Министерство образования Российской Федерации
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет информатики
Кафедра теоретических основ информатики

УДК 004.627 : 004.932.2

ДОПУСТИТЬ К ЗАЩИТЕ В ГАК
Зав. кафедрой, проф., д.т.н.
_____ Ю. Л. Костюк
«___» _____ 2008 г.

Дружинин Денис Вячеславович

РАЗРАБОТКА БЫСТРОГО АЛГОРИТМА СЖАТИЯ ИЗОБРАЖЕНИЙ БЕЗ ПОТЕРЬ ИНФОРМАЦИИ. ОПТИМИЗАЦИЯ АЛГОРИТМОВ СЖАТИЯ ЭКРАННОГО ВИДЕО С ПОМОЩЬЮ РЕСУРСОВ ВИДЕОКАРТЫ

Дипломная работа

Научный руководитель, доцент, к. т. н. — В. А. Лавров

Исполнитель, студ. гр. 1431 — Д. В. Дружинин

Томск – 2008

Реферат

Дипломная работа 51 с., 5 рис., 5 диаграмм, 12 табл., 24 источник, 1 прил.

СЖАТИЕ ВИДЕО, ПИКСЕЛЬНЫЕ ШЕЙДЕРЫ, NVIDIA CUDA, СЖАТИЕ ИЗОБРАЖЕНИЙ, БЫСТРЫЕ АЛГОРИТМЫ, ЗАПИСЬ АКТИВНОСТИ ПОЛЬЗОВАТЕЛЯ

Объекты исследования:

1. Алгоритмы сжатия дискретно-тоновых изображений, обладающие линейной трудоёмкостью, без потерь информации;

2. Технологии, предоставляющие интерфейс к мощностям видеокарты.

Цели работы:

1. Разработка быстрого алгоритма сжатия дискретно-тоновых изображений без потерь информации. Этот алгоритм можно будет использовать для сжатия ключевых кадров экранного видео;

2. Провести практическое сравнение технологии NVidia CUDA и пиксельных шейдеров при сжатии экранного видео.

Метод исследования – вычислительный эксперимент.

Результаты работы:

1. Был разработан быстрый алгоритм сжатия без потерь информации, демонстрирующий высокую степень сжатия дискретно-тоновых изображений и превосходящий аналоги по скорости выполнения;

2. Два алгоритма сжатия экранного видео, основанные на сравнении изображений, реализованы с помощью технологии NVidia CUDA и пиксельных шейдеров;

3. С помощью технологии NVidia CUDA на видеокарту было перенесено выполнение финального этапа сжатия – кодирования методом Хаффмана.

4. На основе указанных в пунктах (1), (2) и (3) технологий и алгоритмов был реализован кодек, предназначенный для сжатия экранного видео.

Область применения – сжатие экранного видео, быстрое сжатие дискретно-тоновых изображений.

Прогноз о развитии исследования: в дальнейшем предполагается внести в Гибридный алгоритм такие изменения, которые позволят объединять в группы не только одноцветные пиксели, но и часто встречающиеся последовательности различных пикселей.

Перспективы более обширного использования пиксельных шейдеров и технологии NVidia CUDA при сжатии экранного видео подробно рассмотрены в Разделе «2. Сжатие потокового видео с помощью видеокарты. Сравнение технологий».

.

Содержание

Перечень условных обозначений

Прямой обход пикселей изображения – обход пикселей изображения в том порядке, в котором информация о них хранится в области памяти, доступ к которой происходит как к массиву байтов (при просмотре массива от начала до конца). В Windows XP обычно такой обход осуществляется по строкам изображения, начиная с нижней его части.

Изображение, типичное для Windows XP – изображение, большую часть которого занимают окна Windows XP (например, окна каталогов). Такое изображение типично для экрана пользователя, работающего в среде Windows XP.

Экранное видео – видео происходящего на экране пользователя. При этом фиксируется активность пользователя: движения курсора мыши, скроллинг, сворачивание и открытие свёрнутого окна, перемещение окна, ввод текста и т. д.

Введение

Сжатие видео – одна из наиболее трудоёмких 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, очень проблематично. Даже с учётом постоянно возрастающей производительности центральных процессоров, этот процесс останется чрезвычайно трудоёмким. Наилучшим вариантом был бы запуск процесса сжатия видео высокого разрешения в фоновом режиме, что позволило бы пользователю продолжать привычную деятельность на своём компьютере. Как же этого добиться?

При решении трудоёмких вычислительных задач часто оказываются незаслуженно забыты вычислительные мощности, предоставляемые видеокартой. Производительность современной видеокарты измеряется сотнями гигафлопов, в то время как производительность центрального процессора – 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. А центральный процессор за счёт сбалансированности количества транзисторов, отведённых для арифметико-логических устройств, под кэш и для устройств, обеспечивающих выполнение инструкций потока управления, не обладает такой высокой производительностью, как современные видеокарты. Зато центральный процессор показывает стабильную производительность при решении как алгоритмических задач с большим количеством условных переходов, так и математических задач с большим количеством вычислений.

В настоящий момент значительная часть персональных компьютеров оснащена отдельными видеокартами. Итак, очевидным способом разгрузить центральный процессор при сжатии потокового видео является использование видеокарты для выполнения тех этапов сжатия, которые требуют однообразных математических или логических расчетов для большого числа элементов и могут быть реализованы с минимальным количеством условных переходов.

В этой работе рассмотрены две технологии, позволяющие получить доступ к мощностям видеокарты: пиксельные шейдеры и NVidia CUDA.

Работы, посвящённые реализациям алгоритмов кодирования / декодирования данных, алгоритмов обработки изображений, использующим вычислительные мощности видеокарты, а также прочие использованные источники рассмотрены в Разделе 1.

Одним из типов видеоданных, сжатие которых необходимо осуществлять в режиме реального времени является экранное видео. Как правило, это видео высокого разрешения. В этой работе рассмотрены два алгоритма сжатия видео без потерь информации, обладающие линейной трудоёмкостью и демонстрирующие высокий коэффициент сжатия экранного видео. Причём зачастую о полезности программ, осуществляющих сжатие и запись на жёсткий диск этого типа видеоданных, можно говорить только в том случае, когда их можно запустить в фоновом режиме. Поэтому при сжатии экранного видео для разгрузки ЦП часть операций разумно 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.

В Разделе 2 рассмотрены реализации этих алгоритмов, основанные на пиксельных шейдерах и технологии NVidia CUDA.

С помощью технологии NVidia CUDA удалось также перенести на видеокарту выполнение финального этапа сжатия – кодирование методом Хаффмана.

В этой работе представлены алгоритмы сжатия видео, которые 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.

Поскольку при сжатии видео ключевые кадры кодируются независимо от других кадров, существует также необходимость в быстрых алгоритмах сжатия изображений. Существует множество различных алгоритмов сжатия изображений. Но многие алгоритмы обеспечивают недостаточно высокий коэффициент сжатия дискретно-тоновых изображений (например, Lossless Jpeg) или работают слишком медленно для сжатия ключевых кадров размером 1024 * 768 пикселей в режиме реального времени (например, Jpeg 2000). Более подробно этот вопрос обсуждается в Разделе 4.3.

Поэтому существует необходимость разработки алгоритмов, сжимающих изображение за линейное время и демонстрирующих стабильно высокую степень сжатия дискретно-тоновых изображений.

Большинство алгоритмов, обладающих высокой степенью сжатия являются алгоритмами с потерями информации. Но при сжатии дискретно-тоновых изображений даже небольшой процент потерь может привести к значительному визуальному ухудшению качества изображения.

В этой работе представлен Гибридный алгоритм, который обладает линейной трудоёмкостью и является алгоритмом сжатия без потерь информации, и демонстрирует высокую степень сжатия двух классов изображений:

1. изображения, типичные для Windows XP;

2. изображения, при прямом обходе пикселей которых, часто чередуются цвета в условиях ограниченного количества этих цветов (например, текст).

Оба этих класса изображений относятся к более обширному классу – классу дискретно-тоновых изображений.

В Разделе 4 рассматривается гибридный алгоритм, а также несколько его модификаций, которые позволяют увеличить коэффициент сжатия дискретно-тоновых изображений при некотором замедлении в работе. В Разделе 4 также проводится сравнение по таким параметрам, как степень сжатия и время работы Гибридного алгоритма с несколькими другими алгоритмами.

1. Обзор литературы

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).

Рассмотрим некоторые работы, посвящённые реализациям алгоритмов обработки изображений, использующим вычислительные мощности видеокарты. В [1] 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, объём математических вычислений, возможность векторной обработки. Такой подход позволил авторам сделать определённые выводы об эффективности выполнения нескольких типов алгоритмов на видеокарте по результатам тестирования нескольких алгоритмов, отнесённых к различным типам. В [2] сравнивается производительность пиксельных шейдеров и ЦП при выполнении алгоритма преобразования изображения из пространства цветов YCoCg(-R) в RGB. В [3] описана реализация алгоритма сжатия DXT1 с помощью технологии NVidia CUDA. Также приводятся результаты тестирования, которые показывают многократное ускорение CUDA-реализации этого алгоритма по сравнению с ЦП-реализацией. В [4] описаны CUDA-реализации двух алгоритмов подавления шума в изображении: фильтр ближайших соседей (Nearest Neighbors Filter) и фильтр нелокальных средних (Non Local Means Filter). Авторы также приводят результаты тестирования CUDA-реализаций этих алгоритмов.

Многие исследователи также изучают способы применения видеокарты для кодирования и декодирования данных. В [5] описывается метод сжатия больших изображений, полученных лазерным сканированием трёхмерных объектов. Декодирование осуществляется с помощью вершинных шейдеров. Авторы работ [6] и [7] достигли значительных успехов в декодировании видео высокого разрешения с помощью видеокарты. Им удалось организовать конвейер, где первые стадии обработки выполняются ЦП, а последующие – пиксельными и вершинными шейдерами. Авторы [8] реализуют значительную часть декодирования кодека Дирака [9] с помощью технологии NVidia CUDA, добиваясь при этом многократного ускорения в работе декодера по сравнению с ЦП-реализацией. В [10] описывается реализация алгоритма оценки движения с помощью пиксельных шейдеров, приводятся результаты тестирования, демонстрирующие её ускорение относительно ЦП-реализации. В [11] описано сканирование трёхмерных объектов, которое включает в себя компенсацию движения, реализованную с помощью вершинных и пиксельных шейдеров. Авторы [12] применили алгоритм компенсации движения, реализованный с помощью вершинных и пиксельных шейдеров, для цифровой рентгенографии кровеносных сосудов. Авторы [13] используют компенсацию движения, реализованную с помощью пиксельных шейдеров, как этап при сжатии видео, снимаемого одновременно несколькими камерами с разных позиций.

Удалось найти подробное описание двух алгоритмов сжатия экранного видео. Один из этих алгоритмов использует компенсацию движения (он описан в [14]). Второй из этих алгоритмов - алгоритм, используемый TightVNC [15] для сжатия экранного видео. Сравнение этих двух алгоритмов с алгоритмом, основанном на попиксельном сравнении приведено в Разделе «2.2 Алгоритмы сжатия экранного видео». Работ по сжатию экранного видео с помощью видеокарты найти не удалось.

Теперь кратко рассмотрим прочие использованные источники. В [16] описывается модель программирования NVidia CUDA, даются рекомендации по оптимизации скорости

выполнения программ, использующих NVidia CUDA. В [17] содержится описание компилятора nvcc, который необходимо задействовать при компиляции программ, применяющих технологию NVidia CUDA. В [18] рассматриваются способы программирования игр с помощью DirectX 9, в том числе пиксельные шейдеры. В [19] представлен Гибридный алгоритм сжатия изображений и дано его сравнение с RLE и Сдвиговым алгоритмом. В [9] приводится спецификация кодека Дирака. [20] является фундаментальным источником информации о сжатии вообще, а также о специфике сжатия различных видов информации и о разработанных алгоритмах в этих областях.

2. Сжатие экранного видео с помощью видеокарты. Сравнение технологий

2.1 Обзор используемых технологий

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

2.1.1. Пиксельные шейдеры. Пиксельные шейдеры являются программами, написанными на си-подобном языке программирования (в случае использования HLSL – High Level Shader Language – высокоуровневого языка шейдеров), и выполняются на процессоре видеокарты без участия центрального процессора. Пиксельный шейдер выполняется для каждого пикселя изображения, и результатом однократного выполнения шейдера является задание цвета соответствующего пикселя изображения [18].

Этот раздел построен следующим образом. Сначала рассматриваются преимущества и недостатки ps_2_0, а затем последовательно указываются отличия ps_3_0 от ps_2_0 и отличия ps_4_0 от ps_3_0.

2.1.1.1. ps_2_0. Преимущество: пиксельные шейдеры поддерживаются практически на всех современных видеокартах.

Недостатки:

1. Способ распараллеливания жёстко фиксирован – шейдер выполняется один раз для каждого пикселя результирующей текстуры, причём предполагается, что изменяться будут только те байты результирующей текстуры, которые соответствуют этому пикселю. Такой подход далеко не всегда удобен и эффективен.

2. 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 будет производиться отрисовка, строго регламентирован, однобитовый формат не поддерживается. Поэтому часто приходится передавать из памяти видеокарты в оперативную память в несколько раз больше данных, чем необходимо.

2.1.1.2. ps_3_0. 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 сравнения с ps_2_0;

2. В ps_3_0 увеличено количество различных регистров по сравнению с ps_2_0;

3. В ps_3_0 по умолчанию все операции выполняются с максимальной точностью (не ниже 32 бит). [21]

Эти отличия свидетельствуют о том, что шейдеры ps_3_0 несколько более приспособлены для выполнения общих вычислений.

Тем не менее, значительные сходства между ps_2_0 и ps_3_0 позволяют сделать общий вывод об их применимости. Модель программирования, используемая в пиксельных шейдерах ps_2_0 и ps_3_0, узко специализирована и предназначена для расчёта цвета пикселей результирующего изображения. Рассмотренные ограничения, накладываемые ps_2_0 и ps_3_0, значительно сужают их область применения для общих вычислений.

2.1.1.3. ps_4_0.

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

1. В ps_4_0 количество слотов инструкций увеличено по сравнению с ps_3_0, а количество выполняемых инструкций в ps_4_0 не ограничено.

2. В ps_4_0 целые числа (int и uint) отображаются в 32-битовые целые числа на видеокарте. А в ps_3_0 и более ранних версиях пиксельных шейдеров целые числа отображались в вещественные числа на видеокарте (например, при передачи в шейдер текстуры, цвета пикселей которой закодированы целыми числами);

3. В ps_4_0 поддерживаются побитовые операторы для целых типов int и uint. В ps_3_0 побитовые операции не поддерживаются;

4. В ps_4_0 регистры были заменены буферами констант и текстур. [22]

Таким образом, ps_4_0 предоставляет несколько большие возможности для общих вычислений, чем ps_3_0. Но поскольку программируемые конвейеры Direct3D (в том числе и Direct3D 10) проектировались для отрисовки изображений, многие ограничения, накладываемые ps_2_0 и ps_3_0, распространяются и на ps_4_0. Поэтому хотя область применения ps_4_0 в общих вычислениях несколько шире, чем область применения ps_2_0 и ps_3_0, но по-прежнему весьма ограничена.

Для реализации попиксельного сравнения двух изображений на равенство были выбраны пиксельные шейдеры ps_2_0, как наиболее ранняя версия пиксельных шейдеров из тех, с помощью которых можно провести эту операцию, так как более ранние версии шейдеров поддерживаются на большем числе видеокарт.

