УДК 004.627:004.932.2

СЖАТИЕ ЭКРАННОГО ВИДЕО С ПОМОЩЬЮ ВИДЕОКАРТЫ. СРАВНЕНИЕ ТЕХНОЛОГИЙ

Д. В. Дружинин[1]

Рассмотрены два алгоритма сжатия видео происходящего на экране пользователя, часть операций которых выполняется на видеокарте. Сравниваются две технологии, позволяющие получить доступ к мощностям видеокарты: пиксельные шейдеры и NVidia CUDA. Приведены результаты тестирования алгоритмов, реализованных на основе этих технологий. Рассмотрены перспективы дальнейшего, более обширного применения пиксельных шейдеров и технологии NVidia CUDA при сжатии экранного видео.

Введение. Сжатие видео — одна из наиболее трудоемких по времени задач, которые приходится решать не только профессионалам, но и рядовым пользователям. Сжатие потокового видео высокого разрешения на классе компьютеров, используемых рядовым пользователем, при помощи одного только центрального процессора чрезвычайно трудоемко. Наилучшим вариантом был бы запуск процесса сжатия видео высокого разрешения в фоновом режиме, что позволило бы пользователю продолжать работу на своем компьютере. Использование видеокарты для выполнения некоторых этапов сжатия позволяет разгрузить центральный процессор при сжатии потокового видео.

В настоящее время значительная часть персональных компьютеров оснащена дискретными видеокартами. Производительность современной видеокарты измеряется сотнями гигафлопов, в то время как производительность центрального процессора (ЦП) — десятками гигафлопов. Архитектура видеокарты такова, что подавляющая часть транзисторов сосредоточена на alu (arithmetic logic unit — арифметико-логическое устройство), и лишь незначительная часть транзисторов обеспечивает кэширование и выполнение инструкций потока управления. Таким образом, архитектура видеокарты рассчитана на интенсивные арифметические и логические вычисления, но плохо приспособлена для алгоритмических задач с большим количеством условных переходов.

Существует несколько технологий, позволяющих использовать вычислительные мощности видеокарты. С момента появления пиксельных и вершинных шейдеров, которые стали исторически первыми технологиями, позволяющими получить доступ к мощностям видеокарты, активно развивается такое направление программирования как вычисления общего назначения на видеокарте (GPGPU — General-Purpose Computing on Graphics Processing Units).

Имеются примеры использования видеокарты для обработки изображений. В [9] приведены данные о сравнении производительности пиксельных шейдеров и ЦП при выполнении нескольких алгоритмов обработки изображений. Работа интересна тем, что рассматриваемые алгоритмы ранжированы по таким параметрам как интенсивность обращения к памяти, количество условных переходов, объем математических вычислений, возможность векторной обработки. Такой подход позволил авторам сделать определенные выводы об эффективности выполнения нескольких типов алгоритмов на видеокарте по результатам тестирования нескольких алгоритмов, отнесенных к различным типам. В [10] сравнивается производительность пиксельных шейдеров и ЦП при выполнении алгоритма преобразования изображения из пространства цветов YCoCg(-R) в пространство цветов RGB. В [14] описана реализация алгоритма сжатия DXT1 с помощью технологии NVidia CUDA, а также приводятся результаты тестирования, которые показывают многократное ускорение CUDA-реализации этого алгоритма по сравнению с ЦП-реализацией. В [15] описаны CUDA-реализации двух алгоритмов подавления шума в изображении: фильтр ближайших соседей (Nearest Neighbors Filter) и фильтр нелокальных средних (Non-Local Means Filter). Кроме того, приводятся результаты тестирования CUDA-реализаций этих алгоритмов.

Проводятся активные исследования по применению видеокарты для сжатия данных. В [7] описывается метод сжатия больших изображений, полученных лазерным сканированием трехмерных объектов; декодирование осуществляется с помощью вершинных шейдеров. В [5, 6] рассматриваются способы декодирования видео высокого разрешения с помощью видеокарты; создан конвейер, в котором первые

[1] Томский государственный университет, факультет информатики, просп. Ленина, 36, корп. 2, 634050, г. Томск; e-mail: dendru@rambler.ru

© Научно-исследовательский вычислительный центр МГУ им. М.В. Ломоносова

стадии обработки выполняются ЦП, а последующие — пиксельными и вершинными шейдерами. Авторы работы [13] реализуют значительную часть декодирования кодека Дирака [20] с помощью технологии NVidia CUDA, добиваясь многократного ускорения в работе декодера по сравнению с ЦП-реализацией. В [4] описывается реализация алгоритма оценки движения с помощью пиксельных шейдеров и приводятся результаты тестирования, демонстрирующие ускорение относительно ЦП-реализации. В [8] описано сканирование трехмерных объектов, которое включает в себя компенсацию движения, реализованную с помощью вершинных и пиксельных шейдеров. В [11] применяется алгоритм компенсации движения, реализованный с помощью вершинных и пиксельных шейдеров, для цифровой рентгенографии кровеносных сосудов. Авторы работы [12] используют компенсацию движения, реализованную с помощью пиксельных шейдеров, как этап при сжатии видео, снимаемого одновременно несколькими камерами с разных позиций.

