Интерактивная трассировка лучей и фотонные карты на GPU

Владимир Фролов, Алексей Игнатенко
Факультет Вычислительной Математики и Кибернетики
Московский Государственный Университет имени М.В. Ломоносова, Москва, Россия
{vfrolov, ignatenko}@graphics.cs.msu.ru

Аннотация

Данная работа посвящена ускорению алгоритма фотонных карт с использованием графических процессоров (GPU). Фотонные карты - один из наиболее универсальных и реалистичных методов синтеза изображений, но и один из самых требовательных к вычислительным ресурсам. Мы представляем модифицированный алгоритм, работающий на CUDA-совместимых GPU. В работе также предложен подход организации трассировки лучей, позволяющий в высокой степени утилизировать ресурсы GPU. Наша реализация обратной трассировки лучей такая же быстрая, как и в передовых работах по интерактивной трассировке лучей на GPU.

Ключевые слова: *Интерактивная трассировка лучей, фотонные карты, GPU, глобальное освещение.*

1. ВВЕДЕНИЕ

Фотонные карты [1] – один из наиболее часто используемых алгоритмов расчета глобального освещения. Этот метод позволяет учесть самый широкий спектр эффектов – мягкие тени, вторичное освещение, каустики от отражающих и преломляющих объектов, спектральное разложение света. Благодаря своей универсальности он является стандартным методом расчета освещения в большинстве программ фотореалистичной визуализации. Условно метод фотонных карт можно разбить на три подзадачи – трассировка лучей, построение фотонной карты, сбор освещенности.

1.1 Трассировка лучей на GPU

Задача ускорения трассировки лучей на графических процессорах решалась в работах [2], [3], [4]. Для быстрого поиска пересечений луча и сцены, обычно используют специальные рекурсивные структуры пространственного разбиения – kd и BVH деревья.

В работе [2] применялись алгоритмы поиска в kd дереве, не требующие стека (или алгоритмы с коротким стеком) и трассировка пакетов лучей. В работе [3] также использовались пакеты большого размера, BVH деревья и один стек на весь пакет, реализованный на разделяемой памяти CUDA. В работе [4] наилучшие результаты были получены для алгоритма поиска в kd дереве с коротким стеком и использованием пакетов лучей. Позже мы рассмотрим эти работы подробнее.

Существуют две основные проблемы при реализации трассировки лучей на графических процессорах - отсутствие стека и нехватка регистров GPU. Мы вернемся ко второй проблеме после того как рассмотрим архитектуру GPU и технологию CUDA, а пока что остановимся на первой.

Для ускорения поиска пересечений в трассировке лучей используются иерархические структуры – kd и BVH деревья. Фактически, поиск пересечений луча и треугольника (или какого-то другого примитива) сводится к поиску в kd или BVH дереве. Стандартные алгоритмы поиска в этих деревьях имеют рекурсивную природу. Так как изначально на GPU нет стека, авторы работ [2] и [4] разработали алгоритмы поиска, не требующие стека и алгоритм, использующий короткий стек. Однако на современных GPU с архитектурой G80 и GT200, поддерживающих CUDA, возможна реализация стека на локальной памяти. Мы попробовали алгоритмы, не использующие стек и алгоритм с коротким стеком (на регистрах GPU) и сравнили их нашей реализацией классического алгоритма со стеком на локальной памяти (рис. 4,5). По-видимому, в нашем случае латентность локальной памяти покрывается высокой степенью параллелизма, поэтому алгоритм, использующий стек на локальной памяти, работал быстрее, чем любой из его аналогов, не использующих стек (или использующих короткий стек). Мы вернемся к этому после рассмотрения нашей реализации, где покажем, что проблема отсутствия быстрого стека для современных GPU менее существенна, чем проблема нехватки регистров.

1.2 GPU и CUDA

CUDA (Compute Unified Device Architecture) – это технология, позволяющая создавать программы для графических процессоров на подмножестве языка C++.

В CUDA ядром называется некоторая функция (обычно небольшая), работа которой распараллеливается на GPU. Все потоки выполняют эту функцию, и вначале работы отличаются только идентификатором потока (threadId).

Единица исполнения потоков в CUDA – группа из 32 потоков с названием warp. Можно считать, что GPU – это SIMD машина с шириной команды, равной размеру группы warp (то есть 32). Группы warp исполняются SIMD процессорами, которые также называются мультипроцессорами - Symmetric Multiprocessor (SM).

В SIMD парадигме так организованы вычисления, что если 31 поток пошел по одной ветви кода, но хотя бы один поток пошел по другой, процессору придется выполнить обе ветки. Поэтому расхождения потоков в местах ветвления кода внутри одной группы warp следует избегать. Это означает, что нужно так организовывать вычисления, чтобы минимизировать расхождения потоков по разным веткам кода в каждой группе из 32 потоков с подряд идущими идентификаторами потоков threadId.

Рассмотрим простой пример. Пусть имеется всего 64 потока. Мы хотим, чтобы половина из них выполняла какую-то одну задачу, а другая половина – другую. Для этого мы ставим некоторое условие if и ветку else. Условие “if (threadId %2 == 0)” будет порождать расхождение в группах warp, потому что потоки с четными номерами попадут внутрь тела условия if, а потоки с нечетными номерами - нет. Условие “if (threadId < 32)” не будет вызывать расхождений, так как все потоки первой группы warp с threadId от 0 до 31 попадут внутрь тела

условия, а все потоки следующей группы warp с threadId от 32 до 63 не попадут внутрь него.

