Адаптация облачного сервиса для HDR-обработки фотографий под GPU с архитектурой NVIDIA Kepler

М.А. Кривов[1,3], Н.Ю. Захаров[1,3], С.Г. Елизаров [2,3]

ООО «ТТГ Лабс»[1], ООО «ЦИФ МГУ им. М.В. Ломоносова»[2], МГУ им. М.В. Ломоносова[3]

В статье рассматривается процесс оптимизации расчётных модулей для обработки изображений под новое поколение ускорителей NVIDIA Kepler. Приведены результаты их профилировки на разных типах данных и на трёх целевых поколениях GPU (Fermi, Kepler1 и Kepler2), описаны предложенные схемы адаптации узких мест алгоритма под каждую целевую архитектуру. Отдельно рассматривается эффект от применения каждого вида оптимизации и его зависимость от размера обрабатываемых данных. Результатом адаптации стало повышение скорости работы сервиса на новом поколении GPU в среднем в 2.4 раза.

Ключевые слова: графические ускорители, GPU, оптимизация программ, обработка фотографий, облака

1. Введение

Чем больше приложение оптимизировано под конкретное поколение GPU, тем сложнее его перенести на более современные архитектуры. Данная проблема отчётливо наблюдалась при оптимизации облачного web-сервиса HdrConverter.com, целью которого является повышение качества фотографий и видео путём применения HDR-преобразования [1]. При попытке обновить используемые ускорители с GeForce 580 до GeForce 780 Ti выяснилось, что скорость обработки повышается всего на 10-30%, хотя пиковая производительность новых GPU выше примерно в 3 раза. И что является более интересным, в отдельных CUDA-ядрах даже было зафиксировано падение производительности.

Рассматриваемый HDR-сервис осуществляет обработку в гибридном облаке, состоящем из собственных ресурсов, а также узлов g2.2xlarge из Amazon EC2, которые могут быть временно подключены при пиковых нагрузках. В результате этого в проекте должны поддерживаться три поколения GPU – Fermi, Kepler1, Kepler2. Более того, обработка может осуществляться как для видео-файлов, так и фотографий, поэтому при оптимизации нужно учитывать и тот факт, что входные данные могут варьироваться от 2-ух мегапиксельного снимка до ролика размером 30 гигапикселей.

Архитектура сервиса была изначально спроектирована для работы в GPU-облаках [1], оснащённых узлами различных типов, поэтому в ней на момент проведения данного этапа уже была реализована вся облачная функциональность — динамическое распределение запросов, асинхронная обработка данных, возможности укрупнения множества запросов в один и активное переиспользование выделенной GPU-памяти. Таким образом, в рамках данных работ требовалось только провести адаптацию CUDA-ядер, которые изначально были оптимизированы для GPU поколения Fermi, под новые ускорители.

2. Описание сервиса

Большинство современных фотокамер способны запечатлеть, а большинство современных дисплеев воспроизвести ровно 256 оттенков цвета одного цвета, в то время как человеческий глаз может различать существенно большее их число. Результатом этого является тот факт, что достаточно часто фотографии выглядят неестественными — некоторые области кажутся засвеченными, а другие, наоборот, излишне затемнёнными. Это объясняется тем, что некоторые оттенки, различимые человеком, обрезаются при использовании стандартного формата

R8G8B8, в котором на каждый цветовой канал выделяется по 8 бит (или по 256 градаций цвета). Профессиональные фотокамеры поддерживают форматы, в которых на один цветовой канал выделяется порядка 12 бит (или 4096 оттенков), однако полученные с их помощью фотографии всё равно требуется отображаться на обычных дисплеях, поэтому часть информации неизбежно будет утеряна.

Решением этой проблемы является HDR-обработка, суть которой заключается в более «привычном» для человека отображении оттенков из расширенного цветового формата R32G32B32 в стандартный R8G8B8. Благодаря этому преобразованию повышается локальный контраст, в результате чего засвеченные области затемняются, а излишне тёмные — подсвечиваются, и фотография начинает восприниматься как более естественная. Существует множество методов для подобного отображения цветов, однако авторами были разработаны собственные закрытые алгоритмы HDR-обработки, оформленные в виде портала HdrConverter.com.

Данный сервис предоставляет пользователю возможность обработать медиа-объект (фотографию или видео) с помощью одного из 16 проблемно-ориентированных фильтров, каждый из которых, фактически, является набором конфигурационных параметров для одного из трёх базовых алгоритмов. Необходимость поддерживать сразу три разных HDR-преобразования была обусловлена тем, что целевой аудиторией сервиса являются молодые люди возрастом около 20 лет, которым нужно быстро подготовить снимки и ролики со смартфонов к их последующей публикации в социальных сетях. К сожалению, одной из особенностей «канонической» версии HDR-обработки является искусственное старение запечатлённых людей примерно на 5-10 лет, так как из-за повышения локального контраста все морщины и складки кожи становятся отчётливо заметными. Как следствие, востребованность подобного сервиса, особенно среди женской половины пользователей, была бы под большим вопросом. Чтобы этой проблемы избежать, были подготовлены три реализации HDR-преобразования, каждая из которых разрабатывалась под свой тип объектов — (1) природа и строения, (2) люди, (3) смешанные сцены. И, таким образом, адаптировать под GPU поколения Kepler потребовалось сразу три независимых фильтра.