А для реализации поблокового сравнения двух изображений не равенство были задействованы пиксельные шейдеры ps_3_0, так как в ps_2_0 количество выполняемых инструкций оказалось недостаточным.

2.1.2. NVidia CUDA (Compute Unified Device Architecture – Унифицированная вычислительная архитектура). Технология NVidia CUDA как и пиксельные шейдеры позволяет выполнять программы, написанные на си-подобном языке на процессоре видеокарты без привлечения центрального процессора. С помощью этой технологии можно решать практически любые задачи, связанные с трудоёмкими 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 её 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.

Преимущества:

1. Модель памяти технологии CUDA включает несколько типов памяти. В том числе это разделяемая память и константная память. Разделяемая память (shared memory) является внутрипроцессорной, поэтому скорость доступа к ней намного выше, чем к глобальной памяти. Каждому блоку потоков выделяется определённое 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.

2. 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 жёсткое разделение на входные и выходные данные).

3. Программные модули, написанные с использованием технологии NVidia CUDA, легко встраиваются в обычное приложение, написанное на си++. Для этого не требуется задействовать дополнительные программные средства, такие как DirectX или OpenGL.

Вывод: модель программирования чёткая и привычная для программиста. Программисту не требуется проходить долгого дополнительного обучения для использования технологии NVidia CUDA при написании сложных программных комплексов. Эта технология 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 [16].

2.2. Алгоритмы сжатия экранного видео

Описанные ниже два алгоритма сжатия видео, основанные на сравнении изображений, были разработаны нами на основе общих идей сжатия видео, изложенных в [20].

2.2.1. Алгоритм, основанный на попиксельном сравнении изображений.

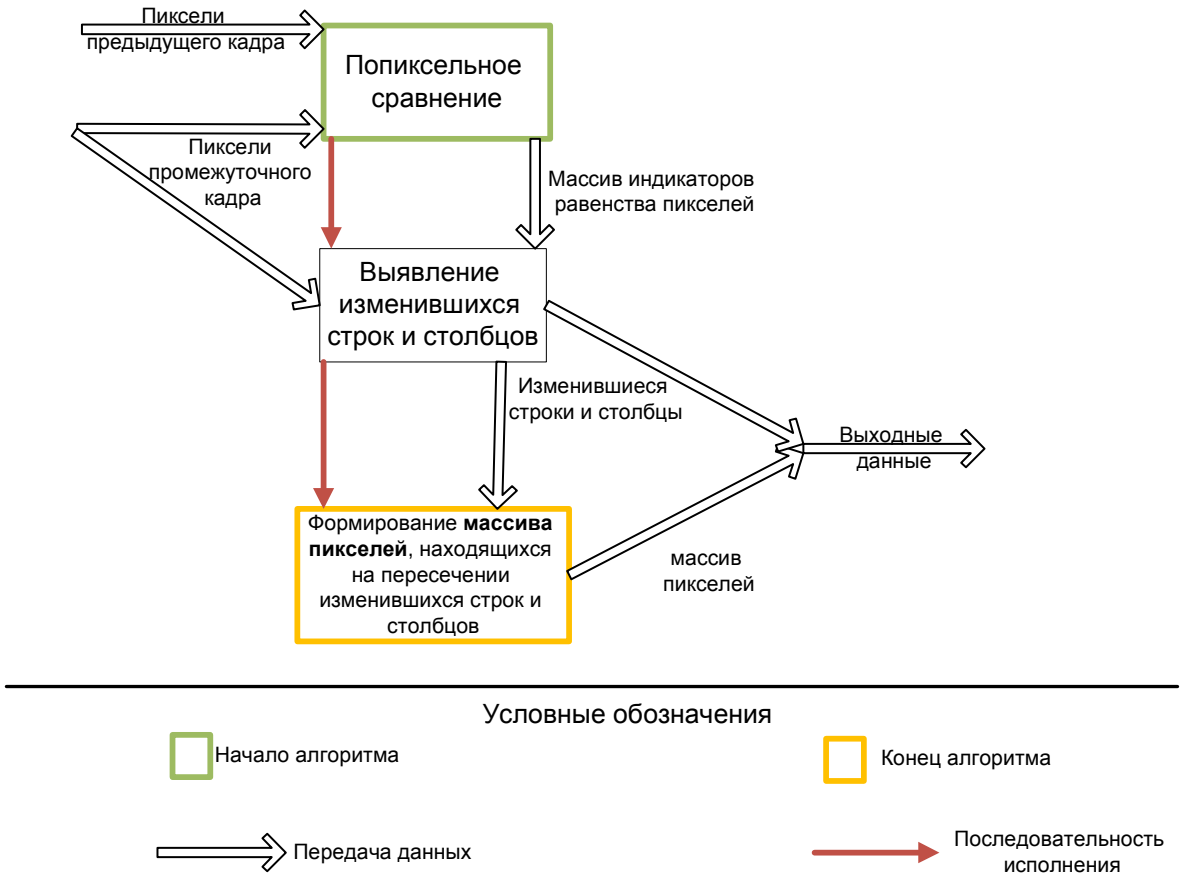

Рисунок 2.1 – Блок-схема алгоритма, основанного на попиксельном сравнении изображений.

Ключевой кадр (каждый десятый) кодируется независимо и записывается в выходной файл. Остальные кадры, называемые промежуточными, сохраняются в выходной файл не полностью. Промежуточный кадр сравнивается с ключевым, и в выходной файл записывается часть промежуточного кадра, включающая изменившуюся часть. А точнее, 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. При этом 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 потерь информации, обладающий линейной трудоёмкостью, что немаловажно для алгоритмов сжатия потокового видео.

Теперь стоит сказать о том, какой именно шаг этого алгоритма выполняется процессором видеокарты. Это сравнение промежуточных кадров с ключевым с целью выявления совпадающих и различающихся пикселей. Входные данные этой операции (назовём её попиксельный xor) – 2 изображения одинакового размера. 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 – 1 бит.

2.2.2. Алгоритм, основанный на поблоковом сравнении изображений.

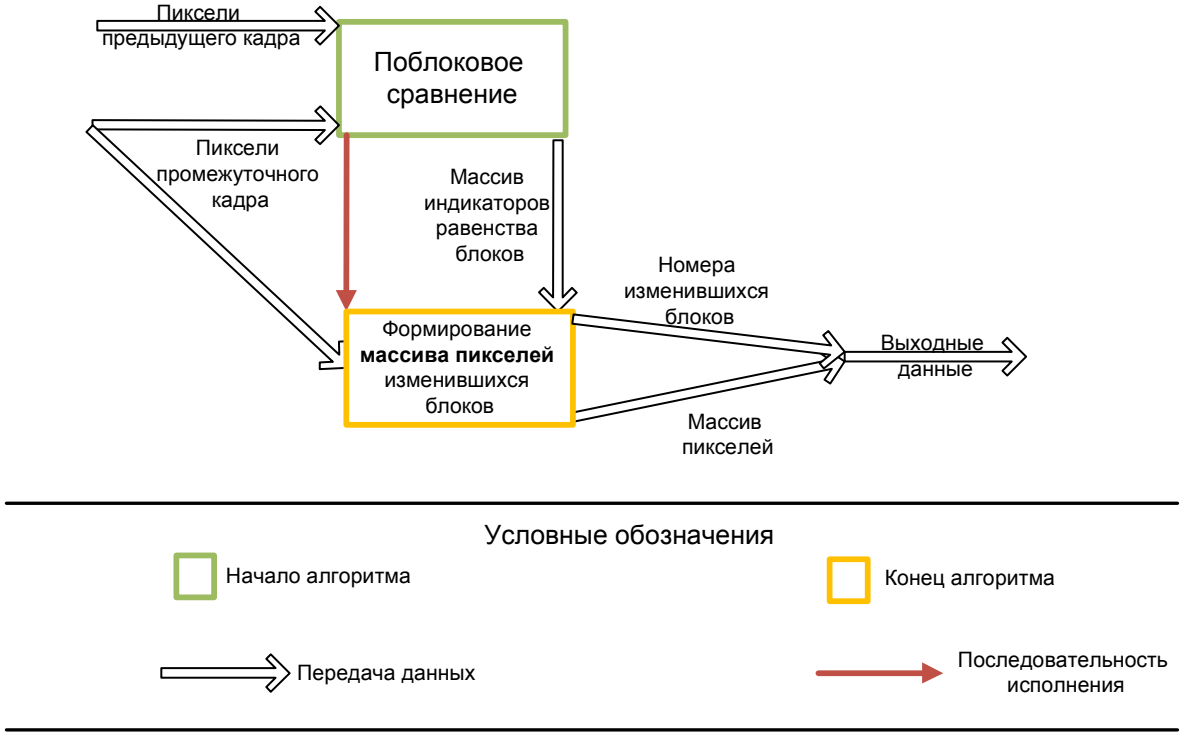

Рисунок 2.2 – Блок-схема алгоритма, основанного на поблоковом сравнении изображений.

Этот алгоритм похож на алгоритм, основанный на попиксельном сравнении изображений. Отличие заключается в том, что в выходной файл записываются те блоки пикселей промежуточного кадра, в которых есть хотя бы 1 изменившийся пиксель относительно соответствующего блока ключевого кадра. Также 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, при уменьшении размера блока увеличиваются накладные расходы, связанные с хранением номеров изменившихся блоков, но увеличивается и точность определения изменившейся области. Был выбран размер блока 8 * 8 пикселей, так как при таком размере блока на большинстве тестов была достигнута максимальная степень сжатия (с учётом количества передаваемых номеров блоков).

С помощью видеокарты выполняется сравнение промежуточных кадров с ключевым с целью выявления совпадающих и различающихся блоков (поблоковый xor). Входные данные те же, что и при попиксельном xor. Существенное отличие состоит в значительно меньшем (в $2^6 = 8 * 8$) размере выходного массива, так как каждый его элемент является индикатором равенства или неравенства двух блоков пикселей. В идеале 1 элемент результирующего массива - 1 бит.

2.2.3. Общий ход инициализации и выполнения операции xor. Общий ход инициализации и выполнения операций попиксельного и поблокового xor одинаков как в случае пиксельных шейдеров, так и в случае CUDA. Рассмотрим последовательность действий, необходимых для инициализации и выполнения операции xor с помощью пиксельных шейдеров и NVidia CUDA.

1) Общий ход инициализации и выполнения операции xor с помощью пиксельных шейдеров. Необходимо:

1.1) создать текстуру в видеопамяти, в которую будет производиться отрисовка, и поверхность с ней связанную;

1.2) создать поверхность в оперативной памяти, куда затем будет скопирован результат выполнения пиксельного шейдера;

1.3) создать текстуры в видеопамяти для входных изображений;

1.4) скомпилировать шейдер из файла. Создать на его основе шейдер, как программный объект, который затем можно установить для устройства D3DDevice;

1.5) скопировать входные изображения в созданные текстуры;

1.6) установить созданную в видеопамяти текстуру в качестве цели отрисовки;

1.7) установить текстуры входных изображений для текущего устройства D3DDevice. При этом текстуры, определённые в пиксельном шейдере, будут ассоциированы с текстурами входных изображений. Таким образом, передача параметров в пиксельный шейдер происходит неявно;

1.8) установить пиксельный шейдер для текущего устройства D3DDevice;

1.9) вызвать функцию DrawPrimitive, чтобы выполнить отрисовку в текстуру. При этом пиксельный шейдер будет вызван для каждого пикселя результирующей текстуры;

1.10) скопировать результирующую текстуру в оперативную память.

При этом пункты 1.1 – 1.4 представляют собой инициализацию, которая должна быть выполнена перед первым выполнением операции xor. А действия, указанные в пунктах 1.5 – 1.10, выполняются при каждом проведении операции xor.

2) Общий ход инициализации и выполнения операции xor с помощью технологии NVidia CUDA. Необходимо:

2.1) выделить память на видеокарте для передачи входных изображений;

2.2) скопировать входные изображения по указателю на выделенную память;

2.3) вызвать функцию, которая будет выполняться на видеокарте, указав количество блоков и количество потоков в каждом блоке;

2.4) скопировать результат выполнения функции в оперативную память.

Пункт 2.1 относится к инициализации, а пункты 2.2 – 2.4 относятся к непосредственному выполнению операции xor.

2.2.4. Некоторые детали реализации попиксельного и поблокового xor с помощью пиксельных шейдеров и NVidia CUDA.

Примечание: в этом разделе все размеры результирующих массивов указаны при ширине и высоте исходных текстур 1024 и 768 пикселей соответственно.

Реализация попиксельного xor с помощью пиксельных шейдеров. Каждый элемент, который является индикатором равенства или неравенства цветов двух пикселей, при использовании пиксельных шейдеров занимает 1 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 2-байтовый формат D3DFMT_R5G6B5. Но была применена следующая техника: каждый байт в 2-байтовом значении цвета пикселя результирующей текстуры используется независимо для кодирования результата операции xor. Это достигается за счёт независимого использования каналов цвета R и B пикселя результирующей текстуры. Таким образом, размер результирующей текстуры составляет 1024 * 768 = 786432 байтов.

Реализация поблокового xor с помощью пиксельных шейдеров. Формат результирующей текстуры был выбран тот же, что и при попиксельном xor - D3DFMT_R5G6B5. Но 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. Тем более что размер результирующей текстуры и так невелик (1024 * 768 * 2 / 64 = 24576 байтов).