В CUDA локальные переменные помещаются по возможности в регистры. Мультипроцессор GPU имеет ограниченное число регистров, причем вычисления организованы таким образом, что эффективность выполнения кода на GPU напрямую зависит от количества регистров, занимаемых ядром. Существует понятие занятости мультипроцессора. Занятость мультипроцессора – это отношение числа активных групп warp на мультипроцессоре к максимально возможному числу активных групп warp, которые могут работать одновременно на этом мультипроцессоре. Чем меньше занимаемое число регистров, тем выше занятость. Например, если ядро занимается в основном арифметическими вычислениями, снижение занимаемых регистров с 32 до 16 увеличит в два раза количество активных групп warp на мультипроцессоре и соответственно скорость возрастет в 2 раза.

Следует обратить особое внимание на работу с текстурами. Если ядро делает много текстурных выборок, латентность которых не скрывается вычислениями, эффективность может очень сильно зависеть от занятости. Так в архитектуре GT200 используется один текстурный кэш на 3 мультипроцессора (рис. 1). Это означает, что если на каждом мультипроцессоре количество активных групп warp возрастет в 2 раза, эффективность использования текстурного кэша может возрасти в 6 раз (при условии, что большинство потоков читают близкие или одни и те же области памяти). Поиск в kd дереве относится именно к такому случаю. Близкие по идентификатору потоки шагают примерно по одним и тем же узлам. Поэтому, несмотря на то, что латентность текстурной памяти не может быть покрыта вычислениями, при поиске в kd дереве, за счет правильного использования текстурного кэша может быть достигнута высокая эффективность.

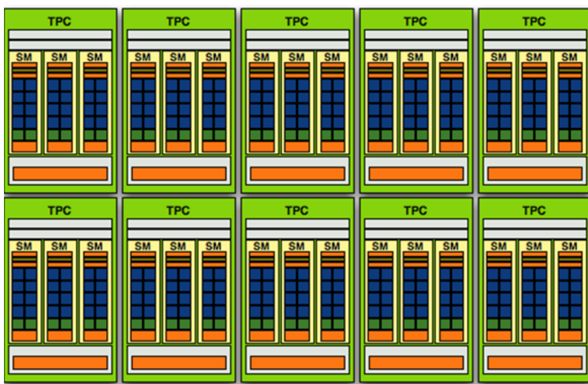

Рис. 1. Архитектура GT200. В каждом TPC на 3 SM один текстурный кэш.

Проблема нехватки регистров не является чем-то специфическим для CUDA, она существует во всех языках программирования для GPU – HLSL, GLSL, Brook, OpenCL, DirectX11 Compute Shaders и прочих.

1.3 Экономия регистров

Сложность реализации трассировки лучей на GPU в том, что алгоритм требует очень много локальных переменных. Причем, все переменные обычно используются. Компилятор старается поместить все переменные в регистры. А те, что не удается поместить в регистры, оказываются в медленной локальной памяти. Обычное пересечение луча и треугольника занимает уже больше 20 регистров (это нетрудно оценить, если выписать алгоритм и посмотреть, от каких переменных компилятор точно не сможет избавиться). То есть пересечение луча и треугольника в принципе не может быть размещено в малом количестве регистров. Но это только самая малая часть алгоритма трассировки лучей. Нужно еще выполнять поиск в ускоряющих структурах (в нашем случае это kd деревья). Также нужно делать затенение, которое может быть очень сложным и занимать произвольное число регистров и локальной памяти.

В случае критической нехватки регистров в качестве стандартного решения используется паттерн uber-kernel (который также называют mega-kernel). Сложный код делится некоторым образом на части. В ядре присутствуют все части, но каждая в своей ветке if. Также имеется флаг, отвечающий за то, какая часть кода должна выполняться. Во время выполнения, процессор может периодически прыгать с одной части кода на другую, сохраняя некоторые важные данные в разделяемой или локальной памяти. Суть такого подхода в том, что он позволяет использовать одни и те же регистры для разных переменных.

Рассмотрим пример того, как мог бы быть реализован паттерн uber-kernel для трассировки лучей. Во время поиска в kd дереве наступает момент, когда луч дошел до листа дерева. Тогда флаг устанавливается в «intersect» и делается переход в начало ядра. Нужные данные могут быть сохранены в разделяемой памяти. Далее считаются пересечения. Если пересечение найдено в пределах узла kd дерева, то данные сохраняются в разделяемой памяти, флаг устанавливается в «shade» и делается переход в начало ядра. После чего ядро начнет выполнять затенение. Если же пересечение не было найдено, флаг устанавливается в «traverse» и выполняется переход в начало ядра. Затем ядро продолжит поиск в kd дереве.

У такого подхода есть ряд серьезных недостатков:

1. Очень усложняется код
2. Самая сложная часть является лимитирующей по регистрам
3. Трудно профилировать код – невозможно понять, какая же именно часть работает медленно.

Наиболее существенный недостаток – второй. Предположим у нас есть три части алгоритма, занимающие разное число регистров. Например, поиск в kd дереве – 16 регистров, подсчет пересечений – 24 а затенение – 32 регистра. Результирующее ядро будет занимать максимум из этих трех чисел. Это означает, что мы очень сильно потеряем в эффективности на поиске в kd дереве из-за того что число активных групп warp уменьшится ровно в 2 раза. При этом, эффективность текстурного кэша может упасть более чем в два раза (в 2-6 раз на архитектуре GT200). А так как поиск в kd дереве – это нагрузка исключительно на текстуры, то и общая скорость может снизиться довольно сильно по сравнению с теоретическим максимумом.