Рис. 1. Пример работы HDR-преобразования (исходная и обработанная фотографии)

В проекте участвует представители двух разных компаний, поэтому отдельное внимание уделялось вопросам разделения зон ответственности. В частности, для этого был разработан собственный C-подобный язык скриптов, на котором осуществляется запись конкретного HDR-алгоритма, и соответствующий ему интерпретатор, встроенный в облачную инфраструктуру. Благодаря подобному разделению стало возможным обновлять алгоритмы HDR-обработки и серверную компоненту полностью независимо. Минусом данного подхода является отсутствие информации об алгоритме, что затрудняет процесс его оптимизации под новые GPU. Это разделение также приводит к необходимости проводить оптимизацию исключительно на уровне отдельных CUDA-функций, которые могут быть вызваны из скрипта с совершенно произвольными параметрами. Именно поэтому далее в работе рассматривается оптимизация отдельных операций над изображениями при условии, что ключевые параметры типа размера ядра свёртки заранее неизвестны. Хотя и имеется возможность их узнать, нет никакой гарантии, что они останутся неизменными через месяц.

3. Результаты профилировки

В разработанном языке скриптов имеются 93 встроенные функции для осуществления операций над изображениями, каждая из которых представляет собой обёртку для одного или нескольких CUDA-ядер. При проведении их профилировки выяснилось, что в рассматриваемом сервисе единое «узкое» место отсутствует, в частности, из-за того, что оптимизировать требуется сразу три независимых алгоритма. К примеру, если выбрать только те функции, вклад которых в общее время работы составляет 70%, 80% и 90%, то их количество будет равно 8, 18 и 27 штукам соответственно.

В следующей таблице 1 приведены замеры для десяти наиболее вычислительно-ёмких функций, на которые приходится более 70% времени работы, при их выполнении на ускорителях GeForce 580 (Fermi), GeForce 680 (Kepler1) и GeForce 780 Ti (Kepler2).

Таблица 1. Время выполнения отдельных этапов на разных поколениях GPU в формате «время, секунды (ускорение, проценты)»

Тип операции	Fermi	Kepler1	Kepler2
Rescale	0,50	0,33 (+50%)	0,21 (+139%)
Max-Min	0,50	0,45 (+13%)	0,56 (-11%)
Saturation	0,52	0,45 (+15%)	0,32 (+62%)
Convolution	0,53	0,67 (-21%)	0,63 (-15%)
Blur	0,83	0,72 (+16%)	0,80 (+4%)
Erode	1,00	0,93 (+7%)	0,59 (+68%)
Offset Filter	1,00	1,46 (-32%)	0,63 (+58%)
Surface Blur	1,85	3,14 (-41%)	3,25 (-43%)
Scale	2,92	3,99 (-27%)	2,30 (+27%)
Recursive Filter	4,97	6,25 (-21%)	2,64 (+88%)

Конкретные времена достаточно сильно зависят от количества обрабатываемых мегапикселей и типа фильтра. К примеру, операция Recursive Filter иногда уступает «лидерство» и занимает второе или третье место, или же вообще не выполняется. Однако исходная проблема наблюдается достаточно отчётливо — достигнутое ускорение от перехода на новое поколение GPU в ряде случаев оказалось отрицательным. Наиболее иллюстративным является пример операции Surface Blur, при выполнении которой было зафиксировано падение производительности на 41-43%. И это с учётом того, что и пиковая производительность, и пропускная способность памяти новых GPU выросли в несколько раз.

На рис. 2 приведены оценки ускорения относительно поколения Fermi уже не для отдельных операций, а сразу для всего фильтра целиком. Они также снабжены теоретическим ускорением, которое можно было ожидать, исходя из технических характеристик новых GPU.

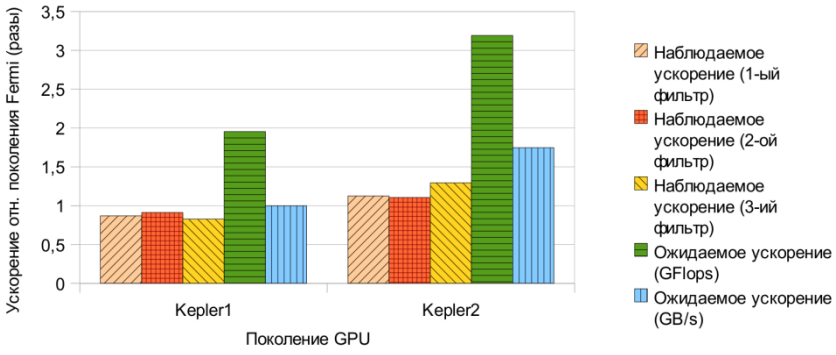

Рис. 2. Ожидаемые и наблюдаемые ускорения на GPU поколении Kepler1 и Kepler2

Как легко заметить, рассматриваемые фильтры не получают заметного выигрыша ни от повышения пиковой производительности (GFlops), ни от пропускной способности памяти (GB/s) ускорителя. Собственно, это объясняется тем, что в архитектуре GPU произошли достаточно существенные изменения, в результате чего требуется перестроить проблемные CUDA-ядра. При этом проблема оказалась не в том, что код исходных CUDA-ядер был низкокачественным или неэффективным, а исключительно в том, что он был спроектирован и оптимизирован для устаревшей архитектуры Fermi, некоторые из рекомендаций для разработки под которую оказались контрпродуктивными при появлении нового поколения ускорителей.