Реализация попиксельного xor с помощью CUDA. Для хранения результата выполнения функции, реализующей операцию xor, был использован однобитовый формат. При этом один поток обрабатывает 8 пар подряд идущих пикселей и сохраняет результат в одном байте выходного массива. Обработка одним потоком 8 пар пикселей исходных изображений имеет смысл, так как иначе возникнет проблема синхронизации обращения нескольких потоков к одному байту выходного массива, что неизбежно приведёт к повышению временных издержек. Использование однобитового формата выходного массива позволило уменьшить время его копирования из видеопамяти в оперативную память, так как в этом случае размер выходного массива в 8 раз меньше, чем при использовании однобайтового формата, и составляет 1024 * 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 использовании технологии CUDA, поэтому реализация операции попиксельного 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 |= (difference && 1) * byteMask;
```

Результатом выполнения операции difference && 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 будет установлен в 1.

Реализация поблокового xor с помощью CUDA. Для реализации поблокового xor применены все те же способы оптимизации, что и при попиксельном xor. Но здесь появляются условные переходы. Они необходимы, чтобы остановить попиксельное сравнение при обнаружении первой пары неравных пикселей в блоке. Существенная экономия времени здесь 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 * 768 / 8 / 64 = 1536 байта.

2.2.5. Сравнение алгоритмов, основанных на попиксельном и поблоковом xor между собой. Выше были описаны два алгоритма, основанные на сравнении изображений. Оба этих алгоритма выполняются за близкое время (см. Раздел « 2.3 Результаты тестирования»).

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 свёрнутое.

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.

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, где пользователь сворачивает окно или наоборот открывает свёрнутое (см. Таблицу 2.1). В этом случае алгоритм, основанный на попиксельном сравнении, выявит и передаст на следующий этап сжатия в точности ту область, которая была изменена. А алгоритм, основанный на поблоковом сравнении, будет считать изменённой область, охватывающую реально изменившуюся область, так как блок 8 * 8 пикселей считается изменённым, если изменился хотя бы 1 пиксель в блоке. То есть, по периметру реально изменившейся области будет захвачена «буферная зона», толщиной максимум в 7 пикселей из неизменившейся области.

Пусть $T_{ср} = T_{max} / 2 = 7 / 2 \approx 4$,

где $T_{ср}$ – это толщина буферной зоны в среднем, а T_{max} – максимальная толщина буферной зоны. Тогда среднее количество неизменившихся пикселей $N_{ср}$ из «буферной

зоны» вокруг изменившейся области, которые алгоритм, основанный на поблоковом сравнении, будет считать изменившимися, можно посчитать по следующей формуле:

$N_{cp} = P * T_{cp} + const,$

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 * 4 * 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, где пользователь перемещает окно на некоторое расстояние по направлению, близкому к диагональному (см. Таблицу 2.1). Для определённости предположим, что окно было перемещено сверху вниз и слева направо (см. Рисунок 2.3). 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). Назовём такую область охватывающим прямоугольником. На Рисунке 2.3 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:

$N = (x_{22} – x_{12}) * (y_{12} – y_{22}) * 2.$

Размер «буферной зоны», захватываемой алгоритмом, основанным на поблоковом сравнении, в большинстве случаев значительно меньше той неизменившейся области, которая попадает в охватывающий прямоугольник.

Таблица № 2.1 – 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

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

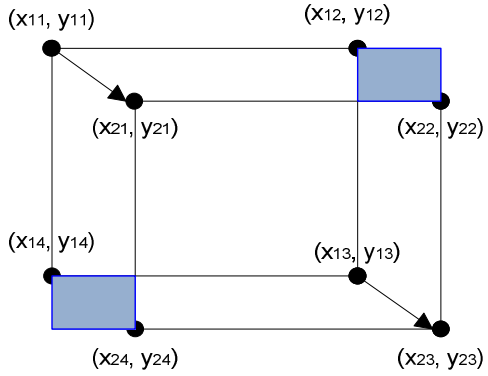

Рисунок 2.3 – Перемещение окна по направлению, близкому к диагональному. Окну до движения соответствует прямоугольник с вершинами (x_{11}, y_{11}), (x_{12}, y_{12}) (x_{13}, y_{13}) (x_{14}, y_{14}). Окну после движения соответствует прямоугольник с вершинами (x_{21}, y_{21}), (x_{22}, y_{22}) (x_{23}, y_{23}) (x_{24}, y_{24}). Синим

цветом обозначены неизменившиеся области, которые попадают в охватывающий прямоугольник.

2.2.6. Сравнение алгоритма, основанного на попиксельном xor и алгоритма, основанного на компенсации движения. Существуют гораздо более сложные алгоритмы сжатия экранного видео, чем два описанных выше алгоритма. Например, рассмотрим алгоритм, описанный в [14].

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, текст) должен быть сжат без потерь информации.

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

Проведём аналитическое сравнение алгоритма, основанного на попиксельном сравнении изображений, и алгоритма [14] при сжатии экранного видео. При таких изменениях промежуточного кадра относительно ключевого, где 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. Но пользователь может и не совершать таких действий во время записи видео происходящего на экране.

А вот при следующих изменениях промежуточного кадра относительно ключевого попиксельное сравнение точно будет обеспечивать степень сжатия не хуже, чем при компенсации движения: пользователь вводит текст, пользователь сворачивает окно или наоборот открывает свёрнутое. При этом происходит появление на экране новых объектов. В таких случаях 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 [11] описано использование видеокарты для сканирования трёхмерных объектов, которое включает в себя компенсацию движения. К сожалению, авторы [11] не приводят таких данных, как размер кадра и время обработки этого кадра алгоритмом компенсации движения. Отсутствие этих данных не позволяет определить, возможно ли применение алгоритма компенсации движения, реализованного авторами статьи, для сжатия потокового видео высокого разрешения.

Авторы [13] используют компенсацию движения, реализованную с помощью видеокарты, как этап при сжатии видео, снимаемого одновременно несколькими камерами с разных позиций. Поскольку в [13] приводятся только результаты тестирования алгоритма сжатия, разработанного авторами, в целом, то сложно оценить с какой скоростью работает реализованный авторами алгоритм компенсации движения.

Авторы [12] применили алгоритм компенсации движения, реализованный с помощью видеокарты, для цифровой рентгенографии кровеносных сосудов. Им удалось обрабатывать алгоритмом компенсации движения 3 кадра размером 1024 * 1024 пикселей в секунду. Но специфика применения компенсации движения здесь такова, что по завершении обработки не требуется передавать данные обратно в ОП и затем сохранять на жёсткий диск. Вместо этого после выполнения компенсации движения на текущий кадр накладывается специальная маска, и результат выводится на экран.

Авторам [10] удалось обработать последовательность из 303 кадров размером 256 * 256 пикселей алгоритмом оценивания движения за 5 секунд. Но в [10], также как и в [12] за выполнением алгоритма компенсации движения следует визуализация, а не копирование данных обратно в ОП с последующим сохранением на жёсткий диск.

Таким образом, не удалось найти ни одной работы, где было бы чётко указано, что авторам 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 мс (более точные данные о скорости выполнения операции xor с помощью пиксельных шейдеров и CUDA приведены в Разделе «2.3. Результаты тестирования»).

Вывод: при сжатии экранного видео компенсация движения хоть и может давать несколько больший коэффициент сжатия по сравнению с попиксельным сравнением, но достигается это ценой значительного замедления в работе.

К алгоритмам сжатия экранного видео предъявляются даже более жёсткие требования по скорости обработки данных, чем к прочим алгоритмам сжатия видео в режиме реального времени – этот тип видеоданных должен сжиматься в фоновом режиме, то есть количество вычислений по возможности должно быть сведено к минимуму. При этом степень сжатия должна быть близка к той степени сжатия, которую демонстрируют более трудоёмкие алгоритмы (например, компенсация движения). Как было показано выше, этим критериям удовлетворяют алгоритмы, основанные на попиксельном и поблоковом сравнении.

2.2.7. Сравнение алгоритма, основанного на попиксельном xor и алгоритма, используемого в VNC для сжатия экранного видео. В этом разделе для сравнения используется реализация VNC TightVNC, исходный текст которой доступен по ссылке [15]. В этой реализации VNC используется технология Mirror Video Driver. Это видеодрайвер для виртуального устройства, который позволяет дублировать любые графические операции физического устройства. Используя эту технологию, приложение может определять любые изменения экрана, что позволяет значительно уменьшить объём обрабатываемых данных при сжатии экранного видео. Но, 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 [23]. Таким образом, эта технология, хоть и позволяет определять любые изменения экрана, но не является

универсальной, так как может быть задействована только в нескольких ОС семейства Windows.

Также в рассматриваемой реализации TightVNC используются специальные перехватчики событий (hooks) [24] для определения изменившихся областей экрана, что позволяет снизить количество копируемых из видеопамяти в ОП данных. Но 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 этой работе рассматриваются алгоритмы, основанные на попиксельном и поблоковом xor, которые не привязаны к определённой ОС. Поэтому имея реализацию этих алгоритмов для одной ОС несложно адаптировать её для другой ОС. Но при этом приходится каждый раз получать снимок всего экрана, а затем выявлять изменившиеся области.

2.3. Результаты тестирования

Ниже приведены результаты тестирования временных показателей пиксельных шейдеров и программы, написанной с применением технологии NVidia CUDA. Поскольку ключевой кадр остаётся неизменным для 9 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 учёта времени копирования входных и выходных данных. Из видеопамяти в ОП всегда копируется 1 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, B используется 1 байт, поэтому «значащими» являются только 3 байта для каждого пикселя. Но, 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. К тому же при применении технологии NVidia CUDA использование 32-битового формата позволяет адресоваться к байтам одного пикселя, как к значению типа unsigned int, что ускоряет работу алгоритма, так как нужно всего одно чтение из видеопамяти, чтобы получить все байты, соответствующие одному пикселю. Таким образом, размер исходного изображения составляет 1024*768*4 байтов.

Тестирование проводилось на платформе со следующими характеристиками

Процессор: Intel Core 2 Duo E6750 2,66 ГГц;

Оперативная память: DDR2 2Гб;

Видеокарта: NVidia GeForce 8600 GTS (подключена через интерфейс PCI Express);

Операционная система: Windows XP.

Все последующие результаты тестирования указаны для этой платформы.

Таблица № 2.2 – Результаты тестирования попиксельного xor.

технология / параметр	время выполнения при копировании двух кадров (мс)	время выполнения при копировании одного кадра (мс)	время выполнения без учёта времени копирования (мс)

Пиксельные шейдеры	13	6	< 1 (менее 1 мс)
NVidia CUDA	6	4	2
ЦП			4

Таблица № 2.3 – Результаты тестирования поблокового xor.

технология / параметр	время выполнения при копировании двух кадров (мс)	время выполнения при копировании одного кадра (мс)	время выполнения без учёта времени копирования (мс)
Пиксельные шейдеры	15	8	< 1 (менее 1 мс)
NVidia CUDA	5	4	2
ЦП			4

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

По результатам тестирования видно, что пиксельный шейдер прорабатывает быстрее, чем программа, реализованная с помощью CUDA, но с учётом копирования данных в видеопамять и обратно CUDA-реализация оказывается более быстрой. Очевидно, что это достигается за счёт 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. Совокупное время выполнения пиксельного шейдера и копирования данных превышает время выполнения ЦП-реализации даже при копировании в видеопамять одного кадра. Но в Разделе «2.4. Перспективы развития исследования» рассматриваются способы сокращения накладных расходов, связанных с копированием данных из ОП в видеопамять, что позволит несколько ускорить работу пиксельного шейдера при выполнении операции xor.

Может показаться, что нет смысла в применении описанных выше технологий для выполнения операции xor, так как центральный процессор выполняет эту операцию за то же время (в случае CUDA) или даже быстрее (в случае пиксельных шейдеров), но не стоит забывать, что передача части вычислений на видеокарту освобождает ЦП для выполнения других задач.