Судя по всему, авторы работ [2] и [3] используют паттерн uber-kernel, так как они утверждают, что реализовали весь алгоритм трассировки лучей в одном ядре. Причем, в работе [3] говорится, что при использовании полного затенения по фонгу и трассировки теневых лучей, занятость мультипроцессоров падает с 63% до 38% по сравнению с простым затенением и трассировкой только первичных лучей. Таким, образом, uber-kernel не позволит сделать код по-настоящему эффективным, если потребуется реализация

сложного затенения, распределенной трассировки лучей или каких-либо других дополнительных эффектов.

Авторы работы [4] использовали совсем другой подход. Они разделили алгоритм поиска на 3 части (на самом деле больше, но мы решили не усложнять изложение) – спуск по kd дереву (kd_down), подъем по kd дереву (kd_up) и подсчет пересечений (intersect). Трассировка ведется блоками. Сначала запускается ядро kd_down, осуществляющее спуск. Как только все лучи блока дошли до листа, ядро спуска останавливается. Запускается ядро подсчета пересечений intersect. Когда все лучи в блоке завершили подсчет пересечений, запускается ядро kd_up. Затем снова ядро спуска kd_down для тех лучей, которые не нашли пересечение. При этом каждый раз происходит остановка и запуск ядра для всего блока, даже для тех лучей, которые уже ударились о поверхность (просто эти потоки сразу же завершают свою работу). Фактически, авторы работы [4] просто разрезали код на части. Ядра kd_down, intersect и kd_up работают последовательно в цикле до тех пор пока все лучи не найдут пересечения или не вылетят за пределы kd дерева.

Это лучше, чем uber-kernel, но проблема такого подхода в том, что на сложных сценах происходит слишком много переключений ядер. Каждое переключение - это дополнительные расходы времени, прежде всего, на работу с памятью. Каждый раз, когда запускается новое ядро, значения всех локальных переменных теряются. Промежуточные данные нужно сохранять в DRAM GPU и загружать из нее, что снижает скорость. В самой статье [4] детали реализации не описаны, поэтому нам пришлось проанализировать исходные коды их трассировщика лучей. Мы полагаем, что в значительной степени производительность программы из работы [4] объясняется именно тем, что авторы разделили код на части и сократили таким способом число занимаемых регистров для каждой отдельной части (ядра).

2. ДЕКОМПОЗИЦИЯ ТРАССИРОВКИ ЛУЧЕЙ

Мы предлагаем новый подход для организации трассировки лучей на CUDA, лишенный недостатков, присущих uber-kernel и множественным переключениям ядер. Основная идея нашего подхода та же самая, что и в работе [4] - нужно разделить алгоритм трассировки лучей на несколько ядер, каждое из которых выполняет только одну узкую задачу. Мы разделили алгоритм на 3 части – поиск в kd дереве, подсчет пересечений и затенение (рис. 2). Отличие от работы [4] заключается в том, что переключение ядер производится всего 3 раза для одного типа лучей (первичных, теневых, отраженных).

Рис. 2: декомпозиция алгоритма трассировки лучей.

Если отделить затенение от всего остального достаточно просто, то для того чтобы разделить поиск в kd дереве и подсчет пересечений мы изменили сам алгоритм поиска. Идея формулируется следующим образом: если мы выполняем какой-то код (например, поиск в kd дереве), то мы должны стараться выполнять его как можно дольше, не останавливая ядро и не теряя значений локальных переменных. На рис. 3 показан классический код поиска в kd дереве. На рис. 4 и 5 изображена наша модификация алгоритма.

```
boolean kd_tree_traversal(Ray ray, Hit* out_hit):
  (hit,t_near,t_far) = IntersectRayAABB(ray, scene_AABB)
  if (!hit): return

  while (true):
    while (!node.Leaf()):
      t_split  = IntersectRayPlane(ray, node.Plane())
      if(t_split < t_near):
        node = Near(node)
      else if(t_split > t_far):
        node = Far(node)
      else:
        node = Near(node)
        stack.push(t_far, &Far(node))

    (hit,t) = IntersectAllPrimitivesInLeaf(ray, out_hit)
    if (hit && t <= t_far):
      return true

    if (stack.empty()):
      return false
    t_near = t_far
    (t_far, farNodeAddress) = stack.pop()
    node = GetNode(farNodeAddress)
```

Рис. 3: Псевдокод классического алгоритма поиска в kd дереве.

```
void kd_tree_traversal_only(Ray ray, Hit* out_hit)
  (hit,t_near,t_far) = IntersectRayAABB(ray, scene_AABB)
  if (!hit): return

  while (true):
    while (!node.Leaf()):
      t_split  = IntersectRayPlane(ray, node.Plane())
      if(t_split < t_near):
        node = Near(node)
      else if(t_split > t_far):
        node = Far(node)
      else:
        node = Near(node)
        stack.push(t_far, &Far(node))

    STORE_DATA_IN_LEAF_LIST(node, t_far)

    if (stack.empty()):
      return false
    t_near = t_far;
    (t_far, farNodeAddress) = stack.pop()
    node = GetNode(farNodeAddress)
```

Рис. 4: Наша модификация классического алгоритма.

Таким образом, на первом этапе мы проходим kd дерево насквозь, составляя для каждого луча список листьев,

которые он посетил. А на следующем шаге мы проходим по списку листьев и считаем в них пересечения.

```
boolean intersect_in_leafes(Ray ray, Hit* out_hit,
                            LeafList* leaf_list):
  while (!leaf_list.End()):
    (node,t_far) = LOAD_DATA_FROM_LEAF_LIST(leaf_list)
    (hit,t) = IntersectAllPrimitivesInLeaf(ray, out_hit)
    if (hit && t <= t_far):
      return true
  return false
```

Рис. 5: Алгоритм подсчета пересечений в списке листьев.

