Унифицированный механизм сэмплирования изображений для современных методов интегрирования освещённости на GPU

Фролов В.А.[1,2]**.** vova@frolov.pp.ru**,**
Галактионов В.А.[1] **.** vlgal@gin.keldysh.ru
(1) ИПМ им. М.В. Келдыша РАН.
(2) МГУ имени М.В, Ломоносова

Аннотация

Сэмплирование изображения - это процесс генерации и учёта вклада выборок (сэмплов) в изображение в методах Монте-Карло интегрирования и рендеринге. В данной работе предлагается общий подход к сэмплированию изображений на GPU в современных методах интегрирования освещённости. Разработанный нами подход протестирован на трёх алгоритмах: Трассировка путей (path tracing), прямая трассировка путей (light tracing) и усечённая двунаправленная трассировка путей (instant bidirectional path tracing). Наш подход не ограничивается в применении перечисленными алгоритмами и может без каких-либо дополнительных модификаций быть использован для других методов интегрирования (например, для всех видов Metropolis Light Transport). Данный подход позволяет нам рассчитывать на GPU изображения в больших разрешениях (до 60Kx60K) без значительных затрат памяти GPU, и является условно-бесплатным в плане производительности.

1 Введение

Необходимость в интегрировании освещенности (или решении проблемы глобальной освещённости) возникает при решении широкого круга задач компьютерной графики, проектирования архитектурных сооружений, оптического моделирования, компьютерной анимации и виртуальной реальности. Интеграл освещённости (рис. 1) в действительности является многомерным, поскольку значения функции L в одной точке трёхмерной сцены зависят от значений самого интеграла I в других точках этой же сцены (подобную зависимость называют уравнением Фредгольма второго рода).

$$I(\varphi_r,\theta_r)=\int_{\varphi_i}\int_{\theta_i} L(\varphi_i,\theta_i)R(\varphi_i,\theta_i,\varphi_r,\theta_r)\cos(n,l_{\varphi_i,\theta_i})d\varphi_i d\theta_i$$

Рис. 1. Интеграл освещённости.

Из-за высокой размерности интеграла его расчёт является вычислительно ёмкой задачей, из-за чего на практике в рендеринге часто используют графичекие процессоры.

Однако, большинство современных рендерсистем на GPU (как промышленных, так и экспериментальных) используют лишь базовый алгоритм трассировки путей (Path Tracing, **PT** [Kajiya, 1986]) из-за того, что реализация более совершенных алгоритмов (например, **BPT** [Veach, 1998], **VCM** [Georgiev et. all, 2012] или **MLT** [Veach and Guibas, 1997; Kelemen et. all 2002; Hachisuka et. all 2014]) встречает ряд принципиальных трудностей при реализации на графических процессорах. Одна из таких трудностей – это *параллельный* учёт вклада сэмплов в изображение из *множества потоков*.

1.1 Сэмплирование изображения

Трассировка путей – алгоритм, который идеально распараллеливается как на CPU, так и на GPU, поскольку интеграл в каждом пикселе изображения вычисляется независимо от других пикселей. В более современных методах (BPT, VCM, MLT и др.) это не так. Интеграл освещённости рассматривается как многомерная функция, спроецированная на плоскость изображения (рис. 2).

$$F(x,y,r1,r2,...)\xrightarrow{project}F(x,y)$$

Рис. 2. Интеграл освещённости как многомерная функция.

Можно сказать, что при таком подходе, вычисляя функцию *F,* эти алгоритмы вычисляют сразу все интегралы для всех пикселей изображения. Для учёта вклада в изображение алгоритмы, основанные на обыкновенном Монте-Карло (BPT, VCM и аналоги), используют операцию проекции точки на плоскость

изображения, из-за чего несколько потоков могут попасть в один пиксел. Что касается алгоритмов, основанных на Монте-Карло по схеме Марковских цепей (MLT и аналоги), то здесь ситуация аналогична, поскольку помимо проекции (для двунаправленных видов MLT), сами цепи Маркова двигаются в пространстве экрана свободно и неконтролируемо (что легко приводит к коллизиям, когда несколько потоков вносят вклад в один и тот же пиксел).