2.4. Перспективы развития исследования

Теперь рассмотрим перспективы более обширного использования описанных выше технологий при сжатии экранного видео.

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

а) Снимок экрана копируется в ОП с помощью функции BitBlt;

б) снимок экрана пересылается обратно в видеопамять для совершения операции xor;

в) результат выполнения копируется в ОП.

Если удастся получать снимок экрана непосредственно в видеопамять, то этап (а) будет заменён на значительно более быструю операцию копирования из видеопамяти в видеопамять, этап (б) вообще не потребуется, а на этапе (в) 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 усовершенствования существующих реализаций, специфичные 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 8 раз меньше, чем в существующей реализации;

2) Широкие возможности, предоставляемые технологией NVidia CUDA, делают реальным перенос выполнения значительной части алгоритма, основанного на попиксельном xor, на видеокарту. В этом случае входными данными алгоритма по-прежнему будут ключевой и промежуточный кадры, а выходными данными – индексы 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 существенно 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.

3. Реализация метода Хаффмана с помощью технологии CUDA

С помощью технологии NVidia CUDA удалось перенести на видеокарту выполнение последнего этапа сжатия экранного видео – кодирование методом Хаффмана. В качестве алфавита был выбран набор всех возможных однобайтовых значений (от 0 до 255).

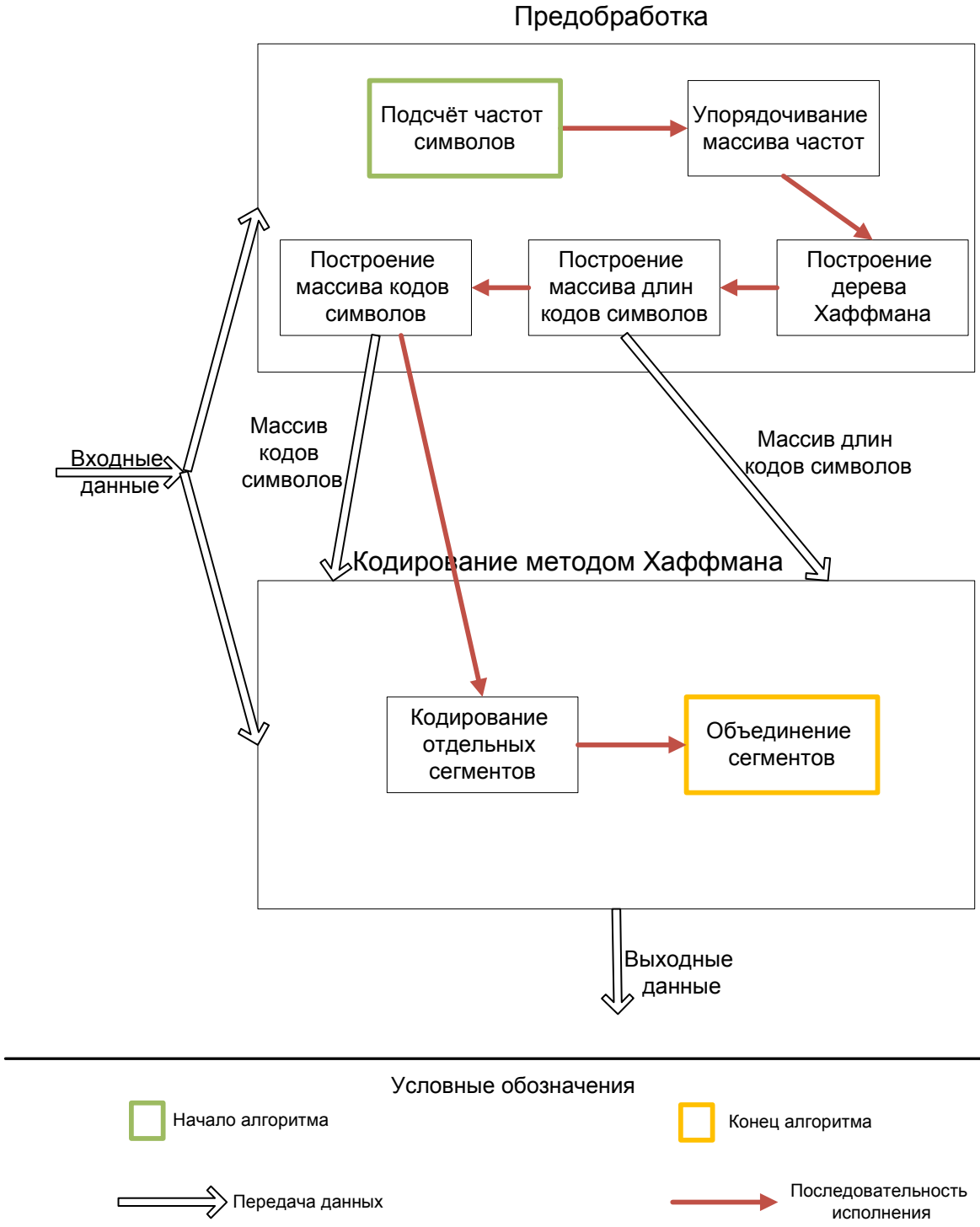

Рисунок 3.1 – Блок-схема CUDA-реализации метода Хаффмана.

3.1. Предобработка

Вначале на ЦП выполняется предобработка, необходимая для кодирования методом Хаффмана. Первые три этапа предобработки являются стандартными при реализации метода Хаффмана:

1. подсчёт частот, с которыми встречаются различные символы алфавита;

2. упорядочивание массива частот;

3. построение дерева Хаффмана.

Затем на основе дерева Хаффмана строятся дополнительные структуры данных:

1. массив кодов символов;

2. массив длин кодов символов.

Было принято решение использовать эти структуры данных при кодировании методом Хаффмана вместо дерева Хаффмана. Применение массива кодов символов вместе с массивом длин кодов символов позволяет реализовать кодирование методом Хаффмана без условных переходов. А условный переход является одной из самых дорогих по времени операцией при выполнении на видеокарте.

Из всех описанных выше шагов предобработки только подсчёт частот символов зависит от размера входных данных. Остальные этапы предобработки являются элементарными шагами.

3.2. Кодирование методом Хаффмана

Рассмотрим более подробно кодирование методом Хаффмана, выполняемое на видеокарте. Массив входных данных делится на сегменты по количеству потоков. Каждый поток обрабатывает свой сегмент.

Заранее неизвестно, сколько потребуется байтов для хранения закодированного методом Хаффмана сегмента. Более того, отдельные сегменты после кодирования методом Хаффмана могут увеличиться в размере, а не уменьшиться, поэтому на основании статистики, полученной при тестировании, было принято решение выделить для хранения одного закодированного сегмента в полтора раза больше байтов, чем исходный размер сегмента. После завершения стадии кодирования отдельных сегментов каждый поток записывает количество занятых битов в его закодированном сегменте в разделяемый массив `bitsCountBlocks`.

После этого необходимо объединить отдельные закодированные сегменты в один массив, который и будет являться результатом кодирования методом Хаффмана. Перед началом этапа объединения сегментов установлена точка синхронизации потоков. В технологии CUDA существует эффективный метод синхронизации потоков, входящих в один блок, поэтому было принято решение создавать только один блок, объединяющий максимально возможное количество потоков (512). Таким образом, переменным является не количество потоков, а размер исходного сегмента.

Количество занятых битов в закодированном сегменте в общем случае не кратно 8, поэтому главная задача этого этапа – провести объединение сегментов и избежать при этом обращения нескольких потоков на запись к одному и тому же байту результирующего массива. Рассмотрим последовательность действий, которые совершает поток с номером ThreadId на этом этапе:

1. поток подсчитывает суммарное количество битов `beforeBitsCount` во всех закодированных сегментах с номерами в интервале [0, ThreadId – 1];

2. поток пропускает в начале своего закодированного сегмента то количество бит, которое необходимо, чтобы дополнить `beforeBitsCount` до числа, кратного 8;

3. поток копирует в выходной массив оставшуюся часть своего закодированного сегмента;

4. если это не поток с последним номером, то он копирует из начала следующего закодированного сегмента то количество бит, которое недостаёт для заполнения текущего байта результирующего массива. Если это поток с последним номером, то он подсчитывает суммарное количество битов в результирующем массиве и сохраняет получившееся число.

Стоит заметить, что не весь входной массив кодируется методом Хаффмана на видеокарте. Количество байтов, равное остатку от деления количества байтов входных данных на количество потоков (512) кодируется методом Хаффмана на ЦП после выполнения кодирования основной части входных данных. Это действие также можно считать элементарным шагом. Использование такой техники позволило сократить количество проверок, а значит и условных переходов при выполнении кодирования методом Хаффмана на видеокарте.

Также для ускорения выполнения кодирования на видеокарте для хранения массива кодов символов и массива длин кодов символов была использована константная память, а для хранения массива `bitsCountBlocks` – разделяемая память (см. Раздел «2.1. Обзор используемых технологий»).

3.3. Результаты тестирования

Размер входных данных при сжатии ключевого кадра составляет порядка 130 Кб (размер изображения после сжатия Гибридным алгоритмом).

Таблица № 3.1 – Результаты тестирования временных показателей CUDA- и ЦП-реализаций метода Хаффмана.

реализация / параметр	Время выполнения на ЦП (мс)	Время выполнения на видеокарте (мс)
ЦП-реализация	8	–
CUDA-реализация	1	16

Как было указано выше, при выполнении кодирования методом Хаффмана на видеокарте предобработка, а также кодирование последних нескольких сот байтов выполняется на ЦП, поэтому для CUDA-реализации также указано время выполнения на ЦП. Очевидно, что метод Хаффмана выполняется дольше на видеокарте. Но, во-первых, суммарное время выполнения алгоритма при CUDA-реализации (17 мс) не слишком велико для осуществления сжатия в режиме реального времени, а во-вторых при этом удаётся значительно снизить использование ЦП.

Таким образом, использование CUDA-реализации метода Хаффмана при сжатии экранного видео позволяет освободить вычислительные мощности ЦП для выполнения других задач.

4. Гибридный алгоритм сжатия изображений

4.1. Описание Гибридного алгоритма и его составных частей

В этом разделе представлен Гибридный алгоритм сжатия изображения без потерь информации, обладающий линейной трудоёмкостью, демонстрирующий высокую степень сжатия двух классов изображений:

1. изображения, типичные для Windows XP;

2. изображения, при прямом обходе пикселей которых, часто чередуются цвета в условиях ограниченного количества этих цветов (например, текст).

Гибридный алгоритм является органическим соединением двух алгоритмов: RLE и Сдвигового.

4.1.1. Алгоритм RLE

Этот алгоритм давно известен и весьма распространён, но поскольку ниже будут приведены результаты тестирования алгоритмов, то стоит описать особенности реализации этого алгоритма.

Идея алгоритма: группа подряд идущих пикселей одного цвета кодируется парой *цвет : количество пикселей.*

Особенности реализации: 3 байта отводится для хранения цвета. Для хранения количества подряд идущих пикселей такого цвета может быть отведено 1 или 2 байта. Если количество подряд идущих пикселей одного цвета меньше, чем 2^7, то используется только 1 байт, первый бит которого равен 0. В противном случае используются оба байта, причём первый бит первого из этих байтов равен 1, а второй байт является младшим. Таким образом, максимальное число пикселей в группе, кодируемой одной парой *цвет : количество пикселей* составляет $2^{15} - 1$.

Коэффициент сжатия в наихудшем случае: 4/3.

Коэффициент сжатия в наилучшем случае: $5 / (2^{15} - 1)$ (при количестве пикселей $>= 2^{15} - 1$).

Преимущества:

1) довольно высокий коэффициент сжатия для изображений, типичных для Windows XP;

2) работает очень быстро, так как в процессе кодирования / декодирования не используются никакие дополнительные структуры данных.

Недостатки:

1) если при прямом обходе пикселей изображения часто чередуются цвета, причём даже в том случае, когда набор цветов ограничен (например, текст), эффективность сжатия резко падает.

4.1.2. Сдвиговый алгоритм

Идея алгоритма: если при прямом обходе пикселей изображения незадолго до текущего пикселя встречался пиксель такого же цвета, то 3 байта, кодирующие цвет пикселя, можно заменить на 1- или 2-байтовую ссылку на пиксель с таким же цветом (далее будет рассмотрена реализация этого алгоритма, использующая 1-байтовую ссылку), а точнее – указать, на сколько байтов нужно сдвинуться назад относительно текущего байта, чтобы получить нужный цвет. Таким образом, может быть выстроено множество списков. Для ускорения работы алгоритма как при кодировании, так и при

декодировании используется хэш-таблица. При кодировании ключом хэш-таблицы является цвет, а значением – номер байта последнего просмотренного пикселя с таким цветом. Если цвет встретился впервые или количество байтов до предыдущего пикселя с таким же цветом превышает $2^8 - 1$, то байт ссылки = 0, а за этим байтом следуют 3 байта, хранящие значение цвета, иначе указывается только ссылка.

При декодировании ключом хэш-таблицы является номер начального байта последнего просмотренного пикселя с таким цветом, а значением – цвет.

Коэффициент сжатия в наихудшем случае: 4/3.

Коэффициент сжатия в наилучшем случае: 1/3.

Преимущества:

1) получаемый при кодировании формат хорошо поддаётся дальнейшему кодированию другими алгоритмами.

Недостатки:

1) даже в наилучшем случае степень сжатия невелика;

2) медленно работает из-за обращения на чтение, а затем на запись к дополнительным структурам данных для каждого пикселя.

Вывод: сам по себе алгоритм неприменим для сжатия изображений, типичных для Windows XP, а также для сжатия изображений, значительную часть которых занимает текст (RLE почти всегда лучше).

4.1.3. Гибридный алгоритм

Этот алгоритм является органическим соединением алгоритмов RLE и сдвигового алгоритма. 2 формата представления закодированной информации объединяются в один следующим образом: *ссылка : цвет : количество пикселей*. При этом количество байтов, отведённых для хранения ссылки, цвета и количества пикселей остаётся таким же, как и в вышеописанных алгоритмах. В процессе кодирования и декодирования производятся все те же действия, что и в предыдущих двух алгоритмах вместе взятых.

Коэффициент сжатия в наихудшем случае: 5/3.

Коэффициент сжатия в наилучшем случае: $6 / (2^{15} - 1)$ (при количестве пикселей $>= 2^{15} - 1$).

Коэффициент сжатия в наихудшем случае хуже, чем у RLE, но худший случай почти никогда не достигается на изображениях, типичных для Windows XP, а также на изображениях, где присутствует текст.

Гибридный алгоритм обладает всеми преимуществами RLE. При его использовании достигается дополнительное сжатие за счёт применения идей сдвигового алгоритма. Гибридный алгоритм не наследует недостатки сдвигового алгоритма. Недостаток (1) устраняется за счёт сжатия алгоритмом RLE. А недостаток (2) присутствует в значительно меньшей степени, так как обращение к дополнительным структурам данных происходит не для каждого пикселя, а для каждой группы, то есть примерно в 16 раз реже при сжатии изображения, типичного для Windows XP и примерно в 7 раз реже при сжатии изображения, значительную часть которого занимает текст (эти цифры получены опытным путём). За счёт применения идей сдвигового алгоритма также устранён недостаток алгоритма RLE – в случае частого чередования цветов при прямом обходе пикселей изображения в условиях ограниченного количества этих цветов степень сжатия значительно выше, чем при сжатии RLE (см. раздел «4.1.4. Результаты тестирования RLE, Сдвигового и Гибридного алгоритмов»).

4.1.4. Результаты тестирования RLE, Сдвигового и Гибридного алгоритмов

Таблица № 4.1 – Результаты тестирования RLE, Сдвигового и Гибридного алгоритмов при сжатии изображения, типичного для Windows XP.

