Вычисления общего назначения на графических процессорах с использованием шейдерных языков

Д.К. Боголепов, В.Е. Турлапов

В работе рассмотрены основные принципы функционирования графического процессора и существующие средства разработки для GPGPU. Предложен подход к оптимизации производительности вычислений при решении задач общего назначения. Подход рассмотрен на примере решения задачи моделирования динамики системы N точечных масс (N = 16384). Он включает: теоретическую оценку производительности решения задачи на графическом процессоре; исследование фактической производительности в условиях вариации тактовых частот основных подсистем GPU; методы оптимизации производительности, основанные на результатах такого исследования. Фактическая производительность в тестовой задаче составила около 150 GFLOPS на графической карте массового производства NVIDIA GeForce 9800 GT.

1. Введение

На протяжении последнего десятилетия графические ускорители стремительно наращивали производительность, чтобы удовлетворить непрерывно растущие запросы разработчиков графических приложений и компьютерных игр. Кроме того, за последние несколько лет изменились некоторые фундаментальные принципы разработки графической аппаратуры, в результате чего она стала более программируемой, чем когда-либо ранее. Сегодня графический ускоритель – это гибко программируемый массивно-параллельный процессор[1] для обработки чисел в формате с плавающей точкой, возможности которого могут быть востребованы для решения целого ряда вычислительно трудоемких задач. К таким задачам относятся различные задачи матфизики и инженерного анализа, высококачественная визуализация, обработка изображений, машинное зрение, задачи финансовой математики и т.д. [1].

2. Средства программирования для графического процессора

2.1 Интерфейсы программирования графики и шейдерные языки

К данной категории относятся проверенные временем интерфейсы программирования графики и связанные с ними *шейдерные языки*. Применительно к вычислениям общего назначения чаще всего используется связка из OpenGL и языка GLSL (OpenGL Shading Language) [2]. Данные средства изначально применялись для визуализации, поэтому при их использовании для задач общего назначения появляется “привязка к графике”. Тем не менее, данный подход ни в коем случае нельзя назвать низкоуровневым. Сама по себе “привязка к графике” вовсе не так сложна и, более того, практически одинакова для различных задач.

Язык шейдеров OpenGL – это высокоуровневый процедурный язык программирования, который основан на синтаксисе и управлении C и C++. Язык определяет богатый набор встроенных типов, включая вектор и матрицу, которые позволяют лаконично записывать многие алгоритмы. Присутствует ряд механизмов, заимствованных из языка C++: перегрузка функций по типу аргумента, объявление переменных непосредственно перед их использованием, конструкторы для сложных типов данных. Язык поддерживает циклы, динамические ветвления и подпрограммы и обеспечивает богатый набор встроенных функций. Наконец, данный язык не накладывает ограничений на длину шейдерных программ.

К достоинствам подхода, основанного на использовании шейдеров, можно отнести отсутствие специфических требований к программному окружению и драйверам. Все необходимое –

[1] Графический процессор AMD Radeon HD 4800 содержит 160 модулей, состоящих из пяти АЛУ (что позволяет говорить о 800 исполнительных устройствах). Графический процессор NVIDIA GeForce GTX 280 содержит 240 скалярных АЛУ. Оба процессора способны выполнять операцию MAD за один такт.

язык шейдеров, компилятор и компоновщик, – определено как часть стандарта OpenGL версии 2.0 и выше. Кроме того, данный подход обеспечивает поддержку графических ускорителей различных производителей (требуется лишь поддержка языка шейдеров OpenGL)[1]. Наконец, графический интерфейс OpenGL и его язык шейдеров являются межплатформенным стандартом, что позволяет использовать единый программный код для поддержки самых различных платформ. Следует отметить, что среди недостатков данного подхода иногда отмечается тот факт, что на шейдерных языках можно писать только шейдеры, а не полные программы. С другой стороны, это позволяет использовать шейдерные языки вместе с любым языком программирования[2], что может обеспечивать большую гибкость решения.

2.2 Специализированные средства от производителей

Некоторое время назад оба ведущих производителя предоставили специализированные средства разработки для графического процессора – CUDA (Compute Unified Driver Architecture) от компании NVIDIA [3] и Stream Computing от компании AMD [4]. Следует отметить, что данные библиотеки имеют много общего. Входящие в их состав компоненты можно разделить на два уровня: *верхний* и *нижний*.

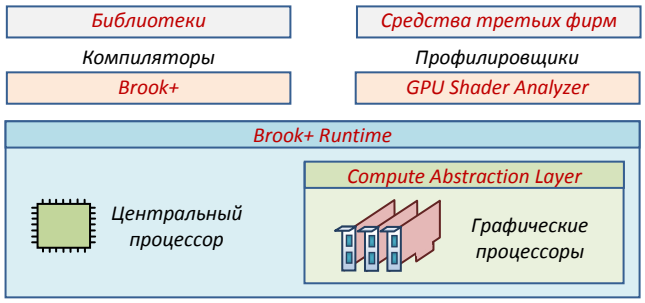

Рис. 1. Компоненты AMD Stream Computing

Нижний уровень нацелен, прежде всего, на разработчиков компиляторов и средств программирования, позволяя взаимодействовать с графическим процессором на низком уровне для достижения максимальной производительности. Данный уровень включает в себя ассемблер для графического процессора, менеджер памяти и средства взаимодействия с графической аппаратурой. На верхнем уровне разработчику предоставляется вариант языка C с потоковыми расширениями, учитывающими особенности конкретного графического процессора.

