Особенности использования регистровой памяти GPU в OpenCL на примере текстурного компрессора

Илья Перминов
Факультет компьютерных технологий и управления
Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики, Санкт-Петербург, Россия
studentikispam@gmail.com

Аннотация

В статье представлена модификация алгоритма текстурной компрессии LSDxt [2] для работы на графическом процессоре. Кратко описана реализация алгоритма с использованием локальной разделяемой памяти. Рассмотрены проблемы, возникающие при реализации алгоритмов с использованием только регистровой памяти. Приведены результаты тестирования производительности.

Ключевые слова: *OpenCL, GPGPU, сжатие текстур, производительность, регистровая память*

1. ВВЕДЕНИЕ

Алгоритмы и задачи, обладающие высокой степенью внутреннего параллелизма, при выполнении на графическом процессоре могут иметь существенно большее быстродействие по сравнению с традиционными процессорами. Но для получения максимальной производительности необходимо существенно модифицировать алгоритм для эффективного использования ресурсов GPU. У разных моделей GPU архитектура может сильно различаться, но основные рекомендации состоят в следующем:

- Разбиение тредов на группы, кратные размеру аппаратно выполняемых групп (warp для GPU компании NVidia или wavefront для AMD)
- Работа с глобальной памятью с учетом её блочной организации. Если все треды одной аппаратной группы обращаются к одному и тому же блоку памяти, то этот запрос может быть удовлетворён за одно обращение в память.
- Уменьшение количества конфликтов банков памяти при использовании локальной разделяемой памяти. Если все адреса, по которым обращаются в память треды одной аппаратной группы, относятся к разным банкам памяти, то такой запрос может быть выполнен за одно обращение в память.
- Аккуратное использование условных переходов. Желательно, чтобы все треды одной аппаратной группы выбирали один и тот же переход. Иначе обе ветви условия будут выполняться последовательно.
- Поддержание высокого значения показателя загруженности тредов (Occupancy), чтобы скрыть задержки при обращении к памяти. Если ресурсы мультипроцессора позволяют разместить на нём несколько групп тредов, то во время ожидания операций с памятью для одной группы, мультипроцессор может исполнять инструкции другой готовой группы.

Однако наибольшим быстродействием доступа обладают именно регистры и для некоторых алгоритмов достаточно объема регистровой памяти. Тем не менее, такой подход не всегда может увеличить производительность.

2. СЖАТИЕ ТЕКСТУР

Сжатие текстур используется для экономии пропускной способности памяти, так как текстуры передаются в графический процессор в сжатом виде и распаковываются аппаратно между блоками кэш-памяти L1 и L2. Так же это позволяет уменьшить объем памяти, занимаемый текстурами. Поскольку в 3D-графике крайне важна эффективность произвольного доступа к отдельным текселям, все современные форматы сжатия текстур обладают фиксированным уровнем сжатия и, следовательно, осуществляют сжатие с потерями. Как правило, это блочные кодеки.

Наиболее распространенным форматом является DXT/S3TC и его вариации BC1-BC5. При использовании DXT1 (Рис.1) для каждого блока исходного изображения размером 4x4 сохраняется 2 ключевых цвета в формате RGB565 и таблица индексов, определяющая в какой пропорции смешивать ключевые цвета при восстановлении блока. Кроме этого, взаимным расположением ключевых цветов неявно кодируется еще один бит, используемый при распаковке.

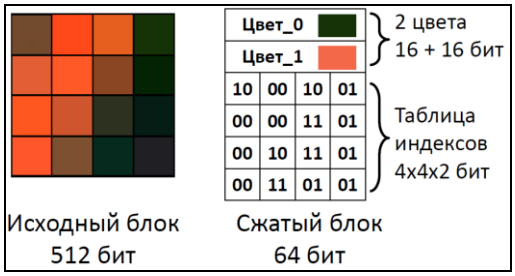

Рис 1: Формат блока DXT1/BC1.

Распаковка такого блока довольно проста, но качество восстановленного изображения сильно зависит от выбора ключевых цветов. Компрессия с высоким качеством очень ресурсоёмка и, как правило, для неё используются методы последовательного перебора.