1.2 Формулировка решаемых проблем

Таким образом, в современных алгоритмах интегрирования освещённости невозможно назначить сэмпл на определённый пиксел заранее: *(проблема №1) любой сэмпл может попасть в любой пиксел изображения*.

При реализации на центральных процессорах сэмплирование изображения не является проблемой, т.к. количество потоков, как правило, небольшое (4-16). Для разрешения конфликтов можно использовать как атомарные операции, так и различные виды блокировок. Однако для GPU эффективность подобных механизмов падает. Кроме того, появляется *(№2) проблема нехватки памяти в больших разрешениях*.

Последнее требует некоторых пояснений. В фотореалистичном рендеринге в целях устранения артефактов ступенчастости часто используют подразбиение пикселей или просто увеличенное в 2-4 раза разрешение. В индустрии кино и архитектуры нормой являются разрешения 4000x2000. При увеличении такого разрешения в 4 раза мы получим 16000x8000, что потребует 2 GB памяти при хранении изображения в HDR формате (т.е. просто плавающая точка ‘float4’ без сжатия). Для трассировки путей это не является проблемой, поскольку (т.к. все пиксели можно интегрировать независимо) изображение может быть разбито на тайлы меньшего размера и хранить всё изображение в памяти GPU не нужно. Однако, для более современных алгоритмов, которые вычисляют всё изображение сразу, данных подход не применим.

2 Предыдущие работы

Для учёта вкалада в изображение при реализации обратной трассировки путей (PT), как уже обсуждалось выше, основной упор делается на возможность явного назначения конкретных потоков на конкретные пикселы. Сюда можно отнести разбиение на тайлы [Frolov et. all 2010], регенерацию путей [Novak et. all 2010; Wald. 2011; Frolov and Galaktionov, 2016] и различные варианты повышения эффективности при помощи уплотнения и сортировки потоков, блочной “русской рулетки” и др. [Antwerpen 2011 (1)]. Данные методы при всех своих преимуществах не позволяют решить проблемы, описанные в разделе 1.2.

В работах [Bogolepov et. all 2013; Antwerpen 2011 (2)] реализован метод усечённой двунаправленной трассировки путей на GPU с использованием технологии CUDA, называемый “IBPT”, и являющийся частным случаем полной двунаправленной трассировки путей (BPT), поскольку он использует ограниченное число стратегий генерации сэмплов (а именно 3 стратегии). Благодаря этому ограничению данный метод может быть реализован на GPU с минимальным объемом *дополнительной памяти* по сравнению с PT. В обеих работах были использованы атомарные операции с плавающей точкой, которые приводят к снижению производительности. Кроме того, в стандарте OpenCL атомарные операции с плавающей точкой отсутствуют. Поэтому необходимо отдельно решать вопрос их реализации или замены [Suhorukov. 2011].

В работе [Davidovič et. all 2014] исследовались различные аспекты повышения эффективности двунаправленных алгоритмов – BPT и VCM. Основное внимание уделено таким вопросам как эффективное построение пространственных хэш-таблиц для сбора фотонов, распределение работы (включая регенерацию путей), компактной (в плане памяти) реализации соединений в BPT и схемам вычисления MIS весов. Однако обозначенные нами проблемы в данной статье не были затронуты.

В работе [Frolov, Galaktionov 2017] для решения описанных проблем при реализации MLT исследовались 2 подхода: (1) имитация атомарных операций при помощи метода [Suhorukov. 2011] и (2) сортировка сэмплов с последующим бинарным поиском. Оба исследуемых подхода показали приблизительно одинаковые временные характеристики. С одной стороны, это является следствием фундаментальной трудоёмкости операции вклада от множества потоков, попадающих в одинаковые пиксели. С другой стороны, это показывает, что атомарные операции действи-

тельно дороги, т.к. используемый в этой работе подход тратит вычислительные ресурсы GPU и обладает сложностью

$O(N_{sam}*(log(N_{sam}))^2 + O(M_{pixels}*log(N_{sam}))$

где N_{sam} – число сэмплов, а M_{pixels} – количество пикселей. Авторы отмечают, что сортировка является условно-бесплатной поскольку отстортированные лучи обрабатываются графическим процессором быстрее. Однако, скользким местом данного подходя является последующий бинарный поиск для M_{pixels} пикселей, который в больших разрешениях становится узким местом. Таким образом, данных подход не может рассматриваться как условно-бесплатный без существенных оговорок.

