights[9] =
{
  4.0 / 9.0 ,
  1.0 / 9.0 , 1.0 / 9.0 , 1.0 / 9.0 , 1.0 / 9.0,
  1.0 / 36.0, 1.0 / 36.0, 1.0 / 36.0, 1.0 / 36.0
};

// Скоростные каналы
const int D2Q9Cell::c[9][2] =
{
  { 0, 0},
  { 1, 0}, { 0, 1}, {-1, 0}, { 0,-1},
  { 1, 1}, {-1, 1}, {-1,-1}, { 1,-1}
};
```

Теперь реализуем несколько вспомогательных функций для работы со структурой `D2Q9Cell`.

Listing 2. Функции для работы со структурой D2Q9Cell

```
// Функция для расчета равновесного распределения, формула (2.5)
inline void calcEquilibrium(LBfloat p,
  LBfloat ux,
  LBfloat uy,
  D2Q9Cell & cell)
{
  // Коэффициэнты из формулы (2.5)
  const LBfloat C1 = (LBfloat) 1.0;
  const LBfloat C2 = (LBfloat) 3.0;
  const LBfloat C3 = (LBfloat) 9.0 / 2.0;
  const LBfloat C4 = (LBfloat) -3.0 / 2.0;

  LBfloat u2 = ux * ux + uy * uy;

  for (int i = 0; i != SITE_NUM; ++i)
  {
    LBfloat eu = D2Q9Cell::c[i][0] * ux
      + D2Q9Cell::c[i][1] * uy;
    LBfloat eu2 = eu * eu;
    cell[i] = D2Q9Cell::weights[i] * p
      * (C1 + C2*eu + C3*eu2 + C4*u2);
```

```
  }
}

// Вычисление локальной плотности, формула (2.1)
inline LBfloat calcDensity(const D2Q9Cell & cell)
{
  LBfloat res = 0;
  for (int i = 0; i != SITE_NUM; ++i)
    res += cell.site[i];
  return res;
}

// Вычисление локальной плотности импульса
inline void calcMomentum( const D2Q9Cell & cell,
  LBfloat & ux,
  LBfloat & uy )
{
  ux = 0;
  uy = 0;
  for (int i = 0; i != SITE_NUM; ++i)
  {
    ux += D2Q9Cell::c[i][0] * cell.site[i];
    uy += D2Q9Cell::c[i][1] * cell.site[i];
  }
}

// Вычисление локальной плотности и скорости по
// формулам (2.1) и (2.2)
inline void calcState( const D2Q9Cell & cell,
  LBfloat & p,
  LBfloat & ux,
  LBfloat & uy )
{
  calcMomentum(cell, ux, uy);
  p = calcDensity(cell);
  if (!cg::eq_zero(p))
  {
    ux /= p;
    uy /= p;
  }
}
```

Теперь перейдем непосредственно к реализации алгоритма. Код демонстрационного приложения для этой работы содержит различные особенности реализации, которые имеют достаточно косвенное отношение к рассматриваемому алгоритму. Поэтому здесь приводится несколько упрощенный вариант кода, содержащий только ключевые моменты.

Для начала определим класс `CPUflow`.

Listing 3. Объявление класса CPUflow

```
class CPUflow
{
public:
  // Конструктор класса. Аргументы width и height задают
размеры решетки.
  CPUflow(int width, int height);

  // Этот метод производит count итераций клеточного автомата.
Обновляются только клетки, которые не лежат на границах
решетки. Таким образом, поддерживаются постоянные граничные
условия. Это также позволяет избавиться от написания отдельных
процедур для обработки пограничных ячеек.
  void Iterate(int count);

  // Инициализация всех клеток решетки состоянием,
соответствующим равновесному распределению для скорости (ux, uy)
и плотности равной 1.0.
  void FillGrid(float ux, float uy);

  // Данный метод задает граничные условия. Клетки,
находящиеся на краях решетки, устанавливаются в равновесное
состояние, соответствующее скорости (ux, uy) и плотности 1.0.
  void SetBoundaryVelocity(float ux, float uy);