алгоритм \ параметр	время (кодирование / декодирование), мс	коэффициент сжатия
Сдвиговый алгоритм	23 / 64	0,36
RLE	3 / 4	0,06
Гибридный алгоритм	9 / 9	0,05

Таблица № 4.2 – Результаты тестирования RLE, Сдвигового и Гибридного алгоритмов при сжатии изображения, значительную часть которого занимает текст.

алгоритм \ параметр	время (кодирование / декодирование), мс	коэффициент сжатия
Сдвиговый алгоритм	21 / 62	0,36
RLE	4 / 4	0,19
Гибридный алгоритм	11 / 13	0,11

Тестирование проводилось при разрешении монитора 1024*768 и глубине цвета в 32 бита. При этом для хранения каждого компонента цвета пикселя: R, G, B используется 1 байт. Таким образом, размер исходного изображения составляет 1024*768*3 байтов.

Сдвиговый алгоритм показал наихудшие результаты на обоих тестах. А Гибридный алгоритм при допустимом замедлении в работе по сравнению с RLE обеспечил несколько лучший коэффициент сжатия на (1) тесте и значительно лучший коэффициент сжатия на (2) тесте.

Алгоритм RLE показал нестабильную эффективность сжатия дискретно-тоновых изображений (довольно высокую степень сжатия изображений, типичных для Windows XP и значительно более низкую степень сжатия изображений, значительную часть которых занимает текст). Степень сжатия дискретно-тоновых изображений, продемонстрированная Гибридным алгоритмом, более стабильна.

4.2. Модификации гибридного алгоритма сжатия изображений

С момента создания гибридного алгоритма было проведено несколько модификаций алгоритмов RLE и Сдвигового, которые затем были встроены в Гибридный алгоритм, а также был добавлен финальный этап сжатия – кодирование методом Хаффмана. Эти модификации позволили увеличить коэффициент сжатия изображений указанных классов при некотором замедлении в работе.

Кодирование Гибридным алгоритмом (с учётом проведённых модификаций) схематично представлено на Рисунке 4.1. В зависимости от количества подряд идущих пикселей одного цвета, а также в зависимости от количества байтов до предыдущего пикселя с таким же цветом исходный формат (RGB) преобразуется в один из четырёх результирующих форматов.

4.2.1. Финальное кодирование методом Хаффмана

Многие алгоритмы сжатия используют один из статистических методов в качестве финального этапа сжатия. Пожалуй, наиболее распространённые из статистических методов: метод Хаффмана и арифметическое кодирование. Известно, что метод Хаффмана производит идеальное сжатие (сжимает данные до их энтропии), если вероятности символов точно равны отрицательным степеням двойки [20]. Если же это условие не выполнено, то коэффициент сжатия арифметическим кодированием будет

выше, чем коэффициент сжатия методом Хаффмана. Тестирование показало, что при финальном сжатии закодированного Гибридным алгоритмом изображения, метод Хаффмана даёт коэффициент сжатия очень близкий к энтропии. А поскольку арифметическое кодирование работает несколько дольше, чем метод Хаффмана, то было принято решение остановиться именно на методе Хаффмана в качестве финальной стадии сжатия.

В качестве алфавита был выбран набор всех возможных однобайтовых значений (от 0 до 255).

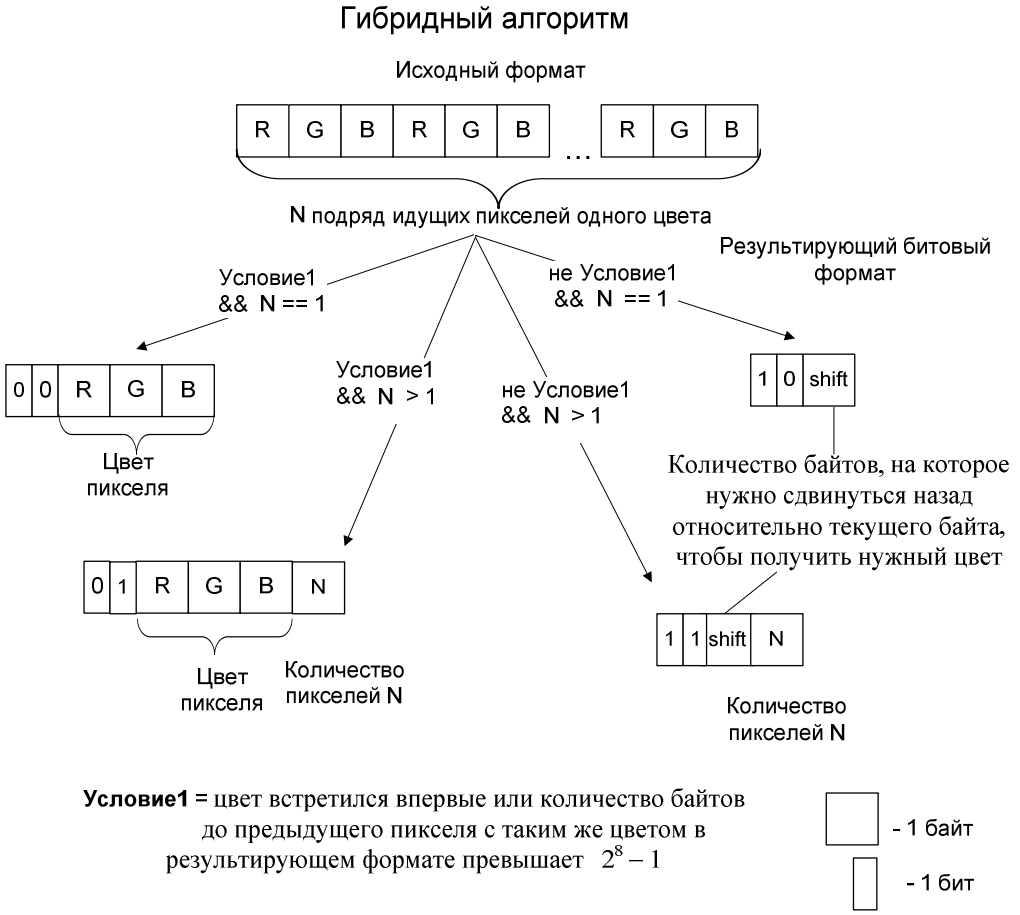

Рисунок 4.1 – Кодирование Гибридным алгоритмом.

4.2.2. Использование битового формата для RLE в составе Гибридного алгоритма

В начальной версии Гибридного алгоритма применялся алгоритм RLE, где для хранения количества подряд идущих пикселей определённого цвета может быть отведено 1 или 2 байта. Если количество подряд идущих пикселей одного цвета меньше, чем 2^7, то используется только 1 байт, первый бит которого равен 0. В противном случае используются оба байта, причём первый бит первого из этих байтов равен 1, а второй байт является младшим. Этот формат является байтовым, так как для хранения ссылки используется 1 или 2 байта, сдвига на некоторое количество битов, не равное 8 не происходит.

Был предложен следующий альтернативный формат RLE: если встречается только 1 пиксель определённого цвета, то за тремя байтами цвета пикселя следует бит, равный 0. Иначе, за тремя байтами цвета пикселя указывается бит, равный 1, а затем однобайтовое количество подряд идущих пикселей этого цвета. Этот формат является битовым, так как после каждой группы пикселей одинакового цвета указывается специальный бит – флаг –

вследствие чего происходит сдвиг на 1 бит. Поэтому, байты цвета пикселя или байт ссылки могут начинаться не с первого бита текущего байта результирующего файла.

Интуитивно понятно, что байтовый формат RLE будет более выгоден при длинных группах одноцветных пикселей, а битовый формат RLE будет выигрывать при значительном количестве единичных пикселей определённого цвета. Максимальная длина серии одноцветных пикселей в байтовом формате составляет $2^{15} – 1$, а в битовом формате всего $2^8 – 1$. Зато на каждом одиночном пикселе определённого цвета битовый формат экономит 7 бит по сравнению с байтовым форматом. Таким образом, при сжатии различных типов изображений предпочтительнее может оказаться тот или иной формат RLE.

При тестировании Гибридного алгоритма на изображениях типичных для Windows XP, а также на изображениях, значительную часть которых занимает текст, выяснилось, что битовый формат RLE в составе Гибридного алгоритма обеспечивает большую степень сжатия, поэтому именно этот вариант RLE был выбран в качестве составляющей Гибридного алгоритма.

4.2.3. Использование битового формата для Сдвигового алгоритма в составе Гибридного алгоритма

В состав начальной версии Гибридного алгоритма входил Сдвиговый алгоритм, где используется однобайтовая ссылка. Если цвет встретился впервые или пиксель такого же цвета встречался слишком далеко от текущей позиции, чтобы можно было на него сослаться, то указывается ссылка равная 0. Затем следуют байты цвета пикселя. Это байтовый формат.

Был разработан следующий альтернативный формат Сдвигового алгоритма: если цвет встретился впервые или количество байтов до предыдущего пикселя с таким же цветом превышает $2^8 – 1$, то в выходной файл записывается бит, равный 0, а затем байты цвета пикселя. Иначе в выходной файл записывается бит, равный 1, а затем однобайтовая ссылка. Это битовый формат.

Байтовый формат Сдвигового алгоритма предпочтительнее в двух случаях:

1. когда количество цветов, присутствующих в изображении, мало;

2. когда расстояние между пикселями одного цвета в большинстве случаев не превышает длину ссылки: $2^8 – 1$ байтов.

Условие (1) является одной из предпосылок для возникновения условия (2). В таких случаях байтовый формат выигрывает за счёт меньшего числа накладных расходов (не используются флаги).

Битовый формат напротив, экономичнее при невыполнении вышеописанных условий, так как экономия битового формата относительно байтового составляет 7 бит на каждом цвете пикселя, встреченном впервые, а также на пикселе, цвет которого встречался слишком далеко от текущей позиции, чтобы на него сослаться.

При тестировании Гибридного алгоритмов на изображениях типичных для Windows XP, а также на изображениях, значительную часть которых занимает текст, оказалось, что битовый формат Сдвигового алгоритма позволяет достичь большей степени сжатия. Но сравнение суммарных коэффициентов сжатия, то есть коэффициентов сжатия после последовательного применения Гибридного алгоритма и метода Хаффмана, битового и байтового формата Сдвигового алгоритма в составе Гибридного алгоритма не позволяет определить однозначно лучший из них для двух типов изображений. Подробнее об этом будет рассказано в следующем Разделе «4.2.4. Объединение флагов в группы по 8 бит (1 байт)».

4.2.4. Объединение флагов в группы по 8 бит (1 байт)

В двух предыдущих разделах обосновывается целесообразность использования битового формата для RLE и Сдвигового алгоритма в составе Гибридного алгоритма. Но тогда частоты различных значений байтов результирующего формата Гибридного алгоритма в значительной степени уравняются, так как каждый флаг будет производить сдвиг на 1 бит. Поэтому эффективность финального сжатия методом Хаффмана, основанного на сопоставлении часто встречающимся символам более коротких кодов и сопоставлении редко встречающимся символам более длинных кодов, резко падает.

Один из способов уменьшить этот эффект – это объединение флагов в группы по 8 бит. Для этого заводится небольшой буфер (несколько десятков байтов), в который записываются данные нескольких последовательных групп, которым соответствует в сумме 8 флагов. Сначала в выходной файл записывается байт флагов, а затем все данные из буфера. Буфер очищается.

При использовании этого способа, хаотичные изменения, вносимые в результирующий формат битами флагов, локализуются в отдельных байтах, а частоты байтов цвета пикселей не изменяются по сравнению с байтовым форматом. Конечно, за счёт наличия байтов флагов степень сжатия методом Хаффмана всё равно несколько ниже, чем при использовании байтового формата.

Понятно, что при использовании битового формата и RLE и Сдвигового алгоритма в составе Гибридного алгоритма количество байтов флагов вдвое больше, чем при использовании битового формата только для RLE или только для Сдвигового алгоритма. По этой самой причине, как свидетельствуют результаты тестирования Гибридного алгоритмов на изображениях, значительную часть которых занимает текст, байтовый формат Сдвигового алгоритма в составе Гибридного алгоритма показал более высокий суммарный коэффициент сжатия. Хотя после первой стадии сжатия – применения Гибридного алгоритма – коэффициент сжатия битового формата Сдвигового алгоритма был выше, чем у байтового формата. В то же время битовый формат Сдвигового алгоритма в составе Гибридного алгоритма показал более высокую суммарную эффективность при сжатии изображений, типичных для Windows XP.

Таким образом, оказалось, что как битовый, так и байтовый форматы Сдвигового алгоритма в составе Гибридного алгоритма имеют право на существование и могут быть задействованы каждый в своей, несколько сузившейся сфере применения.

4.2.5. Отсчёт ссылки по исходному файлу, а не по закодированному

Была также рассмотрена следующая идея изменения Сдвигового алгоритма: отсчёт ссылки производить не по закодированному файлу, а по исходному. При этом стало возможно отсчитывать ссылку не в байтах, а в пикселях. Следовательно, ссылаться можно до 256 пикселей назад. Такая модификация Сдвигового алгоритма увеличила его степень сжатия дискретно-тоновых изображений.

Но для Сдвигового алгоритма в составе Гибридного алгоритма такая модификация ухудшила бы коэффициент сжатия, так как среднее количество пикселей в группе для него составляет несколько десятков (такие результаты получены опытным путём), а на одну группу при этом затрачивается всего несколько байтов. Поэтому при отсчёте ссылки по закодированному файлу реально можно покрыть гораздо большее количество пикселей. Поэтому в Разделе «4.2.6. Результаты тестирования различных модификаций Гибридного алгоритма» не приводятся данные о Гибридном алгоритме, в состав которого включен Сдвиговый алгоритм с отсчётом ссылки по исходному файлу.

4.2.6. Результаты тестирования различных модификаций Гибридного алгоритма

Таблица № 4.3 – Результаты тестирования модификаций гибридного алгоритма при сжатии изображения, типичного для Windows XP.

модификация алгоритма / параметр	время (кодирование / декодирование), мс	время (код-ие / дек-ие алгоритмом Хаффмана), мс	коэффициент сжатия Гибридным алгоритмом	суммарный коэффициент сжатия	размер изображения после основного сжатия	размер изображения после финального сжатия
1. Стандартный гибридный алгоритм	5 / 8	8 / 10	0,08083	0,0603	190697	142274
2. Гибридный алгоритм с полностью битовым форматом и объединением флагов в группы	6 / 9	6 / 8	0,0553	0,0505	130468	119151
3. Гибридный алгоритм с битовым форматом RLE и объединением флагов в группы	6 / 9	7 / 8	0,06348	0,05499	149779	129748

Таблица № 4.4 – Результаты тестирования модификаций гибридного алгоритма при сжатии изображения, значительную часть которого занимает текст.

модификация алгоритма / параметр	время (кодирование / декодирование), мс	время (код-ие / дек-ие алгоритмом Хаффмана), мс	коэффициент сжатия Гибридным алгоритмом	суммарный коэффициент сжатия	размер изображения после основного сжатия	размер изображения после финального сжатия
1. Стандартный гибридный алгоритм	8 / 14	8 / 13	0,12182	0,0789	287404	186138
2. Гибридный алгоритм с полностью битовым форматом и объединением флагов в группы	8 / 14	8 / 13	0,09801	0,07637	231238	180171
3. Гибридный алгоритм с битовым форматом RLE и объединением флагов в группы	9 / 13	8 / 12	0,09879	0,07444	233070	175623

Замечание: к размеру изображения после финального сжатия надо прибавлять размер массива частот символов, который также записывается в закодированный файл. Этот размер составляет 256 * 4 = 1 килобайт.

