© 2012 г. И.В. ПЕРМИНОВ
(Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики)

ДИНАМИЧЕСКАЯ КОМПРЕССИЯ ВИЗУАЛЬНЫХ ДАННЫХ В ГРАФИЧЕСКИХ ПРОЦЕССОРАХ

Рассмотрены существующие подходы к сжатию текстур для формата DXT1, приведено описание нового алгоритма сжатия “LSpiro”. Предложены методы адаптации алгоритма “LSpiro” для выполнения на графическом процессоре. Приведены результаты тестирования производительности полученного кодека.

ONLINE COMPRESSION OF VISUAL DATA ON GPU / I.V. Perminov (St. Petersburg national research university of information technologies, mechanics and optics, St. Petersburg 197101, Russia, E-mail: i.am.perminov@gmail.com). The article considers existing approaches to DXT1 texture compression and describes new algorithm named “LSpiro”. Modifications of “LSpiro” algorithm for executing on GPU are examined. The results of performance testing of modified algorithm are given.

1. Введение

Текстуры повсеместно применяются в компьютерной трехмерной графике и в простейшем случае представляют собой двухмерное изображение, накладываемое на трехмерную поверхность с целью улучшения визуальной детализации без усложнения геометрии.

В современных системах визуализации все чаще используются динамически генерируемые данные (карты окружений для имитации отражения, и т.д.), что делает актуальной задачу быстрого сжатия текстур в темпе генерации кадров ресурсами графического процессора. Однако, в настоящее время такие кодеки, нацеленные на быстрое сжатие, отсутствуют. Так же текстурное сжатие в реальном времени используется при передаче по сети видеопотоков высокого разрешения, например, в системе UltraGrid[1]. Но для этого используются многопроцессорные системы. Применение кодеков, исполняющихся на графическом процессоре, сможет повысить производительность таких систем и заменить дорогостоящие многопроцессорные системы однопроцессорными, за счет переноса нагрузки с центрального процессора на видеокарту.

Однако для эффективного исполнения алгоритма на графическом процессоре необходимо обеспечить высокий уровень внутреннего параллелизма. При сжатии текстур это условие выполняется естественным образом, так как каждый блок изображения обрабатывается независимо. Для достижения высокой скорости сжатия, алгоритм обработки отдельного блока также должен быть модифицирован, в том числе для параллельной обработки.

2. Существующие кодеки

В настоящее время доминирующим форматом текстурного сжатия является формат DXT в различных версиях. Это блочный кодек, оперирующий блоками размером 4x4 пикселя и обеспечивающий фиксированный уровень сжатия. Важно отменить, что формат DXT использует сжатие с потерями: исходное изображение разбивается на блоки 4x4. Для каждого блока сохраняется 2 ключевых цвета c0 и c1 в формате RGB565 и таблица двухбитовых индексов, определяющая, в какой пропорции смешивать ключевые цвета при восстановлении блока. При распаковке из ключевых цветов формируется локальная палитра, состоящая из четырех цветов, а индекс указывает какой из этих цветов необходимо выбрать. Упрощенно для целей статьи можно считать, что цвета L0, L1, L2, L3 в локальной палитре вычисляются так:

(1) $$L0 = C0 \qquad L1 = \frac{2*C0 + C1}{3} \qquad L2 = \frac{C0 + 2*C1}{3} \qquad L3 = C1$$

Таким образом, задача сжатия исходного блока сводится к нахождению двух базовых цветов, позволяющих получить после распаковки блок, максимально близкий к оригиналу. Качество восстановленного изображения сильно зависит от правильного выбора этих двух цветов. И хотя распаковка такого блока довольно проста, качественное сжатие чрезвычайно ресурсоемко. Сами алгоритмы сжатия, как правило, основаны на методах перебора или используют большое количество условных переходов. Архитектурные особенности современных видеокарт, рассчитанных на массовые параллельные вычисления, не позволяют эффективно исполнять подобные алгоритмы на графическом процессоре.

Среди существующих DXT кодеков (Таблица 1) на онлайн сжатие нацелен только FastDXT [2]. Он используется в системе UltraGrid, но не поддерживает сжатие на графическом процессоре. Такой подход не применим для сжатия динамически генерируемых данных, так как он требует пересылки данных из памяти GPU в общую память и обратно.