3. АЛГОРИТМ СЖАТИЯ LSDXT

Алгоритм сжатия LSDxt позволяет варьировать скорость и качество сжатия и состоит из трёх основных шагов:

1. Получение ключевых цветов. Для этого отдельно в каждом цветовом канале берутся уникальные значения цвета, и к ним применяется линейная регрессия.
2. Улучшение результата. Воспроизводится сжатый блок, и линейная регрессия применяется к ошибкам отдельно в каждом цветовом канале, что позволяет уточнить результат.
3. Перебор значений в малой окрестности полученных результатов с целью минимизации ошибки.

Шаг 2 является итеративным. Чем больше итераций проводится на шаге 2 и более широкая окрестность выбирается на шаге 3, тем выше качество итогового изображения.

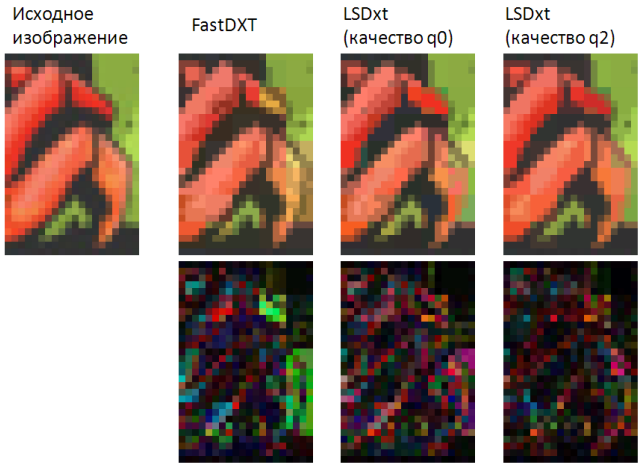

Рис 2: Пример сжатия различными кодеками (верхний ряд) и ошибки сжатия (нижний ряд).

4. АЛГОРИТМ С ИСПОЛЬЗОВАНИЕМ ЛОКАЛЬНОЙ ПАМЯТИ

Рабочая группа (workgroup) была выбрана размером 64 так как она кратна и размеру warp (32) и размеру wavefront (64). Каждый блок 4x4 исходного изображения может обрабатываться независимо, и можно было бы использовать отдельный тред (workitem) для каждого блока. Но это потребовало бы выделения 32 KiB локальной памяти на рабочую группу для хранения значений цветов и ошибок, что означало бы очень низкую Occupancy и простои вычислительных ресурсов на каждой операции с памятью. Поэтому для сжатия используется один тред на каждый тексель. То есть одна рабочая группа сжимает 4 блока, что уменьшает требования к локальной памяти в 16 раз. Такой подход позволяет вычислять ошибку каждого текселя параллельно и использовать параллельную сортировку на шаге 1.

Так как шаги алгоритма независимы, используются отдельные ядра (kernels) для каждого шага. Это делает необходимым загрузку данных из глобальной памяти в локальную 3 раза (на каждом шаге), но затрачиваемое на это время более чем на два порядка меньше времени работы самого алгоритма.

Размещение данных в локальной памяти в естественном порядке (Рис.3 a) приводит к конфликтам банков памяти при одновременном доступе нескольких тредов. Ликвидировать конфликты можно используя варианты расположения b или c.

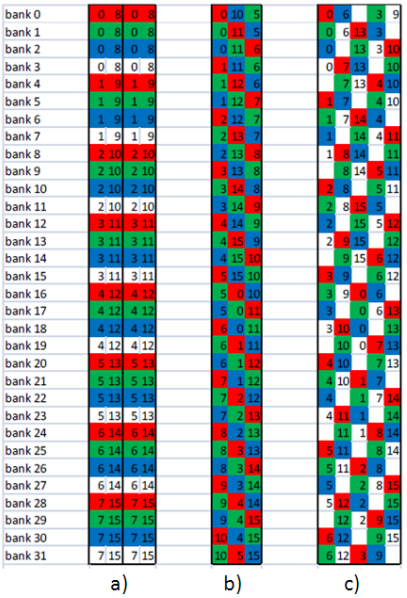

Рис 3: Расположение данных в локальной памяти с конфликтом банков памяти (a) и без (b, c)

При использовании прочих оптимизаций, такой вариант компрессора способен обеспечить 30-кратный прирост производительности по сравнению с CPU вариантом (Табл. 3).