В качестве изображения, значительную часть которого занимает текст, использовался снимок рабочего стола во время просмотра в браузере Opera сайта с преимущественно текстовым наполнением.

Размер исходного изображения составляет 1024 * 768 * 3 = 2359296 байтов. Формат исходного изображения – RGB, причём для хранения каждой из трёх компонент цвета отводится 1 байт.

4.3. Практическое сравнение Гибридного алгоритма с некоторыми другими алгоритмами

Очень важно сравнить показатели Гибридного алгоритма с показателями как можно большего числа алгоритмов, чтобы получить реалистичные данные о конкурентоспособности Гибридного алгоритма и необходимости внесения модификаций в целях увеличения коэффициента сжатия или же ускорения работы алгоритма.

В этом разделе рассмотрены результаты тестирования следующих алгоритмов:

1. Гибридный алгоритм. В соответствии с результатами, описанными в Разделе «4.2. Модификации гибридного алгоритма сжатия изображений», для сжатия изображения, типичного для Windows XP, был применён вариант алгоритма с полностью битовым форматом. А для сжатия изображения, значительную часть которого занимает текст, использовался вариант алгоритма с битовым форматом RLE, но байтовым форматом Сдвигового алгоритма. И в том и в другом случаях применялось объединение флагов в группы, а также метод Хаффмана в качестве финальной стадии сжатия. Размер сжатого файла для Гибридного алгоритма указан с учётом хранения частот символов (1 килобайт). Язык – Си++.

2. Jpeg 2000 (режим работы – без потерь информации). Для тестирования использовалась реализация j2k_with_interface, происхождение которой установить не удалось. Язык – Си++.

3. Lossless Jpeg. В тестировании участвовала реализация University of British Columbia. К сожалению, не удалось замерить время кодирования и декодирования для этой реализации. Найти реализацию под Windows, где получилось бы замерить время кодирования и декодирования, не удалось. Язык – Си++.

4. Deflate. В тестировании была задействована реализация Microsoft. Соответствующий класс реализован в .NET Framework 2.0. Язык – C#.

Замечание: в этом тестировании не участвуют RLE и Сдвиговый алгоритм, так как в разделе «4.1.4. Результаты тестирования RLE, Сдвигового и Гибридного алгоритмов» было показано, что Гибридный алгоритм превосходит RLE и Сдвиговый алгоритм по степени сжатия на обоих рассматриваемых типах изображений. В Разделе 4.1.4. также было показано, что алгоритм RLE, хоть и работает быстрее, чем Гибридный алгоритм, но обеспечивает значительно менее стабильную степень сжатия дискретно-тоновых изображений.

Таблица № 4.5 – Результаты тестирования алгоритмов при сжатии изображения, типичного для Windows XP.

алгоритм / параметр	время (кодирование / декодирование), мс	коэффициент сжатия	размер изображения после сжатия
1. Гибридный алгоритм	18 / 16	0,04704	110977

2. Jpeg 2000	347 / 247	0,04998	117928
3. Lossless Jpeg		0,07076	166937
4. Deflate	47 / 16	0,04544	107200

Таблица № 4.6 – Результаты тестирования алгоритмов при сжатии изображения, значительную часть которого занимает текст.

алгоритм / параметр	время (кодирование / декодирование), мс	коэффициент сжатия	размер изображения после сжатия
1. Гибридный алгоритм	18 / 21	0,05007	118119
2. Jpeg 2000	408 / 247	0,04998	117928
3. Lossless Jpeg		0,07524	177523
4. Deflate	47 / 16	0,04385	103460

Размер исходного изображения составляет 1024 * 768 * 3 = 2359296 байтов. Формат исходного изображения – RGB, причём для хранения каждой из трёх компонент цвета отводится 1 байт.

В качестве изображения, значительную часть которого занимает текст, использовался снимок рабочего стола во время просмотра в текстовом редакторе WordPad текстового документа.

Алгоритмы Jpeg 2000 и Lossless Jpeg предназначены для сжатия непрерывно-тоновых изображений, например, фотографий. Поэтому по совокупности показателей (время сжатия и коэффициент сжатия) они проигрывают двум другим алгоритмам. Jpeg 2000 даёт схожие с Гибридным алгоритмом коэффициенты сжатия, но при этом проигрывает в быстродействии более чем в 10 раз. Lossless Jpeg демонстрирует значительно меньшую степень сжатия, чем остальные участники тестирования. Поэтому даже не так важно, насколько быстро он работает.

Алгоритм Deflate, который предназначен для сжатия дискретно-тоновых изображений (также как и Гибридный алгоритм) показал наилучшие коэффициенты сжатия в обоих случаях. При сжатии изображений, типичных для Windows XP, коэффициент сжатия Deflate незначительно превосходит аналогичный показатель Гибридного алгоритма. А при сжатии изображения, значительную часть которого занимает текст, алгоритм Deflate показал значительно более эффективное сжатие, чем Гибридный алгоритм. Но стоит обратить внимание на тот факт, что время кодирования алгоритма Deflate составляет 47 мс, что более чем в 2 раза дольше, чем у Гибридного алгоритма. Поэтому при кодировании изображений, типичных для Windows XP, в том случае, когда время сжатия является критичным параметром, уже сейчас можно порекомендовать Гибридный алгоритм, как дающий оптимальное соотношение эффективности и продолжительности сжатия. Очевидно также, что необходимо вносить новые усовершенствования в Гибридный алгоритм, чтобы приблизившись по продолжительности сжатия к алгоритму Deflate, превзойти его по степени сжатия или же, значительно уменьшив отставание по степени сжатия, не растерять преимущества по продолжительности сжатия. Такая возможность есть. Тема перспектив развития Гибридного алгоритма будет затронута в Заключении.

5. Описание программы

5.1. Краткое описание архитектуры

Приложение состоит из двух проектов: EncoderMFCInterface и DecoderWinAPIInterface. Первый осуществляет получение моментальных снимков экрана, сжатие видео и запись его на жёсткий диск, а в задачи второго входит чтение видео из файла, декодирование и проигрывание видео.

Оба проекта реализованы на языке Си++. Исключением являются два модуля, написанных на языке Си, которые выполняются на видеокарте с помощью технологии NVidia CUDA. Использование именно языка Си для реализации этих модулей – ограничение, накладываемое технологией CUDA. Оба проекта реализованы под Windows.

В проекте EncoderMFCInterface используются технологии NVidia CUDA (CUDA Runtime), DirectX9.0c для выполнения операций попиксельного и поблокового сравнения на видеокарте. Для создания пользовательского интерфейса использовалась технология MFC.

В проекте DecoderWinAPIInterface для проигрывания видео используется технология Win API.

Способ взаимодействия потоков в кодере и декодере, а также структура и взаимосвязь классов этих проектов достаточно проста. Поскольку пользовательский интерфейс на данный момент слабо развит в кодере и в декодере, то существует всего несколько несложных вариантов использования, которые не требуют документирования. В разработанном ПО можно выделить два компонента: кодер и декодер, причём они практически независимы, так как кодирование и декодирование могут выполняться в разное время и на различных компьютерах, что позволяет отказаться от составления диаграмм компонентов. Таким образом, не требуется составлять полный набор диаграмм, включающий диаграммы использования, диаграммы активности, диаграммы последовательности, структурные диаграммы классов, диаграммы компонентов.

В данном случае для понимания устройства проектов и осуществления поддержки ПО необходимы лишь краткие текстовые описания взаимодействия потоков, а также иллюстрации к структурным диаграммам классов, где отражены только основные методы большинства классов.

5.2. Кодер EncoderMFCInterface

Кодер EncoderMFCInterface создавался для кодирования и записи на жёсткий диск специфического типа видеоданных – экранного видео. Поэтому получение моментальных снимков экрана интегрировано в кодер.

5.2.1. Взаимодействие потоков

В приложении EncoderMFCInterface функционируют два потока. Рассмотрим их назначение, основные действия, выполняемые каждым из этих потоков, и способ их взаимодействия.

При запуске приложения запускается основной поток – поток пользовательского интерфейса. Он проводит инициализацию, в ходе которой создаются необходимые экземпляры классов, и происходит выделение памяти из кучи. После этого пользователю предоставляется возможность запустить запись видео происходящего на экране. Для этого ему нужно нажать на кнопку с надписью «Start screen capture». При нажатии на эту кнопку запускается второй поток.

Сначала второй поток проводит инициализацию, необходимую для использования технологии NVidia CUDA, так как одним из требований этой технологии является

проведение инициализации тем же потоком, который затем будет запускать программу, выполняемую на видеокарте. Затем второй поток запускает основной цикл, в котором последовательно происходит

1. проверка флага `_exitThreadFlag`. Если он равен true, выход. Иначе – переход на следующий шаг.

2. получение моментального снимка экрана;

3. сжатие этого снимка алгоритмом, выбираемым в зависимости от того, является ли текущий кадр ключевым или промежуточным;

4. запись сжатого кадра на жёсткий диск.

Если пользователь нажмёт кнопку с надписью «Stop screen capture», то будут выполнены действия, направленные на завершение второго потока. Для этого флаг завершения потока `_exitThreadFlag` устанавливается в true. Как видно из описания основного цикла второго потока, после установления `_exitThreadFlag` в true произойдёт выход из основного цикла после завершения записи на жёсткий диск текущего кадра. После этого второй поток завершает своё выполнение, так как выходит из функции `MainThreadProcedure()` класса EncodeThreadProcessor, которая является для него стартовой.

Если пользователь закрыл главное окно приложения, запустив второй поток, но не нажав кнопку с надписью «Stop screen capture», то выполнение второго потока также будет корректно завершено. Для этого сначала `_exitThreadFlag` устанавливается в true, а затем идёт ожидание, пока не освободится семафор `_threadExistenceSemaphore`. Этот семафор устанавливается в положение «занят» при запуске второго потока, и устанавливается в положение «свободен» непосредственно перед завершением выполнения второго потока.

5.2.2. Описание структуры проекта

Дадим общее функциональное описание классов и прочих модулей исходного кода.

Сначала рассмотрим основные классы, которые используются как в кодере, так и в декодере.

1. Класс **Coder** является абстрактным и содержит данные и методы их обработки, необходимые всем классам, содержащим реализации алгоритмов кодирования и декодирования данных. Это набор битовых масок `_masks`, вычисляемых в конструкторе класса, а также буфер `_buffer`, используемый для хранения закодированного кадра.

2. Класс **HashTable** является реализацией хэш-таблицы.

3. Класс **ThreadProcessor** является классом-обёрткой системной функции `CreateThread()`. Этот класс позволяет назначать в качестве стартовой функции создаваемого потока метод класса, чего нельзя сделать напрямую, использую системную функцию `CreateThread()`. Для того, чтобы воспользоваться этой функциональностью нужно вызвать метод `CreateThread()`. Стартовой функцией создаваемого потока, является метод `MainThreadProcedure()`, который объявлен как абстрактный. Его реализация возложена на потомков класса.

4. Класс **ClassicHaffmanCoder** явлется потомком класса **Coder** и содержит данные и методы их обработки, необходимые как при кодировании, так и при декодировании методом Хаффмана.

Кратко рассмотрим прочие классы, которые используются в кодере и в декодере, но не отражённы ни на одной из диаграмм по причине их второстепенности.

5. Класс **Utility** содержит набор разнообразных функций, используемых для отладки и подбора оптимальных параметров алгоритмов. Например, метод `GeneratePrimeNumbers()` позволяет сгенерировать набор простых чисел. Этот метод использовался при подборе количества списков в хэш-таблице. А метод `CreateBMPFile()` позволяет сохранить изображение в виде файла с расширением bmp в целях отладки.

6. Класс **Tester** содержит набор тестирующих методов. Некоторые из этих методов моделирует поведение процессора видеокарты при выполнении операций попиксельного и поблокового сравнения с помощью технологии CUDA.

7. Класс **Logger** содержит метод `LogToFile()`, позволяющий записывать некоторые данные в файл в ходе работы приложения, что бывает полезно для отладки.

8. Класс **Converter** содержит методы, позволяющие выполнять различные преобразования. Например, метод `FromBytesToWORD()`преобразовывает 2 соседних байта в массиве в двухбайтовое беззнаковое целое, а метод `FromWORDToBytes()` наоборот преобразует двухбайтовое беззнаковое целое в набор из двух байтов и записывает их в массив переданный в качестве параметра по указателю.

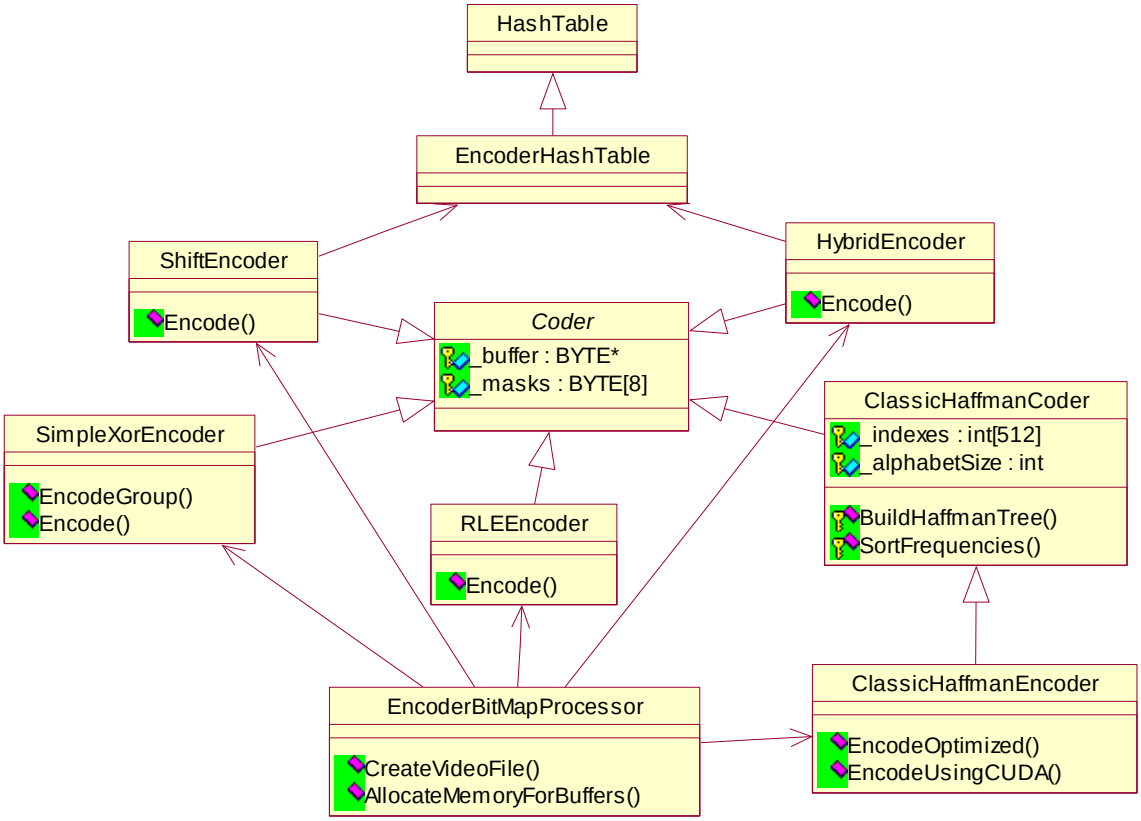

