ПРОГРАММИРОВАНИЕ, 2016, № 4, с. 3–9

COMPUTER GRAPHICS

УДК 004.421.6

Регенерация путей с низкими накладными расходами

© 2016 г. В.А. Фролов [1,2], В.А. Галактионов [1],

[1] *Институт прикладной математики имени М.В. Келдыша РАН. Москва, Россия.*

[2] *Московский государственный университет. Москва, Россия.*

E-mail: vfrolov@graphics.cs.msu.ru, vlgal@gin.keldysh.ru

Поступила в редакцию 15.01.2016

Монте-Карло трассировка путей является центральным алгоритмом расчёта освещенности, вокруг которого строятся более современные методы (такие как BDPT, MLT, VCM и другие). Одна из основных проблем, стоящих на пути к реализации эффективной трассировки путей на GPU - малая загрузка GPU вычислениями вследствии сильно различной глубины трассировки: небольшое число потоков трассируют пути на большой глубине, в то время как остальные потоки простаивают. Обычно для решения этой проблемы используется техника, называемая регенерацией путей. Мы предлагаем новый подход к реализации регенерации путей, названный нами “блочной регенерацией по месту”. По сравнению с предыдущими подходами наш алгоритм обладает более низкой стоимостью и не перемещает данные лучей в памяти, за счёт чего упрощается его интеграция в более сложные методы. Мы протестировали наш алгоритм для Монте-Карло трассировки путей используя CUDA и OpenCL

1. Введение в проблему

Монте-Карло трассировка путей генерирует выборки путём создания случайного пути от камеры к источнику. Каждый путь начинается в виртуальной камере, трассируется в сцену и случайно отражается некоторое число раз пока не встретит источник или не улетит в окружающую панорамму. Другие методы интегрирования освещённости могут начинать трассировку на источнике, либо одновременно трассировать пути как от камеры, так и от источника (IBPT [2], BDPT [3], MMLT [4]).

Из-за стохастической природы процесса все современные алгоритмы интегрирования освещённости содержат одну проблему при реализации на GPU: при большой глубине переотражений (отскоков, bounce) лишь небольшое число потоков остаются активными; при этом GPU вынужден будет выполнять почти все потоки, - а именно, - все потоки каждой SIMD группы (такие группы называются термином warp), если активен хотя бы один из её потоков. Более того, даже полностью неактивные SIMD группы warp могут занимать ресурсы мультипроцессора, производя чтение из DRAM флага, отмечающего активность данного потока (см. архитектуру “многих ядер” далее). Это может происходить также из-за неэффективной стратегии распределения работы, реализованной на уровне CUDA или OpenCL драйвера: когда группа потоков (обычно содержащая несколько SIMD групп, называемая в OpenCL рабочей группой) продолжает оставаться на мультипроцессоре до тех пор, пока все его SIMD-группы не закончат работу. Полностью неактивные SIMD группы в этом случае занимают ресурсы GPU бесцельно. Решение последней проблемы возможно при помощи техники называемой “persistent threads” [1]. Однако данную технику труднее применять в ситуациях, когда потоки используют разделяемую память (это нужно во многих случаях - уплотнение потоков, русская рулетка для групп, эффективный “append” фотонов во внешнюю память и другие).

Архитектуры “одного” и “многих” ядер

При реализации Монте-Карло трассировки путей на GPU существуют две основные архитектуры - “одного ядра” и “многих ядер”. Наибо-

лее простая архитектура “одного ядра” реализует всю трассировку путей практически также, как это делается на центральном процессоре - в одной большой функции. Такая реализация неэффективна из-за ограниченного количества регистров GPU и периодической необходимости сохранения локальных переменных в DRAM [5]. Чтобы использовать регистры более эффективно, архитектура “убер-ядра” изменяет архитектуру “одного ядра” таким образом, что большой и сложный код преобразуется в конечный автомат (finite state mashine), где каждое отдельное состояние более простое. Этот подход используется в NVIDIA OptiX [6], и он позволяет использовать одни и те же регистры при выполнении кода для разных состояний.

