ПРОГРАММИРОВАНИЕ, 2020, № 4, с. 65–72

КОМПЬЮТЕРНАЯ ГРАФИКА И ВИЗУАЛИЗАЦИЯ

УДК 004.421.6

ИССЛЕДОВАНИЕ ТЕХНОЛОГИИ Nvidia RTX

© 2020 г. В. В. Санжаров[a,*], В. А. Фролов[b,c,], В. А. Галактионов[b,***]**

[a] *Российский государственный университет нефти и газа (Национальный исследовательский университет) имени И.М. Губкина 119296 Москва, Ленинский пр., д. 65, Россия*

[b] *Институт прикладной математики имени М.В. Келдыша РАН 125047 Москва, Миусская пл., д. 4, Россия*

[c] *Московский государственный университет имени М.В. Ломоносова 119899 Москва, Ленинские горы, д. 1, стр. 8, Россия*

**E-mail: vs@asugubkin.ru*

***E-mail: vfrolov@graphics.cs.msu.ru*

****E-mail: vlgal@gin.keldysh.ru*

Поступила в редакцию 25.12.2019 г.
После доработки 09.01.2020 г.
Принята к публикации 13.01.2020 г.

Nvidia RTX – это закрытая аппаратно-ускоренная технология трассировки лучей от компании Nvidia. Поскольку детали реализации неизвестны, в сообществе разработчиков было много вопросов о том, что на самом деле представляет из себя аппаратная реализация: какие именно части в конвейере трассировки лучей ускорены аппаратно, а что может быть эффективно реализовано программно. В этой статье мы представляем результаты наших экспериментов с RTX, направленные на понимание внутренней работы этой технологии. В нашей работе мы постарались ответить на вопросы, волнующие разработчиков по всему миру: какое ускорение можно на практике получить по сравнению с программной реализацией и в чем его технологическая основа? Насколько трудоемко будет разрабатывать рендер-систему с поддержкой аппаратного ускорения, которая в то же время может работать на GPU и без RTX (т.е. реализуя трассировку лучей программно), или даже производить вычисления на CPU? Насколько эффективна программная эмуляция RTX, доступная на предыдущем поколении видеокарт Nvidia, и насколько возможно приблизить ее эффективность к аппаратной? Какова будет трудоемкость создания аналога RTX при необходимости запуска приложения на видеокартах других производителей?

DOI: 10.31857/S0132347420030061

1. ВВЕДЕНИЕ

Трассировка лучей является базовой операцией не только в реалистичной компьютерной графике, но и во многих других приложениях (в том числе физическая симуляция, обнаружение столкновений, компьютерная геометрия, симуляция переноса нейтронов в реакторах, визуализация медицинских данных, научная визуализация и др.). В прошлом известно множество реализаций аппаратного ускорения трассировки лучей, однако ни одна из них не была массовой в том смысле, что она была бы интегрирована в штатный графический ускоритель. Поэтому важность появления такой технологии как Nvidia RTX трудно переоценить.

Однако для исследователей и разработчиков по всему миру, использующих трассировку лучей в своих решениях уже сегодня, важно понимать целесообразность интеграции технологии аппаратного ускорения (либо полного перехода на нее). Во-первых, трудоемкость разработки приложений на GPU (особенно при использовании специфической функциональности GPU) в 2–5 раз выше, чем на CPU. Во-вторых, Nvidia RTX – закрытая технология, монопольно предоставляемая на сегодняшний день лишь одним производителем графических ускорителей.

2. ОБЗОР СУЩЕСТВУЮЩИХ РАБОТ

Первыми специализированными аппаратными решениями, связанными с трассировкой лучей, были PCI-карты для визуализации объемных данных, в которых реализовано маршрирование по лучу (ray marching) и затенение по фонгу (например, [1, 2]).

Еще одной заметной ранней реализацией была архитектура SaarCOR [3] и ее обновленная версия

в ПЛИС [4]. Чип SaarCOR реализовал весь алгоритм трассировки лучей – данные сцены и камеры помещались в отдельную DRAM память, соединенную с чипом. Как и рассмотренные ранее работы, SaarCOR использовал трассировку пакетов (в группах по 64 луча). SaarCOR использовал глубокую конвейеризацию для сокрытия высокой латентности доступа к памяти по аналогии к современными GPU: пока одни группы лучей загружают данные, другие, для которых данные уже загружены на чип, могут вычислять пересечение с треугольниками. Подобные системы не получили широкого распространения. Их основной недостаток – узкая направленность системы в целом.