Диаграмма 5.1 Иллюстрация к структурной диаграмме классов кодера (часть 1)

Теперь рассмотрим классы и другие файлы, содержащие исходный код программы, относящиеся только к проекту EncoderMFCInterface.

1. Класс **EncoderBitMapProcessor** является ключевым классом проекта. В его методе `CreateVideoFile()` содержится основной цикл, выполняемый вторым потоком, описанный выше. Этот класс координирует действия различных классов, содержащих конкретные реализации алгоритмов сжатия, а также производит запись закодированной информации в файл. Метод `AllocateMemoryForBuffers()` вызывается во время

инициализации, и создаёт экземпляры классов, инкапсулирующих различные алгоритмы сжатия, а также производит выделение памяти из кучи.

2. Класс **SimpleXorEncoder** инкапсулирует алгритмы сжатия видео, основанные на сравнении изображений. Алгоритм, основанный на попиксельном сравнении изображений, реализован в методе `Encode()`. А алгоритм, основанный на поблоковом сравнении изображений, реализован в методе `EncodeGroup()`.

3. Класс **ClassicHaffmanEncoder** является потомком класса **ClassicHaffmanCoder** и инкапсулирует кодирование метода Хаффмана. Для выполнения кодирования на ЦП нужно вызвать метод `EncodeOptimized()`. А чтобы задействовать CUDA-реализацию метода Хаффмана, надо вызвать метод `EncodeUsingCUDA()`.

4. Класс **RLEEncoder** инкапсулирует алгоритм сжатия RLE. Его можно использовать, вызвав метод `Encode()`.

5. Класс **ShiftEncoder** инкапсулирует Сдвиговый алгоритм сжатия. Его можно использовать, вызвав метод `Encode()`.

6. Класс **HybridEncoder** инкапсулирует Гибридный алгоритм сжатия. Его можно использовать, вызвав метод `Encode()`.

7. Класс **EncoderHashTable** является реализацией хэш-таблицы, оптимизированной для использования при сжатии Сдвиговым и Гибридным алгоритмами.

Замечание: на Диаграмме 5.1 не указаны методы классов **HashTable** и **EncoderHashTable**, так как эти классы реализуют стандартный интерфейс, предоставляемый хэш-таблицей.

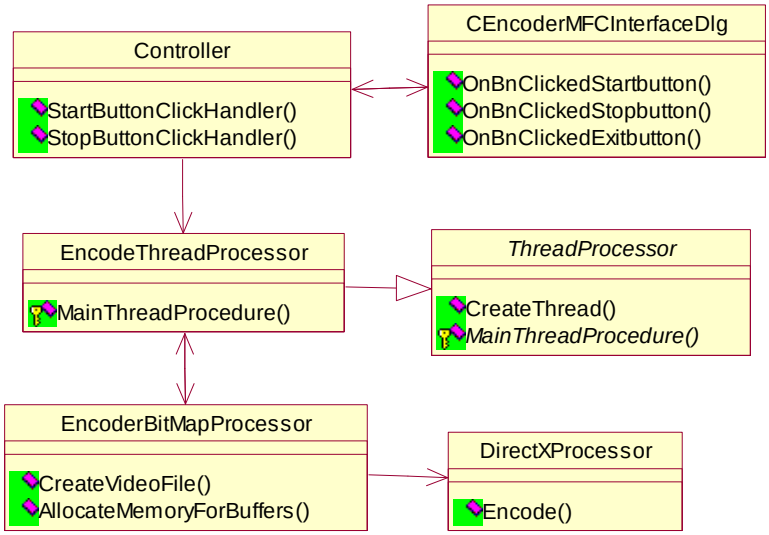

Диаграмма 5.2 Иллюстрация к структурной диаграмме классов кодера (часть 2)

8. Класс **DirectXProcessor** содержит данные и методы их обработки, необходимые для проведения операций попиксельного и поблокового сравнения изображений с помощью пиксельных шейдеров. Для того, чтобы воспользоваться этой функциональностью, нужно вызвать метод `Encode()`, причём тип выполняемой операции (попиксельное или поблоковое сравнение) определяется по значениям параметров метода `Encode()` `factorX` и `factorY`. Если оба этих параметра равны 1, то выполняется попиксельное сравнение, иначе – поблоковое.

9. Класс **CEncoderMFCInterfaceDlg** содержит методы, которым передаётся управление при возникновении некоторого события главной формы приложения. Метод `OnBnClickedStartbutton()` перехватывает событие нажатия на кнопку `Startbutton`. Метод `OnBnClickedStopbutton()` перехватывает событие нажатия на кнопку `Stopbutton`. Метод `OnBnClickedExitbutton()`перехватывает событие нажатия на кнопку `Exitbutton`. Этот класс можно рассматривать как Вид в схеме Модель-Вид-Контроллер. При возникновении некоторого события, его обработка предоставляется классу **Controller**.

10. Класс **Controller** содержит обработчики событий главной формы приложения. Метод `StartButtonClickHandler()` является обработчиком события нажатия на кнопку `Startbutton`. Метод `StopButtonClickHandler()` является обработчиком события нажатия на кнопку `Stopbutton`. Этот класс можно рассматривать как Контроллер в схеме Модель-Вид-Контроллер.

11. Класс **EncodeThreadProcessor**, являясь потомком класса **ThreadProcessor**, реализует метод `MainThreadProcedure()`, который является стартовой функцией второго потока. В этом методе вызывается метод `CreateVideoFile()` экземпляра класса **EncoderBitMapProcessor** (см. описание класса **EncoderBitMapProcessor**).

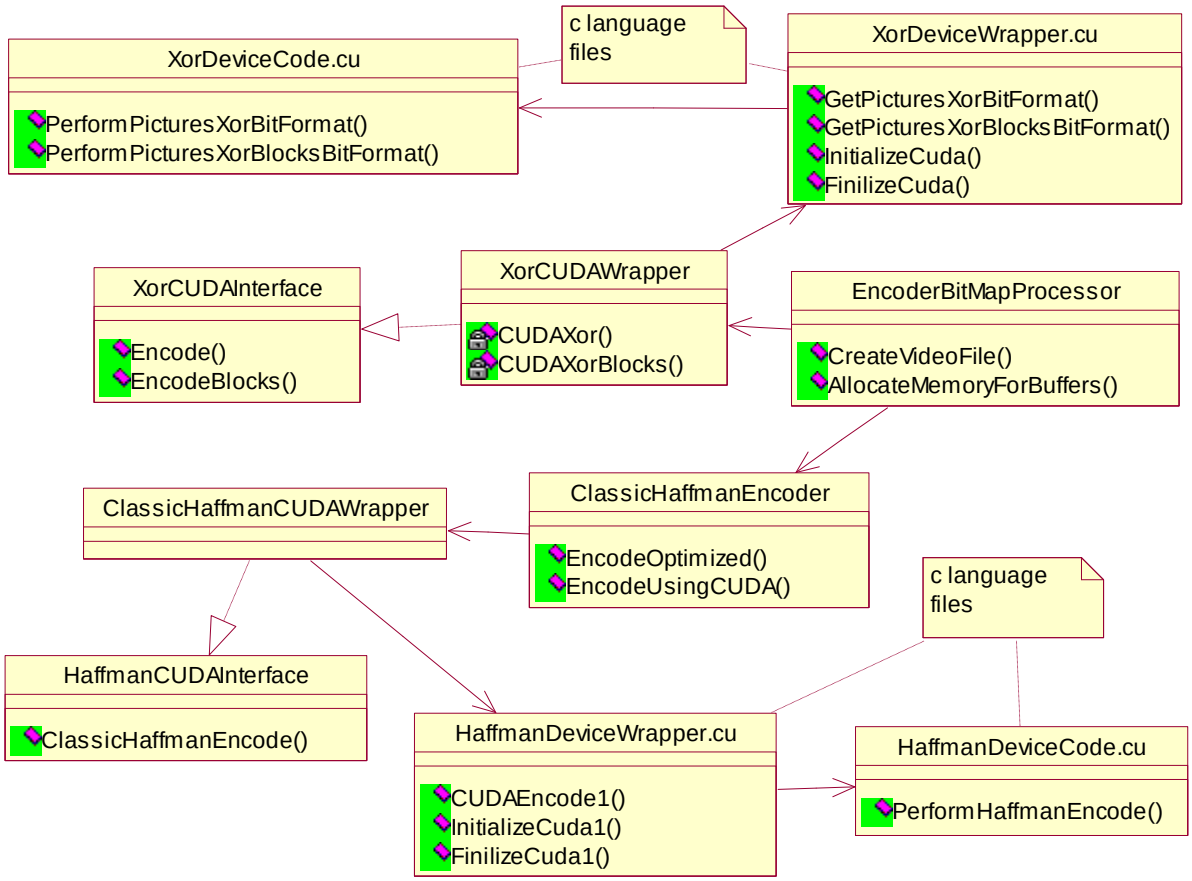

Диаграмма 5.3 Иллюстрация к структурной диаграмме классов кодера (часть 3)

12. Класс **XorCUDAWrapper** реализует интерфейс **XorCUDAInterface** для обращения к функциям, находящимся в файлах, содержащим исходный код программы, **XorDeviceCode.cu** и **XorDeviceWrapper.cu**, которые написаны на языке Си, что является требованием технологии CUDA. Выполнить операцию попиксельного сравнения изображений с помощью технологии CUDA можно, вызвав метод `Encode()`. А выполнить операцию поблокового сравнения изображений можно, вызвав метод `EncodeBlocks()`.

13. Файл **XorDeviceWrapper.cu** содержит функции, позволяющие осуществлять доступ к вычислительным мощностям видеокарты с помощью технологии CUDA при выполнении поблокового и попиксельно сравнения. Для инициализации необходимо вызвать функцию `InitializeCuda()`. Для выполнения операции попиксельного сравнения, необходимо вызвать функцию `GetPicturesXorBitFormat()`, а для выполнения операции поблокового сравнения нужно вызвать функцию `GetPicturesXorBlocksBitFormat()`. Для освобождения использованных ресурсов необходимо вызвать функцию `FinilizeCuda()`.

14. Файл **XorDeviceCode.cu** содержит функции, выполняемые непосредственно видеокартой. Попиксельное сравнение выполняется функцией `PerformPicturesXorBitFormat()`, а поблоковое сравнение выполняется функцией `PerformPicturesXorBlocksBitFormat()`.

15. Класс **ClassicHaffmanCUDAWrapper** реализует интерфейс **HaffmanCUDAInterface** для обращения к функциям, находящимся в файлах, содержащим исходный код программы, **HaffmanDeviceWrapper.cu** и **HaffmanDeviceCode.cu**, которые написаны на языке Си, что является требованием технологии CUDA. Для выполнения кодирования методом Хаффмана с помощью видеокарты нужно вызвать метод `ClassicHaffmanEncode()`.

16. Файл **HaffmanDeviceWrapper.cu** содержит набор функций, позволяющих осуществлять доступ к вычислительным мощностям видеокарты с помощью технологии CUDA при выполнении кодирования методом Хаффмана. Для инициализации необходимо вызвать функцию `InitializeCuda1()`. Для выполнения кодирования методом Хаффмана необходимо вызвать функцию `CUDAEncode1()`. Для освобождения использованных ресурсов необходимо вызвать функцию `FinilizeCuda1()`.

17. Файл **HaffmanDeviceCode** содержит набор функций, выполняемых непосредственно видеокартой. Кодирование методом Хаффмана выполняется функцией `PerformHaffmanEncode()`.

5.2.3. Результаты тестирования

В ходе тестирования замерялись 3 параметра: размер результирующего файла, средняя загрузка ЦП, а также интервал, в котором варьируется загрузка ЦП. При этом проводилась обработка и запись в файл двадцати кадров в секунду. Запись ключевого кадра происходит один раз в секунду. При тестировании каждой реализации в течение минуты выполнялись следующие действия:

1. скроллинг;

2. перемещение окна;

3. выделение текста;

4. сворачивание окна и открытие свёрнутого.

Размер результирующего файла также замерялся после записи видео в течение одной минуты.

5.2.3.1. Результаты тестирования кодера EncoderMFCInterface при различных настройках.

В этом разделе приведены результаты тестирования кодера EncoderMFCInterface при установке различных значений двух параметров: типа операции сравнения и используемой технологии. При тестировании CUDA-реализации, последний этап сжатия – кодирование методом Хаффмана – выполнялся на видеокарте. При тестировании пиксельных шейдеров, кодирование методом Хаффмана выполнялось на ЦП.

Таблица № 5.1 – Результаты тестирования кодера EncoderMFCInterface при различных настройках.

Тип операции сравнения	Размер результирующего файла (Мб)	Используемая технология	Средняя загрузка ЦП (%)	Интервал загрузки ЦП (%)
Попиксельное сравнение	8,9	NVidia CUDA	10	7 - 12
		пиксельные шейдеры	12	10 - 15
		ЦП	15	10 - 18
Поблоковое сравнение	8,7	NVidia CUDA	10	7 - 12
		пиксельные шейдеры	12	10 - 15
		ЦП	13	11 - 16

Проведём анализ результатов тестирования. Размеры результирующих файлов близки при использовании попиксельно и поблокового xor. CUDA-реализация обеспечила наименьшую загрузку ЦП как в случае попиксельного, так и в случае поблокового сравнения. Наибольшая загрузка ЦП была продемонстрирована ЦП-реализацией в обоих случаях. Вывод: CUDA-реализация попиксельного и поблокового сравнения изображений превосходит ЦП-реализацию, так как CUDA-реализация демонстрирует меньший процент загрузки ЦП. Пиксельные шейдеры не продемонстрировали значительного уменьшения загрузки ЦП относительно ЦП-реализации. При этом, как было показано в Разделе «2.3. Результаты тестирования», пиксельный шейдер выполняет сравнение изображений медленнее, чем ЦП-реализация. Поэтому для того, чтобы использовать пиксельные шейдеры при сжатии экранного видео требуется провести дополнительные исследования. Таким образом, результаты, полученные при тестировании различных модификаций кодера EncoderMFCInterface, вполне согласуются с результатами, описанными в Разделе «2.3. Результаты тестирования».

5.2.3.2. Практическое сравнение с некоторыми кодерами

В этом разделе приведены результаты тестирования нескольких кодеров, предназначенных для сжатия экранного видео.

Таблица № 5.2 – Результаты тестирования нескольких кодеров.

Кодер \ Параметр	Размер результирующего файла (Мб)	Загрузка ЦП (%)	Интервал загрузки ЦП (%)
1. Quick Screen Recorder 1.5	144	20	17 - 26
2. Easy Screen Capture Video	144	19	16 - 25
3. Freeze Screen Video Capture	147	20	18 - 29
4. BB FlashBack 2.0.1	7	15	7 - 25
5. Super Screen Recorder 4	148	17,5	15 - 22
6. CamStudio 2.0	149	20	18 - 30

Замечание: кодер FlashBack тестировался в режиме GDI, не использующем mirror видеодрайвер, так как mirror видеодрайвер не является универсальной технологией и может быть задействована только в нескольких ОС семейства Windows (см. Раздел «2.2.7. Сравнение алгоритма, основанного на попиксельном xor и алгоритма, используемого в VNC для сжатия экранного видео»).

Участники тестирования под номерами 1, 2, 3, 5 и 6 продемонстрировали высокий уровень загрузки ЦП и очень большой размер результирующего файла. Все эти кодеры значительно уступают кодеру EncoderMFCInterface (при любых настройках). Единственный из шести участников тестирования, который при близком проценте

загрузки ЦП продемонстрировал несколько меньший размер результирующего файла по сравнению с кодером EncoderMFCInterface, это - BB FlashBack 2.0.1 (участник под номером 4). Стоит заметить, что BB FlashBack 2.0.1 отличается довольно высокой стоимостью – 225 $ за одну копию, в то время как стоимость одной копии Quick Screen Recorder 1.5 составляет всего 29 $.

Результаты тестирования показывают, что реализованный в ходе работы над дипломом кодер EncoderMFCInterface превосходит по совокупности показателей значительную часть существующих коммерческих реализаций. Но очевидно, что существуют кодеры (такие как BB FlashBack 2.0.1), предназначенные для сжатия экранного видео, которые превосходят кодер EncoderMFCInterface по размеру результирующего файла. Поэтому необходимо продолжать исследования в этой области с целью увеличения степени сжатия экранного видео, обеспечиваемой кодером EncoderMFCInterface.

5.3. Декодер DecoderWinAPIInterface

Декодер DecoderWinAPIInterface производит чтение видеоданных с жёсткого диска и их декодирование. Проигрыватель видео интегрирован в декодер.

5.3.1. Взаимодействие потоков

В приложении DecoderWinAPIInterface функционируют два потока. Рассмотрим их назначение, основные действия, выполняемые каждым из этих потоков, и способ их взаимодействия.

При запуске приложения запускается основной поток – поток пользовательского интерфейса. Он проводит инициализацию, в ходе которой создаются необходимые экземпляры классов, и происходит выделение памяти из кучи. После этого запускается второй поток, который выполняет следующие действия в цикле:

1. Проверка флага `_exitThreadFlag`. Если он равен true, выход. Иначе – переход на следующий шаг;

2. если файл, содержащий закодированные данные, прочитан до конца, выход. Иначе, перейти на следующий шаг;

3. прочитать из файла закодированные данные, соответствующие одному кадру;

4. декодировать данные, прочитанные на предыдущем шаге;

5. установить декодированный на предыдущем шаге кадр в качестве текущего;

6. известить поток пользовательского интерфейса об изменении текущего кадра.

После того, как текущий кадр изменился, поток пользовательского интерфейса выполняет его отрисовку в главном окне приложения.

Когда пользователь закрывает главное окно приложения, второй поток должен быть корректно завершён. Для этого используется тот же механизм, что и в приложении EncoderMFCInterface. То есть сначала `_exitThreadFlag` устанавливается в true, а затем идёт ожидание, пока не освободится семафор `_threadExistenceSemaphore`. Этот семафор устанавливается в положение «занят» при запуске второго потока, и устанавливается в положение «свободен» непосредственно перед завершением выполнения второго потока, то есть перед выходом из стартовой функции второго потока.