Архитектура “многих ядер” перемещает эти состояния в отдельные ядра, что ведёт к существенному снижению числа используемых регистров каждым отдельным ядром [5]. Однако, архитектура “многих ядер” вынуждает сохранять промежуточные данные на каждый луч (передаваемые между ядрами) в DRAM. За исключением простых сцен это не влияет на производительность, однако, может значительно уменьшить гибкость в реализации алгоритмов интегрирования освещённости. Например, рекурсию достаточно легко реализовать при помощи стэка в архитектуре “одного ядра”, но это чрезвычайно сложно сделать для “многих ядер”. Так как в архитектуре “многих ядер” место хранения в буфере промежуточных данных для каждого луча обычно жёстко привязано к индексу потока, другие примеры, где архитектура “многих ядер” значительно усложняет жизнь - это уплотнение, сортировка лучей и регенерация путей. В течение нашего исследования мы дополнительно обнаружили, что некоторые компиляторы OpenCL (HD 5770, CPU от Intel-а и AMD) были не способны скомпилировать тяжелые ядра. Таким образом, архитектура “многих ядер” помимо повышения эффективности служит естественным способом уменьшения сложности кода и страхует “свежие” OpenCL компиляторы от непредвиденных падений.

Накладные расходы регенерации путей Можно подытожить, что обе архитектуры имеют свои преимущества и недостатки. Поскольку мы хотели иметь стабильную мультиплатформенную реализацию, мы предпочли использовать архитектуру “многих ядер”. Далее мы обнаружили, что с этой архитектурой существующие подходы к регенерации путей в среднем замедляли трассировку путей: в то время как на некоторых сценах регенерация давала незначительный выигрыш, на других сценах скорость, наоборот, упала (рис. 2-5).

Таким образом, нашей основной целью была разработка алгоритма с низкими накладными расходами, который мы могли бы использовать на всех сценах, не опасаясь потерять в производительности. Следующими целями были гибкость и простотота. Имея планы на реализацию более сложных алгоритмов интегрирования освещённости в будущем, мы бы хотели сохранить код трассировки путей простым и гибким. Существующие подходы к регенерации путей не отличаются этими качествами.

2. Существующие методы

Регенерация путей Идея регенерации путей впервые была представлена в работе [7]. В этой работе завершившиеся потоки создавали в том же месте памяти новые пути из того же самого пиксела. Новые пути для пиксела почти всегда можно сгенерировать, - например для антиалиасинга, реализации эффекта глубины резкости (DOF) или просто делая больше Монте-Карло сэмплов (выборок) за 1 проход. Основной недостаток данного подхода заключается в том, что он значительно увеличивает число расхождений потоков в SIMD группах на ветвлениях, поскольку когерентные первичные лучи (и даже частично-когерентные вторичные) смешиваются с некорегентными третичными, четвертичными и.т.д. В результате скорость трассировки первичных лучей может упасть в 2-3 раза, что сведёт на нет весь выигрыш от регенерации.

Подход, называемый “уплотнением потоков” [8], перемещает все активные лучи в начало массива потоков и заполняет неактивные области в конце новыми потоками. Когерентные первичные лучи всегда попадают в массив потоков последовательно и, таким образом, решается проблема с расхождениями, присутствующая в алгоритме из работы [7]. Однако “уплотнение по-

токов" обладает двумя основными недостатками - наличием накладных расходов и повышенной сложностью реализации:

1.В то время как стоимость алгоритма регенерации из работы [7] близка к нулю, "уплотнение потоков" из [8] может стоить 10-20% от производительности трассировки лучей. Таким образом, типичный выигрыш в 20-30%, свойственный для регенерации, в значительной степени нивелируется ценой алгоритма уплотнения. Основная причина высоких накладных расходов - необходимость перемещать промежуточные данные в DRAM либо использовать индексы. В архитектуре "многих ядер" почти все локальные данные лучей (такие как позиция и направление луча, нормаль к поверхности, текстурные координаты, накопленный цвет и другие) хранятся во внешней памяти, причём для каждого луча место хранения в буфере жёстко привязано к индексу потока. Например, мы можем прочитать текущий накопленный цвет по аналогии с (a), см. ниже. "Уплотнение потоков" изменяет фактические индексы потоков и, таким образом, мы вынуждены либо перемещать данные, либо использовать обращения к ним по аналогии с (b). Такие непрямые чтения нарушают механизм объединения запросов в память (coalescing) GPU, что негативно влияет на производительность.