Стек реализован на локальной памяти. Мы пробовали различные безстековые алгоритмы из работ [2] и [4], в частности алгоритм “kd-tree restart” и короткий стек на регистрах. Но они дали худший результат чем классический вариант. По-видимому, это связано с тем, что благодаря высокой занятости мультипроцессоров, нам удалось скрыть латентность локальной памяти за счет высокой степени параллелизма.

Конечно, с алгоритмической точки зрения, наш подход менее эффективен, чем классический вариант, так как мы все время проходим kd дерево насквозь. Но на практике, за счет того, что мы выделили поиск в kd дереве в отдельное ядро и уместили его в 16 регистров, наша реализация трассировки лучей столь же быстрая, как и в работе [2]. Благодаря тому что мы уложились в 16 регистров для поиска в kd дереве, нам удалось добиться 67% занятости мультипроцессора на архитектуре G80 и 100% занятости на архитектуре GT200. Более того, на сценах размером вплоть до 1.5 миллиона треугольников поиск в kd дереве не является узким местом.

По сравнению с работой [2], наш подход имеет ряд преимуществ:

1. За счет того, что мы провели декомпозицию, нам удалось измерить скорость работы отдельных частей алгоритма трассировки лучей, что не может быть сделано, используя подход из работы [2].
2. Все части системы слабо связаны и легко заменяемы. Например, можно заменить kd деревья на BVH и придется поменять только одно ядро – то что отвечает за поиск в kd дереве. Но весь остальной код останется без изменений. То есть наш подход будет работать и для BVH деревьев. Аналогично,
3. Ядро, отвечающее за затенение может быть сколь угодно сложным и это никак не влияет на производительность остальной части системы.
4. Во время прохода по kd дереву, количество ветвлений в warp-ах значительно меньше, чем в случае с объединенным вариантом, так как никакие лучи не простаивают, пока другие считают пересечения в листьях (такое происходит если какая-то часть лучей из warp-а попала в лист, а другая часть не попала в него). В нашем случае во время поиска в kd дереве пересечения вообще не считаются.

Недостаток нашего подхода в том, что при больших разрешениях тратится довольно много памяти, потому что для каждого пиксела нужно сохранять список листьев, которые прошел луч. Эта проблема, однако, легко решается путем разбиения экрана на несколько более мелких частей и их последовательной либо параллельной (на нескольких GPU) обработки.

Мы также сравнивали наш алгоритм с подходом, реализованным в работе [4]. Для этого мы уменьшали размер списка листьев в n раз и запускали наши ядра последовательно, в цикле из n итераций. На видеокарте GF8800GTX производительность падала в 1.5 раза даже для n=2. Это означает, что для CUDA и текущих архитектур G80 и GT200 подход с простым разрезанием кода, использованный в работе [4] (где использовалась GPU с иной архитектурой), будет работать намного хуже нашей декомпозиции.

2.1 Результаты для обратной трассировки лучей

Мы провели сравнение с другими работами на сцене Conference Room (280 тысяч треугольников). Эта сцена была выбрана по причине того, что она упоминается практически во всех статьях по интерактивной трассировке лучей. Замеры были сделаны в разрешении 1024x1024 на GF8800GTX.

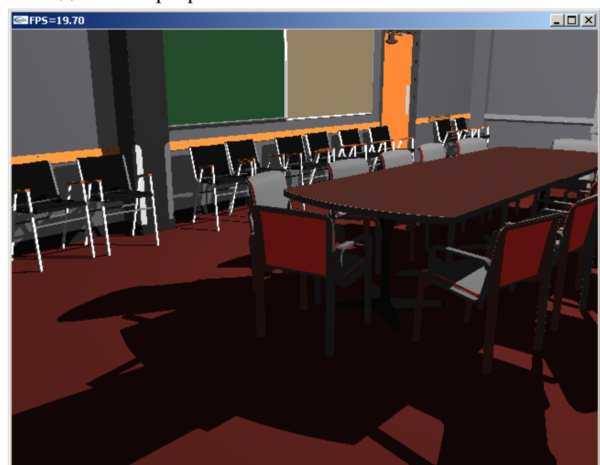

Рис. 6. Conference Room (280 тысяч треугольников). На картинке разрешение 640x480, тени от одного точечного источника, GTX260.

Работа	Ускоряющая структура	Только первичные лучи	Первичные лучи + теневые от одного точечного источника
Наша реализация	kd tree	12 FPS	5.8 FPS
[2]	kd tree	16.7 FPS	6.7 FPS
[3]	BVH	16 FPS	6.1 FPS
[4]	kd tree	15.2* FPS	7* FPS
[5]	BVH	8** FPS	-

Таблица 1: Сравнение скорости обратной трассировки лучей с другими работами. Числа приведены в кадрах в секунду (FPS).

В работах [2], [3] и [4] для первичных лучей использовалась трассировка пакетов, поэтому падение производительности при переходе к вторичным лучам в них больше чем в 2 раза. Мы не использовали пакеты в явном виде (неявно, на уровне групп warp, они конечно присутствуют). Поэтому в нашей

работе меньше падение производительности при переходе к вторичным лучам – теневым и отраженным.

На рис. 8 приведен график зависимости производительности от количества треугольников для сцены, состоящей из стульев (рис. 9). Один стул содержит 1464 треугольника. Мы добавляем стулья в сцену, расставляя их по квадрату, и стараемся сделать так, чтобы все стулья были видимы и занимали экран полностью. На графике каждая следующая отметка содержит в 4 раза больше треугольников, чем предыдущая. Измерялось время только для первичных лучей. Мы откладываем кривые 1/log(N) и 1/(log(N)*log(N)) от первой точки, чтобы оценить характер кривой зависимости производительности от количества треугольников (где N – количество треугольников).

