Опыт портирования среды для HDR-обработки изображений на GPU и APU

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

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

В статье описаны проблемы, с которыми столкнулись авторы при переносе открытого пакета LuminanceHDR на современные архитектуры с графическими ускорителями. Результатом переноса стали две версии пакета, разработанные с использованием технологий NVIDIA CUDA и OpenCL и отдельно оптимизированные для использования на GPU от NVIDIA и новых APU от AMD. На примере рассмотренной задачи проводится сравнение как методов оптимизации под соответствующие технологии, так и аппаратных решений от NVIDIA и AMD.

1. Введение

Одной из областей, в которой крайне востребованы современные подходы к параллельной организации вычислений, является обработка изображений. Благодаря тому факту, что практически во всех алгоритмах каждый пиксель изображения обрабатывается независимо, большинство программ из этой области обладают огромным потенциальным параллелизмом, что делает возможным их оптимизацию с применением различных параллельных вычислительных технологий, будь то использование векторных расширений x86-совместимых процессоров, задействование всех ядер SMP-систем, или же применение графических ускорителей.

В связи с тем, что разрешение матриц цифровых фотоаппаратов постоянно растет, поддержка разнообразных возможностей новых многоядерных процессоров и ускорителей становится крайне актуальной для различных графических пакетов типа Adobe Photoshop, GIMP и прочих. Причём если раньше параллельные реализации алгоритмов обработки изображений были просто ещё одним конкурентным преимуществом, то сейчас поддержка параллельных вычислителей становится жизненной необходимостью. Ведь в противном случае пользователи будут постоянно сталкиваться с ситуациями, когда обработка одной фотографии будет занимать от десятков секунд до нескольких минут.

Крупные компании вроде Adobe и Autodesk постоянно совершенствуют свои решения, распараллеливая используемые алгоритмы и внедряя новые технологии типа NVIDIA CUDA. Однако у разработчиков бесплатных графических пакетов, распространяемых под лицензиями типа GPL, обычно отсутствуют ресурсы и возможности для проведения качественной оптимизации своих программ, в результате чего пользователям приходится либо мириться с низкой скоростью их работы даже на современных процессорах, либо использовать различные ухищрения, жертвуя качеством обработки ради повышения производительности.

Одним из подобных открытых пакетов является среда LuminanceHDR [1], разрабатываемая Рафаэлем Мантиуком и содержащая набор алгоритмов для HDR-обработки изображений, суть которой заключается в изменении локального контраста в целях улучшения восприятия фотографии человеком. Благодаря качественным алгоритмам данный пакет завоевал определённую популярность среди фотографов, однако скорость его работы является предметом постоянной критики — обработка одной 12-мегапиксельной фотографии может занять несколько минут. При этом для получения качественного результата требуется как минимум несколько попыток подбора нужных параметров алгоритма, в результате чего обработка только одной фотографии длится десятки минут.

Компании ООО «ТТГ Лабс» было предложено провести оптимизацию данного пакета путём портирования на графические ускорители одного из самых популярных алгоритмов среды LuminanceHDR под названием Mantiuk06. В результате были созданы CUDA- и OpenCL-

версии требуемого алгоритма, что позволило многократно повысить скорость его работы, а также обеспечить эффективное использование всех возможностей новых вычислительных архитектур, будь то GPU, многоядерный CPU или APU.

2. Описание оптимизируемого пакета

LuminanceHDR является полноценным графическим пакетом с пользовательским интерфейсом, в котором реализованы два этапа обработки изображений: (1) построение по нескольким одинаковым фотографиям с разной экспозицией одной HDR-фотографии и (2) изменение локальной контрастности в HDR-снимке с его последующим переводом в обычную фотографию.

Суть первого этапа заключается расширении диапазона цветов. В обычных форматах изображений на каждый цветовой канал (красный, зелёный и синий) отводится по 8 бит, в результате чего удаётся сохранить только 256 градаций для каждого цвета. Все значения, которые не попадают в этот диапазон (например, более яркие цвета) игнорируются, что приводит к уменьшению реалистичности фотографии, т.к. глаз человека отлично замечает недостающие оттенки.

Предложено два способа обойти это ограничение. Первый заключается в использовании профессиональных фотоаппаратов, способных сохранять фотографии в специальных форматах, поддерживающих 10-12 и даже большее число бит на канал. Второй, реализованный в пакете LuminanceHDR, сводится к использованию нескольких фотографий одного и того же объекта с разной выдержкой. Далее, с помощью специального алгоритма эти фотографии «складываются» в одну HDR-фотографию, в которой на каждый цветовой канал выделяется по 32 бита, интерпретируемых как число с плавающей точкой одинарной точности. При этом качество результирующей фотографии уже целиком зависит от мастерства фотографа, подбирающего исходные снимки с подходящими значениями выдержки.