(a) $color = in\_color[threadId]$;
(b) $color = in\_color[original\_threadId[threadId]]$;

2. Попытки использовать индексы только для некоторых атрибутов (многие из локальных данных живут в течении расчёта одного переотражения, а при следующем отскоке луча рассчитываются заново) вынуждают нас запоминать, какие данные подвержены влиянию уплотнения, а какие нет. Каждый раз при обращении в память мы должны помнить какой вид чтения использовать - (a) или (b).

3. Более сложные алгоритмы интегрирования (такие как BDPT и VCM) могут ещё больше усложнить реализацию уплотнения и замедлить выполнение ядер. Необходимость (присутствующая в них) хранить данные для всех отскоков трансформирует простые индексы в списки, т.к. каждый поток может поменять свой фактический индекс несколько раз.

Чтобы избежать описанных выше сложностей и для того чтобы обращения в глобальную память были эффективны (т.е. без индексов, с работающим объединением запросов в память), в работе [9] была использована архитектура "одного ядра". Ранее мы обсудили преимущества и недостатки архитектуры "одного ядра" по отношению к архитектуре "многих ядер". Кроме того, регенерация из работы [9] даёт приблизительно такой же выигрыш как и "уплотнение потоков" из работы [8].

В работе [10] делается вывод, что мёртвые потоки не влияют значительно на производительность, поскольку оставшиеся потоки выполняются быстрее (чем заполненный пул потоков и регенерация с её накладными расходами). Этот результат в некоторой степени расходится с работами [7, 8, 9], но он согласуется с нашими экспериментами в том, что существующие подходы к регенерации путей имеют слишком высокие накладные расходы. Следует дополнительно отметить, что может быть как минимум две причины, по которым регенерация путей не дала выигрыша в работе [10]: это простые материалы и низкая глубина трассировки (максимум 8).

В работе [11] было отмечено, что простые материалы могут быть причиной низкого выигрыша регенерации. Далее был представлен алгоритм "wavefront path tracing", ориентированный на то, чтобы эффективно вычислять сложные BRDF с архитектурой "многих ядер". Однако в итоге в этой работе также отмечается лишь незначительный выигрыш от регенерации.

Истоки предложенного метода Прежде чем рассмотреть предложенный алгоритм, необходимо упомянуть техники, не связанные напрямую с регенерацией, но связанные с повышением эффективности реализации трассировки путей на GPU. Это техника тайлового (или блочного) распределения работы [5] и техника "русской рулетки на warp" [12].

Тайловое распределение работы разбивает экран на тайлы (блоки) по 16x16 пикселей и использует их как атомарную единицу для распределителя работы. Таким образом, тайлы уменьшают собственную стоимость алгоритма распределения работы в 256 раз.

Русская рулетка на warp впервые была предложена в работе [12]. Оригинальная русская ру-

летка случайно терминирует пути, ограничивая их глубину таким образом, чтобы на большую глубину трассировалось меньше путей, но эти пути брались бы с большим весом. Независимая для каждого потока русская рулетка будет ещё больше увеличивать количество "разреженных" групп warp. Русская рулетка на warp принимает решение о терминировании сразу для всей SIMD группы warp, немного изменяя схему вычисления весов.

Кобинируя тайлы из работы [5] и русскую рулетку на warp из работы [12], мы разработали новый алгоритм регенерации путей, который так же прост в использовании как подход из работы[7], и при этом обладает эффективностью работы [8] в смысле выигрыша в скорости трассировки от регенерации, но в отличие от неё обладает низкими накладными расходами.