  // Этот метод позволяет настраивать кинематическую вязкость
жидкости.
  void SetViscosity(float viscosity);

  // В этом листинге опущены методы, отвечающие за
визуализацию поля скоростей и создание препятствий.

private:
  // Обновление состояния каждой ячейки происходит в этом
методе. Он вызывается из метода Iterate.
  void ProcessCell(int cellId);
```

```
  // Состояние решетки представляет собой двухмерный массив
структур D2Q9Cell. В данном коде этот массив упакован в
одномерный STL-контейнер std::vector. Номер элемента вектора,
соответствующего ячейке с координатами (x,y) находится по
формуле m_width * y + x.
  typedef std::vector<D2Q9Cell> GridStorage;

  // Рабочие буферы
  GridStorage m_buf1, m_buf2;
  // Указатель на буфер, содержащий старое состояние решетки,
и на тот, в который будет записано новое
  GridStorage * m_readBuf, * m_writeBuf;

  // Этот массив хранит конфигурацию препятствий. Единицами
помечены стенки, а свободные клетки -- нулями.
  std::vector<int> m_walls;

  // Эта переменная содержит значение -1/τ из формулы (2.4).
Её значение определяет вязкость жидкости.
  LBfloat m_relaxCoef;

  // Ширина и высота решетки
  int m_width, m_height;
};
```

В описании класса опущены методы, отвечающие за визуализацию поля скоростей и создание препятствий. Здесь предполагается, что препятствия уже определены и соответствующие им клетки помечены в массиве `m_walls` единицами. Визуализация подробно рассмотрена в разделе 3.

Теперь приведем реализацию методов класса `CPUflow`

Listing 4. Реализация класса CPUflow

```
// Конструктор
CPUflow::CPUflow(int width, int height)
  : m_width(width)
  , m_height(height)
  , m_buf1(width * height)       // выделяем память для
хранения решетки
  , m_buf2(width * height)       // выделяем память для
хранения решетки
  , m_readBuf(&m_buf1)           // указатель на буфер
чтения
  , m_writeBuf(&m_buf2)          // указатель на буфер
записи
```

```
  , m_walls(width * height, 0)  // создание массива
стенок
{
  // Здесь можно создать препятствия, записав в
соответствующие ячейки вектора m_walls единицы.
}


// Итерация
void CPUflow::Iterate(int count)
{
  for (int i = 0; i != count; ++i)
  {
    for (int y = 1; y != m_height - 1; ++y)
    {
      int curCell = y * m_width + 1;
      for (int x = 1; x != m_width - 1; ++x)
        ProcessCell(curCell++);  // здесь происходит
основная работа
    }
    swap(m_readBuf, m_writeBuf); // буфера чтения и
записи меняются местами
  }
}

// Именно этот метод выполняет перенос частиц и применение
оператора столкновений, формула (2.3). Аргумент содержит
номер ячейки, которую нужно обработать. Эта ячейка не должна
находиться на краю решетки.
void CPUflow::ProcessCell(int cellId)
{
  D2Q9Cell cell; // здесь мы создадим наше новое состояние

  // Фаза переноса. Копируем данные из скоростных каналов
соседних ячеек, направленных в нашу сторону.
  for (int i = 0; i != SITE_NUM; ++i)
  {
    // Вычисляем смещение i-го соседа относительно текущей
ячейки
    int shift = -D2Q9Cell::c[i][0]
      - m_width * D2Q9Cell::c[i][1];
    cell[i] = (*m_readBuf)[cellId + shift][i];
  }
```