Второй этап обработки, осуществляемой в среде LuminanceHDR, состоит из обратного перевода HDR-фотографии в обычный формат, где на каждый цветовой канал снова приходится по 8 бит. Очевидно, что современные мониторы и дисплеи портативных устройств технически не в состоянии отобразить полноценное HDR-изображение, поэтому для дальнейшей его визуализации требуется сузить диапазон цветов, при этом минимизировав потери информации. Для этого существует множество алгоритмов, известных под общим названием Tone Mapping, которые изменяют локальный контраст отдельных участков фотографии, делая их более светлыми или тёмными в зависимости от изображённых на них объектах. Получившаяся в результате фотография хоть и станет менее правдоподобной, чем набор исходных снимков, но зато будет намного лучше воспринимается человеком, так как в ней не потеряна цветовая информация об объектах.

Для выполнения обратного перевода используются различные алгоритмы, каждый из которых имеет свои плюсы и минусы и лучше подходит для конкретных типов фотографий (города, пейзажи, фотопортреты и т.д.). В пакете LuminanceHDR реализовано девять подобных алгоритмов, наиболее популярным из которых стал метод Mantiuk06 [2], предложенный основным разработчиком пакета.

Возвращаясь к вопросам производительности, стоит отметить, что самым долгим является именно второй этап, в котором не только требуется выполнить множество операций над числами с плавающей точкой, но и подобрать правильные параметры алгоритма, что выливается в необходимость выполнения однотипной обработки несколько десятков раз. Поэтому было решено провести оптимизацию именно этого этапа как основного узкого места пакета.

Для иллюстрации базовых идей, заложенных в алгоритм Mantiuk06, следует описать схему его работы, которая состоит из трех стадий:

1) Выделение контраста из исходного изображения. С этой целью для каждого пикселя HDR-изображения вычисляется яркость, а под контрастом понимается логарифмическое отношение яркостей двух соседних пикселей. Другими словами, для каждой точки исходного изображения генерируется N вспомогательных точек контраста, где число N определяется исходя из количества рассматриваемых точек-соседей. В зависимости от параметров обработки

N может варьироваться от 2 до 20-30 [2].

2) Обработка контраста. Данная стадия является ключевым моментом описываемого алгоритма, и может быть по разному реализован в зависимости от требуемого вида обработки изображения. В оптимизируемом пакете на этой стадии вычисляется градиент от контраста, после чего полученное значение подставляется в некоторую линейную функцию, трактуемую в дальнейшем как градиент от нового контраста. Последующее численное интегрирование полученных значений позволяет восстановить новый контраст, который будет лучше восприниматься глазом человека.

3) Наложение обновлённого контраста на исходное изображение. Последним этапом является обновление исходного изображения с использованием нового контраста. Фактически такое обновление сводится к обратной задаче (по известным соотношениям яркости соседних точек необходимо восстановить исходное изображение), которая в общем случае не имеет единственного решения. В рассматриваемом алгоритме для выполнения этого шага строится специальный функционал, определяющий погрешность между некоторым произвольным и целевым изображениями, после чего с помощью итерационного метода подбирается такое изображение, функционал погрешности от которого не превышает 10^{-3}.

Легко заметить, что на всех стадиях алгоритма Mantiuk06 активно используются численные операции над большими массивами данных. Например, при обработке 10-мегапиксельной фотографии потребуется численное интегрирование дискретных функций, заданных массивами размера более 3000x3000, а итерационный алгоритм минимизации функционала погрешности сведется к построению пирамид изображений, для каждого уровня которого будут применяться такие операции, как дивергенция и градиент.

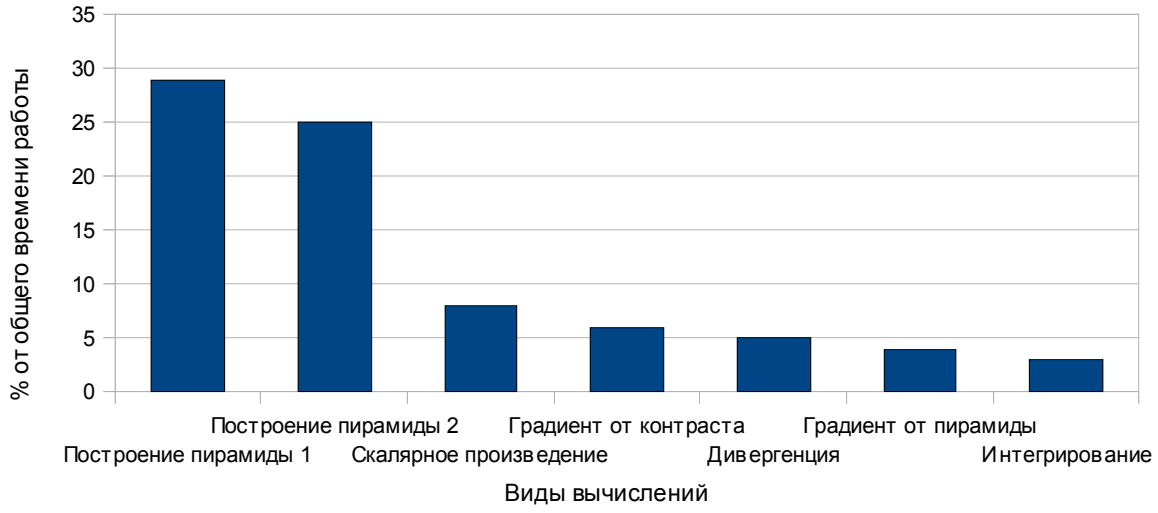

Рис. 1. Распределение времени работы алгоритма Mantiuk06 по основным операциям.