3. Предложенный метод

Разработанный алгоритм опирается на предположение, что для того чтобы достигнуть высокого параметра фактической занятости GPU нам не обязательно иметь все потоки активными. Если в массиве потоков имеются непрерывные неактивные области (длиной в размер рабочей группы OpenCL), занимающие от 25% до 50% его размера, это не влияет существенно на производительность, поскольку аппаратный менеджер потоков с этим хорошо справляется. Далее наш алгоритм объединяет в себе несколько идей:

1. Аналогично работе [5] мы разбиваем экран на множество тайлов по 16x16 пикселей. Это разделяет пиксели и потоки. Каждый тайл соотносится с непрерывной областью в массиве потоков (и, как следствие, во всех массивах данных) размером в 256 потоков.

2. По аналогии с [7] мы производили регнерацию "по месту", назначая новые тайлы на место умерших. В отличие от [7] мы не регенерируем по 1 пикселу, а регенерируем сразу весь тайл, т.е. 256 непрерывных потоков. Каждая регенерировавшая рабочая группа (256 потоков) забирает тайл из нового, ещё не обработанного места на экране.

3. Чтобы терминировать некоторой тайл (соответствующий одной рабочей группе) целиком, мы расширили подход "русской рулетки на warp" до "русской рулетки на рабочую группу". Таким образом, мы принимаем решение о терминировании для тайла целиком.

4. Во время трассировки путей мы сохраняли для каждого тайла его максимальную глубину (максимальнео число отскоков среди всех путей тайла) и на следующем шаге сортировали все тайлы по этой глубине. Таким образом, тайлы с большим числом отскоков всегда будут первыми приходить в массив потоков и последними из него уходить. А тайлы с небольшим числом отскоков будут приходить в массив потоков в конце и быстро уходить из него. Эта стратегия уменьшает суммарное число запусков ядер на 30 - 60% в зависимости от сцены и максимальной глубины трассировки.

5. Мы не выполняем регенерацию при каждом отскоке. Для каждого второго отскока мы проверяем число тайлов, которые собираются регенерировать. Если это число больше некоторого порога (половины активных тайлов), мы запускаем регенерацию. Назовём эту идею регенерацией по порогу.

6. Исключительно для ядер, выполняющих трассировку лучей, мы производили локальное уплотнение потоков (внутри одной рабочей группы) при помощи разделяемой памяти. Поскольку ядра, выполняющие трассировку лучей, читают и записывают небольшой объем локальных данных, нарушение объединения запросов в память (coalescing) в этом случае не вызывает потери производительности. Однако, поскольку локальное уплотнение снижает количество дивергентных групп warp внутри рабочей группы до 1, производительность трассировки лучей выросла на 8-15%. Следует подчеркнуть, что локальное уплотнение не ведёт к глобальному перемещению данных лучей или использованию индексов, поскольку индексы потоков меняются временно только внутри одного ядра на период

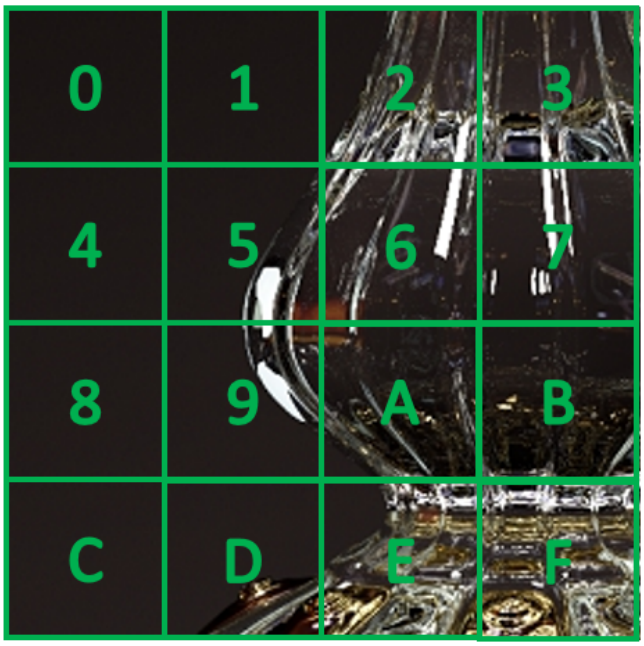

Рис. 1.: Пример изображения, разделённого на 16 тайлов. Каждый тайл пронумерован от 0 до F. Мы сортировали тайлы, основываясь на их максимальной глубине. Тайлы E и F имеют наибольшую глубину. Они приходят в массив потоков первыми. В противовес им тайлы 0,1,4,8,C имеют нулевую глубину и они приходят в массив потоков последними. Таким образом, мы уменьшаем суммарное число вызовов ядер.