Специализированные средства от производителей позволили записывать программу на диалекте языка C и отказаться от специфической терминологии компьютерной графики. Однако существенного прорыва здесь не произошло: программирование выглядит практически также, как и при использовании шейдеров, а разработка эффективных программ по-прежнему требует знания архитектурных особенностей аппаратуры[3]. Фактически произошла лишь замена терминов компьютерной графики на термины *потокового программирования* [5]. Кроме того, обе библиотеки не являются универсальными: они разработаны для поддержки оборудования соответствующего производителя, и перенос программы с одной архитектуры на другую может повлечь значительные изменения кода.

2.3 Сторонние специализированные средства

К данной категории относятся все остальные средства разработки для графических процессоров. Как правило, данные инструменты используют подход *метапрограммирования*: исход-

[1] Поддержку имеют графические ускорители NVIDIA серий GeForce 200, GeForce 9000, GeForce 8000 и GeForce 7000, ATI/AMD серий Radeon HD4000, Radeon HD3000, Radeon HD2000 и Radeon X1000.

[2] Поддержка OpenGL обеспечена практически на всех языках программирования: от традиционных для графических задач (C/C++, Pascal, Basic, Java, C#) до более экзотических (Python, Perl, Ruby).

[3] Согласно ресурсу [6], стандартный пример умножения матриц из CUDA SDK имеет в *четыре раза меньшую* производительность по сравнению с аналогичной функцией библиотеки CUBLAS.

ная программа разрабатывается на диалекте языка C с потоковыми расширениями, а затем транслируется в шейдеры, исполняемые на графическом процессоре. Наиболее известными примерами служат системы BrookGPU [7] и Sh [8]. В качестве примера, рассмотрим принцип функционирования системы BrookGPU.

Система BrookGPU состоит из *двух* компонент. На верхнем уровне функционирует *метакомпилятор*, который транслирует программу на языке BrookGPU в стандартный язык C++. На нижнем уровне функционирует *библиотека времени выполнения* – архитектурно-независимый слой, реализующий поддержку конструкций BrookGPU для конкретной аппаратуры.

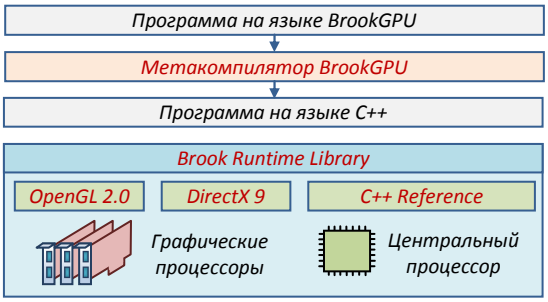

Рис. 2. Архитектура системы BrookGPU

Технически данная библиотека предоставляет набор абстрактных классов для компилятора BrookGPU и их реализации для различной поддерживаемой аппаратуры (центральные и графические процессоры). Конкретная реализация выбирается во время исполнения в зависимости от доступной аппаратуры и предпочтений пользователя.

С одной стороны, подобные системы упрощают процесс программирования по сравнению с шейдерными языками и обеспечивают независимость от оборудования конкретного производителя. С другой стороны, данные системы являются устаревшими и более не поддерживаемыми проектами, имеют много ограничений, не выполняют оптимизаций под конкретную архитектуру и не обеспечивают должного уровня эффективности.

3. Основы формирования изображений на видеоадаптере

3.1 Преобразования координат

Задача графического процессора состоит в преобразовании переданных приложением данных о трехмерной сцене в изображение на экране монитора. Данный процесс называют графическим конвейером (а также *визуализацией* или *рендерингом*). Для обеспечения графического конвейера в компьютерной графике вводится несколько координатных пространств, востребованных в процессе моделирования и визуализации трехмерной сцены.

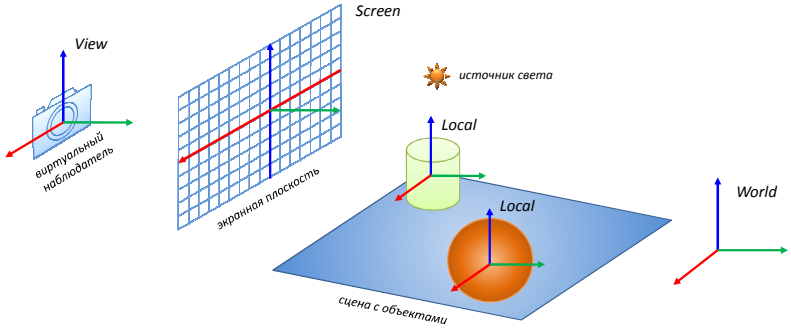

Рис. 3. Преобразования координат в процессе рендеринга

Атрибуты трехмерных объектов, такие как координаты вершин и нормали к поверхности, задаются в *локальном* или *модельном* пространстве.

Чтобы скомпоновать сцену из нескольких объектов, каждый из которых задан в своем локальном пространстве, необходимо перенести их в систему координат *модели* сцены (*мировую*).

В системе модели определяются отношения между объектами, источники света и устанавливается *виртуальный наблюдатель* (глаз или камера). Преобразование координат объекта из локального пространства в мировое осуществляется с помощью матрицы *модели*.

Задав положение и ориентацию виртуального наблюдателя, мы определяем еще одну систему координат – пространство *вида* (*обзора*). Начало координат этой системы находится в точке обзора. Данное пространство полезно тем, что позволяет легко вычислить расстояние от точки обзора до различных объектов сцены. Для удобства последние два преобразования могут быть объединены в одну матрицу *модели-вида*, посредством которой координаты объекта можно преобразовать напрямую из локального пространства в пространство обзора.