На рис. 1 приведена диаграмма распределения времени работы алгоритма по основным вычислительноёмким операциям, на которые в сумме приходится уходит 85% от общего времени работы программы. Наиболее ресурсоёмким оказался этап восстановления изображения по обновлённому контрасту — он отнимает более половины всего времени счета. Сложность остальных этапов практически равномерно распределена по оставшимся 35%, поэтому при портировании данного алгоритма на графические ускорители необходимо разработать GPU-версии для всех этапов обработки, так как в противном случае (при переносе на GPU только узких мест) потребуются частые пересылки больших объемов данных между CPU и GPU.

3. Проведённые оптимизации

Все изменения, которые были внесены в исходную реализацию алгоритма, логически можно разделить на две категории: (1) проведение общих модификаций, призванных повысить степень параллелизма и уменьшить количество вычислений без учёта особенностей

используемых технологий, и (2) разработка отдельных версий алгоритма с использованием технологий NVIDIA CUDA и OpenCL с последующей оптимизацией под конкретный вычислитель.

Если рассматривать первый тип изменений, то сразу стоит оговориться, что необходимость их проведения была вызвана недостаточным качеством исходного кода, что легко объяснить как бета-версией пакета, так и большим объёмом работ, выполняемых основным разработчиком LuminanceHDR, в результате чего на рефакторинг программы, скорее всего, просто не хватило времени. Основные проведённые модификации:

– Оптимизация работы с памятью. В исходном пакете активно использовались функции динамического выделения и освобождения памяти, в результате чего при обработке только одной фотографии блоки памяти размером в десятки мегабайт создавались и уничтожались в общей сложности более 10 000 раз. В принципе, для последовательной программы это не является особым минусом, так как объём системной памяти достаточно велик, а при её недостатке будет задействован файл подкачки. Однако при использовании графического ускорителя это может привести к нехватке памяти, в результате чего программа будет вынуждена аварийно завершиться. Особенно это важно при использовании современных APU, у которых объём доступной памяти ограничен 128 — 1024 Мбайт.

Авторами создан собственный менеджер памяти, который в начале обработки создаёт набор буферов, используемых в дальнейшем вместо динамически выделяемой памяти, а также переписан ряд функций с целью реализации более оперативного освобождения используемых буферов. Это позволило существенно уменьшить потребление памяти: так, для обработки фотографии размером 1600x1200 теперь достаточно всего 40 блоков памяти. Одновременно удалось избежать утечки памяти, которая наблюдалась в исходной реализации.

– Изменение пропорций пирамид. Как уже отмечалось, в алгоритме Mantiuk06 активно используется построение пирамид изображений, в которых каждый уровень в четыре раза меньше предыдущего. Основой подобных пирамид является исходное изображение, поэтому для некоторых форматов фотографий подобные пирамиды получались не выровненными — некоторые уровни имели нечётную длину или ширину. Как следствие, в коде во многих местах присутствовали проверки, призванные не допустить выход за пределы массивов при работе с такими уровнями, так как на краях массивов требовалась соответствующая логика работы.

При портировании на графический ускоритель подобные ветвления крайне не желательны, поскольку из-за специфики GPU будут выполнены все ветки каждого условного оператора, вследствие чего объём вычислений заметно возрастёт. Более того, из-за часто изменяющегося размера массивов не удастся реализовать выровненный доступ к памяти, что обернётся четырехкратной потерей производительности на ускорителях серии Tesla C1060 и двукратной – на Tesla C2050. Чтобы избежать этих проблем, было решено дополнять каждую обрабатываемую фотографию рамочкой, благодаря которой размер обрабатываемого изображения всегда позволит построить ровные пирамиды. К примеру, при обработке фотографии 500x375 на самом деле вычисления будут выполнены для изображения 512x384, которое потом будет обрезано до исходного формата.

– Оптимизация информационного графа программы. Отдельным видом оптимизации стал анализ информационного графа с целью последующего удаления избыточных обращений к памяти и вычислений. Благодаря этому удалось укрупнить некоторые циклы и функции, уменьшив число обращений к памяти и отказавшись от ряда вспомогательных переменных и промежуточных массивов. В дальнейшем при портировании алгоритма на графический ускоритель это позволило более активно использовать разделяемую/локальную память, так как после укрупнения функций во многих местах программы стало возможным поместить обрабатываемую область в кэш, и тем самым уменьшить количество обращений к глобальной памяти.