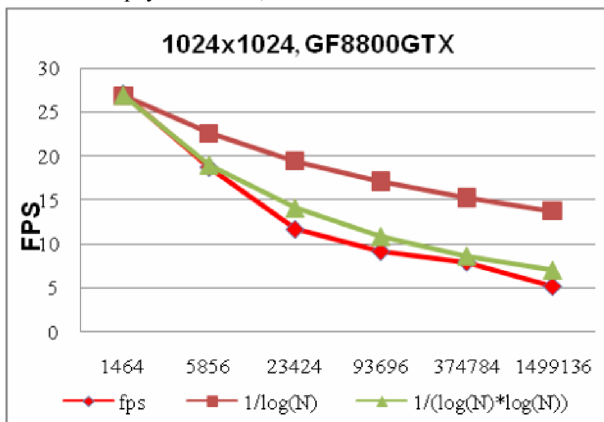

Рис. 7: график зависимости производительности от количества треугольников в тестовой сцене из стульев

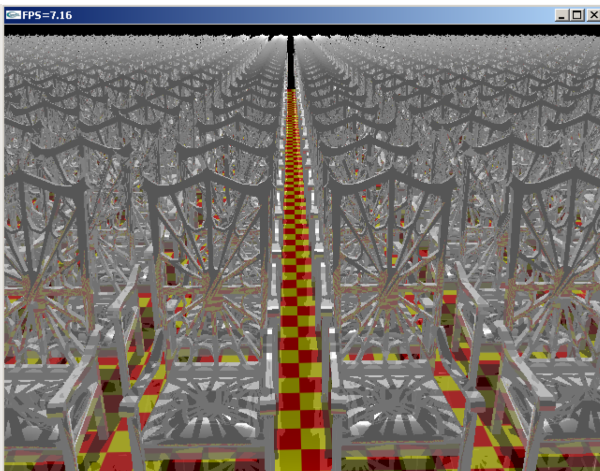

Рис. 8. Сцена из стульев, на которой производились замеры производительности. На картинке разрешение 640x480, один уровень отражения и тени от точечного источника, 1.5 миллиона треугольников, GTX260.

2.2 Измерения скорости отдельных частей алгоритма

На сцене Conference Room (280 тысяч треугольников):

- поиск в kd дереве занимает 15% общего времени
- подсчет пересечений - 74% общего времени
- все остальное (в том числе затенение) - 11%.

Таким образом, мы можем утверждать, что наша реализация поиска в kd дереве крайне эффективна, потому как занимает всего 15% общего времени. Это означает, что на современных GPU (с архитектурами G80 и GT200) не требуются специальные безстековые алгоритмы, о которых говорится в работах [2] и [4].

На сцене из стульев для 1.5 миллиона треугольников:

- поиск в kd дереве занял 46% времени.
- подсчет пересечений – 45% времени.

Это является довольно неплохим результатом, так как на такой сцене kd дерево очень глубокое и равномерное. То есть это худший пример для нашего алгоритма.

2.3 Трассировка фотонов

Особенность трассировки фотонов заключается в том, что в отличие от лучей, идущих из глаза, теневых или даже отраженных лучей, начальные позиции и направления фотонов распределены случайно. Причем, это верно как для первичных фотонов так и для отраженных. Если генерировать случайное направление для каждого фотона, то лучи, представляющие фотоны, будут очень сильно расходиться. Это создает проблемы для GPU так как сильно расходящиеся лучи порождают множество ветвлений в группах warp. В результате производительность может упасть в несколько раз. Вместо того чтобы вычислять случайно направление для каждого фотона, мы поступили следующим образом:

- разделили все фотоны на группы по 32.
- Для каждой группы вычислили случайное направление
- Для каждого фотона в группе ввели небольшое отклонение от заданного направления

Таким образом, мы трассируем фотоны группами, в каждой из которых фотоны расходятся не сильно друг от друга. Для отраженных фотонов мы поступаем аналогичным образом, хотя для них эта техника менее эффективна, так как даже после сортировки по узлам kd дерева, фотоны с близким адресом могут лежать достаточно далеко друг от друга на поверхности.

3. СБОР ОСВЕЩЕННОСТИ

После завершения трассировки фотонов в некоторых точках сцены происходит сбор освещенности с ближайших фотонов. Существует несколько способов выбирать, в каких именно точках собирать освещенность и как визуализировать фотонную карту. Мы выбрали простой, но универсальный способ - трассируем лучи из виртуального глаза для каждого пиксела и собираем освещенность во всех точках, где луч пересек сцену. Мы не производили какую-либо фильтрацию. Этот способ часто называют “визуализацией фотонной карты напрямую”.

Обычно задача сбора освещенности формулируется следующим образом: в каждой точке нужно найти N ближайших фотонов, сложить их энергию, возможно с некоторой зависимостью по расстоянию, и поделить полученную сумму на площадь поверхности сферы, радиус которой равен расстоянию до самого дальнего фотона. Этот алгоритм хорош тем, что для заданного числа N он динамически выбирает радиус сбора в зависимости от того, сколько фотонов лежит вблизи точки сбора. Если фотонов мало, радиус сбора большой, фотонов много – маленький. Качество получаемого изображения зависит только от числа

* Результаты измерены на ATI X1900 XTX с другим ракурсом

** Результаты измерены на GTX280 и масштабированы на 8800 GTX

N. Однако такой алгоритм создает множество проблем. Во-первых, чтобы эффективно искать N ближайших фотонов, каждый раз после трассировки необходимо построить специальное kd дерево, содержащее в своих узлах фотоны. Можно использовать и другую ускоряющую структуру, например регулярную сетку, как в работе [5]. Во-вторых, алгоритм поиска N ближайших фотонов довольно сложен для эффективной реализации на GPU, хотя сделан в работах [5], [6] и [7].