Альтернативой узко-специализированному чипу является размещение большого числа обычных процессоров на одной плате с PCI-e интерфейсом [5–8]. Такие решения обладают наибольшей гибкостью и могут быть использованы не только для ускорения трассировки лучей. Но они не стали популярны в основном из-за их высокой стоимости.

Наконец, существует группа работ, направленных на разработку аппаратных расширений для графических процессоров (либо разработку похожих массивно-параллельных программируемых систем). Одно из первых программируемых решений такого типа было представлено в работе [9]. Обход дерева и поиск пересечений был реализован в специальном блоке с фиксированной функциональностью, в то время как пользовательские программы (шейдеры) выполнялись на т. н. Shader Processing Unit (SPU), очень похожих по архитектуре на ранние процессорные ядра GPU. Как и SaarCOR, работа [9] использовала трассировку пакетов, из-за чего скорость сильно падала на расходящихся в разные стороны (т.н. некогерентных) лучах. Такая же проблема наблюдается во многих GPU реализациях трассировки лучей [10, 11].

Одно из решений проблемы случайного доступа к памяти предложено в работе [12]. Этот подход позразумевает разделение потока запросов к памяти как минимум на 2 потока – поток данных для лучей (ray stream), и поток данных для сцены (BVH дерево, scene stream). Можно сказать, что традиционный подход сокрытия латентности памяти при помощи глубокой конвейеризации, широко используемый в GPU, в работе [12] расширяется таким образом, чтобы загруженный один раз в кэш трилет (фрагмент BVH-дерева) был пройден всеми лучами, которые в данный момент обрабатываются на графическом процессоре. Авторы [12] уверяют что таким образом им удается избегать случайного доступа.

Кроме некогерентных лучей для GPU существует еще проблема нерегулярного распределения работы. Когда в SIMD группе потоков (warp) остается мало активных потоков/лучей, эффективность SIMD процессора GPU существенно снижается. Для решения этой проблемы в работах [10, 11] было использовано уплотнение потоков и регенерация путей, а в [13] была предложена техника блочной регенерации.

В [8, 10, 14] была использована идея группировки BVH дерева в т. н. трилетах (treelets) – небольших фрагментах BVH-дерева. Основное отличие работы [14] состоит в том, что в трилетах можно хранить данные об ограничивающих объемах в BVH с пониженной точностью в 5 бит на 1 плоскость (вместо 32 бит для стандартного типа float). Благодаря этому снижается нагрузка на память и улучшается эффективность работы кэша GPU. Кроме того, решение, предложенное в [14], является относительно дешевым в плане занимаемой площади кристалла (то есть по количеству используемых транзисторов).

Некоторые работы были направлены на аппаратную реализацию трассировки лучей для мобильных систем, где важен такой параметр как энергопотребление системы [15, 16]. Эти работы были нацелены в основном на реализацию классической трассировки лучей [17], и в отличие от многих работ рассмотренных выше, используют MIMD архитектуру с VLIW процессорами, чтобы уменьшить потери энергии/эффективности во время вычислений для расходящихся лучей.

Итого, за последнее время было разработано множество аппаратных реализаций трассировки лучей. Более полный обзор можно найти в работе [18]. Кроме того, некоторые коммерческие компании также презентовали свои решения [19], хотя в настоящее время они не доступны публично. Таким образом, RTX является первой технологией, доступной широкой общественности. Но т. к. эта технология закрыта, неясно какие методы ускорения она использует. Чтобы это понять, мы исследовали Nvidia RTX как черный ящик, проводя различные эксперименты и измеряя производительность. Для этой цели мы реализовали базовый интегратор освещенности на основе трассировки путей, используя интерфейс Vulkan.

2.1. Трассировка путей на GPU

Трассировка лучей на GPU сама по себе является ограниченной и компактной задачей, которую можно решать эффективно различными способами. Однако, проблема в корне меняется, когда на основе трассировки лучей необходимо построить расширяемую программную систему с большим количеством различных функциональностей. При этом необходимо хотя бы приблизительно сохранить исходный уровень производительности. Эта задача во-многом нетривиальна даже для CPU реализаций, но на GPU она требует применения особых подходов. На данный момент известно три основных подхода:

1) “Убер-ядро” (uber-kernel) – подход, при котором код организуется вручную или автоматически (обычно последнее) в виде конечного автомата внутри одного вычислительного ядра. Автомат используется для того, чтобы сократить регистровое давление, поскольку каждое состояние в верхнем операторе switch получает в свое распоряжение все доступные для программы (вычислительного ядра) регистры. Основные недостатки этого подхода – существенные потери производительности на ветвлениях (когда разные потоки выполняют разные состояния), и влияние различных состояний на производительность друг друга, т. к. итоговое вычислительное ядро требует столько регистров, сколько нужно самому тяжелому состоянию [20, 21].