Ещё одним интересным моментом оказалось использование в исходном алгоритме быстрой сортировки, которую, как известно, на графическом ускорителе реализовать довольно сложно. При анализе графа выяснилось, что для работы требуются далеко не все значения отсортированного массива, а лишь некоторые из них, поэтому в GPU- и APU-версиях программы требуемую сортировку удалось заменить обычной параллельной свёрткой.

Проведённые модификации второго типа, по сравнению с только что описанными

являются, скорее, техническими, нежели концептуальными. В большинстве своём они сводились к опытному перебору нескольких альтернативных реализаций того или иного подхода и выбору оптимального варианта.

– Подбор оптимальной топологии данных. В зависимости от количества мультипроцессоров на графическом ускорителе требуется использовать различные разбиения данных на блоки, каждый из которых будет выполняться на отдельном мультипроцессоре. При этом не существует заранее известного оптимального размера блока, ведь этот параметр зависит от особенностей используемого GPU и размера обрабатываемых данных. Так, при крупном разбиении повышается скорость вычислений на отдельном мультипроцессоре, а при мелком гарантированно загружаются все ресурсы. Для дополнительного повышения скорости работы оптимальные размеры блоков были подобраны экспериментальным путём. Как показал ряд тестов, в среднем данный вид оптимизации позволил повысить скорость работы на 20-30% как в CUDA-, так и в OpenCL-версии.

– Использование разделяемой/локальной памяти. Одним из основных подходов к оптимизации программ для графических ускорителей является использование расположенной в GPU «быстрой» памяти, которую программист может использовать как буфер для временных результатов. Основная её особенность заключается в доступности записанных в неё значений для всех ядер одного мультипроцессора, что позволяет минимизировать количество чтений из глобальной памяти. В рассматриваемом алгоритме данный механизм оказался применим к четырём наиболее вычислительноёмким функциям. Так, при построении уровней пирамиды разделяемая/локальная память позволяет в четыре раза уменьшить число обращений к глобальной памяти, а в функциях вычисления градиента и дивергенции — в два раза.

– Использования аппаратной интерполяции. Поскольку графические ускорители долгое время разрабатывались исключительно для отображения 3D-объектов, то до сих пор даже в профессиональных вычислителях сохранился ряд атавизмов, изначально разработанных для растеризации геометрических объектов. Одним из таких атавизмов являются блоки аппаратной интерполяции, реализованные как возможность адресации массива дробными индексами. При подобном обращении к памяти результат будет являться линейной интерполяцией двух соседних элементов массива, причём данная операция будет выполнена за один такт. В ходе оптимизации данные блоки были использованы для масштабирования исходного изображения, что позволило отказаться от достаточно долгой операции, просто заменив схему индексации исходного массива.

4. Оценка ускорения на GPU

Версия пакета, оптимизированная для графических ускорителей, тестировалась на двух системах на базе GPU NVIDIA GeForce 580 GTX и GPU NVIDIA GeForce 555M. Оба ускорителя основаны на архитектуре Fermi, одной из основных особенностей которой является наличие кэша L1/L2. В первом GPU содержится 16 мультипроцессоров, суммарно состоящих из 512 CUDA-ядер, что обеспечивает пиковую производительность в 1,5 TFlops. Второй ускоритель разработан для ноутбуков, поэтому обладает пониженной частотой. Содержащиеся в нём 144 CUDA-ядра позволяют достичь 340 GFlops при вычислениях с одинарной точностью. Данные ускорители находятся в разных ценовых категориях, что позволяет оценить ускорение, которые гарантированно получат пользователи при использовании как высокопроизводительных рабочих станций, так и обычных компьютеров или ноутбуков.

На каждой системе проводилась обработка набора эталонных изображений с использованием пяти различных реализаций алгоритма. Две реализации используют технологию NVIDIA CUDA (с разделяемой памятью и без неё), ещё две — технологию OpenCL (с локальной памятью и без неё), а последняя является слегка оптимизированной версией исходного кода, исполняемого на одном ядре центрального процессора и скомпилированного с использованием Visual C++ 2008.

Результаты тестирования системы на базе GPU NVIDIA GeForce 580GTX и CPU Intel Xeon E3-1230 приведены на рис. 2.