его выполнения. После окончания выполнения трассировки результаты записываются на старые места.

Таким образом, мы запускаем регенерацию, когда достаточно много тайлов этого требуют (рис. 1). Регенерация по порогу дополнительно уменьшает стоимость собственно процесса регенерации и увеличивает фактическую занятость GPU во время выполнения ядра, реализующего собственно алгоритм регенерации, поскольку мы знаем, что если это ядро запущено, достаточно большое число его потоков действительно выполнят полезную работу и регенерируют достаточно много новых тайлов. Этим мы предотвращаем ситуацию, при которой ядро осуществляющее регенерацию, постоянно запускается, но почти никогда не выполняет полезной работы.

```
F E A B 6 7 9 5 D 3 2 1 0 4 8 C
```

```
X X X X X X X X | F E A B 6 7 ...
F E A B 6 7 9 5 | D 3 2 1 8 C
F E A B 6 7 9 5 | D 3 2 1 8 C
F E A B 6 7 9 5 | D 3 2 1 8 C (9,5)->(D,3)
F E A B 6 7 D 3 | 2 1 0 4 ..(A,6,7)->(2,1,0)
F E 2 B 1 0 D 3 | 4 8 C
F E 2 B 1 0 D 2 | 4 8 C       (1,0)->(4,8)
F E 2 B 4 8 D 2 | C           (4,8)->(C,X)
F E 2 B C X D 2 |               (C)->(X)
F E 2 B X X D 2 |             (D,2)->(X,X)
F E 2 B X X X X |
...
X X X X X X X X |
```

Листинг 1. Упрощенный пример, иллюстрирующий работу предложенного метода по шагам (без регенерации по порогу). Массив потоков рассчитан на 8 тайлов (8x256 лучей или потоков). Суммарное число тайлов - 16. Левая часть представляет активные тайлы в массиве потоков. Правая часть после верикального слэша - очередь ожидающих тайлов. Символ “X” представляет мёртвые тайлы. Нотация “(9,5)->(D,3)” означает, что тайлы “9” и “5” умерли и новые тайлы были регенерированы, замещая “9” на “D” и “5” на “3”.

Детали реализации Мы использовали один и тот же набор ядер для CUDA и OpenCL. Наш конвейер трассировки путей состоит в общей сложности из 7 ядер: regenerate, trace, surfEval, lightSample, shadowTrace, shade, nextBounce. Столь мелкое разбиение потребовалось из-за большого набора функциональностей, поддерживаемых трассировщиком путей. Слияние ядер в нашем случае приводило к заметному снижению производительности и непредвиденным вылетам OpenCL компилятора на некоторых драйверах.

Почти все наши ядра достаточно тяжёлые. Ядро surfEval реализует алгоритм Parallax Occlusion Mapping для поверхностей с рельефом, ядро lightSample поддерживает HDR панорамы и источники со сложным световым распределением (гониограммы); ядра Shade и nextBounce работают со сложным представлением материала в виде дерева, где листовые узлы представляют BRDF, а нелистовые - способ смешивания. Это похоже на “Mila material” в рендер-системе Mental Ray [13]. Таким образом, в нашем случае не только трассировка лучей выигрывала от регенерации, но и другие ядра.

4. Результаты и обсуждение

Тестовые сцены Наши тестовые сценарии сфокусированы на четырёх различных случаях, в которых регенерация путей проявляет себя по разному (рис. A-D). Тест # 1 имеет очень простую геометрию, но в нём возможно бесконечное число переотражений лучей. Этот тест идеально подходит для того чтобы оценить эффективность и накладные расходы различных алгоритмов регенерации, не принимая в расчёт скорость трассировки лучей. Тест # 2 прост во всех смыслах и в нём от регенерации путей не должно быть никакого выигрыша. Этот тест позволяет нам оценить минимальные накладные расходы, которые будут присутствовать в любом случае. Тесты # 3 и # 4 представляют типичные случаи из реальной жизни, когда регенерация путей может потребоваться из-за большой максимальной глубины трассировки.