Таблица 1. Характеристики существующих DXT-кодеков

Название	Открытый исходный код	Online-сжатие	GPU-реализация
AMD Compressonator [2]	-	-	-
NVidia NVtt [3]	+	-	+
Crunch [4]	+	-	-
FastDXT [5]	+	+	-
Squish [6]	+	-	+

Одним из наиболее очевидных методов генерации ключевых цветов является выбор их из точек внутри сжимаемого блока. Следует отметить, что любая точка исходного блока с цветом (r, g, b) может рассматриваться как точка в трехмерном пространстве с координатами (r, g, b), а также как вектор $\vec{c}$ (r,g,b). Два ключевых цвета задают прямую, а цвета локальной палитры будут располагаться на этой прямой. Сами цвета исходного блока, как правило, образуют в этом пространстве некоторое облако. Один из методов сжатия предполагает выбор в качестве ключевых цветов двух наиболее удаленных друг

от друга точек. При этом потребуется сравнить между собой расстояния между всеми парами точек. В более простом варианте в качестве ключевых выбираются точки с максимальной и минимальной яркостью. В простейшем случае яркость lum цвета (r, g, b) может вычисляться по следующей формуле:

(2) $$lum = r + 2 * g + b$$

Несмотря на свою простоту, оба эти метода уступают по скорости методам быстрого сжатия, а в большинстве случаев и по качеству результирующего изображения.

2.1. Методы быстрого сжатия

Большинство быстрых кодеков (включая FastDXT и ExtremeDXT [7]) основано на методе сжатия, описанном в статье «Real-Time DXT Compression» [8]. В данном подходе в качестве ключевых точек выбираются две вершины прямоугольного параллелепипеда, ограничивающего в пространстве RGB все точки, содержащиеся в исходном блоке. Для этого достаточно отдельно в каждом цветовом канале найти минимальное и максимальное значение. Полученные значения дополнительно сдвигают друг к другу на величину равную 1/16 разности между максимумом и минимумом. В большинстве случаев это позволяет уменьшить ошибку. Для достижения максимального быстродействия, кодек RealtimeDXT был написан на ассемблере с использованием набора команд SSE2.

И хотя подобный алгоритм относительно прост для реализации на GPU, в определенных ситуациях он приводит к очень заметным артефактам сжатия (Рис. 1. **Пример сжатия с использованием кодека FastDXT. Слева направо: исходное изображение, ошибка, сжатое изображение**Рис. 1).

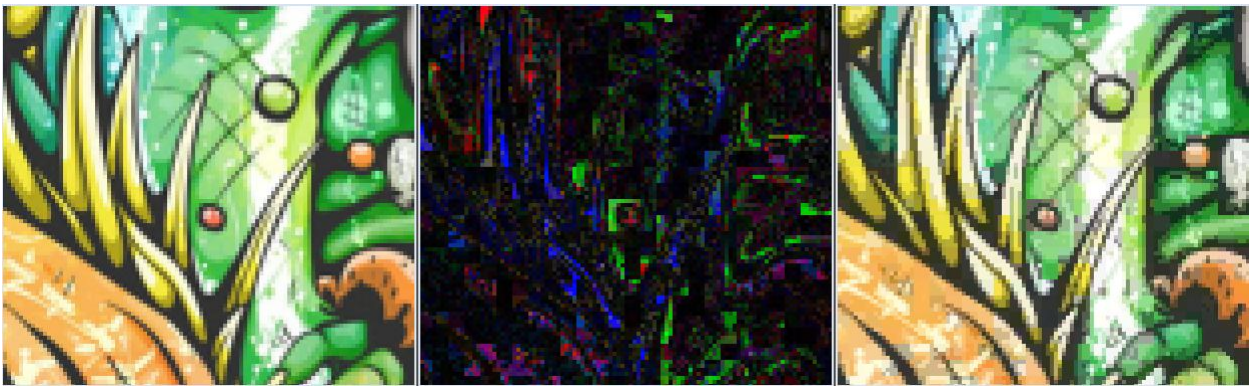

Рис. 1. Пример сжатия с использованием кодека FastDXT. Слева направо: исходное изображение, ошибка, сжатое изображение

2.2. Метод “Cluster Fit”

Метод “Cluster Fit” [9] предназначен для высококачественного сжатия, но при этом его GPU реализация обладает хорошим быстродействием для подобного уровня качества. Данный метод используется в кодеках NVidia NVtt и Squish [10]. Он существенно отличается от остальных кодеков, где поиск нацелен на получение ключевых цветов, а индексы точек в сжатом блоке вычисляются уже на основе этих цветов. Основная идея состоит в том, что если известны индексы сжатого блока, то нахождение ключевых цве-

тов, обеспечивающих минимальную ошибку, является относительно простой оптимизационной задачей. Для поиска индексов используется перебор. С учетом формата DXT блока, всего существует 4^16 возможных комбинаций индексов, и полный перебор за разумное время не возможен.