2) “Разделенное ядро” (separate kernel) – подход, при котором код организуется (как правило вручную) в виде нескольких вычислительных ядер, общающихся между собой явно через буферы данных в памяти [20]. Этот подход решает основные недостатки убер-ядра, и благодаря явному разделению на ядра позволяет сохранять производительность критичных участков кода. Однако он обладает повышенной трудоемкостью разработки (из-за необходимости явной передачи данных, что особенно заметно при наличии сортировки или уплотнения потоков [13]). Кроме того, повышаются накладные расходы на запуск и ожидание ядер, а также на саму передачу данных. Поэтому такой подход может замедлять работу программы на простых сценах/случаях, когда накладные расходы становятся соизмеримы с полезной работой ядер.

3) “Трассировка путей волновыми фронтами” (wavefront pathtracing) – это сложный подход, в котором работа и данные для лучей группируются в отдельные очереди [21]. Очереди выполняются в разных ядрах, а результат сохраняется в нужное место памяти также путем отдельных вызовов вычислительных ядер. Благодаря группировке по условным шейдерам, wavefront pathtracing меньше теряет на ветвлениях, чем предыдущие подходы. Однако сортировка и уплотнение потоков для лучей в этом подходе строго обязательны, поэтому его накладные расходы еще выше, чем в предыдущем случае.

3. ИЗВЕСТНЫЕ ДЕТАЛИ

В настоящий момент технология RTX доступна в таких программно-аппаратных интерфейсах (Aplication Programming Interface или API) как DirectX12, Vulkan и OptiX. Для нашего исследования наибольший интерес представляет API Vulkan, поскольку он был разработан специально для того чтобы предоставлять разработчикам максимально прозрачный доступ к функциональности графических процессоров на низком уровне. Этот подход отличается, например, от OptiX, в котором компания Nvidia стремится скрыть детали, упростив жизнь разработчику прикладных приложений. Что касается DirectX12, при внимательном анализе можно обнаружить, что Microsoft добавляет некоторую собственную функциональность, реализуемую в их компиляторе HLSL. Среди дополнительных возможностей, доступных в DirectX12, следует отметить появившуюся в новой версии (т.н. DXR Tier 1.1) возможность “inline” трассировки лучей – вызова функций трассировки луча в произвольном шейдере (пиксельном, вычислительном и др.) без создания специального конвейера трассировки лучей [22]. В этом случае вызывающий код берет на себя всю работу по использованию результатов трассировки луча – вычисления в случае найденных пересечений с тем или иным видом примитива, в случае промаха и т.д. По этой причине в качестве основного API мы выбрали Vulkan.

Трассировка лучей в Vulkan используется в виде отдельного вида конвейера наряду с традиционным графическим и вычислительным конвейером общего назначения. Для запуска этого конвейера необходимо заранее строить ускорящую структуру в виде двухуровнего дерева. Нижний уровнень дерева (Bottom Level Acceleration Sructure или BLAS) строится над отдельными объектами (RTX поддерживает возможность объявления своего геометрического примитива) или мешами. Верхний уровень дерева (Top Level Acceleration Sructure или TLAS) строится над множеством инстансов (копий) объектов/мешей нижнего уровня. Что касается построения ускоряющих структур, наиболее свежая информация об этом появилась на конференции SIGGRAPH в 2019 году [23].

Сам конвейер трассировки лучей имеет 5 программируемых стадий: (1) генерация луча, (2) промах, (3) ближайшее пересечение (которая вызывается уже после того как пересечение найдено), (4) специальная стадия для реализации прозрачных теней и аналогов (называемая “any hit”, но это название сбивает с толку) и, наконец, (5) пересечение луча с геометрическим примитивом. Вместо пятой стадии есть встроенная реализация пересечения луча с треугольником, которая повидимому реализована аппаратно [24].

Между стадиями конвейера лучи переносят т. н. полезную нагрузку (ray payload) – некоторые данные, например координаты или цвет. Nvidia рекомендует делать полезную нагрузку как можно меньше, так же как и при передаче данных между стадиями графического конвейера [25]. В [23] можно заметить, что данные между различными стадиями конвейера трассировки лучей передаются не через память, а через внутренние очереди. Однако, чем больше размер полезной нагрузки, тем меньше эти очереди могут помочь.