Одним из типов видеоданных, сжатие которых необходимо осуществлять в режиме реального времени, является видео происходящего на экране пользователя. Как правило, это видео высокого разрешения. В настоящей статье рассмотрены два алгоритма сжатия видео без потери информации, обладающие линейной трудоемкостью и демонстрирующие высокий коэффициент сжатия. Одним из требований к программам записи экранного видео является возможность их запуска в фоновом режиме. Поэтому при его сжатии для разгрузки ЦП часть операций имеет смысл перенести на видеокарту. В первом алгоритме на видеокарту переносится выполнение попиксельного сравнения двух изображений на равенство, а во втором алгоритме — выполнение поблокового сравнения двух изображений на равенство.

Существует множество программных продуктов, позволяющих записывать в сжатом виде экранное видео на жесткий диск. Однако в большинстве случаев — это коммерческие программные продукты, поэтому детали реализованных алгоритмов сжатия не раскрываются.

Удалось найти подробное описание двух алгоритмов сжатия экранного видео. Один из этих алгоритмов использует компенсацию движения (см. [18]). Второй — алгоритм, используемый TightVNC [21] для сжатия. Сравнение этих двух алгоритмов с алгоритмом, основанном на попиксельном сравнении, приведено в разделе 2. Работ по сжатию экранного видео с помощью видеокарты автору настоящей статьи найти не удалось.

Рассматриваемые здесь алгоритмы при реализации с помощью видеокарты требуют интенсивного обмена данными с видеопамятью (в обоих направлениях). Поэтому они предназначены для выполнения на видеокартах, подключенных через интерфейс PCI Express.

1. Обзор используемых технологий. В этом разделе дано сравнение возможностей пиксельных шейдеров и технологии NVidia CUDA для выполнения общих вычислений.

1.1. Пиксельные шейдеры. Пиксельные шейдеры являются программами на Си-подобном языке программирования HLSL (High Level Shader Language) и выполняются на процессоре видеокарты без участия центрального процессора. Пиксельный шейдер выполняется для каждого пикселя изображения; результатом его однократного выполнения является задание цвета соответствующего пикселя изображения [3].

Этот раздел построен следующим образом. Сначала рассматриваются преимущества и недостатки ps_2_0 (pixel shader version 2_0), а затем указываются отличия ps_3_0 от ps_2_0 и отличия ps_4_0 от ps_3_0.

1.1.1. Пиксельный шейдер ps_2_0. Преимущество: пиксельные шейдеры поддерживаются практически на всех современных видеокартах.

Недостатком является то, что способ распараллеливания жестко фиксирован — шейдер выполняется один раз для каждого пикселя результирующей текстуры, причем предполагается, что изменяться будут только те байты результирующей текстуры, которые соответствуют этому пикселю. Такой подход далеко не всегда удобен и эффективен.

Кроме того, существует ряд ограничений на формат результирующей текстуры в пиксельных шейдерах. Например, при использовании пиксельных шейдеров совместно с DirectX9.0c под управлением ОС Windows XP формат текстуры, в которую производится отрисовка, строго регламентирован, а однобитовый формат не поддерживается. Поэтому часто приходится передавать из памяти видеокарты в оперативную память в несколько раз больше данных, чем необходимо.

1.1.2. Отличия различных версий пиксельных шейдеров. Отличия ps_2_0 и ps_3_0 (наиболее существенные при выполнении вычислений общего назначения):

1) ps_3_0 поддерживает значительно большее число слотов инструкций и выполняемых инструкций по сравнению с ps_2_0;

2) в ps_3_0 по умолчанию все операции выполняются с максимальной точностью (не ниже 32 бит) [16].

Отличия ps_4_0 от ps_3_0 (наиболее существенные при выполнении вычислений общего назначения):

1) в ps_4_0 целые числа (int и uint) отображаются в 32-битовые целые числа на видеокарте, тогда как в ps_3_0 и более ранних версиях пиксельных шейдеров целые числа отображаются в вещественные числа на видеокарте (например, при передаче в шейдер текстуры, цвета пикселей которой закодированы целыми числами);

2) в отличие от ps_3_0, в ps_4_0 имеются побитовые операторы для целых типов int и uint [17].

Таким образом, с возрастанием версии пиксельных шейдеров наблюдается расширение возможностей для выполнения вычислений общего назначения. Отметим, что программируемые конвейеры Direct3D (в том числе и Direct3D 10) проектировались для отрисовки изображений, что значительно сужает их область применения для общих вычислений.

Для реализации попиксельного сравнения двух изображений на равенство были выбраны пиксельные шейдеры ps_2_0, так как эта версия поддерживается на большем числе видеокарт. Для реализации поблокового сравнения двух изображений на равенство были задействованы пиксельные шейдеры ps_3_0, так как в ps_2_0 количество выполняемых инструкций оказалось недостаточным.

1.2. NVidia CUDA (Compute Unified Device Architecture). Технология NVidia CUDA (как и пиксельные шейдеры) позволяет исполнять программы на Си-подобном языке на процессоре видеокарты без привлечения центрального процессора. С помощью этой технологии можно решать практически любые задачи, связанные с трудоемкими математическими вычислениями. Большое количество потоков на видеокарте могут выполняться параллельно. Потоки объединены в блоки. Каждый блок содержит равное количество потоков. При вызове функции, выполняемой непосредственно процессором видеокарты, эта функция выполняется тем количеством потоков, которое указано при ее вызове, при этом часть работы, выполняемая некоторым потоком, определяется по номеру блока и номеру потока в этом блоке.