Для уменьшения количества комбинаций делается допущение, что в большинстве случаев индексы точек будут отличаться от оптимальных индексов очень незначительно, если в качестве ключевых цветов используются цвета достаточно близкие к оптимальным. Делается предположения, что в качестве прямой, содержащей такие ключевые цвета может использоваться главная прямая, получаемая при помощи метода главных элементов. Можно пронумеровать все точки исходного блока согласно их скалярному произведению на эту прямую (т.е. упорядочить их проекции). Тогда соседние точки будут иметь одинаковый, или отличающийся на 1 индекс, так как все цвета локальной палитры располагаются на этой же прямой, и индекс точки определяется самым близким к ней цветом из палитры. Таким образом, все возможные варианты кластеризации точек с одинаковым индексом будут выглядеть как:

(3) [0,i), [i,j), [j,k), [k,16) для $i, j, k \in [0,16)$

Это позволяет ограничить количество комбинаций индексов, которые необходимо проверить, до 975.

Данный метод обладает хорошим качеством (Рис. 2), но недостаточно быстр для сжатия в реальном времени.

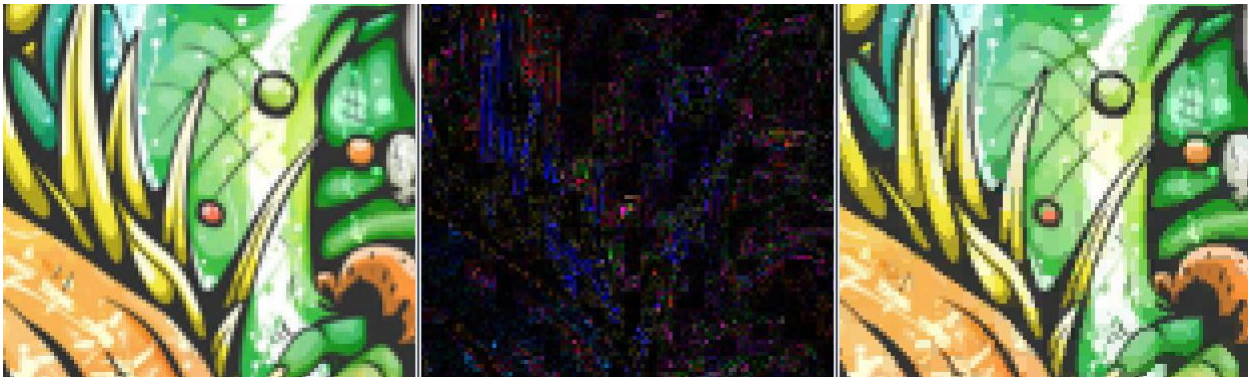

Рис. 2. Пример сжатия с использованием кодека Squish. Слева направо: исходное изображение, ошибка, сжатое изображение

2.3. Метод "LSpiro"

В рамках проекта L.Spiro Engine был предложен новый подход к сжатию текстур в формате DXT [11][12]. Открытой реализации кодека на данный момент не существует, но по прогнозам автора алгоритма он должен обладать высоким качеством и скоростью компрессии. Оригинальный вариант алгоритма предполагает использование 3 шагов для получения ключевых цветов.

На первом шаге для получения начальных ключевых цветов применяется линейная регрессия. При этом каждый цветовой канал обрабатывается независимо. Значения в красном, зеленом и синем каналах сортируются в порядке возрастания и дубликаты удаляются. Для каждого цветового канала применяется линейная регрессия: в качестве координаты Y используется значение цвета, а в качестве координаты X – индекс в отсортированном массиве. Т.е. получается прямая, проходящая максимально близко к за-

данным точкам (Рис. 3). Координаты Y точек c0 и c1 используются в качестве начальных ключевых цветов. Однако точки, которые имеют очень близкие по значению цвета, будут "перекашивать" направление прямой, сильно снижая точность для "редких" цветов. Этот эффект корректируется путем предварительного приведения значения цветов от 32-битного к 16-битному представлению. Таким образом, близкие по значению цвета будут отброшены на этапе сортировки.