4. Проведённая оптимизация

Изначально предполагалось, что наибольшее ускорение будет получено за счёт использования новых возможностей, появившихся в поколении Kepler, в частности, таких как read only data cache и dynamic parallelism. Однако в процессе проведения оптимизации выяснилось, что выигрыш от их использования ограничивается несколькими десятками процентов, в то время как основной эффект обеспечивается дополнительным упорядочиванием операций чтения из памяти и увеличением количества потоков. Что, в принципе, не является каким-то неожиданным результатом — подобная рекомендация верна для любого массивно-параллельного вычислителя. Но интересным оказался тот факт, что новое поколение GPU оказалось крайне чувствительным к подобным модификациям, что делает его похожим с точки зрения процесса программирования на архитектуру GT200, которая использовалась, например, в ускорителях Tesla C1060 (2009 год).

Таким образом, можно заявлять о некотором регрессе — в обязанности программиста вновь вернулась необходимость тщательно следить за операциями чтения из памяти, хотя в предыдущем поколении Fermi с этим зачастую успешно справлялись L1/L2-кэши. Впрочем, наблюдаемому явлению есть и достаточно логичное объяснение. Если проследить за величиной GFlops/GBps, которая характеризует некий абстрактный объём полезных вычислений на один байт, то она описывается следующей последовательностью: 6.68 (GT200), 8.21 (Fermi), 16.07 (Kepler1) и 15.01 (Kepler2). Или, другими словами, теперь на каждый прочитанный из памяти байт требуется выполнить в два раза больше вычислительных операций. Собственно, одно из проявлений этой проблемы и наблюдалось в рассматриваемом сервисе — если на архитектуре Fermi требовалось оптимизировать вычисления, то на Kepler акцент сместился на обращения к памяти.

При детальном рассмотрении проблемных CUDA-функций выяснилось, что в сервисе можно выделить два основных паттерна работы с изображениями, которые в том или ином виде встречались в большинстве ядер — свёртка с двумерным ядром и сканирование по одному измерению. И хотя по структуре все фильтры сильно различаются, основной вклад в итоговое ускорение дали именно оптимизации, проведённые по результатам анализа этих двух схем работы с памятью.

4.1. Оптимизация паттерна «свёртка»

Наиболее иллюстративным является паттерн «свёртка», который оказался не только самым часто используемым, но, в некотором смысле, и наиболее неоднозначным. В данном сценарии для каждого пикселя двумерного изображения производится чтение значений из его окрестности размером (2R+1)x(2R+1) с их последующей свёрткой в скалярную величину, которая и будет определять обновлённое значение соответствующего пикселя. Что, в частности, описывается следующим фрагментом кода, где x и y – координаты обрабатываемого пикселя, а in и out – массивы с входным и выходным изображениями размером width x height:

```
float res = 0.0f;
for (int dy = -R; dy <= R; dy++)
   for (int dx = -R; dx <= R; dx++)
         res = f1(res, x, y, dx, dy,
                  in[x + dx + (y + dy) * width]);
```

```
out[x + dx + (y + dy) * width] = f2(res, x, y);
```

Рис. 3. Схема доступа к памяти в паттерне «Свёртка»

В зависимости от реализаций функций f1(...) и f2(...) данный код может описать совершенно различные действия — от свёртки с несепарабельным ядром до подавления искажений от блочного сжатия. В частности, он используется в ядрах Convolution, Blur, Erode, Surface Blur. Подходы к оптимизации подобных операций были предложены во многих работах, в качестве примера которых можно привести [2, 3], однако в данном случае ключевой особенностью была именно вариативность алгоритма, что требовало применения несколько иных приёмов. Более того, в работах предшественников в основном рассматриваются устаревшие архитектуры, поэтому воспроизвести их результаты оказалось затруднительно.

Как легко заметить, реализация соответствующего CUDA-ядра существенно зависит от функции f1(...) и значения R, которое в рамках даже одной операции варьируется в достаточно широком диапазоне — обычно от 1 до 30. Для оценки оптимизационного потенциала на базе этого ядра были подготовлены две модельные версии, в первой из которых все чтения из глобальной памяти заменены на обращения к разделяемой, в то время как из второй были убраны все вычислительные операции, однако при этом сохранён порядок адресации массивов. Очевидно, что подобные версии не являются корректными и выдают ошибочные результаты, однако с их помощью можно понять, эффект от какого вида оптимизации — вычислений или доступа к памяти — будет наибольшим. Соотношение времени выполнения этих двух модельных ядер на всех трёх ускорителях приведено на рис. 3.