Недостаток: технология NVidia CUDA поддерживается только на видеокартах серии 8 фирмы NVidia. Фирма AMD создала собственную технологию CTM (Close To Metal), которая не рассматривается в данной работе.

Преимущество этого подхода состоит в том, что модель памяти технологии CUDA включает несколько типов памяти, в том числе разделяемую память и константную память. Разделяемая память (shared memory) является внутрипроцессорной, поэтому скорость доступа к ней намного выше, чем к глобальной памяти. Каждому блоку потоков выделяется определенное количество разделяемой памяти. Константная память оптимизирована для доступа на чтение. Этот тип памяти кэшируемый, что в среднем ускоряет чтение данных, находящихся в константной памяти. Наличие этих двух типов памяти позволяет ускорить выполнение алгоритмов, требующих большого количества обращений к памяти.

Кроме того, программист сам определяет количество потоков, которые будут выполнять указанную функцию, при этом любой поток может обращаться на чтение и запись к любому байту обрабатываемых данных (отсутствует жесткое разделение на входные и выходные данные).

И наконец, программные модули, написанные с использованием технологии NVidia CUDA, легко встраиваются в обычное приложение, написанное на Си++ [2]. Для этого не требуется задействовать дополнительные программные средства, такие как DirectX или OpenGL.

Следовательно, эта модель программирования четкая и привычная для программиста. Технология CUDA более гибкая, чем пиксельные шейдеры, так как позволяет в некоторой мере управлять процессом распараллеливания потоков, а также варьировать используемые типы памяти.

Для реализации был выбран уровень CUDA Runtime как обеспечивающий оптимальный баланс предоставляемых возможностей и простоты разработки [1].

2. Алгоритмы сжатия видео происходящего на экране пользователя. Описанные ниже два алгоритма сжатия видео, основанные на сравнении изображений, были разработаны нами на основе общих идей сжатия видео, изложенных в [19].

2.1. Алгоритм, основанный на попиксельном сравнении изображений. Ключевой кадр (каждый десятый) кодируется независимо и записывается в выходной файл. Для каждого промежуточного кадра в выходной файл записываются номера строк и столбцов, в которых есть изменившиеся пиксели относительно ключевого кадра, а затем — цвета пикселей, находящихся на пересечении этих строк и столбцов. Цвета пикселей, находящихся на пересечении этих строк и столбцов, образуют изображение меньшего размера, которое можно сжимать теми же алгоритмами, что и ключевой кадр. Остальную часть промежуточных кадров можно восстановить по ключевому кадру. Это — алгоритм сжатия без потери информации, обладающий линейной трудоемкостью, что немаловажно для алгоритмов сжатия потокового видео.

Сравнение промежуточных кадров с ключевым с целью выявления совпадающих и различающихся пикселей выполняется на видеокарте. Входные данные этой операции — два изображения одинакового

размера; назовем ее попиксельной операцией xor (eXclusive OR). Выходные данные — набор элементов, количество которых равно количеству пикселей во входном изображении. Каждый такой элемент является индикатором равенства или неравенства цветов двух пикселей, поэтому в идеале это один бит.

2.2. Алгоритм, основанный на поблоковом сравнении изображений. Отличие этого алгоритма от алгоритма, основанного на попиксельном сравнении, заключается в том, что в выходной файл записываются те блоки пикселей промежуточного кадра, в которых есть хотя бы один изменившийся пиксель относительно соответствующего блока ключевого кадра. Кроме того, в выходной файл записываются номера изменившихся блоков. Изображение меньшего размера, составленное из изменившихся блоков пикселей, можно сжимать теми же алгоритмами, что и ключевой кадр.

Рассмотрим принципы выбора размера блока. При увеличении размера блока уменьшаются накладные расходы, связанные с хранением номеров изменившихся блоков, но уменьшается и точность определения изменившейся области (и наоборот). Был выбран размер блока 8×8 пикселей, так как при таком размере блока на большинстве тестов была достигнута максимальная степень сжатия (с учетом количества передаваемых номеров блоков).

С помощью видеокарты выполняется сравнение промежуточных кадров с ключевым с целью выявления совпадающих и различающихся блоков (поблоковая операция xor). Входные данные те же, что и при попиксельной операции xor. Существенное отличие состоит в значительно меньшем (в $2^6 = 8 \times 8$) размере выходного массива, так как каждый его элемент является индикатором равенства или неравенства двух блоков пикселей. В идеале один элемент результирующего массива составляет один бит.

2.3. Некоторые детали реализации попиксельной и поблоковой операции xor с помощью пиксельных шейдеров и NVidia CUDA. В этом разделе все размеры результирующих массивов указаны при ширине и высоте исходных текстур в 1024 и 768 пикселей соответственно.