В работе [Schmidt et. all, 2016] предлагается спекулятивно-рекурсивная схема реализации предложений перехода алгоритма MMLT для повышения когерентности потоков GPU. Для учёта вклада путей в гистограмму изображения (в случае MLT изображение рассматривается как гистограмма функции) использовались атомарные операции.

Рассматривая проблему №1 (любой сэмпл может попасть в любой пиксел), можно отметить, что аналогичная проблема возникает при реализации программной растеризации треугольников (когда любой треугольник может перекрыть любое число пикселей) на GPU [Laine and Karras 2011]. Данный подход сочетает в себе двухуровневое разбиение экрана (корзины, тайлы), 2 различных способа назначения потоков GPU – по треугольникам (на этапе bin raster и coarse raster), и по пикселям (на этапе fine raster), --- и эффективное использование разделяемой памяти GPU для сохранения промежуточных атрибутов треугольников, попавших в некоторый тайл. По сравнению с атомарными операциями данных подход существенно быстрее, но только если треуголник перекрывает большое число пикселей (хотя бы 8-16). Если же треугольники выраждаются в точки, как в случае с Монте-Карло сэмплами, эффективность этапа fine raster будет падать, поскольку лишь один поток из блока потоков (называемых обычно warp) на данном этапе будет выполнять полезную работу.

2.1 Выводы по предыдущим работам

В существующих работах для вычисления вклада используются атомарные операции или аналогичные по производительности (и по сути) методы. Данные методы, как уже было сказано, *снижают производительность* расчёта в целом. Помимо этого, ни в одной существующей работе по-прежнему не решена проблема *нехватки памяти в больших разрешениях* для современных методов интегрирования освещённости (BPT, VCM, MLT и др.). На решении этих двух проблем и сосредоточена данная работа.

3 Предлагаемый подход

В основе предлагаемого подхода лежит несколько положений:

1. Необходимо использовать стохастический процесс сэмплирования изображения в целом. Нельзя сэмплировать отдельные пиксели или области изображения (например, тайлы). Тем не менее, для повышения вероятности сэмплирования отдельных областей можно применять выборку по значимости.
2. Для равномерного покрытия изображения сэмплами при генерации как минимум первых двух координат в функции *F(x,y,r1,r1, ...)* ***строго необходимо*** использовать *квази-случайные* последовательности чисел вместо *псевдо-случайных*. На рис. 13 и 14. показана разница в уровне шума при использовании псевдослучайных и квази-случайных чисел в наших экспериментах. Мы использовали квази-случайную последовательность Соболя [Sobol 1967].
3. Необходимо хранить накапливаемое изображение в обыкновенной оперативной памяти (памяти CPU).
4. Координаты и значения (цвета) сэмплов будут копироваться из памяти GPU в память CPU.
5. Для того чтобы копирование с GPU на CPU не являлось узким местом, необходимо использовать конвейер, когда вычисления следующей порции сэмплов перекрываются по времени с копированием с предыдущей порции (рис. 3, 4).
6. Непосредственно операция вклада в изображение выполняется на CPU.
7. Для сохранения когерентности потоков на GPU при трассировке из камеры сэмплы *необходимо сортировать* по Z кривой [Morton 1966]. Данный пункт существенно влияет на скорость трассировки первичных лучей (прирост на первичных лучах до 250%, что даёт + 10-15% прироста производительности в итоге).

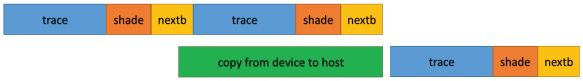

Рис. 3. Предлагаемая схема реализации учёта вклада в изображение в трассировке путей для обратной трассировки путей (PT) и 2 отскоков.

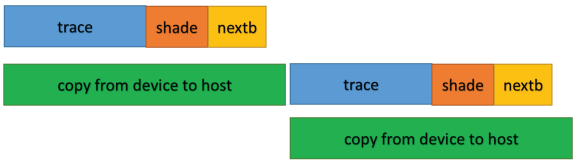

Рис. 4. Предлагаемая схема реализации учёта вклада в изображение для трассировки путей для прямой трассировки путей (LT) и 2 отскоков.