5. ИСПОЛЬЗОВАНИЕ РЕГИСТРОВ

Дальнейшее увеличение быстродействия возможно при использовании только регистровой памяти, потому что она обладает наибольшей пропускной способностью и наименьшей задержкой (Табл. 1).

	Global	Local	Reg
Peak Bandwidth	264 Gb/s	3789 Gb/s	22733 Gb/s
Peak Bandwidth /Stream Core	~0.14 bytes/cycle	8 bytes/cycle	12 bytes/cycle
Size	3Gb	64Kb / CU	256 Kb /CU

Таблица 1: Характеристики различных типов памяти AMD Radeon HD 7970 [3].

Но использование только регистровой памяти затрудняется по следующим причинам:

- Невозможность обмена данными между тредами внутри рабочей группы без использования глобальной или локальной разделяемой памяти.

- Как следствие, необходимость использовать один тред на один блок, что значительно увеличивает расход регистров и снижает Occupancy
- Увеличивается размер исполняемого бинарного кода
- Увеличивается степень дивергенции тредов
- Большие накладные расходы при использовании массивов, если индекс элемента неизвестен на момент компиляции

Фактически, это может привести даже к снижению производительности.

5.1 Обмен данными между тредами

Современные графические процессоры имеют аппаратную возможность обмена данными между тредами без использования глобальной или локальной разделяемой памяти. Примерами могут служить инструкция DS_SWIZZLE_32 для архитектуры AMD GCN (Southern Islands) [5] или инструкция SHFL для архитектуры NVIDIA SM_30 [7]. Однако в OpenCL 1.2 такая возможность не предусмотрена и подобный обмен не может быть использован.

5.2 Уровень Occupancy

Использование одного треда на блок заставляет держать целиком весь блок и весь массив с ошибками в регистрах данного треда, что значительно увеличивает расход регистров на треда и на всю рабочую группу, снижая тем самым уровень Occupancy. Но высокий уровень Occupancy используется только, чтобы скрыть задержку при обращении к памяти. На графических процессорах NVidia наличие нескольких warp'ов на мультипроцессоре так же позволяет скрыть задержку, вызванную зависимостями инструкций по данным [6].

Поэтому в подобной ситуации уровень Occupancy в диапазоне 10%-20% может быть оптимальным[1] и дальнейший рост не приводит к увеличению производительности.

5.3 Размер исполняемого кода

Так как теперь отдельный тред выполняет большее количество работы, размер исполняемого кода может ощутимо увеличиться. Производительность может снизиться, если размер бинарного кода превысит размер кеша инструкций GPU. Для архитектуры AMD GCN он составляет 32 KB [4]. И хотя, как правило, это не является проблемой, для уменьшения размеров кода можно:

- разбить ядро (kernel) на несколько более мелких ядер и выполнить их последовательно
- уменьшить unroll-factor или полностью убрать директивы разворачивания циклов #pragma unroll для больших циклов. В некоторых ситуациях это может увеличить производительность

5.4 Дивергенция

Так как все треды сжимают отдельные блоки, увеличивается шанс, что разные треды одной аппаратной группы пойдут по разным ветвям условного перехода. Впрочем, рекомендации по оптимизации ветвлений не отличаются от обычного случая.

Например, код вида:

```
If (A>B) {
C += D;
} else {
C -= D;}
```

Следует заменять на:

```
int factor = (A>B) ? 1:-1;
C += factor*D;
```

Так как во втором случае не используются инструкции условных переходов.

5.5 Адресация элементов массива

Компилятор старается размещать массивы, объявленные без ключевого слова __global, в регистрах. Если все обращения идут по заранее известным константным индексам, то такой массив может рассматриваться просто как набор переменных, которые будут отображаться на фиксированные регистры. И хотя в архитектурах GPU имеются некоторые механизмы для адресации регистров (например, аппаратный регистр M0 в AMD GCN [5]), адресация по неизвестному индексу выполняется гораздо менее эффективно.