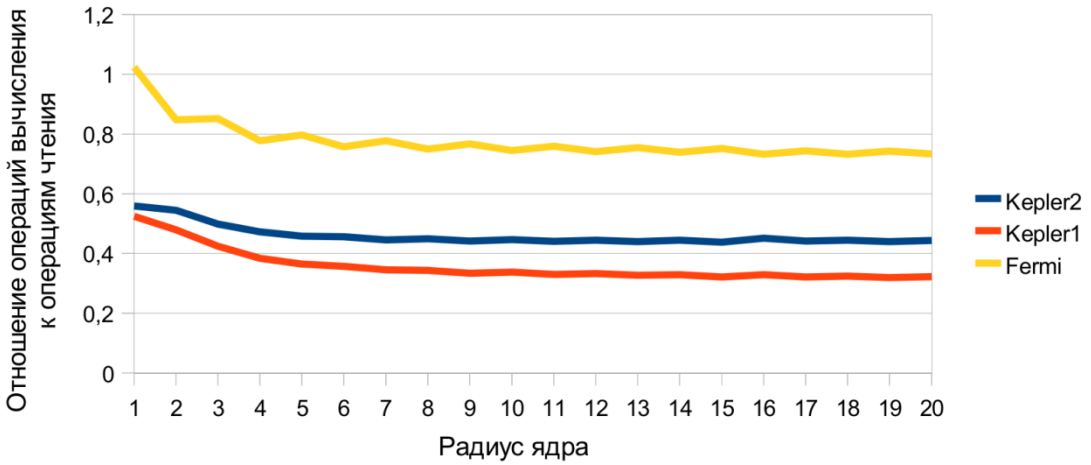

Рис. 3. Соотношение времени выполнения операций для двух модельных ядер для паттерна «свёртка» при обработке изображения из 4 мегапикселей

Полученные результаты хорошо соотносятся со сделанным ранее утверждением, что на новых архитектурах на одно чтение из памяти требуется в несколько раз больше вычислительных операций. Действительно, как видно из приведённой иллюстрации, в случае ускорителя на базе Fermi исходная реализация (которая, к слову, не содержала каких-либо оптимизаций, работая напрямую с глобальной памятью и порождая по одной нити на пиксель) оказалась достаточно сбалансированной — затраты на доступ к памяти и на сами вычисления были примерно равны. В случае же новых ускорителей Kepler1 и Kepler2 отчётливо наблюдается «перекос» в сторону получения значений из памяти — соответствующее соотношение уменьшилось с 74 до 33 и 44 процентов, то есть почти в два раза.

Если учесть, что в рамках архитектуры CUDA данные действия выполняются асинхронно, то общее время определяется не как сумма, а как максимальное значение из этих двух величин. Таким образом, ожидаемое ускорение можно оценить следующим образом — если удастся на порядок уменьшить время работы с памятью, то общая производительность может вырасти на 35%, 203% и 127% при работе на GPU Fermi, Kepler1 и Kepler2 соответственно, в то время как от оптимизации только вычислительных операций какого-либо ускорения ожидать не следует.

Самым простым вариантом снижения затрат на чтение из памяти является помещение в

динамическую разделяемую память сразу всей подобласти, значения из которой используются нитями одного блока. Отметим, что данная *динамическая* память отличается от обычной статической только тем, что её требуемый объём можно задать при запуске ядра, а не на этапе компиляции. Таким образом, если блок определён как 16x16, а объём разделяемой памяти равен 16 килобайтам (гарантированный размер), то максимальным радиусом ядра будет R=24. При увеличении объёма памяти до 48 килобайт (в ущерб кэшу L1) максимально значение для радиуса возрастает до R=47. Или, другими словами, данный вид оптимизации будет работать для всех требуемых ядер свёртки.

Отдельным вопросом является схема выгрузки данных из глобальной памяти. С одной стороны, запросы должны быть по возможности выровнены по 16 элементов (coalesced), но при этом объём вспомогательных операций для вычисления индексов не должен быть существенным. В данной работе были рассмотрены четыре схемы, которые приведены на рис. 4 для случая блока 4x4 и ядра с R=2 — номер в клетке указывает на номер нити, осуществляющей чтение соответствующего элемента, а разными цветами разделена очерёдность операций.

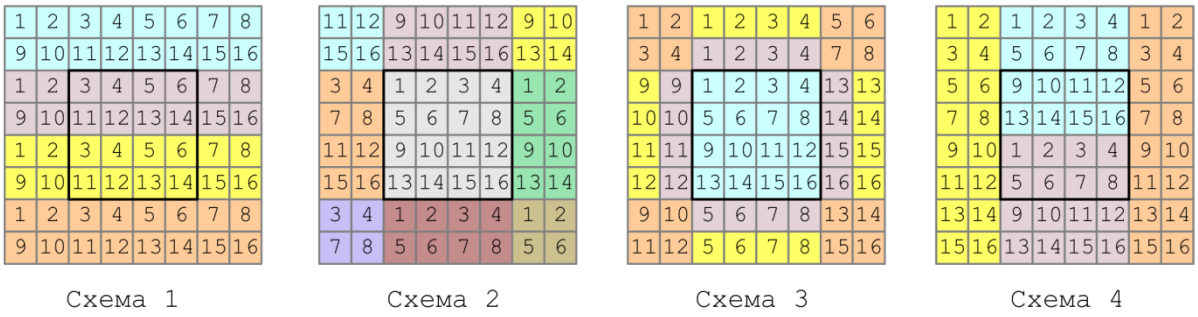

Рис. 4. Схемы выгрузки значений из глобальной в разделяемую память

В схеме 1 для расширенной области размера (2R+blockDim.x) x (2R+blockDim.y) вводится сквозная нумерация, после чего каждая нить с шагом blockDim.x * blockDim.y осуществляет чтение доставшихся ей элементов. Недостаток этого подхода очевиден — для большинства значений R ширина области не будет кратна размеру полуварпа. Из-за этого нарушится выравненность запросов, в результате чего объём реального трафика вырастет в разы. Чтобы этой проблемы избежать, была предложена схема 4, в которой имеется заранее определённый вспомогательный массив, позволяющий по исходному сквозному номеру узнать скорректированный индекс. Этот массив формируется индивидуально для каждого размера R таким образом, чтобы максимизировать число группируемых в один запросов к памяти, при этом он сам может храниться в кэше (константном или текстурном).