Сравнение с существующими методами Мы сравнили наш алгоритм с уплотнением потоков из работы [8] (обозначается "sm" на рис. 2-5) и с трассировкой путей без регенерации. Для всех трёх случаев мы использовали русскую рулетку на тайл (соответствующий одной рабочей группе). Мы использовали подход с индексами при реализации регенерации при помощи уплотнения потоков. Для него мы регенерировали каждый 4-ый или 8-ой отскок (для больших значений максимальной глубины 8-ой, для меньших 4-ый) чтобы получить примерно то же число вызовов ядра регенерации как и в предложенном нами методе. В противном случае (регенерация на каждом отскоке) регенерация при помощи уплотнения потоков ни на одной сцене не давала ускорения. Для реализации уплотнения потоков мы использовали "thrust::copy_if" в CUDA и собственную реализацию уплотнения в OpenCL на основе примера вычисления префиксной суммы в NVSDK.

По сравнению с [8] наш алгоритм даёт такой же чистый выигрыш в производительности (тесты 3, 4) от регенерации, но он обладает меньшими накладными расходами (тесты 1, 3, 4). Основные причины для этого - пункты (1-2), (4), (5). Рабочие группы/тайлы (1-2) наряду с регенераций по порогу (5) снижают стоимость самой регенерации. Сортировка тайлов (4) снижает суммарно число запусков ядер. Тайлы также позволяют выполнять регенерацию по "месту", и не перемещать данные лучей в памяти. Попав в некоторое фиксированное место в массиве потоков, тайл остаётся в нём до самого конца, пока все его потоки не завершат трассировку или тайл целиком не будет терминирован при помощи русской рулетки.

5. Выводы

Наши эксперименты показали, что регенерация даёт существенный выигрыш только для больших значений глубины трассировки (больше или равно 20). Для низких значений глубины трассировки наши результаты в целом согласуются с работой [10].

Таким образом, мы полагаем что проблема расхождений на ветвлениях (решаемая в [7], [8] при помощи регенерации) может быть эффективно решена при помощи русской рулетки на warp или на тайл. Регенерация путей требуется только когда фактическое число активных групп warp становится значительно меньше числа групп warp, которые GPU способен обрабатывать одновременно.

6. Благодарности

Работа поддержана грантами РФФИ 16-31-60048 "мол_а_дк", 16-01-00552.

СПИСОК ЛИТЕРАТУРЫ

1. *K. Gupta, J. A. Stuart, J. D. Owens.* A Study of Persistent Threads Style GPU Programming for GPGPU Workloads // In proceedings of Innovative Parallel Computing. May 2012. San Jose, CA.

2. *Bogolepov,D., Ulyanov,D., Sopin,D., Turlapov,V.* GPU-Optimized Bi-Directional Path Tracing. // Proc. WSCG 2013, 24-28 June 2013, Plzen Czech Republic. PP. 57-60. ISBN: 978-80-86943-76-3

3. *Eric P. Lafortune., Yves D. Willams* Bidirectiona Lapth Tracing // Proceedings of third internetional conference on computational graphics and vizualization techniques. 1993.

4. *Toshiya Hachisuka, Anton S. Kaplanyan, Carsten Dachsbacher* Multiplexed Metropolis Light Transport. // ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2014. url = http://www.ci.i.u-tokyo.ac.jp/ hachisuka/mmlt.pdf

5. *Frolov V., Kharlamov A., Ignatenko A.* 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.

6. *S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A, Robison, and M. Stich.* OptiX: a general purpose ray tracing engine // In ACM SIGGRAPH 2010 papers: ACM, NY, Article 66, 13 pp.

7. *Nocak, J., Havran, V., and Daschbacher* Path regeneration for interactive path tracing // In EUROGRAPHICS 2010, short papers, pp.61–64.