На рис. 3 и 4 изображены схемы конвейера для обратной и прямой трассировки путей для глубины трассировки в 2 отскока. Расчёт одного отскока состоит условно из 3 ядер (в нашей реализации на самом деле из 7, но мы рисуем только 3 для экономии места) – трассировка, шейдинг и вычисление параметров следующего отскока.

Копирование на рис. 3 и 4 изображёно длинным зелёным прямоугольником. В зависимости от оборудования время копирования (и соответственно длина зелёного прямоугольника) могут различаться.

Во время обратной трассировки вклад в изображение делается только после окончания трассировки всех отскоков. Поэтому (особенно при большой глубине) копирование (+ вклад в изображение) может быть практически сколь-угодно долгим, и при этом оно всё равно останется условно-бесплатным. Иная ситуация наблюдается с прямой трассировкой, где копирование перекрывается только вычислениями в течение расчёта единичного отскока лучей.

Поскольку усечённая двунаправленная трассировка путей (IBPT) состоит в сущности из двух проходов – PT и LT, то для неё поочерёдно применяются обе схемы, изображённые на рис. 3 и 4.

4 Детали реализация

Мы реализовали предложенный подход на OpenCL в программном обеспечении с открытым исходным кодом Hydra Renderer. Для асинхронного копирования были использованы 2 очереди команд OpenCL, в одну из которых помещались только команды запуска вычислительных ядер (и копирования необходимых констант на GPU), а вторая очередь использовалась только для асинхронного копирования сэмплов из GPU памяти в CPU память (алгоритм 1).

```
clFlush(cmdQueue1);
clEnqueueCopyBuffer(cmdQueue2,bufGPU,bufCPU);
  data=clEnqueueMapBuffer(cmdQueue2, bufCPU);
  AddSamplesContrib(data, image);
clEnqueueUnmapMemObject(data);
clFinish(cmdQueue1);
clFinish(cmdQueue2);
```

Алгоритм. 1. Псевдокод, демонстрирующий используемую последовательность OpenCL вызовов для реализации конвейера с асинхронным копированием.

Функция *AddSamplesContrib* предполагает, что массив *data* состоит из четвёрок чисел с плавающей точкой, причём в первых трёх числах хранится цвет, а четвёртое число хранят две запакованные координаты в пространстве экрана – x и y. Она в цикле считывает *i-ый* сэмпл и вносит вклад в нужный пиксел изображения. Её упрощённый листинг на языке Си приведён ниже:

```
void AddSamplesContrib(
      float4* out_color,
      float4* in_colors,
      int a_size, int a_width, int a_height)
{
  for (int i = 0; i < a_size; i++)
  {
    float4 color = in_colors[i];

    int packed = as_int(color.w);
    int x      = (packed & 0x0000FFFF);
    int y      = (packed & 0xFFFF0000) >> 16;
    int offset = y * a_width + x;

    out_color[offset].x += color.x;
    out_color[offset].y += color.y;
    out_color[offset].z += color.z;
  }
}
```

Алгоритм. 2. Листинг реализации функции вклада сэмплов в изображение. a_size – число сэмплов во входном буфере in_colors; a_width и a_height – разрешение выходного изображения out_color.

5 Анализ и сравнение

Хотя основное время вклада (изображённое зелёным прямоугольником на рис. 3 и 4) тратится на операцию *clEnqueueCopyBuffer* (рис. 5), время, затрачиваемое функцией *AddSamplesContrib* нельзя полностью игнорировать. В данной функции практически сразу обращает на себя внимание тот факт, что доступ к памяти при сохранении сэмплов в изображение *out_color* хаотичный (т.к. координаты x и

у непредсказуемы), а это означает гарантированный кэш промах. Причём, поскольку размер изображения *out_color* может быть в действительности очень большим (для разрешения 16384x16384 это уже 4GB), то здесь имеет место уже промах в TLB кэш [Wikipedia 2017]. Это легко увидеть, если выделить память для изображения с увеличенным размером страниц, т.к. использование больших страниц даёт выигрыш ровно в 2 раза (рис. 5).