В схемах 2 и 3, напротив, сначала загружаются значения из центра, и только после этого из R-окрестности. В схеме 2 все чтения осуществляются сразу блоками, в результате чего она предпочтительна для больших значений R, в разы превосходящих ширину или высоту CUDA-блока, в то время как в схеме 3 каждое из восьми направлений последовательно загружается отдельным полуварпом, что должно было обеспечивать наилучшую скорость для небольших значений порядка R=1,2,3.

Как показало предварительное тестирование, схемы 3 и 4 оказываются заведомо неэффективными на рассматриваемых архитектурах, хотя для устаревших моделей ускорителей с CC 1.3 именно они, по мнению авторов, были бы предпочтительными. В первом случае проблемой оказались излишние расходы на вычисление смещений и выполнение проверок на выход за пределы области, во втором — отсутствие подходящего типа кэша для массива со скорректированными индексами. Константная память имеет задержку до одного такта, но плохо обслуживает запросы на чтение по последовательным индексам, сериализуя их по 16 штук, а текстурная память не обеспечила существенно выигрыша. Таким образом, тестирование проводилось только для первых двух схем, результаты которого приведены на рис. 5 в виде ускорений относительно исходной версии.

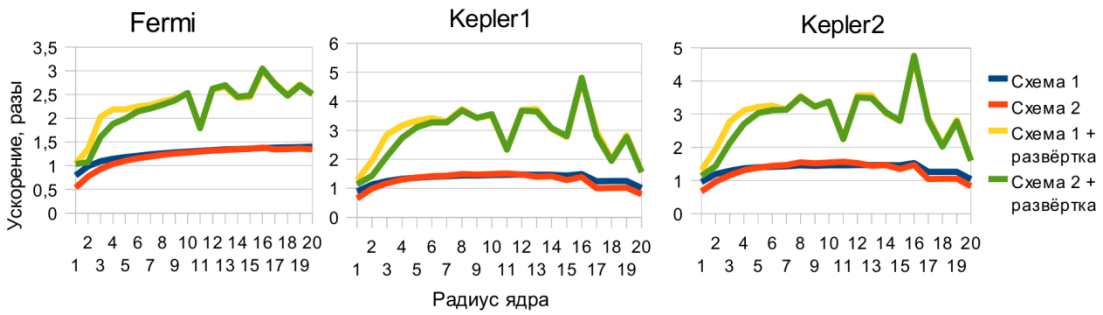

Рис. 5. Производительность различных схем выгрузки данных при обработке изображения из 4 мегапикселей

В первую очередь стоит отметить, что из-за добавления барьерной синхронизации через __syncthreads() пропала асинхронность ядра — теперь все данные сначала выгружаются в разделяемую память, и только после этого проводятся вычисления. Как следствие, «суммарный» эффект от перехода к разделяемой памяти оказался не таким уж существенным, а местами и вовсе отрицательным — от 0.79 до 1.4 для архитектуры Fermi, от 0.95 до 1.52 — для Kepler2. При этом, как и предполагалось, схема 1 оказалась более производительной, хотя в некоторых сценариях и немного проигрывала схеме 2.

Другой проблемой, появившейся в результате внесённых изменений, стало увеличение вспомогательных операций по вычислению индексов. К счастью, её можно избежать, осуществив развёртку циклов на этапе компиляции с помощью шаблонов — для этого достаточно размер ядра объявить в виде шаблонного параметра, после чего с помощью макросов сгенерировать вызовы для всех требуемых значений. Далее компилятор сможет автоматически раскрыть все циклы по y, что позволяет ему заметно уменьшить объём вычислений — соответствующие результаты тестов на рис. 4 имеют суффикс «развёртка». Как легко заметить, благодаря этой модификации в лучшем случае ускорение поднялось с 1.5 до 4.5 раз. Стоит отдельно подчеркнуть, что подобный эффект наблюдается только при использовании разделяемой памяти — в исходной версии развёртка циклов не обеспечивала заметного выигрыша.

Последним этапом оптимизации данного паттерна стала динамическая подстройка размера CUDA-блока под значения радиуса ядра R и формат кадра, так как во всех предыдущих замерах использовалось статическое значение 16x16. Как известно, максимальная эффективность использования ускорителей NVIDIA достигается в том и только в том случае, когда суммарный объём ресурсов, выделяемых для всех блоков одного мультипроцессора, равен предельным значениям. К подобным типам ресурсов относятся количество регистров (не более 32 — 128 тысяч), число блоков (8 — 32) и объём разделяемой памяти (48 — 112 КБ). Изменяя размер блока, мы можем неявно варьировать первый параметр и явно влиять на два последних, и, таким образом, корректировать потребности в каждом виде ресурсов мультипроцессора. Как следствие, удаётся повысить эффективность выполнения конкретного CUDA-ядра, однако стоит помнить, что это не обязательно приведёт к повышению производительности — к примеру, 50% загрузка мультипроцессора в ряде случаев лучше 100%-ой [4].

Подстройка размера блока осуществлялась с помощью автотюнера TTG Apptimizer [5]. Тестирование проводилось при одновременном варьировании размера кадра и радиуса ядра, соответствующие результаты приведены в таблице 2.