Последнее преобразование состоит в проецировании объектов на *экранную плоскость* (при этом с точностью до масштаба мы получаем оконные координаты). Данное преобразование выполняется с помощью матрицы *проекции* (используется параллельная или перспективная проекция). В пространстве оконных координат оси *x* и *y* параллельны сторонам экранной плоскости, а ось *z* ортогональна к ней. В данном пространстве выполняется *отсечение* графических примитивов, которые находятся вне прилегающей к оси *z* усеченной пирамиды видимости. Все три преобразования могут быть объединены в одну матрицу *модели-вида-проекции*, с помощью которой объекты можно спроектировать из локального пространства на экранную плоскость.

3.2 Графический конвейер операций

В процессе рендеринга графический процессор заполняет область памяти, называемую *буфером кадров* (в оконной системе термину “буфер кадров” соответствует термин “окно”). Рассмотрим упрощенную схему формирования изображения на современных видеоадаптерах.

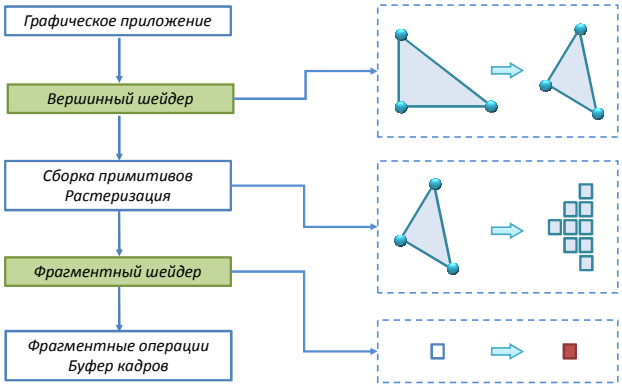

Рис. 4. Упрощенная схема современного графического конвейера

Приложение, работающее на центральном процессоре, инициирует вывод некоторых *геометрических примитивов* (точек, линий, треугольников, многоугольников). При этом разработчик в своей программе задает набор *атрибутов* вершин (положение, цвет, нормаль и текстурные координаты) и информацию о *связях* между ними. Данному этапу визуализации соответствует блок “Графическое приложение”.

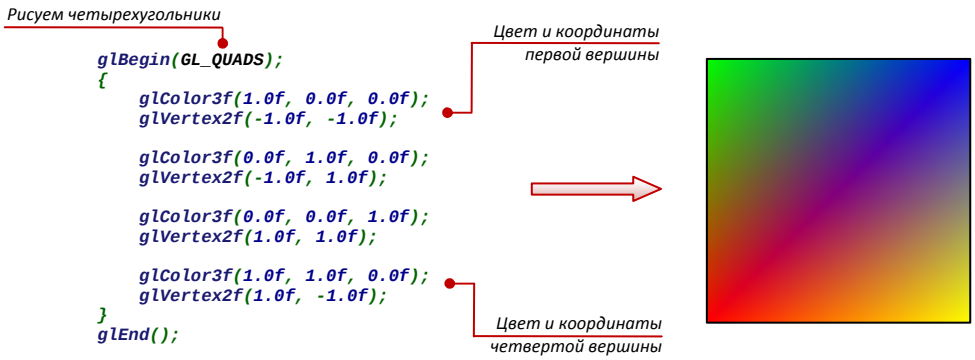

Рис. 5. Вызов команд интерфейса OpenGL для рисования четырехугольника

Задачей следующего этапа является расчет *освещенности* и *трансформация* вершин (проецирование вершин на экранную плоскость). На данном этапе каждая вершина рассматривается *отдельно*, поэтому их обработка может выполняться параллельно. Приложение может передать исполнение этапа стандартному графическому конвейеру или использовать *вершинный шейдер*. Вершинным шейдером называется программа, которая исполняется непосредственно на графическом процессоре и определяет алгоритм обработки вершинных атрибутов. Используя вершинный шейдер, можно выполнить нестандартный расчет освещения (применив модель освещения Кука-Торренса) и трансформацию вершин (генерация ландшафта из плоской поверхности или анимация объектов). На схеме данный этап обозначен как “Вершинный шейдер”.

На следующем этапе преобразованные вершины объединяются в геометрические примитивы (например, отрезок строится из двух вершин, треугольник из трех, а четырехугольник из четырех). Данный этап называется *сборкой примитивов*; в процессе его выполнения используются не только данные об отдельных вершинах, но также информация о связях между ними.

Сформированные примитивы разбиваются на меньшие части в соответствии с расположением пикселей в буфере кадров. Данный процесс называется *растеризацией*. Полученные части примитива называется *фрагментами*. Например, если отрезок, заданный двумя вершинами, будет занимать на экране пять пикселей, то в процессе растеризации он будет разбит на пять фрагментов. Последние два этапа представлены блоком “Сборка примитивов и растеризация”.

После растеризации над фрагментами выполняется еще ряд операций, которые в целом называются *обработкой фрагментов*. Наиболее важной является операция *текстурирования*. Подобно вершинам, отдельные фрагменты обрабатываются *независимо*, поэтому их обработка может производиться параллельно. На данном этапе разработчик может использовать *фрагментный шейдер* – программу для графического процессора, определяющую алгоритм обработки фрагментов. С помощью данного шейдера можно задать нестандартные методы расчета цвета и наложения текстур (процедурное текстурирование, по-фрагментное освещение и нанесение микрорельефа). Данный этап соответствует блоку схемы “Фрагментный шейдер”.