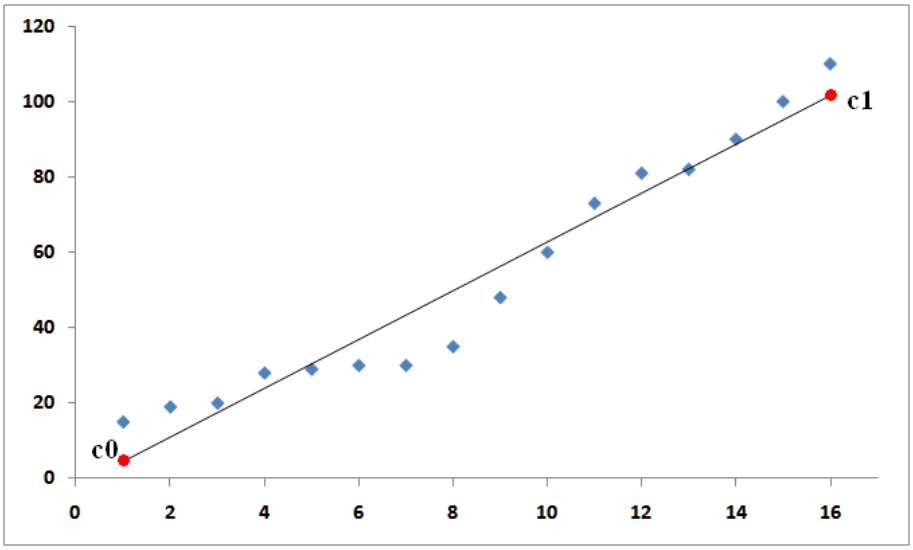

Рис. 3. Использование линейной регрессии для нахождения ключевых цветов

На втором шаге линейная регрессия используется для уточнения результата. По ключевым цветам воспроизводится сжатый блок, и линейная регрессия применяется к разности между полученным и оригинальным блоком. Второй проход линейной регрессии рассчитан на тот факт, что после сжатия внутри блока доступно только 4 различных цвета. Первый шаг дает оптимальный результат для варианта, когда в сжатом блоке можно использовать любой цвет, находящийся на прямой, задаваемой двумя базовыми цветами. Используя эти цвета в качестве отправной точки, можно скорректировать направление прямой так, чтобы разница между доступными и исходными цветами была минимальна. Данный процесс является итеративным.

На последнем шаге производится перебор всех комбинаций цветов, находящихся в небольшой окрестности полученных результатов. Цвета, дающие минимальную ошибку выбираются в качестве ключевых.

Отличительной особенностью алгоритма, является возможность его использования, как для быстрого, так и для качественного сжатия, так как шаги 2 и 3 являются опциональными. С точки зрения быстрого сжатия, дополнительным преимуществом является возможность варьирования соотношения скорость/качество за счет изменения количества итераций на шаге 2 и размера окрестности на шаге 3. При заданных ограничениях на скорость сжатия, это позволит улучшить качество сжатия при увеличении производительности аппаратной платформы.

При использовании всех трех шагов алгоритма (Рис. 4), качество сжатия близко к качеству сжатия методом Cluster Fit.

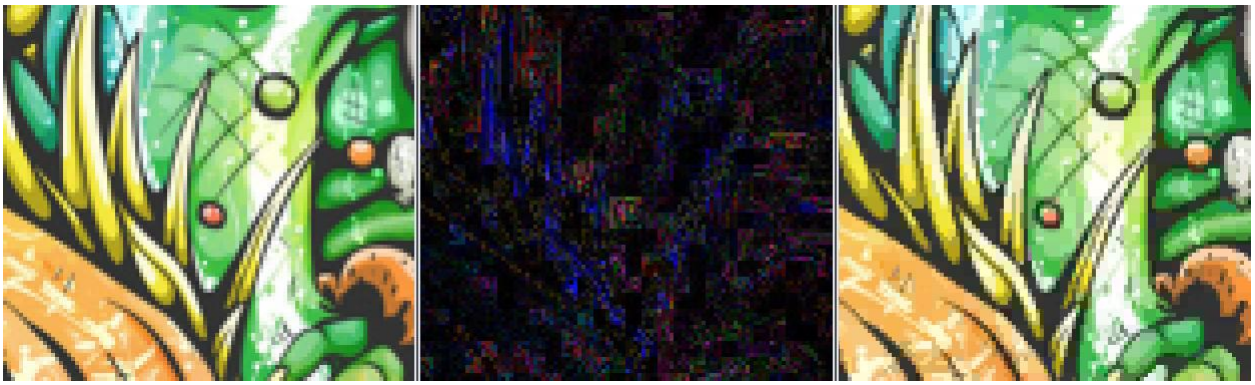

Рис. 4. Пример сжатия с использованием алгоритма LSpiro. Слева направо: исходное изображение, ошибка, сжатое изображение

Исходный алгоритм рассчитан на исполнение на центральном процессоре и не предполагает быстрое сжатие, но сама базовая идея может быть использована для построения быстрого параллельного кодека, работающего на графическом процессоре.