Интересно отметить, что функциональность под-проходов (subpass) в API Vulkan в принципе

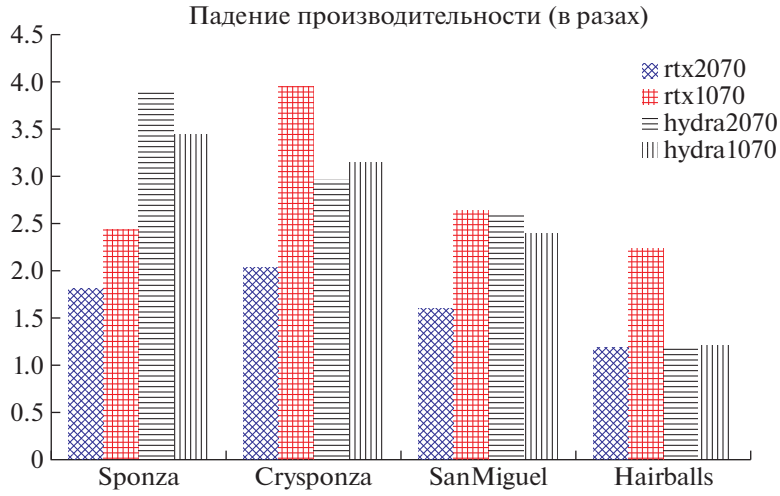


Рис. 1. Падение производительности в разах при переходе от первичных лучей ко вторичным.

позволяет передавать данные между различными вычислительными ядрами не выгружая их в память, и таким образом, симулировать механизм передачи данных, используемый в RTX (в соответствии с [25]). Однако для этого вычисления придется организовывать через графический конвейер специфическим образом, поскольку подпроходы были придуманы для специального алгоритма отложенного затенения. Кроме того, подпроходы предназначены в первую очередь для мобильных графических процессоров, и на настоящий момент нет гарантии того, что на десктопных GPU они не игнорируются полностью (т.е. в реальности все данные могут передаваться через память).

Наконец, стоит сказать, что Nvidia занимается разработкой трассировки лучей в продукте OptiX последние 5–7 лет. Поэтому мы еще раз обращаем внимание на работу [21], в которой предлагается т. н. “wavefront path tracing”.

Таблица 1. Миллионы лучей в секунду для разрешения 1024 × 1024 и 1 сэмпла на пиксел (spp), видеокарта GTX1070 (программная реализация RTX от Nvidia)

Сцена	перв.	втор.	трет.
Sponza, RTX	103	42	38
Sponza, Hydra	**214**	**62**	**59**
Crysponza, RTX	95	24	14
Crysponza, Hydra	**132**	**42**	**37**
San Miguel, RTX	26	10	6
San Miguel, Hydra	**55**	**23**	**20**
Hairballs, RTX	22	9.5	7
Hairballs, Hydra	**27.6**	**22.8**	**25**

4. ЭКСПЕРИМЕНТЫ

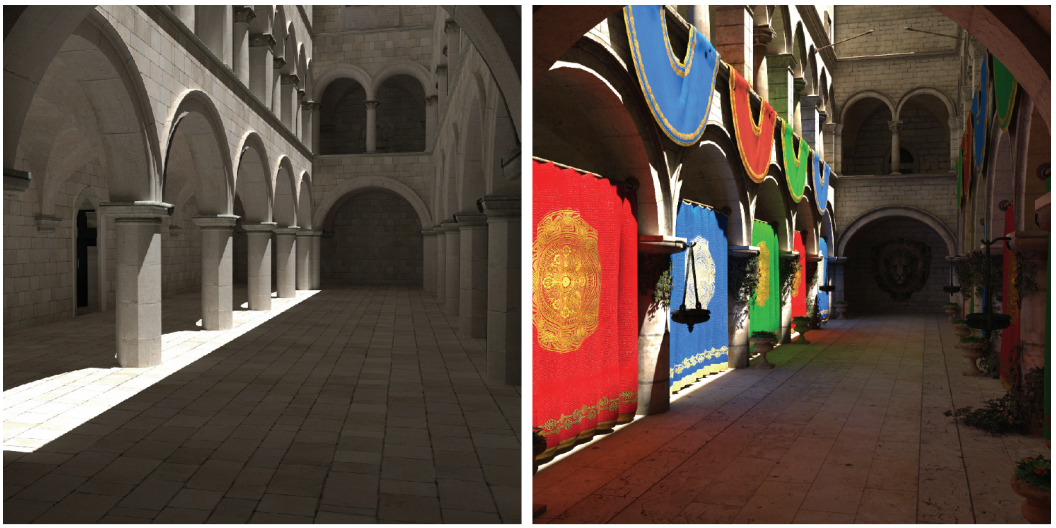
Чтобы проверить наши гипотезы о том как RTX устроен внутри, мы реализовали базовый алгоритм трассировки путей и сравнили его с открытой программной реализацией трассировки путей на OpenCL в Hydra Renderer [26]. Для измерения производительности во всех наших экспериментах мы использовали 2 графические карты: GTX1070 и RTX2070. При этом известно, что в GTX1070 трассировка лучей внутри Vulkan реализована программно, в то время как RTX2070 имеет аппаратное ускорение.