Таблица 2. Оптимальный размер CUDA-блока и достигаемое за счёт перехода к нему ускорение (относительно блока 16x16)

Размер кадра	**Fermi**		**Kepler1**		**Kepler2**	
	Ускорение	Блок	Ускорение	Блок	Ускорение	Блок
R = 1						
640 x 480	+23.5%	32x16	+47.2%	32x8	+101.5%	32x8
1024 x 768	+24.1%	32x12	+19.2%	32x8	+36.4%	32x16

1600 x 1200	+14.8%	32x16	+20.4%	32x8	+26.0%	32x8
2560 x 1600	+8.8%	32x16	+14.5%	32x8	+16.5%	32x8
R = 5						
640 x 480	+3.7%	32x10	+11.4%	32x8	+2.9%	32x10
1024 x 768	+8.4%	32x12	+21.4%	32x8	+13.8%	32x12
1600 x 1200	+8.4%	32x16	+19.4%	32x8	+17.9%	32x12
2560 x 1600	+8.9%	32x12	+19.8%	32x8	+20.6%	32x12
R = 10						
640 x 480	+10.9%	32x16	+30.6%	32x8	+37.0%	32x12
1024 x 768	+13.5%	32x16	+30.2%	32x8	+28.7%	32x12
1600 x 1200	+13.5%	32x16	+32.0%	32x16	+32.8%	32x12
2560 x 1600	+14.2%	32x16	+32.3%	32x16	+32.9%	32x12

Эффект от автотюнинга оказался достаточно неожиданным в том плане, что для ускорителей Kepler1 и Kepler2 потребовались разные размеров блоков, хотя структура их мультипроцессоров практически идентична. Если смотреть на полученное ускорение, то, на первый взгляд, оно кажется не столь существенным — в среднем скорость обработки увеличилась на 12%, 24% и 30% для поколений Fermi, Kepler1 и Kepler2 соответственно. Однако если учесть, что эти величины определены относительно уже оптимизированных версий, то результаты начинают выглядеть несколько иначе — например, для случая R=10 и ускорителя Kepler2 итоговое ускорение выросло с 3.4 до 4.65 раз.

4.2. Оптимизация паттерна «сканирование по измерению»

Второй из рассматриваемых паттернов сводится к поочерёдному сканированию изображения по строкам и столбцам, что можно схематично описать следующим фрагментом кода:

```
for (int y = 0; y < height; y++)
{
   for (int x = 1; x < width; x++)
        img[x + y * width] = f(img[  x   + y * width],
                               img[x - 1 + y * width]);
}

for (int x = 0; x < width; x++)
{
   for (int y = 1; y < height; y++)
        img[x + y * width] = f(img[x +     y     * width],
                               img[x + (y - 1) * width]);
}
```

Рис. 6. Схема работы с памятью в паттерне «сканирование по измерению»

В данном случае имеется две основные проблемы, а именно — (1) ограниченный параллелизм, так как число порождаемых CUDA-нитей не может превышать высоту или ширину изображения, и (2) невыровненный доступ к памяти при X-проходе. Рассматриваемая операция даже для одного кадра выполняется многократно, поэтому эти две особенности алгоритма зачастую становятся ключевыми недостатками — например, в таблице 1 вариация данного паттерна идёт под именем «Recursive Filter», и на её выполнение уходило более 40% от всего времени работы.

Первую проблему удаётся преодолеть техническим путём за счёт механизма укрупнения изображений — если за один запуск CUDA-ядра обрабатывается по 100 кадров видео в формате

HD Ready, то будет порождено не менее 72 тыс. нитей, что более чем достаточно для эффективной загрузки GPU. Вторая же проблема, наоборот, лишь усугубляется при росте количества обрабатываемых мегапикселей, так как соотношение времени, затрачиваемого на X- и Y-проходы, возрастает с 3:1 до 10:1.

Для оптимизации данной операции было добавлено дополнительное транспонирование изображения до и после X-прохода. Таким образом, чтения из памяти всегда будут выровненными, однако в дополнение к «полезным» действиям теперь добавляются два «служебных» вызова CUDA-ядер, которые лишний раз копируют всё изображение из памяти в регистры и обратно. Чтобы этого избежать, два данных транспонирования были также встроены в ядра для X- и Y-проходов. На рис. 7 показано ускорение, которое удалось достичь за счёт предложенных изменений — линия «Оптимизация #1» соответствует результатам для транспонирования изображения через вызовы вспомогательных ядер, в то время как в варианте «Оптимизация #1 и #2» реализована схема с укрупнёнными ядрами.