Мы использовали более простой алгоритм - выбрали некоторый фиксированный радиус сбора и взяли для оценки освещения все фотоны, попавшие в сферу с заданным радиусом сбора. Такой алгоритм намного проще реализовать на GPU. Проще в данном случае означает, что можно добиться от него более высокой утилизации ресурсов GPU, чем от алгоритма, выполняющего поиск N ближайших фотонов. Более того, этот алгоритм дал нам возможность не перестраивать ускоряющие структуры для фотонов.

3.1 Построение пространственного индекса

Если сделать предположение, что фотоны могут быть распределены только по поверхности (отказаться от объемных эффектов), то для заданной геометрии можно построить kd-дерево для фотонов всего один раз. Мы использовали SAH kd дерево для геометрии и разбили каждый его лист равномерно до некоторой величины, сравнимой с радиусом сбора.

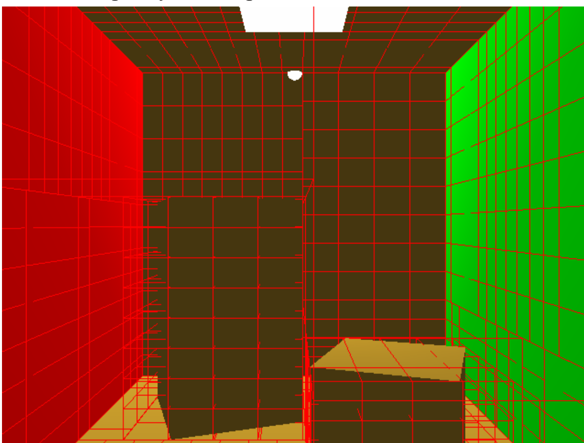

Рис. 9. kd дерево для фотонов.

После окончания трассировки фотонов нужно определить для каждого фотона, в какой узел kd дерева он попал. А затем нужно отсортировать фотоны по узлам kd дерева. И в каждый узел записать смещение для списка фотонов и их число в узле. Мы определяем индекс узла kd дерева для каждого фотона с помощью поиска в kd дереве на GPU. Затем мы сортируем по узлам kd дерева фотоны, используя линейную сортировку индексов фотонов на CPU с помощью быстрых списков. После этого мы загружаем отсортированные индексы на GPU и за одну операцию копирования для каждого массива (мы использовали принцип хранения Structure Of Arrays – SOA, поэтому фотоны храняться в нескольких массивах) перемещаем все остальные данные о фотонах.

4. МНОГОПРОХОДНОЙ АЛГОРИТМ

Первая проблема, с которой мы столкнулись при реализации фотонных карт на GPU – это учет фотонов, отразившихся от поверхности один или более раз. Для того чтобы обрабатывать фотоны параллельно, в CUDA на каждый фотон заводится свой поток. И каждый поток должен записывать информацию об ударах в фотонную карту строго в свою область памяти.

Пусть на начальном этапе имелось n фотонов и на каждый фотон отводится k байт. Пусть глубина трассировки фотонов равна m. Тогда даже если в соответствие с принципом “русской рулетки” на каждом этапе отражается только часть фотонов (например половина), нам все равно потребуется k*n*m байт. Нетрудно посчитать, что на CPU в случае если каждый раз отражается только половина фотонов, нам потребуется не больше чем k*m*2 байт.

Следующая проблема заключается в том, что для достижения хорошего качества нужно больше фотонов, чем способно поместиться в памяти GPU. Обе проблемы усугубляются тем, что для эффективного доступа и соблюдения правила объединения запросов к памяти в CUDA (coalescing), мы были вынуждены потратить 56 байт на фотон (3 массива из элементов float4 и один массив float2). Мы сохраняли позицию фотона, направление, цвет, индекс узла kd дерева, флаг активности фотона и некоторую информацию о поверхности, в которую ударился фотон. Мы могли бы сократить эту информацию до 48 байт, используя битовые операции, но от этого ничего принципиально бы не изменилось. Памяти все равно бы тратилось слишком много.

Мы модифицировали многопроходный алгоритм, предложенный в статье [9] и переложили его на GPU. Сначала мы испускаем некоторое число (например, один миллион) фотонов из источника света. Трассируем их, строим фотонную карту и собираем освещенность. Затем, мы обрабатываем фотоны – часть из них отражается, часть умирает. Все активные фотоны трассируется. Строится фотонная карта, собирается освещение. При этом мы используем те же области памяти, что и для первичных фотонов. Так как с первичных фотонов мы уже собрали освещение, они нам больше не нужны, и мы можем задействовать их память для вторичных фотонов. Аналогично мы поступаем с третичными, четвертичными фотонами и так далее. Таким образом, мы можем обработать любое число переотражений фотонов, имея ограниченное количество памяти. Та же самая идея позволяет нам рассчитать фотонную карту для вообще любого числа фотонов. Мы обрабатываем фотоны последовательно - блоками по N фотонов. Если у нас имеется 70 миллионов фотонов, то мы можем разделить их на блоки, например по 2 миллиона фотонов, и за 35 проходов обработать все 70 миллионов. Таким способом можно легко распараллелить расчет фотонной карты на несколько GPU. Часть проходов рассчитывается на одной GPU, часть на другой. Результирующее освещение в каждой точке просто складывается. Это дает возможность использовать суперкомпьютер Tesla для расчета фотонных карт достаточно простым способом.