На последнем этапе фрагменты проходят некоторые дополнительные стадии обработки, которые в целом называются *фрагментными операциями*. К ним относится, например, тест на видимость, отсечение по шаблону, проверка глубины, α-смешивание и затуманивание. Все эти стадии не зависят от других и могут быть эффективно выполнены на графической аппаратуре. Данный этап соответствует блоку схемы “Фрагментные операции и буфер кадров”.

Выполнение всех описанных этапов приводит к тому, что данные о трехмерной сцене, переданные приложением, оказываются преобразованными в пиксели в буфере кадров для последующего отображения на экране.

4. Использование видеоадаптера для вычислений общего назначения

4.1 Общие принципы вычислений общего назначения на видеоадаптере

Графические ускорители предназначены для решения задач визуализации на базе алгоритма растеризации, который может быть существенно расширен за счет использования программируемых вершинных и фрагментных процессоров. Таким образом, для решения задачи общего назначения на видеоадаптере ее необходимо переформулировать в виде задачи растеризации.

Обратимся, например, к задаче сложения двух матриц большой размерности. А именно, пусть A и B – матрицы размера $n \times m$ и требуется вычислить матрицу $C = A + B$. Вполне очевидно, что матрицы A *и* B легко представить в виде прямоугольных текстур размера $n \times m$ в формате с плавающей точкой [9].

Для инициирования вычисления матрицы C следует нарисовать прямоугольник в окне размера $n \times m$. При этом, конечно, для пользователя данное изображение не имеет смысла, поэтому вывод следует осуществлять не на экран, а в текстуру в формате с плавающей точкой. На этапе растеризации данного прямоугольника будет сгенерировано ровно $n \times m$ фрагментов, каждый из которых будет обработан фрагментным шейдером. Заметим, что отдельные фрагменты будут обрабатываться независимо на множестве шейдерных процессоров (сотни независимых “потоков”).

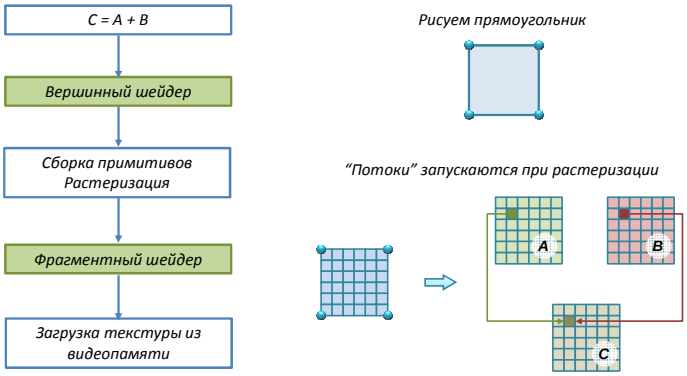

Рис. 6. Процесс решения задачи сложения двух матриц

Все вычисления выполняются во *фрагментном* шейдере. На вход данный шейдер получает координаты обрабатываемого фрагмента, которые могут быть использованы в качестве *текстурных координат* для доступа к соответствующим элементам матриц *A* и *B*. Таким образом, в качестве четырехкомпонентного вектора цвета шейдер может вернуть сумму выборок из текстур с матрицами *A* и *B*. Важно отметить, что если рендеринг производится в текстуру в формате с плавающей точкой, то компоненты цвета пикселя, рассчитанные фрагментным шейдером, *не усекаются* по отрезку [0, 1] (данное поведение введено расширением [10]). Иными словами, в качестве цвета можно возвращать *произвольный* четырехкомпонентный вещественный вектор (в рассматриваемом примере нас интересует лишь первая его компонента).

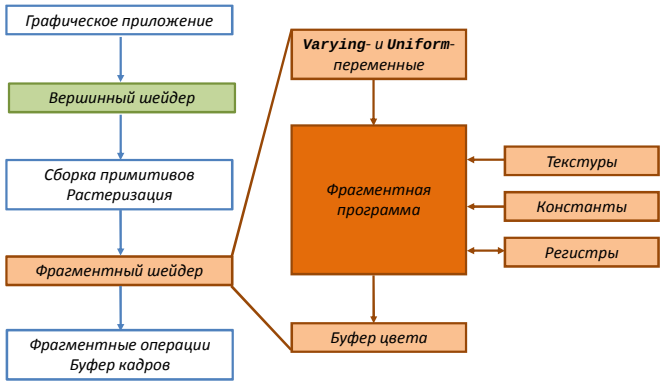

Рис. 7. Все полезные вычисления выполняются во фрагментном шейдере

Заключительный этап решения задачи состоит в загрузке результирующей текстуры в массив чисел в формате с плавающей точкой (в нем будут храниться значения вычисленной матрицы *C*). При этом данные из видеопамяти загружаются в память центрального процессора. Данная операция может сильно нагружать системную шину и негативно влиять на производительность, поэтому ее следует использовать осторожно.

4.2 Ограничения шейдерных программ

Существует ряд особенностей и ограничений при разработке программ для графического процессора по сравнению с приложениями для традиционных архитектур. Во-первых, это существенная “привязка к графике”, которая очень часто преподносится в виде серьезного недостатка, хотя должна быть отнесена к особенностям. Во-вторых, отдельные фрагменты генерируемого изображения *изолированы*. Поэтому для реализации алгоритмов, требующих информации о значениях соседних фрагментов, необходимо прибегать к технике *многопроходного рендеринга*. В-третьих, из фрагментного шейдера невозможна произвольная запись в буфер кадра (каждый фрагмент связан с определенным пикселем результирующего изображения). Наконец, существуют определенные ограничения самих шейдерных языков, среди которых следует отметить запрет рекурсии и отсутствие возможности работать с указателями.