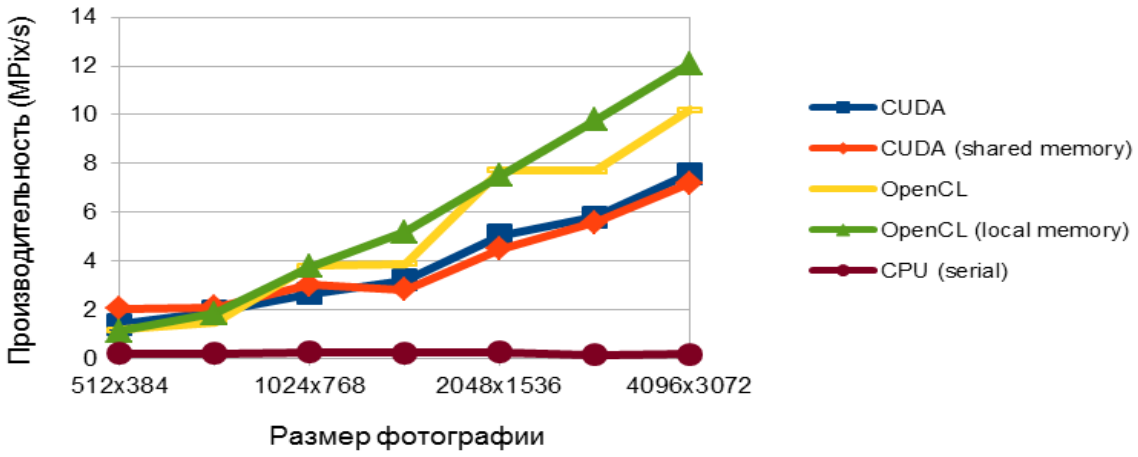

Рис. 2. Тестирование CUDA и OpenCL версий алгоритма на ускорителе NVIDIA GeForce 580 GTX.

Наиболее интересным моментом оказался рост производительности при увеличении размера обрабатываемой фотографии. Если для центрального процессора размер практически не сказывался на скорости работы, то графические ускорители обрабатывали большие изображения примерно в 10 раз быстрее, чем изображения небольших размеров. Это легко объяснить исходя из архитектурных особенностей GPU – чем больше фотография, тем больше порождается параллельных потоков, и тем более эффективно загружаются все вычислительные блоки.

Заслуживает отдельного комментария тот факт, что на данной задаче использование локальной/разделяемой памяти не всегда приводило к повышению производительности. Например, в технологии NVIDIA CUDA при обработке небольших фотографий программное кэширование обращений в глобальную память позволило получить 10-процентное ускорение. Однако на больших изображениях эта оптимизация лишь замедлила программу. В случае OpenCL-версии ситуация оказалась ещё более неоднозначной — в большинстве случаев локальная память давала заметное ускорение, но для некоторых форматов фотографий скорость обработки, наоборот, понижалась.

Также важно подчеркнуть, что оптимизация CUDA- и OpenCL-версий программы проводилась независимо и отчасти разными способами, поэтому более высокая скорость работы в последнем случае, скорее, является результатом больших усилий разработчика, чем отражением особенностей используемой технологии.

В качестве последнего замечания к рис. 2 следует привести итоговое ускорение относительно центрального процессора. Оно находилось в диапазоне от 7 до 60 раз, что является хорошим результатом для задачи, скорость решения которой упирается в пропускную способность глобальной памяти. Также стоит отметить, что на других тестовых данных, имеющих более «правильный» для GPU размер, авторами было достигнуто 100-кратное ускорение. Если перевести озвученные значения во время ожидания пользователем окончания обработки одного изображения, то для 12-мегапиксельной фотографии он уменьшилось с 76 до 1,2 с.

Аналогичное тестирование было проведено на другой системе, состоящей из GPU NVIDIA GeForce 555M и CPU Intel Core i7 2670 (рис. 3). На данном графике наблюдается аналогичное поведение производительности всех версий в зависимости от размера сеток. Единственным отличием является более заметное доминирование CUDA-версий программы над OpenCL-реализациями алгоритма при обработке небольших фотографий (размером до 1024x768). Поэтому в данной задаче при использовании ускорителей от NVIDIA имеет смысл использовать OpenCL-и CUDA-версии совместно, выбирая оптимальный вариант в зависимости от входных данных. Ускорение по сравнению с одним ядром CPU на данной системе составило от 5,5 до 20 раз.