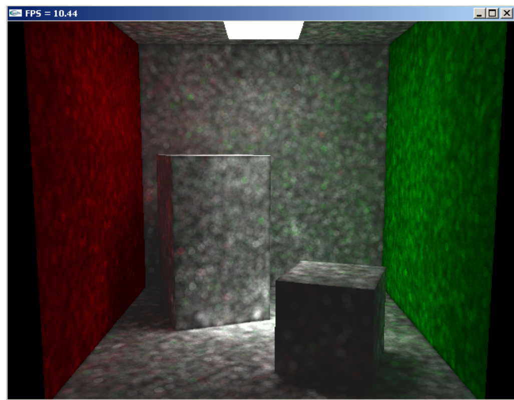

Рис. 9. Cornel Box, 260 тысяч фотонов, 1 проход, 10.4 fps.

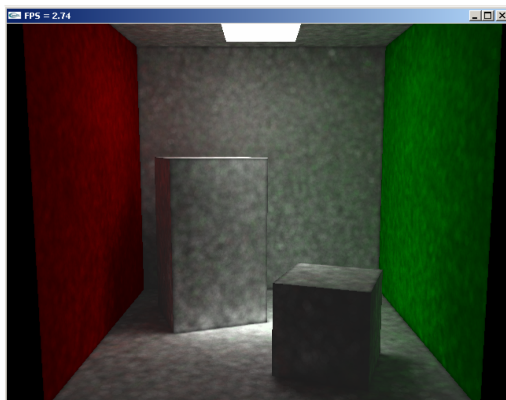

Рис. 10. Cornel Box, 1 миллион фотонов, 1 проход, 2.7 fps.

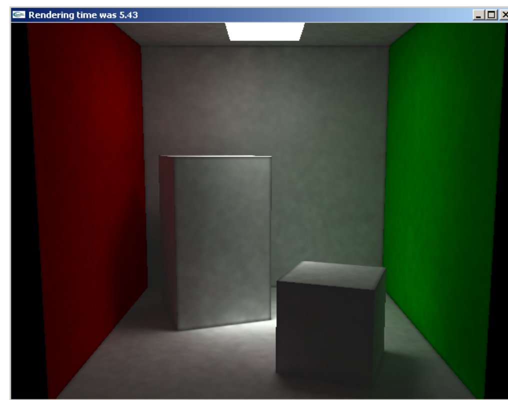

Рис. 11. Cornel Box, 2 миллиона фотонов, 8 проходов, время рендеринга – 5.4 секунд.

Рис. 12. Cornell Box со стулом, 260 тысяч фотонов, 1 проход 6.6 fps.

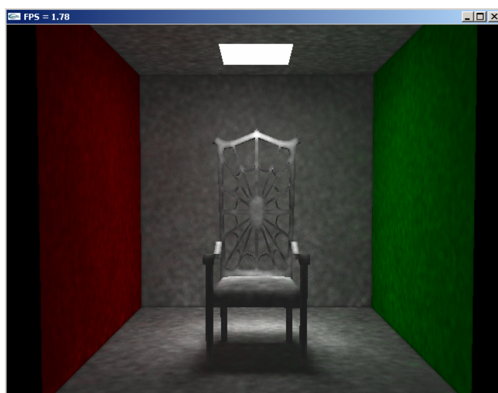

Рис. 13. Cornell Box со стулом, 1 миллион фотонов, 1 проход, 1.8 fps.

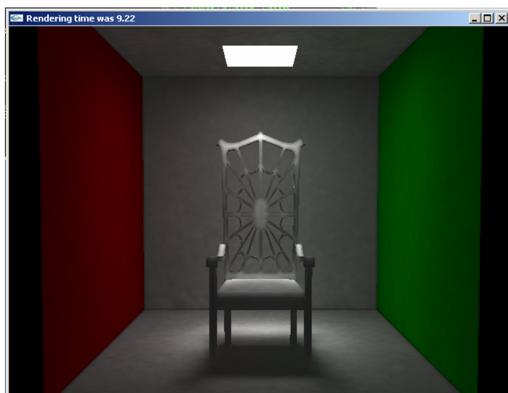

Рис. 14. Cornell Box со стулом, 2 миллиона фотонов, 8 проходов, время рендеринга – 9.26 секунд.

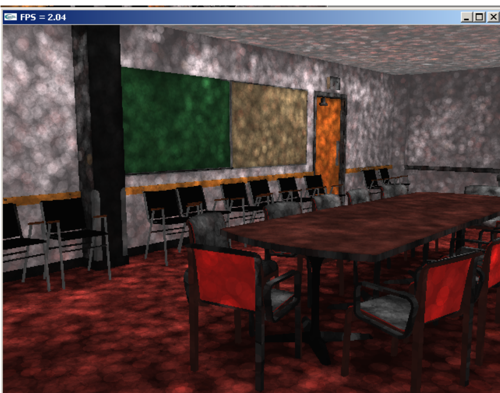

Рис. 15. Conference Room (280 тысяч треугольников), 260 тысяч фотонов, 2.0 fps.

Рис. 16. Conference Room, 1 миллион фотонов, время рендеринга – 1.64 секунды.

Рис. 17. Conference Room, 2 миллиона фотонов, 8 проходов, время рендеринга – 25 секунд.

На сцене Conference Room под потолком располагаются 4 протяженных источника света, которые не были визуализированы.