5. Моделирование динамики системы *N* точечных масс

5.1 Постановка задачи

В качестве примера использования графического ускорителя рассмотрим задачу моделирования динамики системы *N* точечных масс, взаимодействующих по закону тяготения Ньютона. Ранее данная задача уже решалась на графическом процессоре с использованием библиотеки CUDA [11] и языка потокового программирования BrookGPU [12]. В данной работе рассматривается решение указанной задачи с использованием шейдерных языков.

Как известно, эволюция системы *N* гравитирующих тел описывается следующей системой дифференциальных уравнений:

$$\begin{cases} \dfrac{dr_i}{dt} = v_i, \\ \dfrac{dv_i}{dt} = G\sum\limits_{j\neq i} m_j \dfrac{r_j - r_i}{\left(r_j - r_i\right)^3}, \end{cases} \tag{1}$$

где r_i, m_i и v_i – масса, радиус-вектор и скорость i-го тела соответственно (i изменяется от 1 до N), G – гравитационная постоянная.

Рис. 8. Примеры моделирования в реальном времени системы из N = 16384 частиц при разных начальных условиях

Массы тел, а также положения и скорости в начальный момент времени считаются известными. Необходимо найти положения и скорости всех частиц в произвольный момент времени.

5.2 Метод решения

Для моделирования данной системы воспользуемся простым подходом, рассчитывая взаимодействия между *всеми* парами гравитирующих тел. В чистом виде такой подход практически неприменим для исследования динамики больших систем (его трудоемкость растет как $O(N^2)$). Тем не менее, он представляет определенный интерес, поскольку используется для расчета взаимодействия достаточно близких тел в составе более сложных методов.

Прежде всего, введем в рассмотрение состояние Y_i i-ой частицы системы:

$$Y_i = \begin{pmatrix} r_i \\ v_i \end{pmatrix}, \tag{2}$$

где r_i и v_i – радиус-вектор и скорость i-го тела соответственно (i изменяется от 1 до N). Тогда исходная система (1) перепишется следующим образом:

$$\frac{dY_i}{dt} = \begin{pmatrix} v_i \\ G\sum\limits_{j\neq i} m_j \dfrac{r_j - r_i}{\left(r_j - r_i\right)^3} \end{pmatrix} = \begin{pmatrix} v_i \\ a_i(r_i) \end{pmatrix} = F(Y_i). \tag{3}$$

Такой вид системы позволяет удобно записать различные численные методы. Для решения задачи применим наиболее распространенные методы *прямого* интегрирования:

Таблица 1. Распространенные методы прямого интегрирования

Название метода	*Значение в следующей точке*	*Ошибка на каждом шаге*	*Суммарная ошибка*
Эйлера	$Y_{i+1} = F(Y_i) \cdot h$	$O(h^2)$	$O(h)$
Рунге-Кутты 2-ого порядка	$Y_{i+1} = Y_i + h \cdot F(Y_i + h \cdot F(Y_i)/2)$	$O(h^3)$	$O(h^2)$
Рунге-Кутты 4-ого порядка	$Y_{i+1} = Y_i + h \cdot (k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4)/6,$ $\begin{cases} k_1 = F(Y_i), \\ k_2 = F(Y_i + h \cdot k_1/2), \\ k_3 = F(Y_i + h \cdot k_3/2), \\ k_4 = F(Y_i + h \cdot k_3). \end{cases}$	$O(h^5)$	$O(h^4)$

Следует заметить, что данные численные методы практически невозможно использовать при тесных сближениях тел, поскольку это вызывает быстрый рост численных ошибок. Чтобы частично сгладить ситуацию, добавим к знаменателю малую положительную величину $\varepsilon^2 > 0$, которая позволит исключить бесконечный рост силы притяжения:

$$a_i(r_i) \approx G \sum_j m_j \frac{r_j - r_i}{\left[(r_j - r_i)^2 + \varepsilon^2\right]^{3/2}}. \quad (4)$$

При этом равенство получается уже не точным, а суммирование можно производить по всем индексам i от 1 до N (поскольку соответствующее слагаемое обращается в ноль).

5.3 Решение задачи на графическом процессоре

Для решения задачи на графическом процессоре необходимо использовать две распространенные техники: *рендеринг в текстуру* [13] и *рендеринг в несколько буферов цвета* одновременно [14]. В совокупности данные техники позволяют фрагментному шейдеру записывать результат одновременно в несколько двумерных текстур заданного формата. В рассматриваемом примере необходимо использовать *четыре* текстуры в формате с плавающей точкой для записи *положений* и *скоростей* материальных точек: первые две текстуры (*CurrentPositionTexture* и *CurrentVelocityTexture*) описывают состояние частиц в *текущий* момент времени, а вторые две текстуры (*NextPositionTexture* и *NextVelocityTexture*) – в *следующий* момент времени.