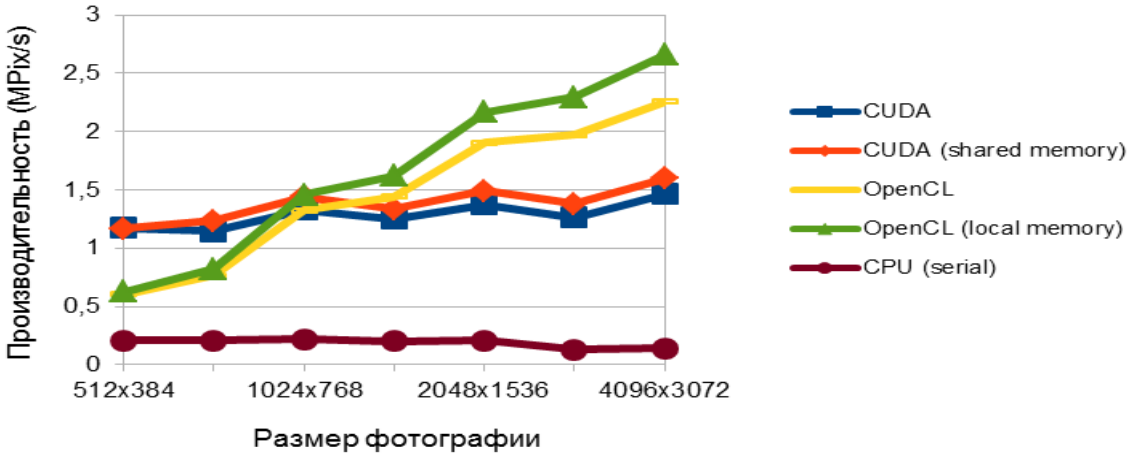

Рис. 3. Тестирование CUDA и OpenCL версий алгоритма на ускорителе NVIDIA GeForce 555M.

5. Оценка ускорения на APU

В настоящее время в качестве одной из альтернатив графическим ускорителям рассматриваются системы на базе гибридных процессоров, в состав которых входят как ядра CPU, так и программируемые потоковые процессоры GPU. Гибридные процессоры разрабатываются всеми крупнейшими производителями: AMD (Fusion), NVIDIA (Project Denver) и Intel (Ivy Bridge).

На момент написания данной статьи в свободном доступе имелись только решения от компании AMD, известные как Accelerated Processing Unit (APU). Основным их достоинством является общая память, в результате чего отсутствует необходимость в пересылке данных по шине PCI-Express, что в перспективе позволит существенно ускорить GPGPU-программы. К сожалению, в текущей версии это оказывается, скорее, недостатком — программная модель OpenCL не позволяет эффективно задействовать общую память при активной записи в неё из каждого типа вычислителя [3], а объём выделенной памяти для GPU-ядер лимитирован 128-1024 Мбайт в зависимости от модели APU и версии BIOS. Другим недостатком является ограниченность пропускной способности оперативной памяти, которая не только ниже, чем у памяти дискретного графического ускорителя, но и совместно используется и CPU- и GPU-ядрами.

Другими словами, хотя с точки зрения программиста архитектура гибридных процессоров практически не отличается от архитектуры обычного графического ускорителя и при этом обладает рядом аппаратных недостатков, ее ожидают неплохие перспективы в случае появления адаптированных под неё программных моделей и интеграции CPU- и GPU-ядер в последующих версиях APU.

Ниже приведены результаты тестирования двух систем на базе подобных процессоров, один из которых имеет 4 ядра CPU и 400 ядер GPU (суммарно порядка 0,5 TFlops), а второй — 2 ядра CPU и 80 ядер GPU (порядка 0,1 TFlops). При тестировании GPU-ядер использовались две OpenCL-версии пакета (с локальной памятью и без неё), а для оценки производительности CPU-ядер были задействованы OpenCL-версия и исходная последовательная реализация алгоритма, скомпилированная при помощи Visual C++ 2008. Результаты тестирования APU AMD A8-3850 приведены на рис. 4.

Стоит отметить, что на системах данного типа использование локальной памяти практически не сказалось на общей производительности при вычислениях на GPU-ядрах. В остальном встроенный графический ускоритель ведёт себя так же, как и его дискретный аналог: чем больше размер фотографий, тем выше скорость обработки. При использовании ядер центрального процессора выигрыш от OpenCL-версии неоднозначен. С одной стороны, она позволяет задействовать все ядра, а с другой, ускорение — по сравнению с последовательной

версией программы оказалось лишь двукратным при использовании четырёх ядер.

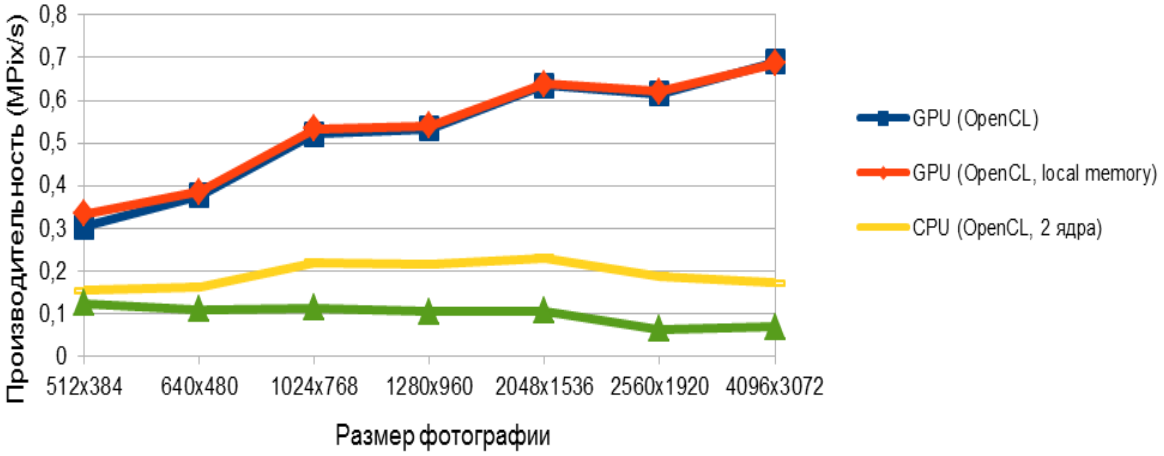

Рис. 4. Тестирование OpenCL версии алгоритма на AMD APU A8-3850.