5.3.2. Описание структуры проекта

Дадим общее функциональное описание классов проекта.

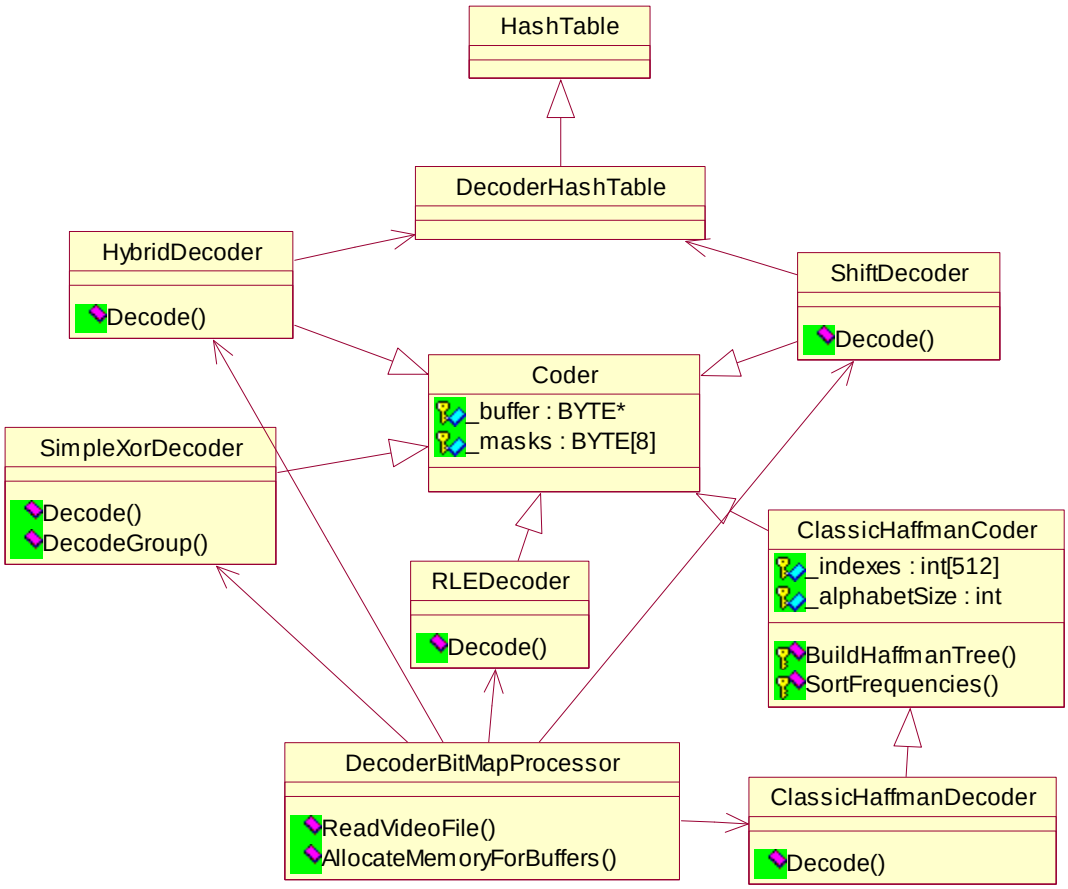

Диаграмма 5.4 Иллюстрация к структурной диаграмме классов декодера (часть 1)

1. Класс **DecoderBitMapProcessor** является ключевым в процессе декодирования экранного видео. В методе `ReadVideoFile()`содержится основной цикл, выполняемый вторым потоком, описанный выше. Этот класс координирует действия различных классов, содержащих конкретные реализации алгоритмов декодирования, а также производит чтение закодированной информации из файла. Метод `AllocateMemoryForBuffers()` вызывается во время инициализации, и создаёт экземпляры классов, инкапсулирующих различные алгоритмы декодирования, а также производит выделение памяти из кучи.

2. Класс **SimpleXorDecoder** инкапсулирует алгритмы декодирования видео, основанные на сравнении изображений. Алгоритм, основанный на попиксельном сравнении изображений, реализован в методе `Decode()`. А алгоритм, основанный на поблоковом сравнении изображений, реализован в методе `DecodeGroup()`.

3. Класс **ClassicHaffmanDecoder** является потомком класса **ClassicHaffmanCoder** и инкапсулирует декодирование методом Хаффмана. Его можно использовать, вызвав метод `Decode()`.

4. Класс **RLEDecoder** инкапсулирует алгоритм декодирования RLE. Его можно использовать, вызвав метод `Decode()`.

5. Класс **ShiftDecoder** инкапсулирует Сдвиговый алгоритм сжатия (декодирование). Его можно использовать, вызвав метод `Decode()`.

6. Класс **HybridDecoder** инкапсулирует Гибридный алгоритм сжатия (декодирование). Его можно использовать, вызвав метод `Decode()`.

7. Класс **DecoderHashTable** является реализацией хэш-таблицы, оптимизированной для использования при декодировании Сдвиговым и Гибридным алгоритмами.

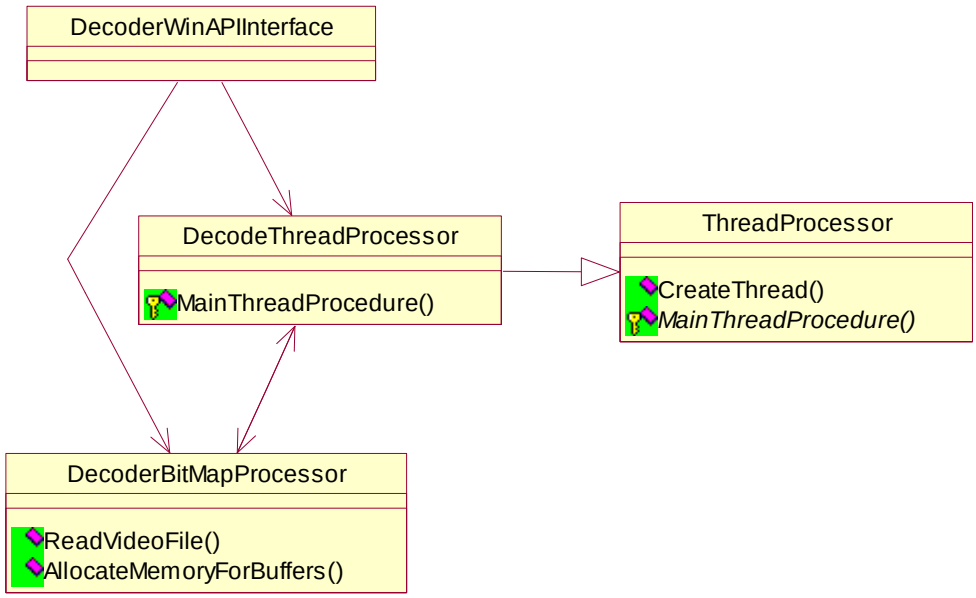

Диаграмма 5.5 Иллюстрация к структурной диаграмме классов декодера (часть 2)

8. Класс **DecoderWinAPIInterface** содержит методы, которым передаётся управление при возникновении некоторого события главного окна приложения.

9. Класс **DecodeThreadProcessor**, являясь потомком класса **ThreadProcessor**, реализует метод `MainThreadProcedure()`, который является стартовой функцией второго потока. В этом методе вызывается метод `ReadVideoFile()` экземпляра класса **DecoderBitMapProcessor** (см. описание класса **DecoderBitMapProcessor**).

Замечание: классы, используемые в кодере и в декодере, описаны в Разделе «5.2.2 Описание структуры проекта».

Заключение

Представленные в этой работе реализации алгоритмов сжатия экранного видео, основанные на технологии NVidia CUDA, подтвердили свою эффективность при тестировании. Преимуществом технологии NVidia CUDA является наличие значительно большего потенциала для реализации различных алгоритмов сжатия видео по сравнению с пиксельными шейдерами. Также 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 (см. Раздел «2.4 Перспективы развития исследования») с целью довести скорость выполнения пиксельного шейдера, реализующего операцию xor хотя бы до уровня ЦП-реализации.

Как подтвердили результаты тестирования, представленный в этой работе Гибридный алгоритм превосходит аналоги по скорости выполнения, демонстрируя при этом стабильно высокую степень сжатия дискретно-тоновых изображений.

В дальнейшем предполагается внести в Гибридный алгоритм такие изменения, которые позволят объединять в группы не только одноцветные пиксели, но и часто встречающиеся последовательности различных пикселей. Такие усовершенствования Гибридного алгоритма предоставят возможность для увеличения коэффициента сжатия дискретно-тоновых изображений.

Был реализован кодек, предназначенный для кодирования / декодирования экранного видео. В этом кодеке для сжатия ключевых кадров, а также изменившихся относительно ключевого кадра частей промежуточных кадров используется Гибридный алгоритм, затем метод Хаффмана, реализованный с помощью технологии NVidia CUDA. А для выявления изменившихся частей используется попиксельное и поблоковое сравнение, также выполняемое на видеокарте.

Таким образом, все поставленные задачи выполнены.

По материалам дипломной работы были опубликованы или приняты к печати следующие статьи:

1. «Гибридный алгоритм сжатия изображения. Сравнение алгоритмов сжатия изображений». Опубликовано в "Информационные технологии и математическое моделирование: Материалы VI Международной научно-практической конференции (9 – 10 ноября 2007 г.) ".

2. «Модификации гибридного алгоритма сжатия изображений». Принято к публикации в сборник IV Научно-практическая конференция "Обратные задачи и информационные технологии рационального природопользования".

3. «Гибридный алгоритм сжатия изображений. Практическое сравнение алгоритмов». Принято к публикации в сборник IV Научно-практическая конференция "Обратные задачи и информационные технологии рационального природопользования".

«Сжатие видео происходящего на экране пользователя с помощью видеокарты. Сравнение технологий». На момент написания данной работы, эта статья в целом одобрена рецензентом, но есть несколько мелких недочётов. После их устранения статья будет опубликована в Интернет-журнале "Вычислительные методы и программирование".

Список использованных источников

1. Colantoni, P. Fast and Accurate Color Image Processing Using 3D Graphics Cards. [Электронный ресурс] / P. Colantoni, N. Boukala, J. Da Rugna – Электрон. дан. – Режим доступа: http://colantoni.nerim.net/articles/VMV2003.pdf, свободный.

2. Rijsselbergen, D. YCoCg(-R) Color Space Conversion on the GPU. [Текст] / D. Rijsselbergen //Sixth FirW Symposium / Ghent University, 2005. – статья № 102.

3. High Quality DXT Compression using CUDA. [Электронный ресурс] / NVIDIA Corporation – Электрон. дан. – Режим доступа: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/dxtc/doc/cuda_dxtc.pdf, свободный.

4. Image Denoising. [Электронный ресурс] / NVIDIA Corporation – Электрон. дан. – Режим доступа: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/imageDenoising/doc/imageDenoising.pdf, свободный.

5. Krüger, J. A Structure for Point Scan Compression and Rendering. [Электронный ресурс] / J. Krüger, J. Schneider, R. Westermann; отв. ред. M. Pauly, M. Zwicker – Электрон. дан. – 2005. – Режим доступа: http://wwwcg.in.tum.de/Research/data/Publications/pbg05.pdf, свободный.

6. Accelerate Video Decoding With Generic GPU. [Текст] / [G. Shen и др.] // IEEE transactions on circuits and systems for video technology. – 2005. – VOL. 15, NO. 5. – с. 685–693.

7. Pieters, B. Motion Compensation and Reconstruction of H.264/AVC-coded Pictures using the GPU. [Электронный ресурс] / B. Pieters, D. Van Rijsselbergen, W. De Neve – Электрон. дан. – Режим доступа: http://symposium.elis.ugent.be/archive/symp2006/papers_poster/paper084_Bart_Pieters.pdf, свободный.

8. J. van der Laan, W. Wavelet Lifting on Graphics Hardware for Faster Video Decoding. [Электронный ресурс] / W. J. van der Laan, A. C. Jalba, J. B.T.M. Roerdink – Электрон. дан. – Режим доступа: http://www.ictonderzoek.net/3/assets/File/posters/2007_102/2007_102.pdf, свободный.

9. Dirac Specification. [Электронный ресурс] – Электрон. текстовые дан. – Режим доступа: http://dirac.sourceforge.net/DiracSpec2.2.0.pdf, свободный.

10. Strzodka, R., Garbe C. Real-Time Motion Estimation and Visualization on Graphics Cards. [Текст] / R. Strzodka, C. Garbe // Proceedings IEEE Visualization. – 2004. – с. 545–552.

11. Weise, T. A Fast 3D Scanning with Automatic Motion Compensation. [Электронный ресурс] / T. Weise, B. Leibe, L. Van Gool – Электрон. дан. – Режим доступа: http://www.vision.ee.ethz.ch/~bleibe/papers/weise-motioncompensation-cvpr07.pdf, свободный.

12. Motion compensation in digital subtraction angiography using graphics hardware. [Текст] / [Y. Deuerling-Zheng и др.] // Computerized Medical Imaging and Graphics – 2006. – с. 279–289.

13. Morvan, Y. Incorporating depth-image based view-prediction into h.264 for multiview-image coding. [Электронный ресурс] / Y. Morvan, D. Farin, P. H. N. de With – Электрон. дан. – Режим доступа: http://vca.ele.tue.nl/publications/data/Morvan2007d.pdf, свободный.

14. Motion estimation/compensation for screen capture video. [Электронный ресурс] / FreePatentsOnline.com – Электрон. текстовые дан. – Режим доступа: http://www.freepatentsonline.com/7224731.html, свободный.

15. TightVNC. [Электронный ресурс] / AT&T Laboratories Cambridge - Электрон. дан. – Режим доступа: http://www.sfr-fresh.com/windows/misc/tightvnc-1.3.9_winsrc.zip, свободный.

16. NVIDIA CUDA Compute Unified Device Architecture. Programming Guide. [Электронный ресурс] / NVIDIA Corporation – Электрон. дан. – Режим доступа: http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf, свободный.

17. The CUDA Compiler Driver. [Электронный ресурс] / NVIDIA Corporation – Электрон. дан. – Режим доступа: http://www.nvidia.com/object/io_1195170069217.html, свободный.

18. Луна, Ф. Д. Введение в программирование трехмерных игр с DirectX 9.0. [Электронный ресурс] / Электрон. текстовые дан. – Режим доступа: http://www.proklondike.com/file/Other/Frank_Luna_-_3dGamesProgrammingIntro(RUS).rar, свободный.

19. Дружинин, Д. В. Гибридный алгоритм сжатия изображения. Сравнение алгоритмов сжатия изображений. [Текст] / Д. В. Дружинин // Информационные технологии и математическое моделирование: Материалы VI Международной научно-практической конференции (9 – 10 ноября 2007 г.). – с. 70–73 – Томск: Изд-во Том. ун-та, 2007. Ч. 2.

20. Сэломон, Д. Сжатие данных, изображений и звука [Текст] / Д. Сэломон; перевод с англ. В. В. Чепыжова – М. : Техносфера, 2006. – 365 с. – (Мир программирования).

21. Shader Model 3 (Direct3D 9). / Microsoft Corporation // DirectX SDK (August 2007) Documentation [Электронный ресурс]. – Электрон. дан. – 1 электрон. опт. диск (CD-ROM).

22. Shader Model 4 Features. [Электронный ресурс] / Microsoft Corporation – Электрон. дан. – Режим доступа: http://msdn.microsoft.com/en-us/library/bb509657(VS.85).aspx, свободный.

23. Mirror driver. [Электронный ресурс] / Wikipedia - Электрон. текстовые дан. - Режим доступа: http://en.wikipedia.org/wiki/Mirror_driver, свободный.

24. Hooks. [Электронный ресурс] / Microsoft Corporation - Электрон. дан. - Режим доступа: http://msdn.microsoft.com/en-us/library/ms632589.aspx, свободный.

Приложение А. Список файлов программы

Ниже перечислены файлы исходного текста проекта EncoderMFCInterface с сохранением структуры каталогов:

Header Files

CUDA

XorCUDAWrapper.h

ClassicHaffmanCUDAWrapper.h

Encoders

ClassicHaffmanCoder.h

ClassicHaffmanEncoder.h

Coder.h

HybridEncoder.h

RLEEncoder.h

ShiftEncoder.h

SimpleXorEncoder.h

HashTables

EncoderHashTable.h

HashTable.h

System

Multithreaded

EncodeThreadProcessor.h

ThreadProcessor.h

Converter.h

DataStructures.h

Logger.h

Tester.h

Utility.h

User Inteface

Controller.h

EncoderMFCInterfaceDlg.h

DirectXProcessor.h

EncoderBitMapProcessor.h

EncoderMFCInterface.h

Resource.h

stdafx.h

Source Files

CUDA

ClassicHaffmanCUDAWrapper.cpp

HaffmanDeviceCode.cu

HaffmanDeviceWrapper.cu

XorCUDAWrapper.cpp

XorDeviceCode.cu

XorDeviceWrapper.cu

Encoders

ClassicHaffmanCoder.cpp

ClassicHaffmanEncoder.cpp

Coder.cpp

HybridEncoder.cpp

RLEEncoder.cpp

ShiftEncoder.cpp

SimpleXorEncoder.cpp

HashTables

EncoderHashTable.cpp

HashTable.cpp

System

Multithreaded

EncodeThreadProcessor.cpp

ThreadProcessor.cpp

Converter.cpp

Logger.cpp

Tester.cpp

Utility.cpp

User Interface

Controller.cpp

EncoderMFCInterfaceDlg.cpp

DirectXProcessor.cpp

EncoderBitMapProcessor.cpp

EncoderMFCInterface.cpp

stdafx.cpp

Shaders

LittlePictureXor16.psh

LittlePictureXor32.psh

LittlePictureXor8.psh

LittlePictureXorRGB.psh

XorRGB.psh

Ниже перечислены файлы исходного текста проекта DecoderWinAPIInterface с сохранением структуры каталогов:

Header Files

Decoders

ClassicHaffmanCoder.h

ClassicHaffmanDecoder.h

Coder.h

HybridDecoder.h

RLEDecoder.h

ShiftDecoder.h

SimpleXorDecoder.h

HashTables

DecoderHashTable.h

HashTable.h

System

Multithreaded

DecodeThreadProcessor.h

ThreadProcessor.h

Converter.h

DataStructures.h

Logger.h

Tester.h

Utility.h

DecoderBitMapProcessor.h

DecoderWinAPIInterface.h

Resource.h

stdafx.h

Source Files

Decoders

ClassicHaffmanCoder.cpp

ClassicHaffmanDecoder.cpp

Coder.cpp

HybridDecoder.cpp

RLEDecoder.cpp

ShiftDecoder.cpp

SimpleXorDecoder.cpp

HashTables

DecoderHashTable.cpp

HashTable.cpp

System

Multithreaded

DecodeThreadProcessor.cpp

ThreadProcessor.cpp

Converter.cpp

Logger.cpp

Tester.cpp

Utility.cpp

DecoderBitMapProcessor.cpp

DecoderWinAPIInterface.cpp

stdafx.cpp