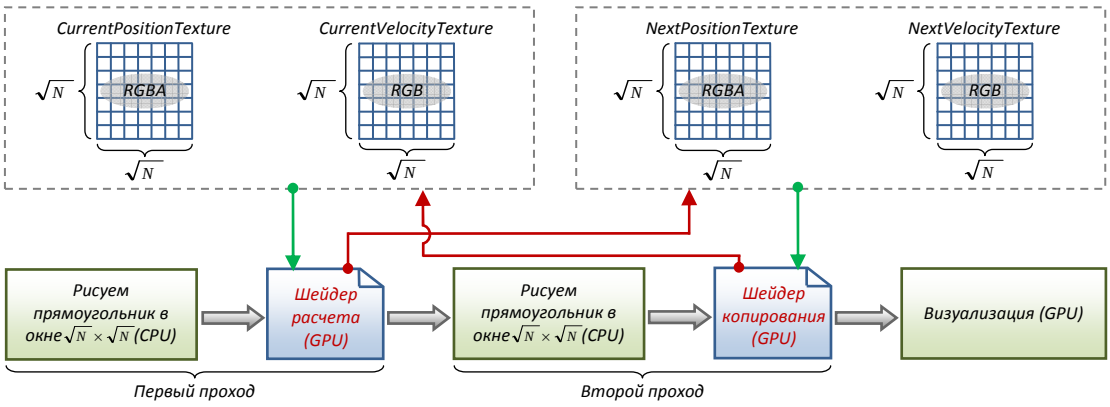

Рис. 9. Схема выполнения одного шага интегрирования

Кроме того, для каждой частицы необходимо сохранить *массу*. Это можно сделать с помощью отдельной текстуры, однако более рационально записать массы в текстуру положений, используя свободные четвертые компоненты. Современные графические ускорители поддерживают работу с прямоугольными текстурами, поэтому размер указанных выше текстур можно выби-

рать любым удобным способом. Требуется лишь обеспечить, чтобы общее число элементов совпадало с числом частиц N (в данной работе использовались текстуры размера $\sqrt{N} \times \sqrt{N}$).

Для выполнения одного шага интегрирования необходимо выполнить два *прохода*. При первом проходе выполняется расчет новых положений и скоростей частиц, при этом на вход фрагментному шейдеру подаются текстуры *CurrentPositionTexture* и *CurrentVelocityTexture*, а результат выводится в текстуры *NextPositionTexture* и *NextVelocityTexture*. Второй проход предназначен для обновления текущих положений и скоростей частиц, а соответствующий фрагментный шейдер выполняет поэлементное копирование текстур *NextPositionTexture* и *NextVelocityTexture* в текстуры *CurrentPositionTexture* и *CurrentVelocityTexture* соответственно.

В основе вычислительного шейдера лежит функция *vec3 Acceleration (vec3 current)*, которая вычисляет ускорение обрабатываемой частицы под воздействием остальных частиц системы[1].

```
 1|vec3 Acceleration( vec3 current )
 2|{
 3|  vec3 acceleration = vec3( 0.0 );
 4|
 5|  for ( float x = 0.0; x < 1.0; x += TextureStep )
 6|  {
 7|    for ( float y = 0.0; y < 1.0; y += TextureStep )
 8|    {
 9|      // Считываем положение и массу очередной частицы
10|      vec4 position = texture2D( CurrentPositionTexture, vec2( x, y ) );
11|
12|      // Находим направление на данную частицу [3 FLOP]
13|      vec3 direction = vec3( position ) - current;
14|
15|      // Находим квадрат расстояния между частицами [6 FLOP]
16|      float squaredist = dot( direction, direction ) + Epsilon;
17|
18|      // Находим шестую степень расстояния между частицами [2 FLOP]
19|      float sixthdist = squaredist * squaredist * squaredist;
20|
21|      // Вычисляем ускорение для данной частицы [8 FLOP]
22|      acceleration += position.w * direction * inversesqrt( sixthdist );
23|    }
24|  }
25|}
```

Данная функция содержит цикл по *всем* частицам системы, выполняя $c \cdot N$ операций с плавающей точкой за один вызов. Доля операций, связанных с самим численным методом, исчезающе мала по сравнению с данной величиной. Таким образом, для оценки трудоемкости конкретного численного метода следует подсчитать число обращений к функции расчета ускорения. Например, в методе Эйлера ускорение вычисляется один раз, поэтому данный метод выполняет $c \cdot N$ операций на одну частицу. Методы Рунге-Кутты 2-ого и 4-ого порядка вычисляют ускорение два и четыре раза соответственно и поэтому выполняют $2 \cdot c \cdot N$ и $4 \cdot c \cdot N$ операций на одну частицу. Для подсчета числа операций на один шаг интегрирования предыдущую оценку необходимо умножить на число N частиц в системе.

Наконец, оценим саму константу c. Для этого подсчитаем число элементарных операций в исходном коде функции *vec3 Acceleration (vec3 current)*. В комментариях нуждается лишь последняя строка (под номером 22), в которой учитывается очередная добавка к ускорению. В соответствии с [15], функция *inversesqrt (float value)* выполняется по меньшей мере 4 такта. Кроме того, современные графические процессоры способны выполнять операции сложения и умножения за один такт (инструкция MAD). Таким образом, имеем 4 такта на вычисление обратного значения квадратного корня, 1 такт на умножение двух скалярных величин и 3 такта на покомпонентное умножение и сложение векторов, что в сумме составляет 8 тактов. В итоге для

[1] Полный исходный код приложения доступен по адресу (*Tutorials 5-7*): http://www.itlab.unn.ru/?doc=959.

рассматриваемой архитектуры получим $c \geq 19$. Указанное значение константы c будет использоваться для оценки производительности на основе подсчета числа итераций в секунду.

