ИССЛЕДОВАНИЕ ВОЗМОЖНОСТЕЙ РАСПАРАЛЛЕЛИВАНИЯ АЛГОРИТМОВ СЖАТИЯ ПОЛНОЦВЕТНЫХ ИЗОБРАЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ ОБЩИХ ВЫЧИСЛЕНИЙ OpenCL

© 2016 г. А.Д. ГОРГУРОВ

Московский технический университет связи и информатики

В области обработки изображений существует множество различных алгоритмов. В этой статье мы рассмотрим способы распараллеливания алгоритма, в основе которого лежит адаптивная развёртка. Под *развёрткой* мы будем понимать способ восстановления («разворачивания») битового изображения из множества структур – *путей развёртки*, в которых хранится направление и (в некоторых модификациях) значение единичного элемента изображения. При генерации путей развертки направление зависит от текущего единичного элемента по некой функции f(px), где px – значение единичного элемента изображения (пикселя) в точке изображения. Эта функция также зависит от набора других параметров, называемых *контекстом функции*. В зависимости от реализации *контекстом функции* является набор статистических данных, по которым строятся пути развёртки. Эти данные рассчитываются функцией stats(img, w, h), где img – битовое изображение, w – ширина, h – высота, и затем передаются функции f(px). Пути в дальнейшем сжимаются любым алгоритмом сжатия двоичных данных для более компактного представления. Такие алгоритмы легко масштабируются на многоцветные изображения с помощью разбиения таких изображений на битовые плоскости и передачи этих битовых плоскостей на вход алгоритму.

Существует множество способов и вспомогательных инструментов для распараллеливания алгоритмов. В [1] и [2] были рассмотрены способы распараллеливания с помощью специальных исполняющихся на GPU программ – шейдеров – которые написаны на диалекте языка Си – языке GLSL и исполняются на программируемой стадии графического конвейера OpenGL 2.x и выше (а также доступны в 1.x, если есть необходимое расширение ARB_fragment_shader). Там же [1][2] было показано, что перенос простых алгоритмов сопряжен с некоторой сложностью из-за технических ограничений языка и графической подсистемы. В этой статье мы рассмотрим возможности распараллеливания алгоритма развертки, описанного ранее, с помощью программного интерфейса OpenCL – Open Computing Library – первые версии которого были разработаны компанией Apple и потом были переданы комитету Khronos на дальнейшее развитие[3]. С первых версий OpenCL имеет возможность тесно взаимодействовать с контекстами исполнения OpenGL, благодаря чему приложения могут выносить различные «тяжелые» вычисления на видеокарту и визуализировать их результаты без участия центрального процессора, что приводит к увеличению производительности, т.к. не приходится пересылать данные туда и обратно, а также центральный процессор может заниматься другой деятельностью, например, GUI. Также OpenCL поддерживает схожий с OpenGL механизм расширений, благодаря которому он может взаимодействовать с такими библиотеками, как Direct 3D (в частности с 9, 10, 11 и 12 версиями), Vulkan (http://khronos.org/vulkan) и другими, если вычислительное устройство поддерживает нужные расширения.

В OpenCL используется модель выполнения «клиент-сервер», где клиентом (хостом) является программа – kernel (ядро), – которая инициализирует контекст OpenCL, выбирает нужное устройство, посылает на него программный код на языке

OpenCL C и входные данные, и забирает выходные данные, а «сервером» является выбранное устройство, на котором могут выполняться параллельно несколько рабочих групп (work groups) рабочих потоков (work-items, рабочих элементов). В качестве примера рабочего потока можно привести ядра процессора, унифицированные потоковые процессоры видеокарты, различные процессоры одного DSP, виртуальные потоки исполнения (Hyper-Threading) и пр. Из такой многослойной архитектуры можно понять, что в классификации OpenCL может существовать несколько видов памяти:

1. Память хоста (Host Memory) – именно в ней мы подготавливаем входные данные для исполнения нашей программы – kernel instance;
2. Память всего устройства (Global / Constant Memory) – в ней хранятся все глобальные / константные переменные программ, выполняемых на устройстве;
3. Память рабочей группы (Local Memory) – доступна только рабочим потокам, входящим в одну группу;
4. Память рабочего потока (Private Memory) – доступна только рабочему потоку. В ней обычно хранятся локальные переменные выполняемых этим потоком функций.