Реализация попиксельной операции xor с помощью пиксельных шейдеров. Каждый элемент, который является индикатором равенства или неравенства цветов двух пикселей, при использовании пиксельных шейдеров занимает один байт. Однобитовый формат текстуры, являющейся результатом отрисовки, не поддерживается в DirectX9.0c под Windows XP. Однобайтовый формат не поддерживается видеокартой, которая использовалась при тестировании, поэтому используется двухбайтовый формат D3DFMT_R5G6B5. Однако была применена следующая техника: каждый байт в двухбайтовом значении цвета пикселя результирующей текстуры используется независимо для кодирования результата операции xor. Это достигается за счет независимого использования каналов цвета R- и B-пикселя результирующей текстуры. Таким образом, размер результирующей текстуры составляет $1024 \times 768 = 786432$ байтов.

Реализация поблоковой операции xor с помощью пиксельных шейдеров. Формат результирующей текстуры был выбран тот же, что и при попиксельном xor — D3DFMT_R5G6B5. Однако поскольку независимое использование каждого байта в этом двухбайтовом формате привело к замедлению в работе алгоритма, то было принято решение не использовать эту технику при поблоковом xor, с учетом того, что размер результирующей текстуры и так невелик: $(1024 \times 768 \times 2)/64 = 24576$ байтов.

Реализация попиксельной операции xor с помощью CUDA. Для хранения результата выполнения функции, реализующей операцию xor, был использован однобитовый формат; при этом один поток обрабатывает восемь пар подряд идущих пикселей и сохраняет результат в одном байте выходного массива. Использование однобитового формата выходного массива позволяет уменьшить время его копирования из видеопамяти в оперативную память, так как в этом случае размер выходного массива в восемь раз меньше, чем при использовании однобайтового формата, и составляет $(1024 \times 768)/8 = 98304$ байтов.

Для уменьшения количества вычислений программа, выполняемая на видеокарте, оперирует с цветом пикселя исходной текстуры как со значением типа unsigned int. Условный переход является одной из самых дорогих операций при выполнении на видеокарте, поэтому реализация попиксельной операции xor не использует условных переходов. Для этого сначала вычисляется разница $difference$ между максимальным и минимальным из двух рассматриваемых значений цветов пикселей, а затем текущий бит в байте, который будет результатом выполнения потока, вычисляется следующим образом:

$$output \mathrel{|}= (difference \mathbin{\&\&} 1) \times byteMask;$$

Результатом выполнения операции $difference \mathbin{\&\&} 1$ будет 1, если разница $difference$ не равна 0, и 0, если разница $difference$ равна 0. В маске $byteMask$ единице равен только один бит (соответствующий текущему биту в байте результата). Таким образом, если цвета сравниваемых пикселей равны (разница $difference$ равна 0), то текущий бит в байте результата $output$ останется равным 0, иначе он будет установлен в 1.

Реализация поблоковой операции xor с помощью CUDA. Для ее реализации применены все те же

способы оптимизации, что и при попиксельном xor. Однако здесь появляются условные переходы, которые необходимы, чтобы остановить попиксельное сравнение при обнаружении первой пары неравных пикселей в блоке. Существенная экономия времени достигается, если все потоки в warp (потоки, которые выполняются одновременно и для которых гарантировано, что они будут выполнять одну и ту же инструкцию в один и тот же такт времени) вышли из цикла проверки пикселей блока досрочно. Иначе потоки, которые вышли из цикла проверки пикселей блока досрочно, вынуждены будут ждать те потоки, которые выполнят проверку до конца. Тестирование показало, что использование условных переходов для выхода из цикла попиксельного сравнения при обнаружении первой пары неравных пикселей в блоке несколько ускоряет выполнение операции поблокового xor (примерно на 0.3 мс, что составляет около 10 % от времени непосредственного выполнения поблокового xor, не использующего условные переходы). Размер результирующего массива составляет $(1024 \times 768)/(8 \times 64) = 1536$ байта.

2.4. Сравнение алгоритмов, основанных на попиксельной и поблоковой операции xor. Выше были описаны два алгоритма, основанные на сравнении изображений. Оба этих алгоритма выполняются за близкое время (см. раздел 3).

В некоторых случаях лучшие результаты по степени сжатия показывает алгоритм, основанный на попиксельном сравнении:

1) пользователь вводит текст;

2) пользователь сворачивает окно или, наоборот, открывает свернутое.

В других случаях более высокую степень сжатия демонстрирует алгоритм, основанный на поблоковом сравнении:

1) скроллинг;

2) перемещение окна на некоторое расстояние по направлению, близкому к диагональному.

Таблица 1

Результаты тестирования алгоритмов, основанных на попиксельном и поблоковом сравнении при сворачивании окна и перемещении окна по диагонали

Тип изменения экрана / алгоритм	Попиксельное сравнение	Поблоковое сравнение
Сворачивание окна	921600	997632
Перемещение окна по диагонали	673554	476928

Рассмотрим более подробно два типа изменений на экране пользователя. Было установлено, что алгоритм, основанный на попиксельном сравнении, демонстрирует более высокую степень сжатия при таком изменении промежуточного кадра относительно ключевого, когда пользователь сворачивает окно или открывает свернутое (табл. 1). В этом случае алгоритм, основанный на попиксельном сравнении, выявит и передаст на следующий этап сжатия в точности ту область, которая была изменена, тогда как алгоритм, основанный на поблоковом сравнении, будет считать измененной область, охватывающую реально изменившуюся область, так как блок размером 8×8 пикселей считается измененным, если изменился хотя бы один пиксель в блоке. Иными словами, по периметру реально изменившейся области будет захвачена "буферная зона", толщиной максимум в семь пикселей, из неизменившейся области.