3. Модификация алгоритма LSpiro для выполнения на графическом процессоре

Для эффективной загрузки вычислительных блоков графического процессора, необходимо обеспечить высокий уровень внутреннего параллелизма исполняемого алгоритма. При сжатии текстур это условие выполняется естественным образом, так как каждый блок изображения обрабатывается независимо. Поэтому на каждый блок можно выделить отдельную группу потоков. Однако, для достижения высокой скорости сжатия, алгоритм обработки отдельного блока также должен быть модифицирован.

При сжатии текстур в темпе генерации кадров нет необходимости в получении максимально возможного качества, поэтому третий шаг алгоритма может быть пропущен. Кроме этого, количество итераций, выполняемых на втором шаге, также может быть сокращенно.

Для оценки эффективности предложенного метода был написан кодек, выполняющий сжатие на графическом процессоре. В качестве языка и платформы выполнения был выбран OpenCL, так как только эта технология обеспечивает кроссплатформенность как на аппаратном, так и на программном уровне.

3.1. Алгоритм сортировки

В силу архитектурных особенностей графических процессоров, для повышения быстродействия следует минимизировать количество циклов и ветвлений в алгоритме. В этом отношении проблемным является этап подготовки данных на первом шаге алгоритма, где для каждого блока требуется выделить и отсортировать уникальные значения цветов и получить их общее количество. Ввиду малого количества сортируемых элементов, в исходном варианте используется сортировка методом вставки. Значения последовательно добавляются в отдельный массив. При уникальности нового значения, проверяемом при каждом добавлении, определяется его положение в массиве, а элементы, располагающиеся на данном и последующих позициях, сдвигаются.

При реализации на графическом процессоре такой подход имеет малую эффективность по следующим причинам:

– внутри одной группы потоков могут работать не более 4-х потоков, независимо обрабатывающих отдельные цветовые каналы;
– из-за ветвлений и разного количества цветов в каждом канале, потоки не могут выполняться одновременно;
– количество инструкций, выполняемых каждым потоком, велико.

Для решения этой проблемы был предложен вариант сортировки, основанный на прямом вычислении ранга каждого элемента. Здесь и далее под рангом понимается количество элементов массива, строго меньше данного. Упрощенный вариант параллельной сортировки с вычислением ранга для случая неповторяющихся элементов выглядит следующим образом:

– каждый элемент обрабатывается отдельным потоком
– отдельный поток вычисляет ранг соответствующего элемента, проходя по всему массиву
– после вычисления ранга, поток копирует данный элемент в выходной массив, используя ранг в качестве индекса

Выделение и сортировка уникальных значений осуществляется в 2 прохода (Рис. 5). Сначала массив «B» инициализируется значениями, заведомо большими, чем элементы исходного массива. Это возможно, за счет того, что диапазон значений исходных элементов строго фиксирован.