При проведении аналогичного тестирования на другом APU (AMD E-350) были получены результаты, практически полностью аналогичные предыдущим, за исключением отсутствия возможности провести тестирование GPU-ядер на больших фотографиях. Как было отмечено ранее, максимальный объём памяти APU довольно мал (в данной модели — 128 Мбайт), поэтому для обработки фотографий размера 2048x1536 и более глобальной памяти попросту не хватает. Потенциальным решением является использование общей системной памяти, однако в этом случае скорость доступа к ней будет в десятки раз ниже, чем к глобальной памяти.

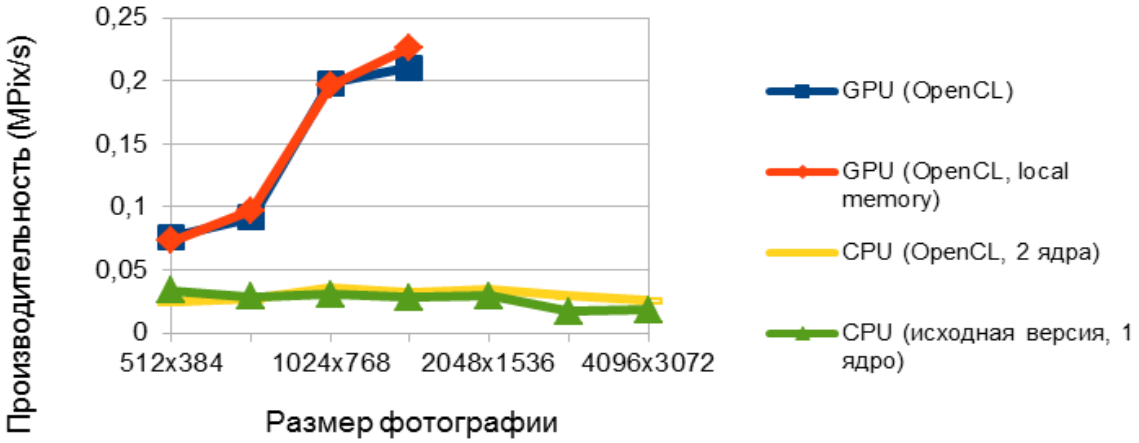

Рис. 5. Тестирование OpenCL версии алгоритма на AMD APU E-350.

Подводя итоги, стоит отметить, что на ускорителях типа APU удалось достичь 3-10-кратного ускорения, причём оно оказалось одинаковым как на самой производительной модели AMD A8-3850, так и на более массовой AMD E-350.

6. Оценка ускорения на многоядерных CPU

Одним из преимуществ технологии OpenCL является тот факт, что написанная на ней программа может быть запущена как на графических ускорителях, так и на многоядерных центральных процессорах. В последнем случае, по заявлению производителей OpenCL-драйверов, которыми являются компании Intel и AMD, будут также задействованы векторные

расширения процессора (SSE или AVX), что принесёт дополнительное ускорение. Впрочем, авторами [4] показано, что даже в синтетических тестах производительность программ на OpenCL, использующих центральный процессор, оказывается крайне низка, что можно объяснить меньшим временем, выделяемым на оптимизацию. В то время как обычный компилятор может потратить практически любое время на анализ исходного кода, драйвер OpenCL должен провести компиляцию практически аналогичной программы, имея на это всего несколько миллисекунд.

На рис.6 приведены результаты сравнения производительности параллельной OpenCL-версии программы и ее последовательной реализации, скомпилированной при помощи Visual C++ 2008. В качестве тестовых машин были выбраны две серверные платформы, одна из которых оснащена двумя 16-ядерными процессорами от AMD, а вторая — 4-ядерным процессором Intel Xeon E3. Так как обе модели вычислителей поддерживают расширения AVX (позволяющие обрабатывать за такт 8 чисел с плавающей точкой одинарной точности), то была использована реализация OpenCL от Intel, являющаяся не только самой быстрой, но и адаптированной для работы с подобными процессорами.