Эксперимент 1. В первом эксперименте мы измеряли время для различной глубины трассировки лучей (из которого последовательным вычитанием мы получали время для разных отскоков), и из него оценивали количество лучей в секунду для каждого отскока по формуле 4.1:

$$rays = \frac{width * height * spp}{t_s} \tag{4.1}$$

При помощи ПО Nvidia Nsight Graphics мы замеряли время вызова “vkCmdTraceRaysNV” в Vulkan и сравнивали его с временем, потраченным на выполнение вычислительного ядра OpenCL “BVH4TraversalInstKernel” в Hydra Renderer [26]. В этом эксперименте нас интересовала производительность для различных случаев: когерентные (первичные) и расходящиеся лучи (вторичные и третичные, рис. 1), а также зависимость скорости от сложности геометрии (рис. 2, таблицы 1, 2).

Эксперимент 2. Наш следующий эксперимент ставил своей целью проверить наличие внутреннего механизма распределения нерегулярной работы, когда некоторые лучи порождают много дочерних лучей, а некоторые – мало, либо не порождают их вовсе. Такая рекурсивная трассировка путей



Sponza (66 K треугольников) Cry Sponza (262 K треугольников)

Sun Miguel (11 M треугольников) Hairballs (224 M треугольников)

Рис. 2. Тестовые сцены; Sponza и Crysponza – сцены с низкой детализацией и преимущественно прямоугольной геометрией. San Miguel содержит большое количество непрямоугольных форм и наиболее близка к практике. Hairballs интенсивно использует инстансинг, при этом базовый меш состоит из неудобных для BVH дерева геометрических форм – тонких волосков.

является в некотором смысле традиционным “вызовом” для GPU реализаций, поскольку наивная реализация этого алгоритма на GPU через стэк в единственном вычислительном ядре чрезвычайно неэффективна [20, 21]. Для того чтобы получить наглядные результаты, мы провели эксперимент следующим образом: в raygen-шейдере мы пускали случайно от 10 до 40 лучей и измеряли падение производительности. Далее, мы провели аналогичный эксперимент с обыкновенными вычислениями, когда некоторые тяжелые вычисления (например, шум перлина) мы также выполняли случайно от 10 до 40 раз (рис. 4а).

Эксперимент 3. В этом эксперименте мы поставили своей целью проверить наличие в RTX внутренних очередей, передающих данные между различными стадиями конвейера трассировки лучей. Для этого мы последовательно увеличивали полезную нагрузку (payload) для луча и измеряли падение производительности в процентах, чтобы понять, в какой момент передача данных становится узким местом (рис. 5).

5. ВЫВОДЫ

Вывод 1. Nvidia RTX в первую очередь нацелена на ускорение случайного доступа к памяти во время трассировки большого числа расходящихся лучей. Этот вывод вытекает из рис. 3, справа. На небольшой сцене (Sponza) аппаратная реали-

Таблица 2. Миллионы лучей в секунду для разрешения 1024 × 1024 и 1 сэмпла на пиксел (spp), видеокарта RTX2070 (аппаратно ускоренная реализация RTX)

Сцена	перв.	втор.	трет.
Sponza, RTX	**970**	**534**	**490**
Sponza, Hydra	480	122	130
Crysponza, RTX	**788**	**386**	**337**
Crysponza, Hydra	276	92	80
San Miguel, RTX	**286**	**180**	**151**
San Miguel, Hydra	127	48	42
Hairballs, RTX	**282**	**238**	**289**
Hairballs, Hydra	61	50	56

зация Nvidia RTX на первичных (когерентных) лучах выигрывает у открытой программной реализации из [26] не более чем в 2 раза. Однако, уже на вторичных лучах эта разница достигает 5–6 раз. Кроме того, на тяжелой сцене (Hair Balls) RTX показывает то же преимущество в 4–5 раз, и тот факт, что ускорение сохраняется для сцен, где память является узким местом, подтверждает наше предположение.