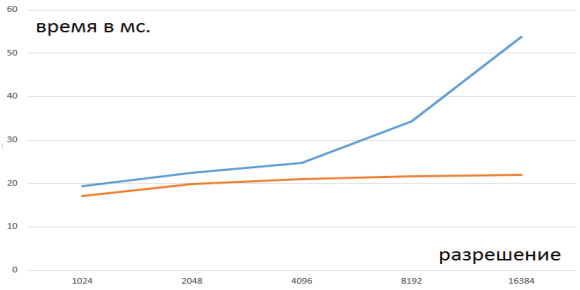

Рис. 5. Время выполнения функции *AddSamplesContrib* для 1 миллиона сэмплов (суммарно за 10 прогонов) в милисекундах для нормального размера страниц (4KB, синий график) и увеличенного (2MB или более в ОС Windows). Цифра 16384 сотвествует разрешению изображения в 16384x16384 пикселей, которое занимает 4 GB памяти. Выигрыш в 2 раза объясняется тем, что при промахе в TLB кэш процессору требуется 2 фактических чтения из памяти вместо одного. При увеличении размера страниц их число резко сокращается и промахи в TLB кэш пропадают.

Анализ времени вклада на простой сцене (Teapot Cornell Box) для различных конфигураций представлен на рис. 6. Из данного рисунка следует что время, требуемое для расчёта одного отскока даже на простой сцене (сцена №1, рис. 11) больше, чем сумма копирования и учёта вклада на CPU. Следовательно, предложенный подход является ***условно бесплатным*** при условии асинхронного выполнения (1) копирования + вклада на CPU и (2) расчёта следующего отскока на GPU. Что касается более тяжёлых сцен (например, сцена №2, рис. 12), то на них время расчёта будет возрастать, а время копирования + вклада не зависит от сложности сцены.

Мы сравнили предложенный подход с подходом из работы [Frolov, Galaktionov 2017] на основе сортировки с последующим сбором, также реализованным в системе Hydra Renderer. Поскольку предложенный подход является условно-бесплатным, мы не сравниваем чистое время выполнения операции вклада, принимая его за ноль. Вместо этого на рис. 7-10 мы демонстрируем увеличение скорости работы рендер-системы в целом на 10-25% за счёт использования предложенного подхода. Интересно отметить, что даже на достаточно тяжёлой в плане трассировки лучей сцене №2, выигрыш от применения предложенного подхода всё еще заметен.

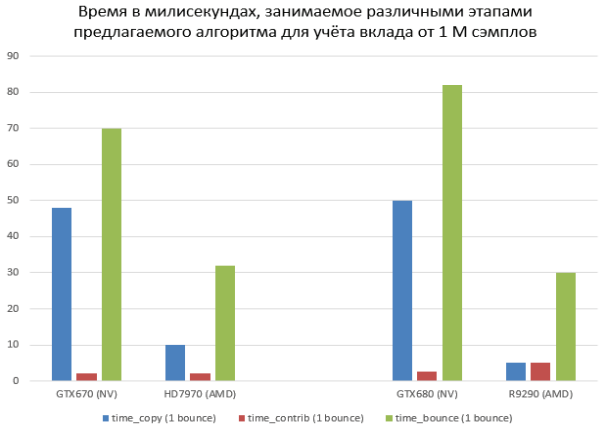

Рис. 6. Диаграмма времени вклада для функции *clEnqueueCopyBuffer* (time_copy, синие столбцы) и *AddSamplesContrib* (time_contrib, красные столбцы). Время расчёта одного сэмпла – зелёные столбцы. В правой части диаграммы отображается 1 машина с 2 видеокартами, в левой – вторая машина также с двумя видеокартами. Тестирование проводилось на сцене №1 (рис. 11).

Что касается затрат памяти, то требуется выделить лишь порядка 8-16MB памяти для вспомогательного буфера на GPU (в зависимости от числа потоков). Само изображение хранится в памяти CPU и ограничено только количеством бит для запакованных координат и объёмом оперативной памяти, доступной на CPU. Т.к. мы используем 16 бит на каждую координату, максимальное разрешение изображения – 65535x65535, что более чем достаточно для подавляющего большинства задач.