Пусть $T_{\text{ср}} = T_{\max}/2 = 7/2 \approx 4$, где $T_{\text{ср}}$ — это толщина буферной зоны в среднем, а $T_{\max}$ — максимальная толщина буферной зоны. Тогда среднее количество неизменившихся пикселей $N_{\text{ср}}$ из "буферной зоны" вокруг изменившейся области, которые алгоритм, основанный на поблоковом сравнении, будет считать изменившимися, можно посчитать по следующей формуле: $N_{\text{ср}} = PT_{\text{ср}} + \text{const}$, где P — это периметр реально изменившейся области, измеряемый в пикселях, а const — некоторая небольшая константа, определяемая формой изменившейся области (например, в случае прямоугольной изменившейся области эта константа будет равна $4 \times 4 \times 4$, то есть 4 угловых квадрата размером 4×4 пикселя).

Алгоритм, основанный на поблоковом сравнении, показывает более высокий коэффициент сжатия при таком изменении промежуточного кадра относительно ключевого, когда пользователь перемещает окно на некоторое расстояние по направлению, близкому к диагональному (табл. 1). Для определенности предположим, что окно было перемещено сверху вниз и слева направо (см. рисунок). В этом случае алгоритм, основанный на попиксельном сравнении, будет считать изменившейся областью прямоугольник, координаты верхнего левого угла которого совпадают с координатами левого верхнего угла окна на ключевом кадре (до движения), а координаты правого нижнего угла совпадают с координатами правого нижнего угла окна на промежуточном кадре (после движения). Назовем такую область охватывающим прямоугольником. На рисунке — это прямоугольник с вершинами (x_{11}, y_{11}), (x_{22}, y_{21}), (x_{23}, y_{23}), (x_{14}, y_{24}). Количество пикселей N в неизменившейся области, которая попадает в охватывающий прямоугольник,

можно посчитать по следующей формуле: $N = 2(x_{22} - x_{12})(y_{12} - y_{22})$. Размер “буферной зоны”, захватываемой алгоритмом, основанным на поблоковом сравнении, в большинстве случаев значительно меньше той неизменившейся области, которая попадает в охватывающий прямоугольник.

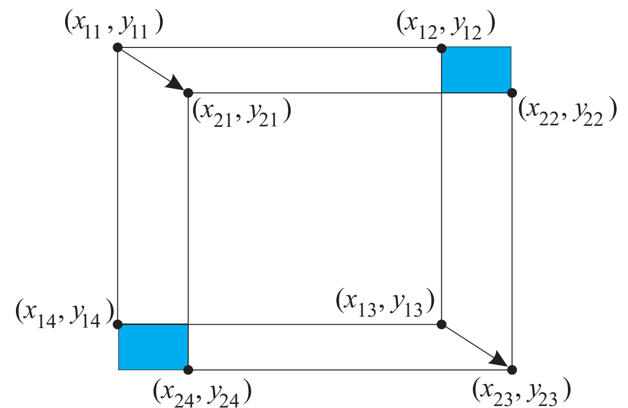
Приведенное здесь сравнение нельзя считать исчерпывающим сравнительным анализом описанных алгоритмов, основанных на попиксельном и поблоковом сравнении. Такое сравнение лишь показывает, что ни один из этих алгоритмов не является безусловно лучшим по степени сжатия.

2.5. Сравнение алгоритма, основанного на попиксельной операции xor, и алгоритма, основанного на компенсации движения. Существуют гораздо более сложные алгоритмы сжатия экранного видео, чем два описанных алгоритма. Например, рассмотрим алгоритм, изложенный в [18].

Ниже приведены основные отличия этого алгоритма от общих алгоритмов, используемых для сжатия видео (таких как MPEG2).

1. При сжатии ключевых кадров используется разделение на непрерывно-тоновые и дискретно-тоновые области. Первый тип областей (это может быть, например, фотография, отображаемая в качестве фона) может быть сжат с потерями, а второй тип областей (например, окна или текст) должен быть сжат без потери информации.

2. При сжатии промежуточных кадров алгоритмом компенсации движения стратегия поиска блока ключевого кадра, соответствующего текущему блоку промежуточного кадра, адаптирована для сжатия экранного видео.