8. *Van Antwerpen.* Unbiased physically based rendering on the GPU // M.S. thesis, Delft University of Technology, the Netherlands, 2011.

9. *Tomas Davidovic, Jaroslav Krivanek, Milos Hasan, and Philipp Slusallek.* Progressive Light Transport Simulation on the GPU: Survey and Improvements // ACM Trans. Graph. 33, 3, Article 29 (June 2014), 19 pages.

10. *Wald, I.* Active thread compaction for GPU path tracing // High Performance Graphics, 2011, pp. 51–58.

11. *Laine, S., Karras, T., and Aila, T.* Megakernels considered harmful: Wavefront path tracing on GPUs // Proc of High-Performance Graphics 2013, pp. 137–143.

12. *Novak J.* Global Illumination Methods on GPU with CUDA // MS Thesis: Ph. D. thesis.: Czech Technical University, Prague, 2009.

13. *NVIDIA* Mental Ray // 2014. url = http://www.nvidia.com/object/nvidia-mental-ray.html

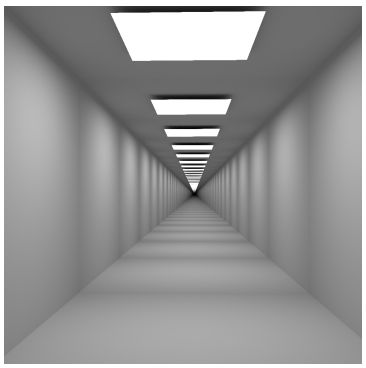

Рисунок A. Тест 1.

Рисунок B. Тест 2.

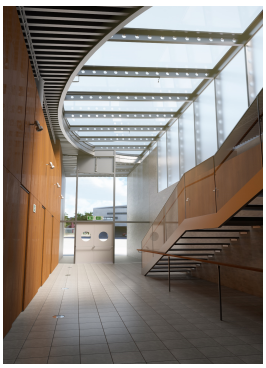

Рисунок C. Тест 3.

Рисунок D. Тест 4.

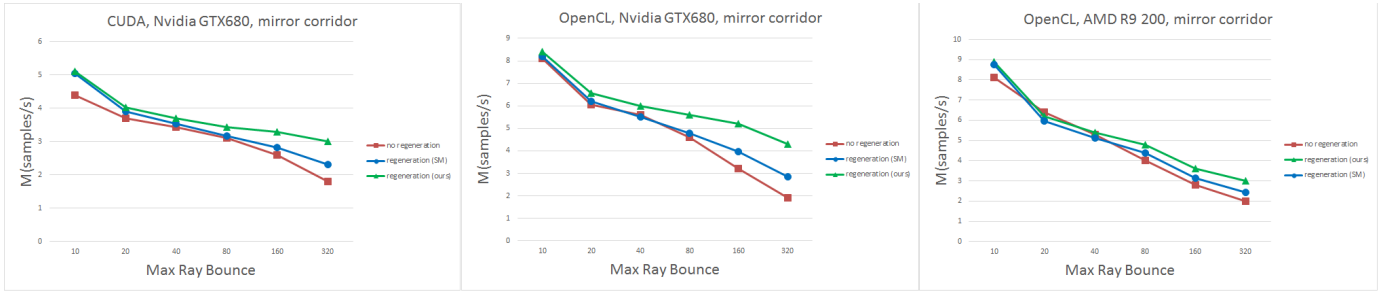

Рисунок 2. Тест 1, зеркальный коридор; сэмплов в секунду (соотвст. рис. A).