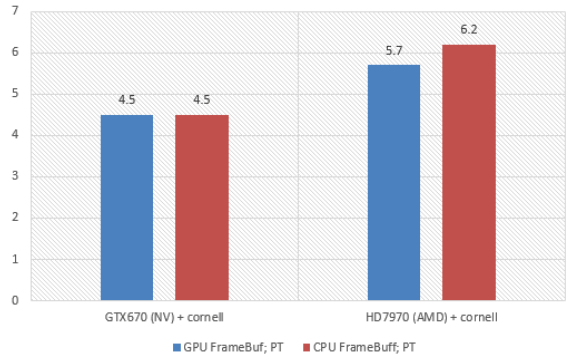

Рис. 7. Производительность трассировки путей (**PT**) в миллионах сэмплов в секунду для подхода из работы [Frolov, Galaktionov 2017] (синий столбец, GPU FrameBuf) и предложенного подхода (CPU frameBuff). На сцене №1.

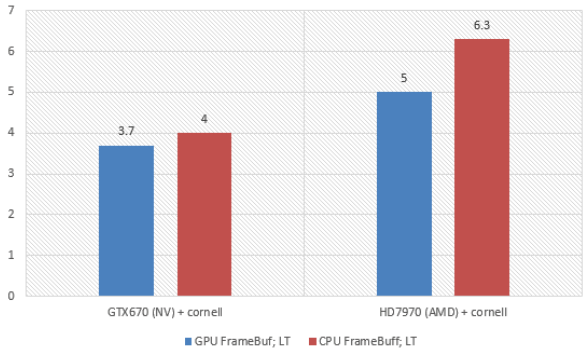

Рис. 8. Производительность прямой трассировки путей (**LT**) в миллионах сэмплов в секунду для подхода из работы [Frolov, Galaktionov 2017] (синий столбец, GPU FrameBuf) и предложенного подхода (CPU frameBuff). На сцене №1.

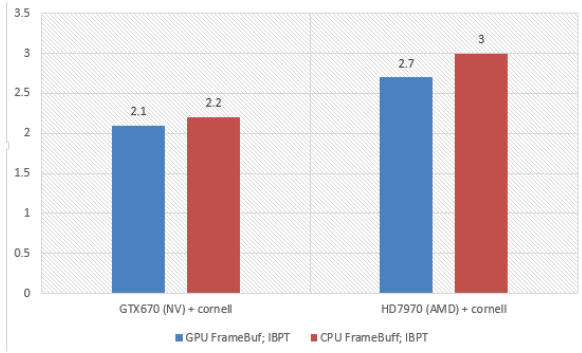

Рис. 9. Производительность усечённой двунаправленной трассировки путей (**IBPT**) в миллионах сэмплов в секунду для подхода из работы [Frolov, Galaktionov 2017] (синий столбец, GPU FrameBuf) и предложенного подхода (CPU frameBuff). На сцене №1.

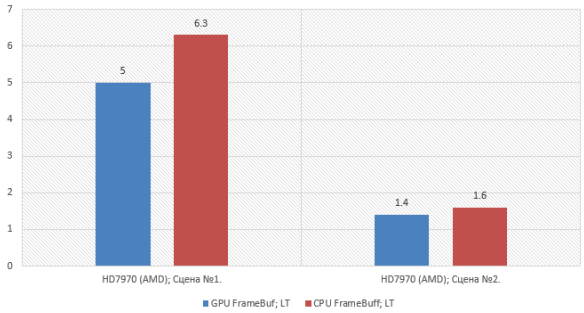

Рис. 10. Сравнение выигрыша в производительности прямой трассировки для “лёгкой” сцены №1 (Рис.11, слева, 25%) и “тяжёлой” сцены №2 (Рис.12, справа, 14%).

Благодарности

Работа поддержана грантом РФФИ №16-31-60048 мол_а_дк и частично поддержана грантом РФФИ №16-01-00552.

Список литературы

James T. Kajiya. 1986. *The rendering equation.* In Proceedings of the 13th annual conference on Computer graphics and interactive techniques (SIGGRAPH '86), David C. Evans and Russell J. Athay (Eds.). ACM, New York, NY, USA, 143-150. DOI=http://dx.doi.org/10.1145/15922.15902

Iliyan Georgiev, Jaroslav Křivánek, Tomáš Davidoič, and Philipp Slusallek. 2012. *Light transport sim lation with vertex connection and merging. ACM Trans. Graph.* 31, 6, Article 192 (November 2012), 10 pages. New York, USA.