Вывод 2. RTX реализует некоторый механизм группировки лучей. Это подтверждается анализом падения производительности трассировки лучей (в процентах или разах), представленном на рис. 1. Можно заметить следующие тенденции: во-первых, аппаратная реализация (rtx2070, первый столбец на рис. 1) существенно лидирует над всеми программными реализациями и ни на одной сцене не замедляется более чем в 2 раза. Во-вторых, на сцене Hairballs, где группировка лучей не может помочь в принципе в силу высокой сложности геометрии, аппаратная реализация и открытые программные реализации (столбцы hydra2070 и hydra1070), не выполняющие группировку лучей, ведут себя одинаково и не теряют в производительности существенно. При этом программная реализация RTX от Nvidia (gtx1070) демонстрирует неожиданное поведение: на простой сцене Sponza она выигрывает, но на остальных, более сложных, существенно проигрывает открытой реализации (то есть имеет существенно больший процент падения производительности при переходе к вторичным лучам). Такой результат может быть вызван одной из двух причин:

1) Если RTX в программной реализации (на GTX1070) выполнен в виде монолитного вычислительного ядра, тогда выигрыш на простой сцене является следствием сниженных накладных расходов, т. к. не нужно передавать данные между разными ядрами; при этом проигрыш на сложных сценах – прямое следствие известных недостатков убер-ядер [20, 21].

2) Если RTX в программной реализации (на GTX1070) выполнен в виде аналога “wavefront pathtracing”, тогда без должной аппаратной поддержки распределения работы этот подход, по-видимому, недостаточно эффективен.

Мы считаем первый сценарий наиболее вероятным, однако, поскольку RTX является закрытой реализацией, исключать второй вариант полностью нельзя.

Вывод 3. RTX реализует некоторый внутренний механизм распределения нерегулярной работы. Этот механизм по-видимому работает по принципу, похожему на “wavefront path tracing” [21]. Этот вывод подтверждается следующим наблюдением во время эксперимента номер 2: когда на каждый пиксел мы сгенерировали случайное (от 10 до 40) количество лучей, мы получили замедление в 2 раза по сравнению с 10 лучами. С другой стороны, когда мы повторили тот же эксперимент для вычисления шума Перлина, мы получили замедление ровно в 4 раза, как и должно быть на GPU из-за того что все потоки в SIMD группе warp должны ожидать завершения самого медленного (рис. 4а).

Вывод 4. RTX в действительности реализует передачу данных между разными стадиями кон-

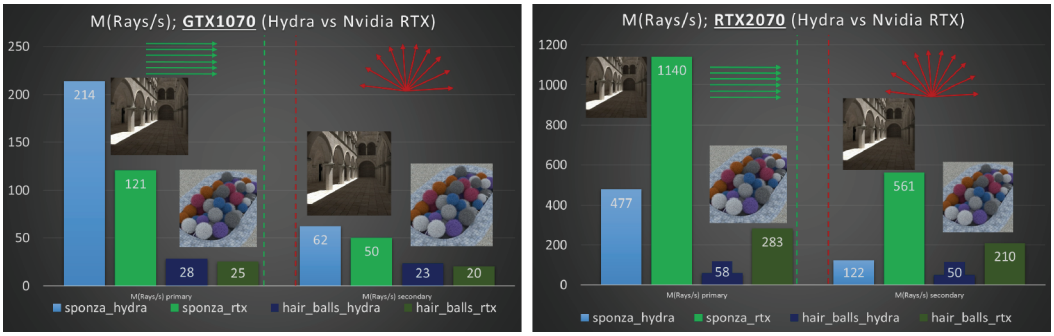
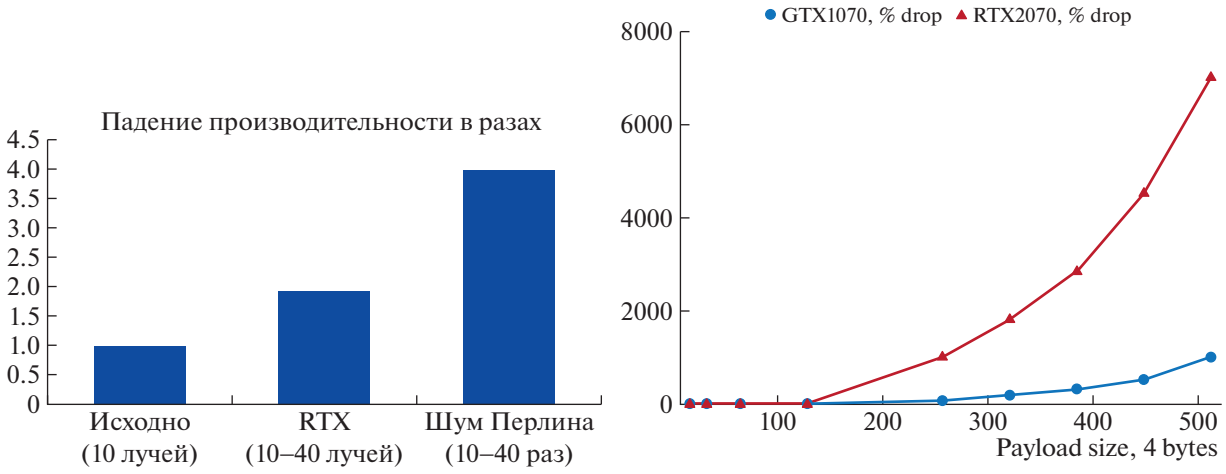