Поэтому, если в теле программы осуществляется доступ к элементам массива по неизвестному на момент компиляции индексу, то в большинстве случаев, компилятор будет вынужден разместить такой массив в глобальной или локальной памяти или использовать большое количество дополнительных инструкций. Особенно сильно это сказывается на производительности в архитектуре AMD GCN. О наличии подобных проблем можно судить по результатам профилирования и показаниям аппаратных счётчиков производительности по следующим признакам:

- Наличие memory stall при невысокой интенсивности обмена данными с памятью и блочному обмену с глобальной памятью и отсутствии конфликтов банков локальной памяти.
- Общий объём записанных и считанных из памяти данных значительно превышает ожидаемое значение. Одно обращение в исходном коде к массиву может порождать несколько десятков KiB трафика с памятью.
- В некоторых ситуациях «раздувание» бинарного кода.

На шаге номер 2 рассматриваемого алгоритма сжатия используется 2 таких массива. Для каждого текселя блока вычисляется, к какому из 4-х цветов, доступных в сжатом блоке, он ближе всего (bestInd), и в элементы массивов used и Bucket заносится информация об ошибке:

```
used[bestInd] = 1;
Bucket[bestInd] += MinError;
```

Так как bestInd не известен на момент компиляции, массивы used и Bucket размещаются в глобальной памяти. Но малый размер данных массивов позволяет произвести перебор по всем индексам:

```
#pragma unroll
for(int i=0; i<4; i++){
       char is = (bestInd == i) ? 1 : 0;
       used[i] |= is;
       Bucket[i] += is * MinError;
}
```

Данная оптимизация позволила значительно ускорить выполнение шага 2 (Табл. 2).

	AMD Radeon HD7970	NVidia GTX560
Local Memory	6.05 ms	18.1 ms
Registers, non compile-time constants indexing	12.37 ms	4.34 ms
Registers only, compile-time constants indexing	0.86 ms	3.88 ms

Таблица 2: Время выполнения шага 2 для тестовой текстуры 512x512

6. РЕЗУЛЬТАТЫ

Результаты тестирования производительности рассматриваемого алгоритма сжатия при выполнении на CPU и GPU представлены в Табл.3. Для запуска на CPU использовался оригинальный многопоточный компрессор от автора алгоритма [2]. Тестирование проводилось на максимальных настройках качества на следующем оборудовании:

- CPU: Intel Core i3 660 (2 cores, 4 threads, 3.33 GHz)
- GPU1: Nvidia GeForce GTX560 (336 CUDA cores, 1.6 GHz)
- GPU2: AMD HD7970 (2048 PE cores, 925 MHz)

	Time	Speedup
Intel Core i3 660	1523 s	-
NVidia GTX560 (local memory)	50.8 s	30 x
NVidia GTX560 (registers)	30.7 s	50 x
AMD HD7970 (local memory)	16.4 s	92 x
AMD HD7970 (registers)	6.8 s	224 x

Таблица 3: Скорость сжатия тестовой текстуры 512x512 на максимальном качестве

7. ЗАКЛЮЧЕНИЕ

Несмотря на то, что использование только регистровой памяти приводит к существенным проблемам при обмене данными между тредами и сложностям при работе с массивами, такой подход может дать ощутимый выигрыш по скорости.

Однако даже для алгоритмов, активно использующих локальную или глобальную память, использование массивов, относящихся к private области памяти, может служить причиной снижения производительности. Зачастую это временные массивы небольшой размерности. Для повышения быстродействия в подобных ситуациях можно попробовать использовать подход, описанный в разделе 5.5, или другой способ, уменьшающий количество обращений по индексам неизвестным на момент компиляции.

8. ССЫЛКИ

[1] Volkov V., *Better Performance at Lower Occupancy*, 2010, http://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf

[2] [Release] LSDxt DXT Compressor, 2012, http://lspiroengine.com/?p=516

[3] AMD Accelerated Parallel Processing OpenCL Programming Guide, 2012, http://developer.amd.com/download/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf

[4] AMD Graphics Cores Next (GCN) Architecture, 2012, http://www.amd.com/la/Documents/GCN_Architecture_whitepaper.pdf

[5] AMD Southern Islands Series Instruction Set Architecture, 2012, http://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf

[6] NVidia OpenCL Best Practices Guide, 2011

[7] NVidia Parallel Thread Execution ISA version 3.0, 2012

Об авторе

Илья Перминов – аспирант кафедры вычислительной техники в НИУ ИТМО (Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики)

Email: studentikispam@gmail.com