Eric Veach. 1998. *Robust Monte Carlo Methods for Light Transport Simulation*. Ph.D. Dissertation. Stanford University, Stanford, CA, USA. Advisor (s) Leonidas J. Guibas. AAI9837162.

Eric Veach and Leonidas J. Guibas. 1997. Metropolis Light Transport. *SIGGRAPH 97 Proceedings* (August 1997), Addison-Wesley, pp. 65-76.

Csaba Kelemen, László Szirmay-Kalos, György Antal and Ferenc Csonka. 2002. *Simple and Robust Mutation Strategy for the Metropolis Light Transport Algorithm.* EUROGRAPHICS 2002 / G. Drettakis and H.-P. Seidel. 2002, v.21, № 3.

Toshiya Hachisuka, Anton S. Kaplanyan, and Carsten Dachsbacher. 2014. Multiplexed metropolis light transport. *ACM Trans. Graph.* 33, 4, Article 100 (July 2014), 10 pages.

Frolov V., Kharlamov A., Ignatenko A. 2010. *Biased Global Illumination via Irradiance Caching and Adaptive Path Tracing on GPUs.* Proceedings of GraphiCon’2010 international conference on computer graphics and vision. St. Petersburg, 2010, pp. 49–56.

Novak, J., Havran, V., AND Daschbacher, C. 2010. *Path regeneration for interactive path tracing.* In EUROGRAPHICS 2010, short papers, pp.61–64.

Ingo Wald. 2011. Active thread compaction for GPU path tracing. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (HPG '11),Stephen N. Spencer (Ed.). ACM, New York, NY, USA, 51-58.

Frolov V.A., Galaktionov V.A. 2016. *Low overhead path regeneration.* Programming and Computer Software, 2016, Vol. 42, №. 6, pp. 382–387. DOI: 10.1134/S0361768816060025

Antwerpen D. 2011. (1) *Unbiased physically based rendering on the GPU*. M.S. thesis, Delft University of Technology, the Netherlands.

Antwerpen D. 2011. (2) *Improving SIMD Efficiency for Parallel Monte Carlo Light Transport on the GPU.* HPG ’11 Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics.

Bogolepov D.K., Sopin D., D. Uljanov. 2013. Turlapov V.E. *GPU-Optimized Bi-Directional Path Tracing.* WSCG 2013 proceedings. Plzen. Czech Republic.

Igor Suhorukov. 2011. OpenCL 1.1: *Atomic operations on floating point values.* WEB publication. URL=http://suhorukov.blogspot.ru/2011/12/opencl-11-atomic-operations-on-floating.html

Tomáš Davidovič, Jaroslav Křivánek, Miloš Hašan, and Philipp Slusallek. 2014. Progressive Light Transport Simulation on the GPU: Survey and Improvements. *ACM Trans. Graph.* 33, 3, Article 29 (June 2014), 19 pages. DOI: https://doi.org/10.1145/2602144

Frolov, V.A. & Galaktionov, V.A. 2017. *Memory compact Metropolis Light Transport.* Program Comput Soft (2017) 43: 196. https://doi.org/10.1134/S0361768817030057

M. Schmidt., Oleg Lobachev., Michael Guthe. 2016. *Coherent metropolis light transport on the GPU using speculative mutations.* WSCG, 2016, vol. 24, no. 1, pp. 1–8.

Samuli Laine, Tero Karras. 2011. High-performance software rasterization on GPUs. High-Performance Graphics 2011, August 2011.

Wikipedia. *Translation lookaside buffer.* 2017. URL =https://en.wikipedia.org/wiki/Translation_lookaside_buffer

Sobol, I. M. 1967. *Distribution of points in a cube and approximate evaluation of integrals*. *Zh. Vych. Mat. Mat. Fiz.* **7**: 784–802 (in Russian); *U.S.S.R Comput. Maths. Math. Phys.* **7**: 86–112 (in English).

G. M. Morton. 1966. *A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing.* In Research Report, IBM Ltd, ON, Canada