Перемещение окна по направлению, близкому к диагональному. Окну до движения соответствует прямоугольник с вершинами (x_{11}, y_{11}), (x_{12}, y_{12}), (x_{13}, y_{13}), (x_{14}, y_{14}). Окну после движения соответствует прямоугольник с вершинами (x_{21}, y_{21}), (x_{22}, y_{22}), (x_{23}, y_{23}), (x_{24}, y_{24}). Затененные прямоугольники — это неизменившиеся области, которые попадают в охватывающий прямоугольник

Проведем аналитическое сравнение алгоритма, основанного на попиксельном сравнении изображений, и алгоритма [18] при сжатии видео происходящего на экране пользователя. При таких изменениях промежуточного кадра относительно ключевого, когда пользователь пролистывает содержимое документа, используя скроллинг, или перемещает окно на некоторое расстояние, выигрыш по степени сжатия алгоритма, основанного на компенсации движения, будет ощутим, так как при этом за доли секунды, зачастую, происходит движение на небольшое расстояние (достаточно малое для успешного нахождения наиболее близкого по некоторому критерию блока алгоритмом компенсации движения). Если пользователь совершает большое количество таких действий во время записи видео происходящего на экране, то коэффициент сжатия видеоданных в среднем окажется несколько выше при использовании алгоритма, основанного на компенсации движения, чем при использовании алгоритма, основанного на попиксельном сравнении. Однако пользователь может и не совершать таких действий во время записи экранного видео.

Если пользователь вводит текст или сворачивает/открывает окно, то попиксельное сравнение, скорее всего, будет давать степень сжатия не хуже, чем компенсация движения. Поскольку при этом происходит появление на экране новых объектов, то в изменившихся областях на экране алгоритму компенсации движения, скорее всего, не удастся найти блок, в достаточной мере схожий с текущим блоком, поэтому исходный блок будет закодирован по тому же алгоритму, который используется для кодирования блоков ключевого кадра.

Рассмотрим работы, в которых описано применение алгоритма компенсации движения, реализованного с помощью видеокарты, в различных сферах с целью определить возможность обработки видео высокого разрешения алгоритмом компенсации движения в режиме реального времени.

В [8] описано использование видеокарты для сканирования трехмерных объектов, которое включает в себя компенсацию движения. К сожалению, автор работы [8] не приводит таких данных как размер кадра и время обработки этого кадра алгоритмом компенсации движения. Отсутствие этих данных не позволяет определить, возможно ли применение реализованного алгоритма компенсации движения для сжатия потокового видео высокого разрешения.

В [12] используется компенсация движения, реализованная с помощью видеокарты, как этап при сжатии видео, снимаемого одновременно несколькими камерами с разных позиций. Поскольку в [12] приводятся только результаты тестирования алгоритма сжатия, разработанного авторами, в целом, то сложно

оценить, с какой скоростью работает реализованный алгоритм компенсации движения.

В [11] применяется алгоритм компенсации движения, реализованный с помощью видеокарты, для цифровой рентгенографии кровеносных сосудов. Авторам удалось обработать алгоритмом компенсации движения три кадра размером 1024×1024 пикселей в секунду. Однако специфика применения компенсации движения здесь такова, что по завершении обработки не требуется передавать данные обратно в оперативную память (ОП) и затем сохранять их на жесткий диск. Вместо этого после выполнения компенсации движения на текущий кадр накладывается специальная маска, а результат выводится на экран.

В [4] обрабатывается последовательность из 303 кадров размером 256×256 пикселей алгоритмом оценивания движения за пять секунд. Однако в [4], так же как и в [11], за алгоритмом компенсации движения следует визуализация, а не копирование данных обратно в ОП с последующим сохранением на жесткий диск.

Таким образом, не удалось найти ни одной работы, где было бы четко указано, что удалось реализовать алгоритм компенсации движения, который бы обрабатывал видео высокого разрешения в режиме реального времени с помощью видеокарты, учитывая время копирования данных из видеопамяти в ОП.

Алгоритмы, основанные на попиксельном и поблоковом сравнении, напротив, выполняются в режиме реального времени. Так, пара изображений размером 1024×768 пикселей обрабатывается всего за несколько миллисекунд (более точные данные о скорости выполнения операции xor с помощью пиксельных шейдеров и CUDA приведены в разделе 3).

Вывод: при сжатии экранного видео компенсация движения хотя и может давать несколько больший коэффициент сжатия чем попиксельное сравнение, но достигается это ценой значительного замедления в работе.

К алгоритмам сжатия видео происходящего на экране пользователя предъявляются даже более жесткие требования по скорости обработки данных, чем к прочим алгоритмам сжатия видео в режиме реального времени — этот тип видеоданных должен сжиматься в фоновом режиме, т.е. количество вычислений по возможности должно быть сведено к минимуму. Кроме того, степень сжатия должна быть близка к той степени сжатия, которую демонстрируют более трудоемкие алгоритмы (например, компенсация движения). Как было показано выше, этим критериям удовлетворяют алгоритмы, основанные на попиксельном и поблоковом сравнении.

2.6. Сравнение алгоритма, основанного на попиксельной операции xor, и алгоритма, используемого в VNC для сжатия экранного видео. Здесь для сравнения используется реализация VNC TightVNC, исходный текст которой доступен по ссылке [21]. В этой реализации VNC используется технология Mirror Video Driver. Это — видеодрайвер для виртуального устройства, который позволяет дублировать любые графические операции физического устройства. На основе этой технологии приложение может определять любые изменения экрана, что позволяет значительно уменьшить объем обрабатываемых данных при сжатии экранного видео. Однако, к сожалению, эту технологию можно применять без дополнительных ограничений только в Windows 2000 и Windows XP. Если задействовать mirror-видеодрайвер в Windows Vista, то режим Aero, являющийся стандартным для Windows Vista, будет автоматически отключен [22]. Таким образом, эта технология, хотя и позволяет определять любые изменения экрана, не является универсальной, так как может быть задействована только в нескольких ОС семейства Windows.