Рис. 3. Сравнение на GTX1070 (слева) и RTX2070 (справа), открытая реализация трассировки путей в HydraRenderer против Nvidia RTX. На каждом изображении его левая часть (слева от двух параллельных пунктирных линий) показывает производительность для первичных (когерентных лучей). Правая часть (справа от двух параллельных пунктирных линий) показывает производительность на вторичных, расходящихся лучах.



(а) Падение производительности для цикла со случайным количеством итерации (от 10 до 40)

(б) Падение производительности (в процентах) в зависимости от полезной нагрузки для луча (в байтах)

Рис. 4. Устойчивость производительности в различных экспериментах.

вейера (т.е. разными пользовательскими программами) через очереди на чипе. Это подтверждается характером падения производительности при увеличении полезной нагрузки на луч (рис. 5). Это дополнительно подтверждается появлением “Mesh шейдеров” в RTX картах.

Вывод 5. Nvidia RTX – это чрезвычайно тяжелая технология, которую трудно эффективно реализовать программно. Этот вывод подтверждается низкой эффективностью программной реализации RTX от самой компании Nvidia на видеокарте GTX1070, которая проигрывает простой открытой программной реализации трассировки лучей в 2–3 раза (таблица 1). Низкая производительность в данном случае, вероятно, является следствием высокой гибкости и стремлением сделать как можно более общую технологию, которая без должной аппаратной поддержки работает медленно.

6. ЗАКЛЮЧЕНИЕ

Технология Nvidia RTX – это довольно общий механизм, сочетающий в себе различную аппаратную функциональность, которая может быть использована не только в трассировке лучей, но и в других приложениях (в качестве примера такого использования можно привести работу [24]). Основные используемые механизмы это: (1) упорядочивание случайного доступа к памяти во время трассировки расходящихся лучей и (2) механизм создания работы на GPU, включающий в себя (3) передачу данных между разными вычислительными ядрами через кэш на чипе. Для пользователя RTX во многом упрощает разработку и предоставляет большую гибкость. С другой стороны, эта технология существенно ограничивает переносимость, поскольку RTX реализован как отдельный тип конвейера в Vulkan, и возможность использовать разработанный под RTX код каким-либо еще образом практически отсутствует. Данная проблема частично решается в DirectX12 (DXR Tier 1.1) за счет “inline” трассировки лучей, позволяющей использовать RTX и в “традиционных” конвейерах, но само по себе использование DirectX12 снижает переносимость еще больше.