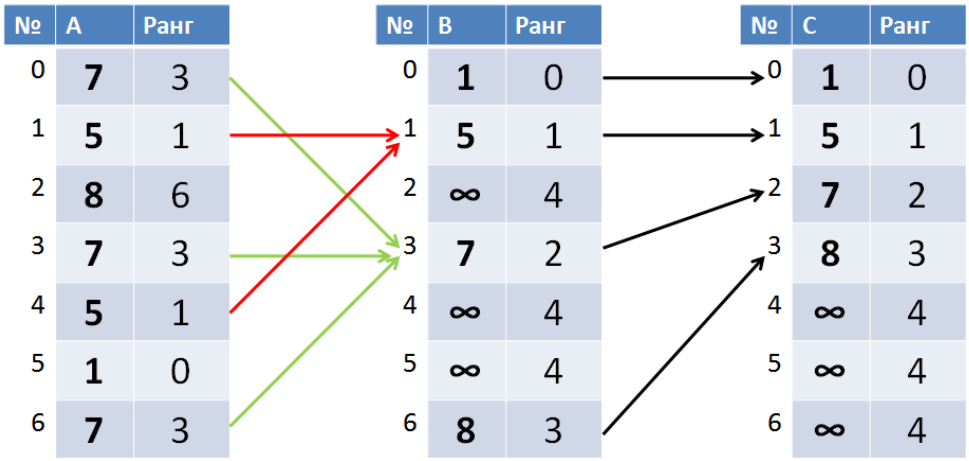

Рис. 5. Пример сортировки методом с вычислением рангов

На первом проходе вычисляются ранги для исходного массива «A», и элементы копируются по соответствующим индексам в массив «B». При этом повторяющиеся элементы будут иметь одинаковый ранг и, следовательно, будут помещены в одну и ту же ячейку массива «B». Копирование повторяющихся элементов на Рисунок 3 обозначено цветными стрелками. На втором проходе вычисляются ранги для массива «B», и элементы копируются в выходной массив «C» в соответствии со своими рангами. При этом все уникальные значения будут располагаться в начале массива «C» в порядке возрастания, а количество уникальных элементов будет равно рангу последнего элемента массива «C».

Такой подход позволяет осуществить сортировку уникальных элементов без ветвлений, поэтому все потоки могут выполняться одновременно. Таким образом, для каждого исходного блока изображения можно использовать 16 потоков для сортировки

значений в отдельном цветовом канале. При этом отдельный поток выполнит только 48 сравнений, 48 инкрементов и 2 записи в память.

Результаты тестов (Таблица 2) на текстурах размером 512x512 пикселей показывают, что предложенный алгоритм сортировки позволяет сократить время выполнения первого шага в среднем в 3 раза. Тестирование проводилось с использованием видеокарты GeForce GTX 560 (7 мультипроцессоров, 336 потоковых процессоров).

Таблица 2. Время выполнения шага №1 для различных методов сортировки

Метод сортировки	Текстура №1	Текстура №2	Текстура №3
Сортировка методом вставки	6,4 мс	5,2 мс	8,6 мс
Сортировка с подсчетом ранга	1,9 мс	1,9 мс	2,1 мс

3.2. Изменение размеров группы потоков

С точки зрения написания кода, наиболее удобно использовать одну группу потоков для сжатия одного DXT-блока. С учетом размера блока, количество потоков в одной группе будет составлять 16. Однако в этом случае ресурсы графического процессора будут использоваться крайне не эффективно, так как показатель Occupancy (отношение количества выполняющихся на мультипроцессоре потоков к максимально возможному) будет ограничен максимальным количеством групп потоков, способных выполняться на мультипроцессоре.

Для увеличения производительности можно сжимать в одной группе потоков сразу несколько DXT-блоков. Это несколько усложняет код, но позволяет увеличить производительность в несколько раз. В Таблица 3 приведены результаты тестирования для текстур размером 1920x1200.

3.3. Размещение данных в локальной памяти

Для увеличения производительности, каждая группа потоков загружает соответствующие ей блоки исходного изображения в локальную память. В графических процессорах NVidia локальная память состоит из 32 банков памяти, которые могут одновременно выполнять запросы чтения/записи [13]. Однако с учетом количества точек в блоке и используемых типов данных, размещение данных в локальной памяти в таком же порядке, как в глобальной, приведет к конфликту банков памяти.

На Рис. 6 приведено распределение данных по банкам памяти для случая сжатия двух блоков в одной группе потоков (цифры в ячейках обозначают порядковый номер пикселя внутри блока, цвет ячейки соответствует цветовому каналу: красный, зеленый, синий, альфа-канал). В варианте “а” данные в локальной памяти расположены в естественном порядке, как в глобальной памяти. Видно, что пиксели номер 0 и 8 обоих блоков памяти расположены в банке 0. Следовательно, потоки 0, 8, 16 и 24 будут создавать конфликт в данном банке при одновременном обращении к соответствующим им пикселям. Вариант размещения данных, не приводящий к конфликтам, представлен на Рис. 6 б, но вычисление адреса необходимого блока в таком случае оказывается слишком сложным. Варианты “в” (отбрасывание альфа-канала) и “г”(дополнение пустым каналом) так же не вызывают конфликтов, но в отличии от “б” усложняют адресную арифметику незначительно. Недостатком варианта “в” является невозможность правильного сжатия изображений с альфа-каналом, а варианта “г” – увеличение требований к объему

локальной памяти. Для предотвращения конфликтов банков памяти был выбран вариант размещения “в”.