```
  // Фаза столкновения.
  if (m_walls[cellId] != 0)
  {
    // Если текущая ячейка является стенкой, то меняем
местами содержимое скоростных каналов, направленных в
противоположные стороны
    swap(cell[SITE_R], cell[SITE_L]);
    swap(cell[SITE_U], cell[SITE_D]);
    swap(cell[SITE_UR], cell[SITE_DL]);
    swap(cell[SITE_UL], cell[SITE_DR]);
  }
  else
  {
    // Иначе применяем оператор столкновения BGK (2.4)
    D2Q9Cell eq;
    LBfloat p, ux, uy;
    calcState(cell, p, ux, uy);
    calcEquilibrium(p, ux, uy, eq); // вычисляем
локальное равновесное распределение

    // Проверка на нулевую плотность. В этом случае
дальнейшие вычисления не имеют смысла.
    if (cg::eq_zero(p))
      return;

    // Выполняем релаксацию
    for (int i = 0; i != SITE_NUM; ++i)
      cell[i] += m_relaxCoef * (eq[i] - cell[i]);
  }

  (*m_writeBuf)[cellId] = cell;
}
```

Приложение B. Реализация на графическом ускорителе

Listing 5. Код шейдера фазы переноса

```
// В данную структуру шейдер сохранит новое состояние
обрабатываемой ячеки после фазы переноса
struct pixelOutput
{
  float3 row0 : COLOR0;
  float3 row1 : COLOR1;
  float3 row2 : COLOR2;
};

pixelOutput main(
  float2 pos : TEXCOORD0,    // Координаты обрабатываемой
ячейки

  // Данные текстуры хранят текущее состояние решетки в
формате, показанном на рис. 7
  uniform samplerRECT bufRow0 : TEXUNIT0,
  uniform samplerRECT bufRow1 : TEXUNIT1,
  uniform samplerRECT bufRow2 : TEXUNIT2
)
{
  pixelOutput pOut;

  // r g b
  // UL U UR - row 0
  // L C R - row 1
  // DL D DR - row 2

  // Читаем данные из соседних ячеек
  float UL = texRECT(bufRow0, pos + float2( 1.0,-1.0)).r;
  float U  = texRECT(bufRow0, pos + float2( 0.0,-1.0)).g;
  float UR = texRECT(bufRow0, pos + float2(-1.0,-1.0)).b;

  float L = texRECT(bufRow1, pos + float2( 1.0, 0.0)).r;
  float C = texRECT(bufRow1, pos + float2( 0.0, 0.0)).g;
  float R = texRECT(bufRow1, pos + float2(-1.0, 0.0)).b;

  float DL = texRECT(bufRow2, pos + float2( 1.0, 1.0)).r;
```

```
  float D  = texRECT(bufRow2, pos + float2( 0.0, 1.0)).g;
  float DR = texRECT(bufRow2, pos + float2(-1.0, 1.0)).b;

  // Формируем новое состояние ячейки
  pOut.row0 = float3(UL, U, UR);
  pOut.row1 = float3(L,  C,  R);
  pOut.row2 = float3(DL, D, DR);

  return pOut;
}
```

Listing 6. Код шейдера фазы столкновения

```
// В данную структуру шейдер сохранит новое состояние
обрабатываемой ячеки после фазы столкновения
struct pixelOutput
{
  float3 row0 : COLOR0;
  float3 row1 : COLOR1;
  float3 row2 : COLOR2;
};

// Функция для расчета равновесного распределения по формуле
(2.5)
float3 calcEqRow(float2 u, float u2, float p, float cy)
{
  const float3 cx = float3(-1.0, 0.0, 1.0);

  const float C1 = 1.0;
  const float C2 = 3.0;
  const float C3 = 9.0 / 2.0;
  const float C4 = -3.0 / 2.0;

  float3 eu = cx * u.x + cy * u.y;
  float3 eu2 = eu * eu;

  return p * (C1 + C2*eu + C3*eu2 + C4*u2);
}

pixelOutput main(
  float2 pos : TEXCOORD0,    // Координаты обрабатываемой
ячейки
```

```
  // Этот параметр содержит значение -1/τ из формулы (2.4)
  uniform float relaxCoef,