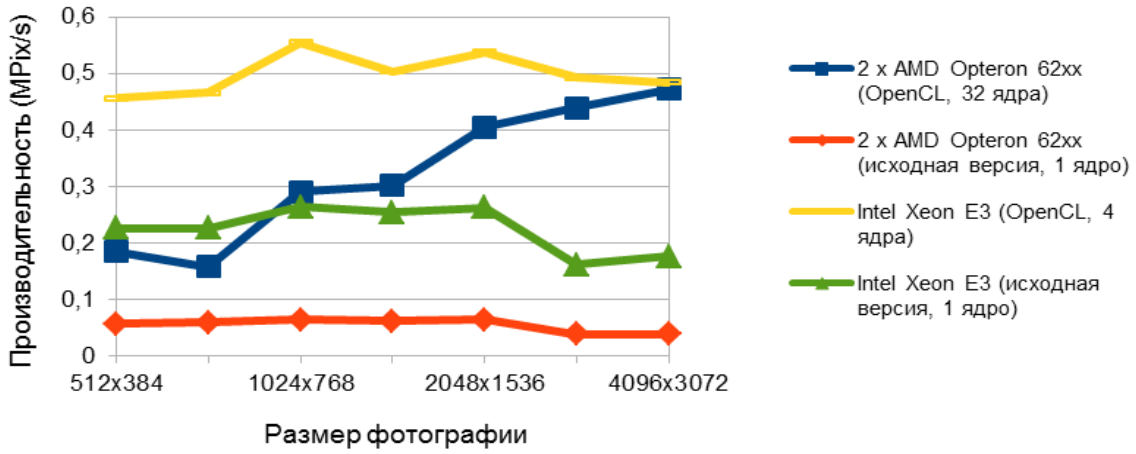

Рис. 6. Тестирование OpenCL-версии алгоритма на многоядерных центральных процессорах.

Результаты тестирования оказались довольно неожиданными. Если сравнивать процессоры от Intel и AMD, то следует объявить о безусловной победе первых. Один 4-ядерный процессор компании Intel с пиковой производительностью порядка 100 GFlops на всех тестовых фотографиях оказался быстрее, чем два 16-ядерных процессора от AMD с пиковой производительностью 560 GFlops. При этом скорость работы программы на 32-ядерной системе практически линейно возрастала с ростом размера картинок, что наводит на мысль о проблемах доступа к памяти при столь большом количестве ядер. Данная система имеет разделённый кэш, поэтому при частых операциях чтениях и записи в один и тот же участок памяти требуется синхронизация кэшей разных процессоров, что, возможно, и объясняет столь низкую скорость работы.

Другим интересным фактом является то, что последовательная версия программы на 4-ядерном процессоре всего в 2-3 раза медленнее, чем параллельная, и в 3,5-10 раз медленнее на 32-ядерной системе. Это лишний раз свидетельствует о том, что на данный момент статические компиляторы создают более быстрые версии программ, чем драйвер OpenCL, компилирующий аналогичные программы динамически.

В заключение стоит отметить, что авторам при ручной оптимизации ряда тестовых программ удавалось на подобных системах достичь 40-50% от пиковой производительности [5], поэтому полученные выше результаты стоит рассматривать лишь как проверку эффективности OpenCL-драйвера для центральных процессоров.

7. Дальнейшее развитие

В результате проведенных работ было создано две оптимизированные реализации алгоритма Mantiuk06, которые на тестовых системах оказались в 5-60 раз быстрее исходной версии. Дальнейшее развитие этой работы планируется осуществить, как минимум, в двух направлениях:

– Разработка гибридной версии алгоритма, которая самостоятельно определяет, на каком вычислителе (CPU или GPU) и какую именно реализацию (CUDA или OpenCL) выгоднее использовать для обработки конкретной фотографии. Как было показано в предыдущих разделах, выбор оптимальной ветки не всегда является очевидным, и в зависимости от архитектурных особенностей динамическое переключение между созданными реализациями позволит повысить суммарную скорость работы на десятки процентов.

– Повышение стабильности и опубликование патча для исходной среды. В настоящий момент во всех созданных версиях алгоритма не реализованы проверки на ряд ошибок времени выполнения типа недостатка памяти, отсутствия требуемых библиотек и OpenCL-драйверов и т.д.. После добавления соответствующих проверок и проведения дополнительного тестирования планируется формализация всех выполненных наработок в виде открытого патча для последней версии среды LuminanceHDR.

Литература

1. Электронный ресурс http://qtpfsgui.sourceforge.net.
2. Mantiuk R., Myszkowski K., Seidel H.-P., A Perceptual Framework for Contrast Processing of High Dynamic Range Images, ACM, 2006.
3. AMD Accelerated Parallel Processing OpenCL™ Programming Guide (v1.3c). Электронный ресурс http://developer.amd.com/sdks/AMDAPPSDK/documentation/Pages/default.aspx
4. Кривов М.А., Зелёно-сине-красная OpenCL // Журнал "Суперкомпьютеры", Осень 2011, с. 47-50.
5. Кривов М.А., Казеннов А.М. Портируем на GPU и оптимизируем для CPU // Журнал "Суперкомпьютеры", Весна 2011, с. 43-45.