Таблица 3. Влияние размещения данных и количества сжимаемых блоков на быстродействие шага №1

	1x	2x	4x	6x	8x	10x	12x	16x
Кол-во потоков в группе	16	32	64	96	128	160	192	256
Occupancy	0,17	0,17	0,33	0,5	0,67	0,73	0,73	0,83
Время выполнения шага №1, мс	13,36	10,11	6,074	4,697	4,535	4,197	4,335	4,211
Время выполнения шага №1 с предотвращением конфликтов банков памяти, мс	12,7	6,549	4,007	3,002	3,001	2,589	2,84	2,699

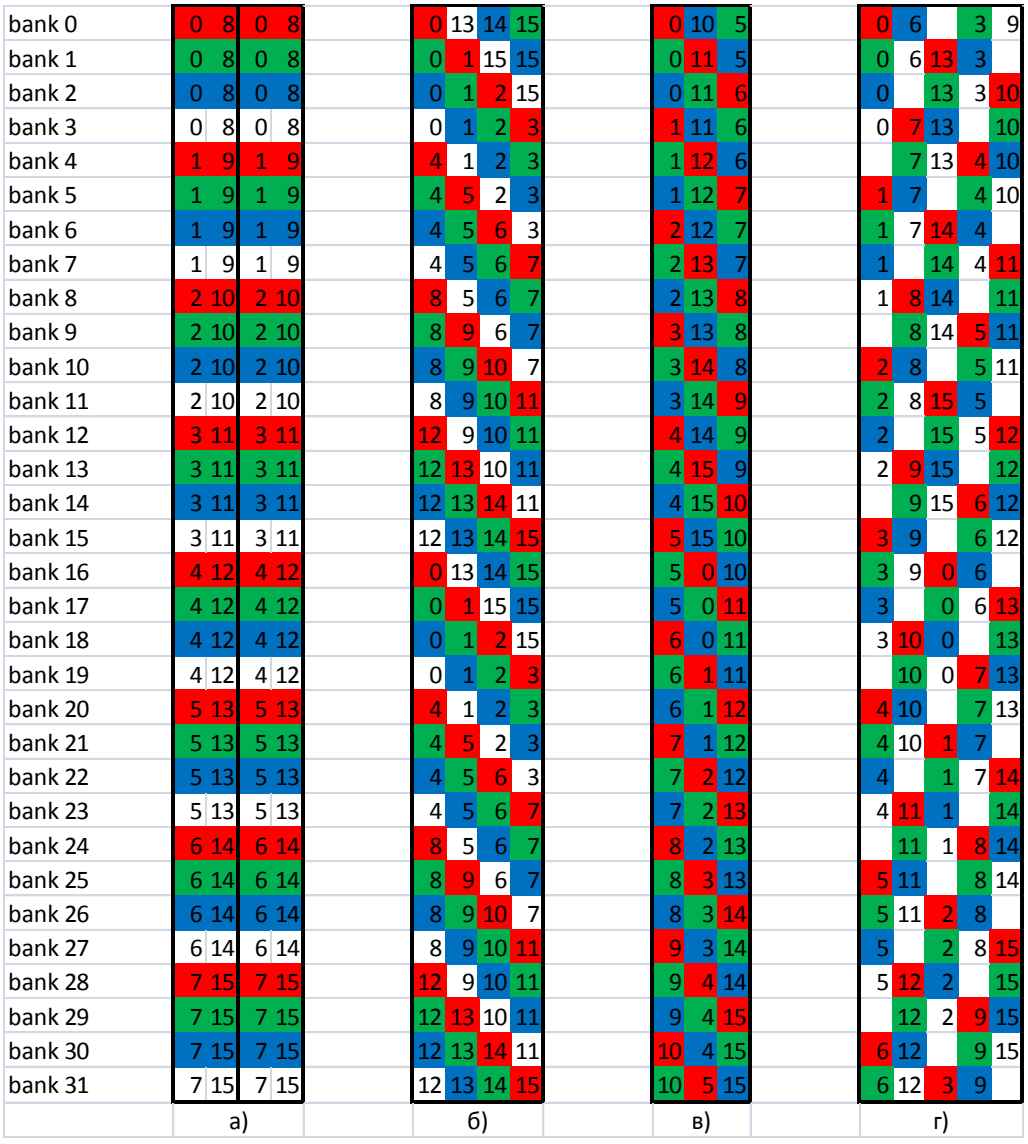

Рис. 6. Варианты размещения данных в локальной памяти

3.4. Дополнительные оптимизации производительности

Кроме вычисления ключевых цветов, для получения сжатого DXT-блока, необходимо вычислить таблицу индексов и сохранить ключевые цвета в формате RGB565. Для выполнения этих функций используется отдельное OpenCL ядро. Выполнение шага алгоритма №1 и упаковки DXT-блока в рамках одного ядра позволяет ликвидировать промежуточную выгрузку данных в глобальную память после завершения первого ядра и повторную их загрузку в локальную память при выполнении второго ядра.