  // Данные текстуры хранят текущее состояние решетки в
формате, показанном на рис. 7
  uniform samplerRECT bufRow0 : TEXUNIT0,
  uniform samplerRECT bufRow1 : TEXUNIT1,
  uniform samplerRECT bufRow2 : TEXUNIT2
)
{
  pixelOutput pOut;

  // r g b
  // UL U UR - row 0
  // L C R - row 1
  // DL D DR - row 2

  // Читаем текущее состояние ячейки
  float3 row0 = texRECT(bufRow0, pos).rgb;
  float3 row1 = texRECT(bufRow1, pos).rgb;
  float3 row2 = texRECT(bufRow2, pos).rgb;

  // Локальной плотности и скорости по формулам (2.1) и (2.2)
  float p = row0.r + row0.g + row0.b
    + row1.r + row1.g + row1.b
    + row2.r + row2.g + row2.b;

  float2 u = float2(0.0, 0.0);
  u.x += row0.b + row1.b + row2.b;
  u.x -= row0.r + row1.r + row2.r;

  u.y += row0.r + row0.g + row0.b;
  u.y -= row2.r + row2.g + row2.b;

  u /= p;

  float u2 = dot(u, u);

  // Релаксация
  const float Wcorn = 1.0 / 36.0;
```

```
  const float Wside = 1.0 / 9.0;
  const float Wcent = 4.0 / 9.0;

  pOut.row0 = row0 + relaxCoef * ( calcEqRow(u, u2, p, 1.0)
    * float3(Wcorn, Wside, Wcorn) - row0 ) ;
  pOut.row1 = row1 + relaxCoef * ( calcEqRow(u, u2, p, 0.0)
    * float3(Wside, Wcent, Wside) - row1 );
  pOut.row2 = row2 + relaxCoef * ( calcEqRow(u, u2, p,-1.0)
    * float3(Wcorn, Wside, Wcorn) - row2 );

  return pOut;
}
```

Список литературы

[1] *Brian J. N. Wylie* Application of two-dimensional cellular automaton lattice-gas models to the simulation of hydrodynamics. University of Edinburgh, 1990.

[2] *Maxwell J. B.* Lattice Boltzmann methods for interfacial wave modeling. University of Edinburgh, 1997.

[3] *Wolf-Gladrow D. A.* Lattice-gas cellular automata and Lattice-Boltzmann models. Springer, 2000.

[4] *Ван-Дайк М.* Альбом течений жидкости и газа. М.: Мир, 1986.

[5] *Лойцянский Л. Г.* Механика жидкости и газа. М.: Государственное издательство технико-теоретической литературы, 1950.

[6] *Фейнман Р., Лейтон Р., Сэндс М.* Фейнмановские лекции по физике. Том 7. Физика сплошных сред.

[7] *Черняк В. Г., Суетин П. Е.* Механика сплошных сред. М.: Физматлит, 2006.

[8] *Ferziger J. H., Peric M.* Computational methods for fluid dynamics, 3ed. Springer, 2001.

[9] *Von Neumann J.* Theory of self-reproducing automata. University of Illinois Press, 1966.

[10] *Gardner M.* Mathematical Games: The fantastic combinations of John Conway's new solitare game "Life". Scientific American 223, 1970.

[11] *Chopard B., Luthi P., Masselot A.* Cellular automata and Lattice-Boltzmann techniques: an approach to model and simulate complex systems. University of Geneva, 1998.

[12] HSL Color Space. `http://en.wikipedia.org/wiki/HSL_color_space`

[13] Cg Toolkit User's Manual. NVIDIA Corporation, 2005.

[14] *Xiaowen S., Hudong C.* Lattice-Boltzmann model for simulating flows with multiple phases and components. Dartmouth College, Hanover, 1993.

[15] *Zhen-Hua C., Bao-Chang S., Lin Z.* Simulating high Reynolds number flow in two-dimensional lid-driven cavity by multi-relaxation-time Lattice-Boltzmann method. Chin. Phys. Soc. and IOP Publishing Ltd, 2006.