СПИСОК ЛИТЕРАТУРЫ

1. *Meibner M. et al.* VIZARD II: a reconfigurable interactive volume rendering system // ACM Eurographics Proceedings of High-Performance Graphics on Graphics hardware, Eurographics Association. 2002. P. 137–146.
2. *Pfister H. et al.* The VolumePro real-time ray-casting system // Computer graphics and interactive techniques. N.Y.: Association for Computing Machinery, 1999. P. 251–260.
3. *Schmittler J., Wald I., Slusallek P.* SaarCOR: a hardware architecture for ray tracing // ACM Special Interest Group on Computer Graphics conference on Graphics hardware, Eurographics Association. 2002. P. 27–36.
4. *Schmittler J. et al.* Realtime ray tracing of dynamic scenes on an FPGA chip // ACM SIGGRAPH/EUROGRAPHICS conf. on Graphics hardware. ACM. 2004. P. 95–106.
5. *Hall D.* The AR350: Today’s ray trace rendering processor // Eurographics/SIGGRAPH workshop on Graphics hardware – Hot 3D Session 1. 2001.
6. *Seiler L.* et al. Larrabee: a many-core x86 architecture for visual computing // ACM Transactions on Graphics. V. 7. № 3, Article 18 (August 2008), 15 pages.

7. TRaX: *Spjut J.* et al. A multi-threaded architecture for real-time ray tracing // Symposium on Application Specific Processors, Institute of Electrical and Electronics Engineers. 2008. P. 108–114.
8. *Kopta D. et al.* An energy and bandwidth efficient ray tracing architecture // High-performance Graphics, ACM. 2013. P. 121–128.
9. *Woop S., Schmittler J., Slusallek P.* RPU: a programmable ray processing unit for realtime ray tracing // ACM Transactions on Graphics (TOG), ACM, 2005. V. 24. № 3. P. 434–444.
10. *Aila T., Karras T.* Architecture considerations for tracing incoherent rays // High-performance Graphics, Eurographics Association. 2010. P. 113–122.
11. *Nocak J., Havran V.* and Daschbacher. Path regeneration for interactive path tracing // EUROGRAPHICS 2010, short papers. P. 61–64.
12. *Shkurko K. et al.* Dual streaming for hardware-accelerated ray tracing // High Performance Graphics, ACM. 2017. P. 12.
13. *Фролов В.А., Галактионов В.А.* Регенерация путей с низкими накладными расходами // Программирование. 2016. № 6. С. 67–74. English translation: V.A. Frolov, V.A. Galaktionov. Low overhead path regeneration // Programming and Computer Software. 2016. V. 42. № 6. P. 382–387.
14. *Keely S.* Reduced precision hardware for ray tracing // Proceedings of High-Performance Graphics. 2014. P. 29–40.
15. *Nah J.H.* et al. RayCore: A ray-tracing hardware architecture for mobile devices // ACM Transactions on Graphics (TOG), ACM. 2014. V. 33. № 5. P. 162.
16. *Lee W. J. et al.* SGRT: A mobile GPU architecture for real-time ray tracing // High-performance graphics Proceedings of High-Performance Graphics, ACM. 2013. P. 109–119.
17. *Whitted T.* An improved illumination model for shaded display // ACM Special Interest Group on Computer Graphics and Interactive Techniques. 1979. V. 13. № 2. P. 14.
18. *Deng Y. et al.* Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques // ACM Computing Surveys (CSUR). 2017. V. 50. № 4. P. 58.
19. Imagination technologies., PowerVR Ray Tracing, 2019, https://www.imgtec.com/graphics-processors/architecture/powervr-ray-tracing/
20. *Фролов В.А., Харламов А.А., Игнатенко А.В.* Смещенное решение интегрального уравнения светопереноса на графических процессорах при помощи трассировки путей и кэша освещенности // Программирование. 2011. Т. 37. № 5. English translation: V. A. Frolov. A. A. KharlamovA. V. Ignatenko. Biased solution of integral illumination equation via irradiance caching and path tracing on GPUs // Programming and Computer Software. 2011. V. 37.
21. *Laine S., Karras T., Aila T.* Megakernels considered harmful: wavefront path tracing on GPUs // Proceedings of the 5th High-Performance Graphics Conference (HPG '13), ACM, New York, NY, USA. P. 137–143.
22. Microsoft. DirectX, спецификация трассировки лучей (DXR), 2019, https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html
23. Viitanen Timo. Acceleration data structure hardware (and software), Jul 27, 2019, Special Interest Group on Computer Graphics and Interactive Techniques, LA, USA.
24. *Wald I.* et al. RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location. Authors' Preprint – to be presented at High-Performance Graphics 2019.
25. Nvidia. RTX Ray tracing developer resources, 2019, https://developer.nvidia.com/rtx/raytracing
26. Ray Tracing Systems, Keldysh Institute of Applyed Mathematics, Moscow State Uiversity. Hydra Renderer. Open source rendering system, 2019, https://github.com/Ray-Tracing-Systems/HydraAPI