5.4 Оценка теоретической производительности

Прежде чем измерять *фактическую* производительность, сделаем некоторые оценки относительно *теоретически* возможной производительности в *данной* задаче. Подобные рассуждения позволят более конкретно говорить о полученных результатах и укажут на “узкие места”, требующие повышенного внимания. В первую очередь, следует ознакомиться с техническими характеристиками используемого графического ускорителя. В данной работе использовалась видеокарта NVIDIA GeForce 9800 GT со следующими характеристиками:

Таблица 2. Технические характеристики графического ускорителя NVIDIA GeForce 9800 GT

Число потоковых процессоров	*Число текстурных модулей*	*Ширина шины памяти*	*Частота ядра / потоковых процессоров*	*Частота памяти реальная / эффективная*
112	*56*	*256 бит*	*600 / 1500 МГц*	*900 / 1800 МГц*

Для оценки итоговой теоретической производительности необходимо оценить производительность *потоковых процессоров*, *блоков выборки из текстур* и *подсистемы памяти*. Рассуждения проведем для случая системы из $N = 16384$ тел и метода Эйлера.

Чистое время вычислений можно вычислить как отношение числа операций на один шаг интегрирования к числу операций, которое способен выполнить графический процессор за единицу времени:

$$ALU = \frac{[\textit{Число фрагментов}]\times[\textit{Число инструкций на фрагмент}]}{[\textit{Число интрукций за такт}]\times[\textit{Частота процессора}]} = \frac{[16384]\times[19\cdot 16384]}{[112]\times[1500\ \textit{МГц}]} \approx 30{,}4\ \textit{мс}.$$

В указанной выше формуле в качестве числа инструкций за такт, которое может выполнить графический ускоритель, было использовано общее число скалярных потоковых процессоров. Возможность потоковых процессоров исполнять за такт *две* операции – сложение и умножение – была учтена в оценке трудоемкости для функции расчета ускорения.

Проводя аналогичные рассуждения, вычислим время выборки из текстур как отношение числа выборок на один шаг интегрирования к общему числу выборок, которое способен выполнить графический процессор за единицу времени:

$$TEX = \frac{[\textit{Число фрагментов}]\times[\textit{Число выборок на фрагмент}]}{[\textit{Число выборок за такт}]\times[\textit{Частота текстурных блоков}]} = \frac{[16384]\times[1\cdot 16384]}{[56]\times[600\ \textit{МГц}]} \approx 8\ \textit{мс}.$$

Следует заметить, что текстурные модули входят в состав графического процессора и, следовательно, функционируют на его частоте.

Наконец, подсчитаем теоретическое время обращения к памяти графического процессора. Для этого необходимо оценить общее число *бит*, которые должны быть прочитаны и записаны в процессе обработки одного фрагмента:

$$MEM = \frac{[\textit{Число фрагментов}]\times[\textit{Число бит на фрагмент}]}{[\textit{Ширина шины памяти}]\times[\textit{Частота памяти}]} = \frac{[16384]\times[16384\cdot 4\cdot 8 + 2\cdot 4\cdot 8]}{[256]\times[900\cdot 2\ \textit{МГц}]} \approx 18{,}7\ \textit{мс}.$$

В графическом процессоре действует многоуровневая структура кэшей, поэтому полученная оценка является пиковой, когда данные действительно будут загружаться из видеопамяти.

Все блоки графического процессора функционируют *параллельно*. Следовательно, в качестве теоретической временной оценки одного этапа интегрирования следует взять *наихудшую* из трех полученных оценок.

$$TIME = max\,(ALU, TEX, MEM) = ALU \approx 30{,}4\ \textit{мс} \text{ или } 33\ \textit{итерации / с}.$$

В действительности, данные величины не всегда являются независимыми. Относительно нашей задачи можно сказать, что все вычисления во фрагментном шейдере начинаются с операции *выборки* из текстуры. Это означает, что потоковые процессоры будут подключаться постепенно по мере завершения операций выборки из текстур. Таким образом, имеем некоторую прибавку к теоретическому времени выполнения одной итерации.

5.5 Фактическая производительность и основные методы ее оптимизации

Из полученных выше теоретических оценок следует, что узким местом для рассматриваемой задачи является производительность потоковых процессоров. Число арифметических операций, которые выполняются во фрагментном шейдере, не может быть уменьшено. Однако можно попытаться повысить эффективность их исполнения на графическом процессоре. Среди различных подходов отметим простой и эффективный метод *разворачивания циклов*: тело цикла повторяется несколько раз, при этом его длина соответствующим образом уменьшается.

Таблица 3. Изменение производительности в зависимости от размера разворачиваемого блока (N = 16384, метод Эйлера, *без* загрузки данных)

Размер разворачиваемого блока	*Производительность итераций в сек / GFLOPS*	*Размер разворачиваемого блока*	*Производительность итераций в сек / GFLOPS*
1 ×1	*23.6 / 123.9*	*4 ×4*	*28.3 / 144.3*
2 ×2	*27.3 / 139.2*	*8 ×8*	*27.0 / 137.7*

Из полученных экспериментальных данных вытекает, что наибольшая эффективность достигается при разворачивании блоков размера 4 × 4. Блоки данного размера будут использоваться для последующих замеров производительности.