Кроме того, в рассматриваемой реализации TightVNC используются специальные перехватчики событий (hooks) [23] для определения изменившихся областей экрана, что позволяет снизить количество данных, копируемых из видеопамяти в ОП. Однако перехватчики событий специфичны для каждой ОС, поэтому для создания программного обеспечения, использующего перехватчики событий, для одной ОС на основе реализации для другой ОС потребуются значительные усилия.

В нашей работе рассматриваются алгоритмы, основанные на попиксельном и поблоковом xor, которые не привязаны к определенной ОС. Поэтому, имея реализацию этих алгоритмов для одной ОС, не сложно адаптировать ее для другой ОС. Однако при этом приходится каждый раз получать снимок всего экрана, а затем выявлять изменившиеся области.

3. Результаты тестирования. Ниже приведены результаты тестирования временны́х показателей пиксельных шейдеров и программы, написанной с применением технологии NVidia CUDA. Поскольку ключевой кадр остается неизменным для девяти подряд идущих промежуточных кадров, то ключевой кадр необходимо копировать в видеопамять только при сравнении с первым из девяти промежуточных кадров. Поэтому приведено время выполнения операции xor как при копировании в видеопамять двух кадров, так и при копировании одного кадра. Интересно также сравнить скорость выполнения операции

xor без учета времени копирования входных и выходных данных. Из видеопамяти в ОП всегда копируется один выходной массив. Для алгоритма, выполняемого на ЦП, указывается только время выполнения, так как в этом случае все данные находятся в ОП и их не нужно копировать в видеопамять и обратно.

При тестировании каждый кадр имел разрешение 1024×768 и глубину цвета в 32 бита. При этом для хранения каждого компонента цвета пикселя R, G и B используется один байт, поэтому "значащими" являются только три байта для каждого пикселя. Однако затраты на перевод кадра из 32-битового в 24-битовый формат превысили бы затраты на копирование лишних 1024×768 байтов из оперативной памяти в видеопамять. Таким образом, размер исходного изображения составляет $1024 \times 768 \times 4$ байтов.

Тестирование проводилось на платформе со следующими характеристиками: процессор Intel Core 2 Duo E6750 2,66 ГГц, оперативная память DDR2 2 Гб, видеокарта NVidia GeForce 8600 GTS (подключена через интерфейс PCI Express), операционная система Windows XP.

Таблица 2

Результаты тестирования временны́х показателей попиксельного xor

Технология / параметр	Время выполнения при копировании двух кадров (мс)	Время выполнения при копировании одного кадра (мс)	Время выполнения без учета времени копирования (мс)
Пиксельные шейдеры	13	6	< 1 (менее 1 мс)
NVidia CUDA	6	4	2
ЦП			4

Таблица 3

Результаты тестирования временны́х показателей поблокового xor

Технология / параметр	Время выполнения при копировании двух кадров (мс)	Время выполнения при копировании одного кадра (мс)	Время выполнения без учета времени копирования (мс)
Пиксельные шейдеры	16	9	< 1 (менее 1 мс)
NVidia CUDA	6	4	2
ЦП			4

Поскольку соотношение результатов, продемонстрированных различными реализациями при проведении операций поблокового и попиксельного xor, примерно одинаково, то можно провести общий анализ результатов тестирования.

Из результатов тестирования (табл. 2 и 3) следует, что пиксельный шейдер работает быстрее, чем программа, реализованная с помощью CUDA, но с учетом копирования данных в видеопамять и обратно CUDA-реализация оказывается более быстрой. Очевидно, что это достигается за счет того, что технология CUDA обеспечивает более быструю работу с памятью, нежели могут обеспечить пиксельные шейдеры. По совокупному времени исполнения CUDA-реализация при копировании одного кадра в видеопамять показала те же результаты, что и ЦП-реализация. Даже при копировании в видеопамять одного кадра совокупное время выполнения пиксельного шейдера и копирования данных превышает время выполнения ЦП-реализации. Ниже рассматриваются способы сокращения накладных расходов, связанных с копированием данных из ОП в видеопамять, что позволит несколько ускорить работу пиксельного шейдера при выполнении операции xor. Перенос сжатия видео на видеокарту позволяет снизить вычислительную нагрузку на центральный процессор.

4. Перспективы развития исследования. Теперь рассмотрим перспективы более обширного использования описанных выше технологий при сжатии видео происходящего на экране пользователя.

В существующей реализации последовательность действий, выполняемых для получения снимков экрана и их сжатия, далека от оптимальной:

1) снимок экрана копируется в ОП с помощью функции BitBlt;

2) снимок экрана пересылается обратно в видеопамять для выполнения операции xor;

3) результат выполнения копируется в ОП.

Если удастся получить снимок экрана непосредственно в видеопамять, то первый этап будет заменен на значительно более быструю операцию копирования из видеопамяти в видеопамять, второй этап вообще не потребуется, а на третьем этапе из видеопамяти в ОП будет пересылаться не только результат выполнения операции xor, но и область промежуточного кадра, охватывающая изменившуюся область. В этом случае CUDA-реализация попиксельного и поблокового xor может превзойти по скорости выполнения

ЦП-реализацию, а программа, использующая пиксельные шейдеры, может сравняться с ЦП-реализацией по этому параметру.