5. РЕЗУЛЬТАТЫ

На рисунках 9-17 представлены результаты работы нашей программы. Снимки получены на машине с GF8800GTX, Core 2 Quad 2.4 GHz. Глубина трассировки фотонов равнялась двум. Все наши демонстрационные программы могут быть найдены на сайте http://ray-tracing.ru. Системные требования – CUDA 2.1 или выше, любая видеокарта с поддержкой CUDA и достаточным количеством видеопамяти. В работе [6] экранизация (рендеринг) сцены Cornell Box занимал порядка минуты, но мы считаем, что проводить сравнение с работой [6] некорректно, так как в ней использовалась очень старая аппаратура (GeForce FX 5900 Ultra). Мы сравнивали нашу реализацию с работами [7], [8] и библиотекой OpenRT.

В работе [7] использовалась фотонная карта только для расчета каустиков. Тестировались две сцены размером 3 и 20 тысяч треугольников. В первой сцене для расчета каустиков было задействовано 200 тысяч фотонов и освещение собиралось с 50 ближайших – 12 fps. На второй сцене из 20 тысяч треугольников задействовано 400 тысяч фотонов, сбор освещенности с 40 ближайших - 7.5 fps. Результаты измерены на видеокарте GF8800 ULTRA в разрешении 800x600.

В работе [8] глобальное освещение вычислялось методом фотонных карт с последующим финальным сбором (Final Gathering - FG) и техникой кэширования вторичного освещения (irradiance caching). Прямое освещение вычислялось обратной трассировкой лучей. Использовались сцены размером от 20 до 80 тысяч треугольников. Было задействовано от 200 до 400 тысяч фотонов. На видеокарте GTX280 в разрешении 1024x1024 было достигнута скорость визуализации 1.5-4.2 fps. Можно условно считать, что GTX280 в 3 раза лучше, чем GF8800GTX. По крайней мере, это верно для нашей реализации трассировки лучей.

По сравнению с работой [8], мы можем обработать гораздо больше фотонов (примерно в 3-4 раза больше) за то же самое время потому что мы не перестраиваем ускоряющие структуры и упрощаем алгоритм сбора освещенности. Более того, мы можем обрабатывать произвольное число фотонов благодаря многопроходному алгоритму. Однако так как мы не использовали кэш освещения (что делалось в работе [8]) и считали прямое освещение фотонными картами, наши изображения достаточно шумные для малого числа фотонов. Также мы не реализовывали метод Final Gathering, поэтому нам сложно привести более точное сравнение с работой [8].

Из существующих решений стоит упомянуть библиотеку OpenRT, которая на сцене Conference Room показывает скорость 1.76 кадров в секунду на 17 двуядерных машинах. Так как разработчики OpenRT не выкладывают данные о количестве задействованных фотонов и о том, как было рассчитано первичное освещение, нам трудно привести точное сравнение.

6. ЛИТЕРАТУРА

[1] Jensen, H. W. 2004. *A practical guide to global illumination using ray tracing and photon mapping.* In ACM SIGGRAPH 2004 Course Notes (Los Angeles, CA, August 08 - 12, 2004). SIGGRAPH '04. ACM, New York, NY, 20.

[2] Popov S., Günther J., Seidel H.-P., Slusallek P. *Stackless KD-Tree Traversal for High Performance GPU Ray Tracing.* In Proceedings of the EUROGRAPHICS conference, vol. 26 (2007), Number 3.

[3] Günther J., Popov S., Seidel H.-P., Slusallek P. *Realtime Ray Tracing On GPU With BVH-Based Packet Traversal.* In Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing. IEEE Symposium on Volume, Issue, 10-12 Sept. 2007. p. 113 – 118.

[4] Horn, D. R., Sugerman, J., Houston, M., and Hanrahan, P. 2007. *Interactive k-d tree GPU raytracing.* In Proceedings of the 2007 Symposium on interactive 3D Graphics and Games (Seattle, Washington, April 30 - May 02, 2007). I3D '07. ACM, New York, NY, 167-174.

[5] Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D., *Fast BVH construction on GPU.* In Proceedings of the EUROGRAPHICS conference , vol. 28, number 2, 2009.

[6] Purcell, T. J., Donner, C., Cammarano, M., Jensen, H. W., and Hanrahan, P. 2005. *Photon mapping on programmable graphics hardware.* In *ACM SIGGRAPH 2005 Courses* (Los Angeles, California, July 31 - August 04, 2005). J. Fujii, Ed. SIGGRAPH '05. ACM, New York, NY, 258.

[7] Zhou, K., Hou, Q., Wang, R., and Guo, B. 2008. *Real-time KD-tree construction on graphics hardware.* In *ACM SIGGRAPH Asia 2008 Papers* (Singapore, December 10 - 13, 2008). J. C. Hart, Ed. SIGGRAPH Asia '08. ACM, New York, NY, 1-11.

[8] Wang, R., Zhou, K., Pan, M., and Bao, H. 2009. *An efficient GPU-based approach for interactive global illumination. ACM Trans. Graph.* 28, 3 (Jul. 2009), 1-8.

[9] Bounded A., Paulin M., Pitot P., Pratmarty D. *Low Memory Spectral Photon Mapping.* In Proceedings of the WSCG conference, 2004.

7. ABSTRACT

This paper is about GPU accelerated ray tracing and photon mapping. We present a modified photon mapping algorithm that is entirely executed on CUDA capable GPUs. We also propose an efficient approach to the GPU ray tracing. Our realization of backward ray tracing is as fast as in top papers on GPU ray tracing.

8. ABOUT THE AUTHORS

Vladimir Frolov is the fifth year student of Moscow State University. His contact e-mail is vfrolov@graphics.cs.msu.ru.

Alexey Ignatenko is a PhD researcher at Computational and Cybernetics department of Moscow State University. His research interests include photorealistic 3D rendering, 3D modeling and reconstruction, image based rendering and adjacent fields. His contact e-mail is ignatenko@graphics.cs.msu.ru.