Первоначальная оптимизация текстурных выборок уже была проведена: для хранения масс использовалась текстура положений, что позволило *вдвое* сократить время выборки. Заметим, что данный показатель может быть улучшен. Поскольку фрагментный шейдер может выводить результат одновременно в *восемь* цветовых буферов, то с каждым фрагментом можно связать обработку *четырех* частиц одновременно. В процессе вычислений прочитанные из текстур данные будут использоваться сразу для четырех точеных масс, поэтому объем текстурных выборок сократится *вчетверо*. В рассматриваемом примере данная оптимизация не проводилась, поскольку производительность не ограничивалась текстурными модулями. Заметим, что данный факт можно проверить, варьируя частоту графического ядра (на которой функционируют текстурные блоки), но оставляя неизменной частоту потоковых процессоров:

Таблица 4. Изменение производительности в зависимости от частоты текстурных блоков (N = 16384, метод Эйлера, *без* загрузки данных)

Частота ядра / потоковых процессоров	*Производительность итераций в сек / GFLOPS*	*Частота ядра / потоковых процессоров*	*Производительность итераций в сек / GFLOPS*
400 / 1500	*18.3 / 93.3*	*600 / 1500*	*28.3 / 144.3*
450 / 1500	*20.6 / 105.1*	*650 / 1500*	*28.3 / 144.3*
500 / 1500	*22.6 / 115.3*	*700 / 1500*	*28.6 / 145.9*
550 / 1500	*25.0 / 127.5*	*750 / 1500*	*28.6 / 145.9*

Как видно из полученных экспериментальных данных, производительность практически не увеличивается при увеличении частоты ядра выше стандартных 600 МГц. Тем нее менее, описанная оптимизация может сыграть решающую роль во многих других задачах.

Обсудив возможные оптимизации, оценим производительность при использовании различных численных методов. Поскольку в некоторых случаях данные требуется загружать в память центрального процессора, приведем производительность и для этого случая:

Таблица 5. Фактически полученная производительность для системы из N = 16384 тел

Название метода	*Производительность без загрузки данных (итераций в сек / GFLOPS)*	*Производительность с загрузкой данных (итераций в сек / GFLOPS)*
Эйлера	*28.3 / 144.3*	*21.3 / 108.6 (-24%)*
Рунге-Кутты 2-ого порядка	*13.3 / 135.6*	*11.6 / 118.3 (-13%)*
Рунге-Кутты 4-ого порядка	*6.6 / 134.6*	*6.3 / 128.5 (-5%)*

Теоретически возможная производительность интегрирования системы N = 16384 тел методом Эйлера составляет 33 итерации/с. На практике рассматриваемая реализация обеспечила 28 итераций/с, что можно считать хорошим показателем.

Заключение

В работе делается попытка привлечь внимание сообщества, занимающегося высокопроизводительными вычислениями, к возможностям высокопроизводительных вычислений общего назначения на графических процессорах с использованием шейдерных языков. Что позволяет, в отличие от технологий CUDA и Stream Computing, не ограничиваться графическими процессорами только одной фирмы. Рассмотрены возможные выгоды использования графических процессоров в вычислениях общего назначения. Изложены сведения, должные снять “потенциальный барьер” компьютерной графики: основы формирования изображений на видеоадаптере, понятия графического конвейера и шейдерной программы; использование видеоадаптера для вычислений общего назначения на примере сложения матриц. Предложен подход к оптимизации производительности вычислений при решении задач общего назначения. Подход рассмотрен на примере решения задачи моделирования динамики системы N тел (N = 16384). Он включает: теоретическую оценку производительности решения задачи на графическом процессоре; исследование фактической производительности в условиях вариации тактовых частот основных трактов GPU; методы оптимизации производительности, основанные на результатах исследования (в том числе, выбор степени развертывания циклов на основе исследования ее эффективности). Изложенный метод позволил достичь производительности около 150 GFLOPS на графической карте массового производства NVIDIA GeForce 9800 GT с пиковой производительностью порядка 330 GFLOPS.

Литература

1. *Ресурс General-Purpose Computation Using Graphics Hardware* (http://www.gpgpu.org).
2. *Официальный сайт OpenGL* (http://www.opengl.org).
3. *NVIDIA CUDA* (http://www.nvidia.com/object/cuda_home.html).
4. *ATI Stream Computing* (http://ati.amd.com/technology/streamcomputing/resources.html).
5. *Stream processing* (*From Wikipedia,* http://en.wikipedia.org/wiki/Stream_processing).
6. *Тутубалин А. SGEMM на видеокарте и CPU* (http://www.gpgpu.ru/articles/sgemm-6.html).
7. *Архитектура BrookGPU* (http://www-graphics.stanford.edu/projects/brookgpu/arch.html).
8. *Архитектура Sh* (http://www.libsh.org/about.html).
9. *Спецификации расширения ARB_texture_float* (*February 19, 2008, Version 7*). (http://www.opengl.org/registry/specs/ARB/texture_float.txt)
10. *Спецификации расширения ARB_color_buffer_float* (*February 15, 2007, Version 8*). (http://www.opengl.org/registry/specs/ARB/color_buffer_float.txt)
11. *Lars Nyland, Mark Harris, Jan Prins. Fast N-Body Simulation with CUDA.* (http://users.ices.utexas.edu/~organism/m368k/hw5/gpu-gems-3--ch-31-N-body.pdf)
12. *Erich Elsen V. Vishal Mike Houston и др. N-Body Simulations on GPUs.* (http://arxiv.org/pdf/0706.3060)
13. *Спецификации расширения EXT_framebuffer_object* (*April 5, 2006*). (http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt)
14. *Спецификации расширения ARB_draw_buffers* (*December 13, 2004*). (http://oss.sgi.com/projects/ogl-sample/registry/ARB/draw_buffers.txt)
15. *NVIDIA G80: Architecture and GPU Analysis* (http://www.beyond3d.com/content/reviews/1/1).