Теперь рассмотрим возможности совершенствования существующих реализаций, специфичных для пиксельных шейдеров и технологии CUDA.

1. Использовать результирующую текстуру однобитового формата в пиксельном шейдере. Для этого можно задействовать DirectX9.0c под Windows Vista, где однобитовый формат для текстуры, являющейся целью отрисовки, допустим. Это позволит уменьшить время копирования результирующей текстуры из видеопамяти в оперативную память, так как в этом случае результирующая текстура будет в восемь раз меньше, чем в существующей реализации.

2. При попиксельном сравнении с помощью технологии CUDA перенести на видеокарту также определение индексов строк и столбцов, в которых есть изменившиеся пиксели. После этого останется только записать байты пикселей, находящихся на пересечении указанных строк и столбцов, в выходной файл. Такое усовершенствование позволит разгрузить центральный процессор и свести время пересылки выходного массива из видеопамяти в оперативную память к минимуму, так как его размер будет составлять всего несколько килобайтов.

5. Заключение. Технология NVidia CUDA может быть применена для сжатия экранного видео. Преимуществом технологии NVidia CUDA является наличие значительно большего потенциала для реализации различных алгоритмов сжатия видео по сравнению с пиксельными шейдерами. Отличительной чертой этой технологии является простота понимания и универсальная модель программирования.

Для того чтобы применять пиксельные шейдеры для сжатия экранного видео, необходимо провести дополнительные исследования с целью доведения скорости выполнения пиксельного шейдера, реализующего операцию xor, хотя бы до уровня ЦП-реализации.

СПИСОК ЛИТЕРАТУРЫ

1. NVIDIA CUDA. Compute Unified Device Architecture. Programming Guide (http://developer.download.nvidia.com/compute/cuda/1_1/ NVIDIA_CUDA_Programming_Guide_1.1.pdf).
2. The CUDA Compiler Driver (http://www.nvidia.com/object/io_1195170069217.html).
3. *Луна Ф.Д.* Введение в программирование трехмерных игр с DirectX 9.0 (http://www.proklondike.com/file/Other/Frank_Luna_-_3dGamesProgrammingIntro(RUS).rar).
4. *Strzodka R., Garbe C.* Real-time motion estimation and visualization on graphics cards // Proc. IEEE Visualization Conf. 2004. 545–552.
5. *Shen G. et al.* Accelerate video decoding with generic GPU // IEEE Transactions on Circuits and Systems for Video Technology. 2005. **15**, N 5. 685–693.
6. *Pieters B.* Motion compensation and reconstruction of H.264/AVC-coded pictures using the GPU (http://symposium.elis.ugent.be/archive/symp2006/papers_poster/paper084_Bart_Pieters.pdf).
7. *Krüger J.* A structure for point scan compression and rendering (http://wwwcg.in.tum.de/Research/data/Publications/pbg05.pdf).
8. *Weise T.* A fast 3D scanning with automatic motion compensation (http://www.vision.ee.ethz.ch/bleibe/papers/weise-motioncompensation-cvpr07.pdf).
9. *Colantoni P.* Fast and accurate color image processing using 3D graphics cards (http://colantoni.nerim.net/articles/VMV2003.pdf).
10. *Rijsselbergen D.* YCoCg(-R) color space conversion on the GPU // Sixth FirW Symp., Ghent Univ., 2005.
11. *Deuerling-Zheng Y. et al.* Motion compensation in digital subtraction angiography using graphics hardware // Computerized Medical Imaging and Graphics. 2006. N 5. 279–289.
12. *Morvan Y.* Incorporating depth-image based view-prediction into h.264 for multiview-image coding (http://vca.ele.tue.nl/publications/data/Morvan2007d.pdf).
13. *Laan J. van der, Wavelet W.* Lifting on graphics hardware for faster video decoding (http://www.ictonderzoek.net/3/assets/File/posters/2007_102/2007_102.pdf).
14. High Quality DXT Compression using CUDA (http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/dxtc/doc/cuda_dxtc.pdf).
15. Image Denoising (http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/imageDenoising/doc/-imageDenoising.pdf).
16. Shader Model 3 (Direct3D 9) / Microsoft Corporation // DirectX SDK (August 2007) Documentation.
17. Shader Model 4 Features (http://msdn.microsoft.com/en-us/library/bb509657(VS.85).aspx).
18. Motion estimation/compensation for screen capture video (http://www.freepatentsonline.com/7224731.html).
19. *Сэломон Д.* Сжатие данных, изображений и звука. М.: Техносфера, 2006.
20. Dirac Specification (http://dirac.sourceforge.net/DiracSpec2.2.0.pdf).
21. TightVNC (http://www.sfr-fresh.com/windows/misc/tightvnc-1.3.9_winsrc.zip).
22. Mirror driver (http://en.wikipedia.org/wiki/Mirror_driver).
23. Hooks (http://msdn.microsoft.com/en-us/library/ms632589.aspx).

Поступила в редакцию
12.05.2008