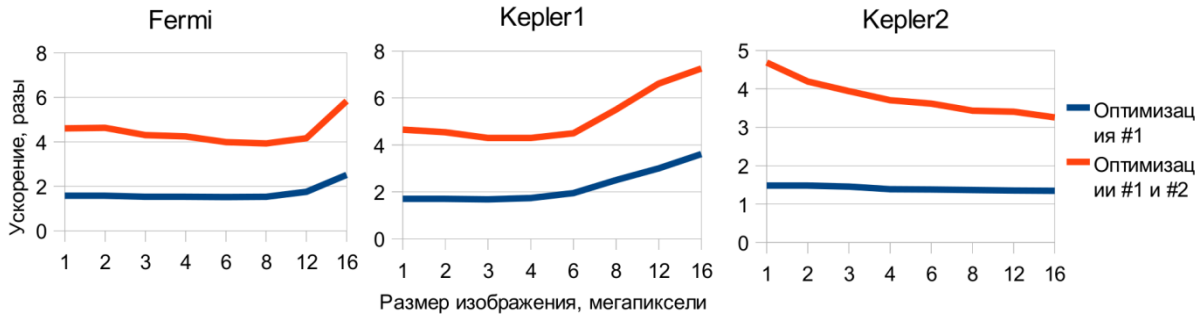

Рис. 7. Ускорение в зависимости от разных видов оптимизации (относительно исходной версии)

Во всех случаях ускорение было многократным, но достаточно неожиданным оказалась его зависимость от размера данных. Если для архитектур Fermi и Kepler1 оно увеличивалось с ростом числа мегапикселей, то для Kepler2 – наоборот, уменьшилось с 4.7 до 3.1 раза. Логично предположить, что причина кроется в аппаратных нововведениях, а именно в read only кэше, благодаря которому проблема невыровненного доступа изначально уже была сглажена.

В отличие от рис. 7, на рис. 8 приведены не ускорения относительно исходных версий, а полезная производительность. Как видно, предложенные модификации действительно помогли в разы повысить скорость обработки, однако положительный эффект был инвариантным и наблюдался вне зависимости от поколения ускорителя. Данный факт объясняется достаточно просто — основным узким местом рассматриваемой операции является работа с памятью, поэтому не стоило ожидать какого-либо ускорения только от роста вычислительной производительности при неизменной пропускной способности памяти.

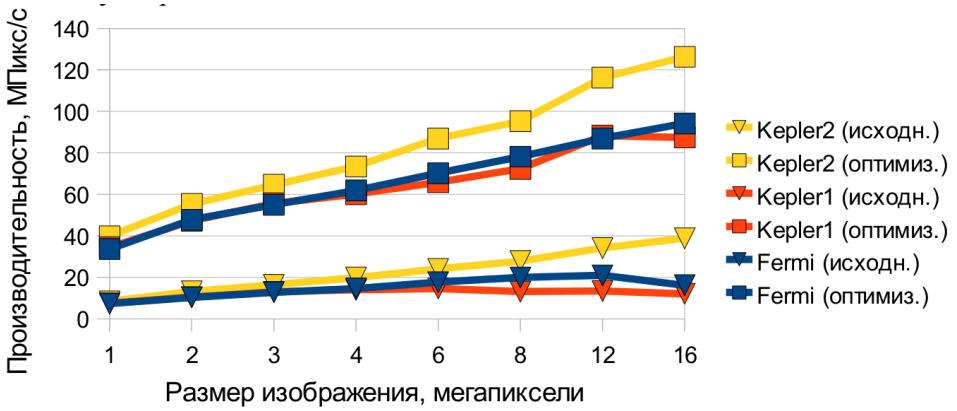

Рис. 8. Производительность исходной и оптимизированной версий на разных поколениях ускорителей

5. Заключение

Основным полученным результатом является проведённая оптимизация сервиса под актуальные поколения ускорителей Kepler1 и Kepler2, которая заключалась не только в применении технических приёмов, но и в исследовании схем отображения используемых паттернов работы с изображениями на рассматриваемые архитектуры. Благодаря предложенным модификациям удалось в среднем повысить скорость работы сервиса в 2-3 раза, что, в частности, можно проиллюстрировать рис. 9 зависимости достигнутого ускорения от размера данных для одного из трёх используемых HDR-преобразований (фильтр третьего типа для смешанных сцен):

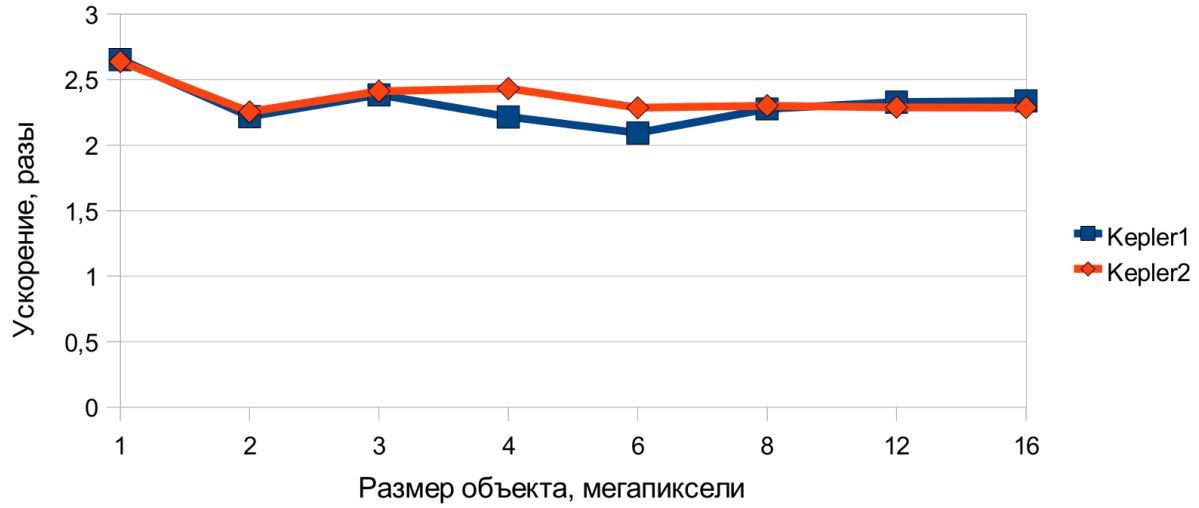

Рис. 9. Достигнутое ускорение на GPU поколении Kepler1 и Kepler2 относительно исходной версии

Внесённые изменения также позитивно сказались и на производительности, достигаемой при использовании устаревшей архитектуры Fermi, однако данный результат является второстепенным, так как изначально скорость обработки на данных моделях GPU и так была удовлетворительной, а срок их службы уже подходит к концу.