Так же подсчет рангов в цикле сортировки одновременно для всех цветовых каналов (в отличие от последовательной сортировки отдельных каналов) позволяет упростить адресную арифметику, так как каждый поток будет обращаться к последовательным адресам.

Некоторого выигрыша производительности можно добиться за счет выставления опции компиляции cl-fast-relaxed-math, позволяющей использовать более быструю арифметику для чисел с плавающей запятой, но не соответствующую стандарту IEEE 754.

Начиная с количества одновременно сжимаемых блоков равного 8, Occupancy для полученного ядра уже ограничивается количеством доступных в мультипроцессоре регистров. Принудительное уменьшение количества используемых одним потоком регистров уменьшает быстродействие отдельного потока, но позволяет увеличить Occupancy.

Результаты тестирования при использовании описанных оптимизаций сведены в Таблица 4. Тестирование проводилось с использованием текстур размером 1920x1200 и количестве потоков в группе равном 160 (10 DXT-блоков на одну группу потоков).

Таблица 4. Влияние дополнительных оптимизаций на производительность сжатия

Описание	Время сжатия
Раздельные ядра для шага №1 и упаковки DXT-блока	5,826 мс
Общее ядро для шага №1 и упаковки DXT-блока	5,517 мс
Реорганизация цикла сортировки	5,185 мс
Использование «не строгой» арифметики	5,12 мс
Ограничение количества регистров	4,813 мс

Полученная скорость сжатия соответствует 470 MP/s или 200 кадрам в секунду для кадра размером 1920x1200 (120 кадров в секунду для 2560x1600). Полученные цифры не включают в себя время, затрачиваемое на копирование данных из оперативной памяти в память графического процессора. На тестируемой системе это время составляет 5 мс для кадра 1920x1200. Однако при сжатии динамически генерируемых визуальных данных необходимость пересылки данных в память графического процессора отсутствует, а в случае применения в системе UltraGrid эту задержку можно скрыть, за одновременного сжатия предыдущего кадра и копирования текущего.

4. Заключение

Полученная скорость с использованием только шага №1 позволяет производить сжатие в темпе более 200 кадров в секунду для FullHD-кадров. И хотя это на порядок ниже лучшей реализации метода, используемого в FastDXT, подобной производительности достаточно для многих применений.

Дальнейшая оптимизация алгоритма и реализация шага №2 может позволить производить сжатие с достаточной для динамического сжатия производительностью и значительно лучшим качеством по сравнению с FastDXT. Дополнительным преимуществом в таком случае будет возможность варьировать количество итераций шага 2. Это позволит изменять соотношение качества и скорости сжатия в зависимости от быстродействия аппаратного обеспечения.

Реализация же всех трех шагов интересна с точки зрения высококачественного сжатия. Теоретически оно может превосходить качество метода ClusterFit при сопоставимых временных затратах.

СПИСОК ЛИТЕРАТУРЫ

1. *Wesley-Smith I., Liška M., Holub P.* Implementation of DXT Compression for UltraGrid // CESNET Technical Report x/2008
2. AMD Developer Central. The compessonator. http://developer.amd.com/Resources/archive/ArchivedTools/gpu/compressonator/Pages/default.aspx
3. NVidia Texture tools. http://developer.nvidia.com/content/gpu-accelerated-texture-compression
4. Crunch Compression Library. http://code.google.com/p/crunch/
5. FastDXT Compression Library. http://www.evl.uic.edu/cavern/fastdxt/
6. Squish Compression Library. http://code.google.com/p/libsquish/
7. *Uličiansky P.* Extreme DXT Compression // Cauldron, Ltd., 2010
8. *J.M.P. van Waveren.* Real-Time DXT Compression // Id Software Inc., 2006
9. *Brown S.* DXT Compression Techniques // http://www.sjbrown.co.uk/2006/01/19/dxt-compression-techniques/, 2006
10. *Castano I.*. High Quality DXT Compression using OpenCL for CUDA // NVIDIA OpenCL SDK Code Samples 2009
11. New DXT Compressor (Algorithm Explained). 2011. http://lspiroengine.com/?p=260
12. DXT Compression Revisited. 2012 http://lspiroengine.com/?p=312
13. OpenCL Programming Guide for the CUDA Architecture, v4.2 // NVidia Corporation, 2012