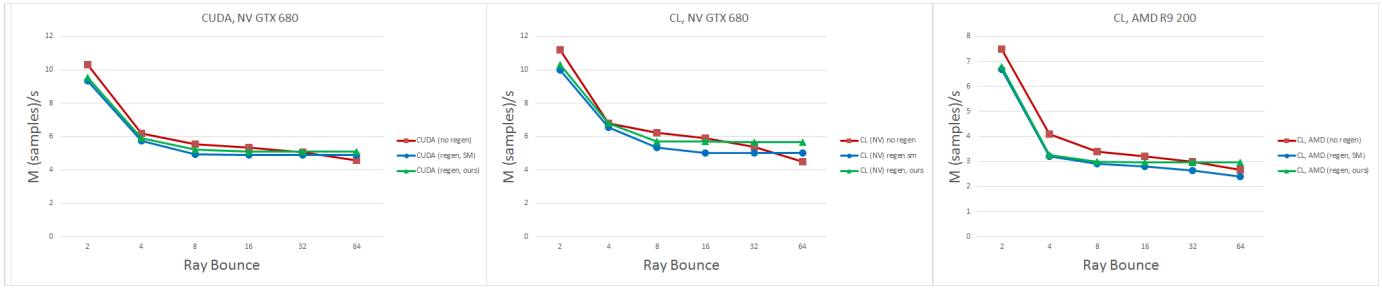

Рисунок 3. Тест 2, cornel box; сэмплов в секунду (соотвст. рис. B).

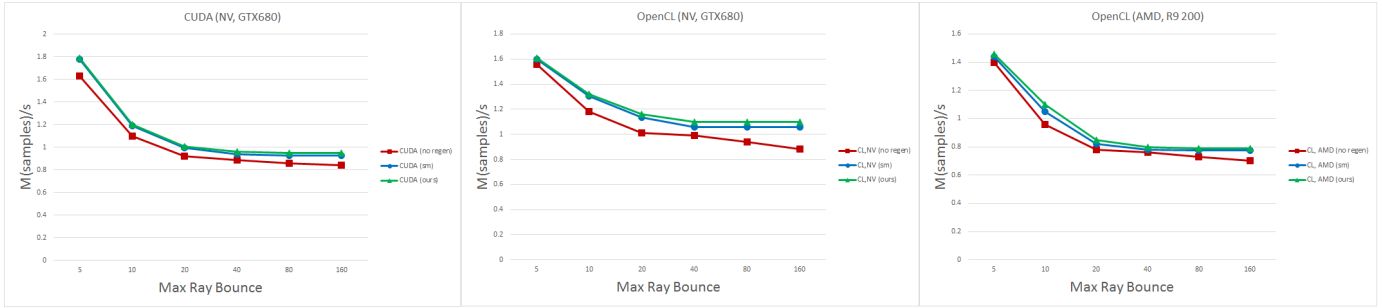

Рисунок 4. Тест 3, коридор; сэмплов в секунду (соотвст. рис. C).

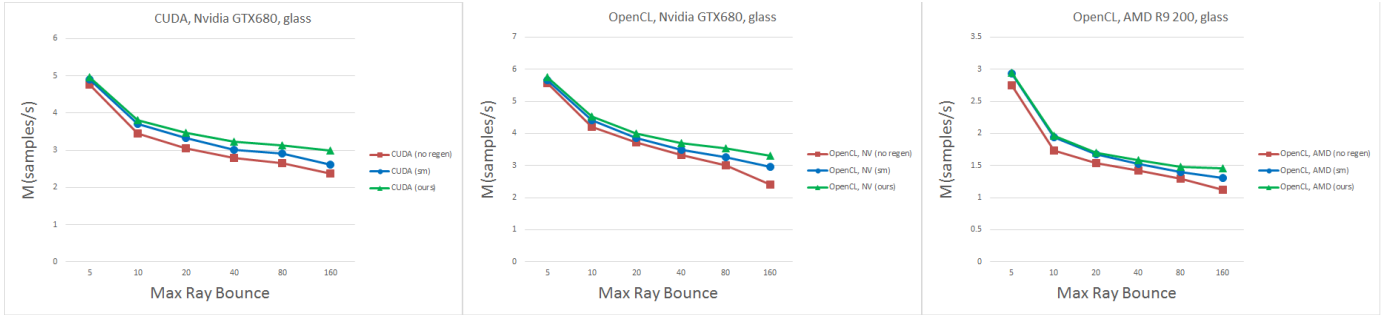

Рисунок 5. Тест 4, glass; сэмплов в секунду (соотвст. рис. D).