В рамках дальнейших работ планируется проведение схожей оптимизации под поколения ускорителей Maxwell, предварительные исследования производительности сервиса при работе на которых показали, что ожидать автоматического повышения скорости при переходе на данные модели GPU также не следует.

Литература

1. Притула М.Н., Кривов М.А., Елизаров С.Г., Опыт разработки сервиса для HDR-обработки видео-данных в GPGPU-облаке // Научный сервис в сети Интернет: поиск новых решений: Труды Международной суперкомпьютерной конференции (17-22 сентября 2012 г., г. Новороссийск), 2012, с. 287 — 291.
2. Silva J., Ansorge R., Jena R., Efficient scatter-based kernel superposition on GPU // Journal of Parallel and Distributed Computing, Volume 84/C, 2015, pp. 15-23.
3. Podlozhnyuk V., Image Convolution with CUDA, NVIDIA Corporation, 2007.
4. Volkov V., Better performance at lower occupancy // Proceedings of the GPU technology conference (GTC 2010), 2010, p. 16.
5. Кривов М.А., Притула М.Н., Иванов П.С., Применение технологий автоадаптации программ для решения CFD задач на структурированных сетках с использованием GPU // Параллельные вычислительные технологии (ПаВТ'2014): труды международной научной конференции (1–3 апреля 2014 г., г. Ростов-на-Дону), 2014, с. 108–117

Adaptation of cloud service for HDR photo processing to GPU with NVIDIA Kepler architecture

M.A. Krivov[1,3], N.U. Zakharov[1,3], S.G. Elizarov [2,3]

TTG Labs LLC[1], EPC of Lomonosov Moscow State University[2], Lomonosov Moscow State University[3]

The paper describes the process of optimizing the computational modules for image processing on a new generation of NVIDIA Kepler accelerators. The results of profiling these modules for different types of data and for three target GPU generations (Fermi, Kepler1 and Kepler2) are presented. We also demonstrate several schemes to adapt the algorithm's bottlenecks to each target architecture. Separately, the effect of application of each type of optimization and its dependence on the size of processed data is analyzed. The accomplished optimizations resulted in an increase of the service performance on the new GPU generation by 2.4-fold on average.

Keywords: graphics accelerators, GPU, software optimization, image processing, clouds

References

1. Pritula M.N., Krivov M.A., Elizarov S.G., Opyt razrabotki servisa dlya HDR-obrabotki video-dannykh v GPGPU-oblake [Development of the service for HDR video processing in a GPGPU-cloud] // Nauchnyy servis v seti Internet: poisk novykh resheniy: Trudy Mezhdunarodnoy superkomp'yuternoy konferentsii (17-22 sentyabrya 2012 g., g. Novorossiysk) [Scientific service on the Internet: search for new solutions: Proceedings of the International Scientific Conference (Novorossiysk, Russia, September 17-22, 2012)], 2012, pp. 287 – 291.
(Притула М.Н., Кривов М.А., Елизаров С.Г., Опыт разработки сервиса для HDR-обработки видео-данных в GPGPU-облаке // Научный сервис в сети Интернет: поиск новых решений: Труды Международной суперкомпьютерной конференции (17-22 сентября 2012 г., г. Новороссийск), 2012, с. 287 — 291)
2. Silva J., Ansorge R., Jena R., Efficient scatter-based kernel superposition on GPU // Journal of Parallel and Distributed Computing, Volume 84/C, 2015, pp. 15-23.
3. Podlozhnyuk V., Image Convolution with CUDA, NVIDIA Corporation, 2007.
4. Volkov V., Better performance at lower occupancy // Proceedings of the GPU technology conference (GTC 2010), 2010, p. 16.
5. Krivov M.A., Pritula M.N., Ivanov P.S., Primenenie tekhnologiy avtoadaptatsii programm dlya resheniya CFD zadach na strukturirovannykh setkakh s ispol'zovaniem GPU [Application of auto-adaptation technology to solve CFD problems on structured grids using GPUs] // Parallelnye vychislitelnye tekhnologii (PaVT'2014): Trudy mezhdunarodnoj nauchnoj konferentsii (Rostov-Na-Donu, 1-3 aprelya 2014) [Parallel Computational Technologies (PCT'2014): Proceedings of the International Scientific Conference (Rostov-Na-Donu, Russia, April 1-3, 2014)], 2014, pp. 108-117. (Кривов М.А., Притула М.Н., Иванов П.С., Применение технологий автоадаптации программ для решения CFD задач на структурированных сетках с использованием GPU // Параллельные вычислительные технологии (ПаВТ'2014): труды международной научной конференции (1–3 апреля 2014 г., г. Ростов-на-Дону), 2014, с. 108–117)