Очевидно, что в зависимости от вида устройства (CPU/GPU) задержка (overhead) пересылки данных между разными данными может быть существенной [3] (к примеру, на CPU данные обычно пересылаются «сразу», за десятки нс, которые уходят на отображение (map) области памяти на нужное ядро, а вот на GPU или DSP данные идут по шине – PCIe для современных GPU и PCI/UART/пр. для DSP, что может приводить и микросекундным задержкам). Поэтому в OpenCL необходимо явно давать команды на пересылку данных из одного вида памяти в другой во время инициализации программы на устройстве и на стадии получения данных. Помимо этого, могут возникнуть задержки следующего рода:

- Задержки на передачу информации с устройства на хост;
- Задержки на инициализацию OpenCL и устройства;
- Задержки на компиляцию для заданного устройства;
- Задержки на синхронизацию, если у нас есть конкурентный доступ у рабочих потоков к разделяемой памяти (local/global/constant).

Анализ алгоритма обработки изображения

Рассмотрим тестовую реализацию стадии подготовки контекста для алгоритма генерации путей развёртки изображения. В качестве данных контекста мы возьмем количество переходов значений пикселя от 1 в 0 и от 1 в 1. Под переходом мы будем понимать изменение значения пикселя из одного значения в другое на определенное направление: 0 – вправо, 1 – вправо-вниз, 2 – вниз, 3 – влево-вниз, 4 – влево, 5 – влево вверх, 6 – вверх, 7 – вправо-вверх. В итоге мы имеем функцию stats(), принимающую на вход одну битовую плоскость, а на выходе возвращающую набор переходов – массив из 16 чисел (первые 8 – количество переходов от 1 в 0, вторые 8 – количество переходов от 1 в 1), которые назовем статистикой алгоритма.

Алгоритм функции будет выглядеть следующим образом:

1. Для каждого пикселя из изображения сделать следующее действие:
2. Если текущий пиксель ≠ 1, то берем следующий пиксель и повторяем этот пункт;
3. Для каждого направления пикселя изображения (от 0 до 7 включительно) необходимо сделать следующие действия:
4. Если мы можем перемещаться в заданном направлении, то мы проверяем пиксель, доступный по тому направлению. Если он равен 0, то мы заполняем соответствующую переходу 1 в 0 и текущему направлению ячейку набора переходов, если равен 1, то делаем аналогичное действие, но для части набора, где переходы 1 в 1.
5. Если направления закончились, то мы берем следующий пиксель и идем в п. 2, если закончились пиксели – то алгоритм завершается. Если не закончились направления, берем следующее направление и идем в п.4.

Для того, чтобы распараллелить эту функцию, нам необходимо сделать следующие шаги:

1. Проанализировать возможные параллельные и конкурентные участки выполнения алгоритма. Под параллельными участками выполнения будем понимать те участки, в которых рабочие элементы могут работать, не мешая друг другу, конкурентные участки – те участки, в которых рабочие элементы могут соперничать за общий ресурс. Основным видом соперничества являются операции чтения и записи разными рабочими элементами в одну и ту же область памяти. Так как порядок выполнения рабочих элементов не определен, то в зависимости от того, какой из рабочих элементов первым сделает операцию чтения или записи в общую область памяти, будет зависеть разный результат. Такая ситуация называется гонкой (race condition);
2. Записать алгоритм на языке OpenCL C (с учетом его ограничений), обозначив главную функцию квалификатором __kernel;
3. Создать контекст необходимого устройства;
4. Скомпилировать текст алгоритма на языке OpenCL C, используя функцию clBuildProgram() [4];
5. Создать очередь команд для этого устройства, в которую мы будем помещать наши задания на выполнения определенных программ-ядер;
6. Передать входные параметры программе-ядру;
7. Отправить запрос на выполнение программы-ядра на устройстве и ждать результата;
8. Запросить результат выполнения программы-ядра.

Найдем возможные параллельные участки в алгоритме. Для этого необходимо проанализировать схожие по действиям последовательности, которые можно запустить параллельно. Таким участком в нашем алгоритме является цикл обхода направлений (п.3-п.5), т.к. мы можем одновременно получать доступ ко всему изображению, что не ведет к конкуренции, т.к. изображение не модифицируется, а модицируемая область – набор переходов – может спокойно изменяться каждым из рабочих элементов, которые теперь отвечают за свое направление. Для такого случая нам необходимо использовать 8 рабочих элементов.

Следует отметить, что также возможно распараллелить алгоритм, чтобы каждый рабочий элемент отвечал за свой пиксель (или группу пикселей). Однако тогда в таком случае у нас будет конкуренция за доступ к набору переходов. Эту проблему можно решить либо синхронизацией рабочих элементов – тогда у нас они начинают работать последовательно, один ждет, пока второй сделает операцию записи в набор; либо вводом дополнительных наборов переходов, которые потом, по завершению вычислений, надо будет сложить для получения конечного результата. Несмотря на то, что второй подход предлагает интересную концепцию работы потоков исполнения (также известную как «fork-join»[5]), мы его оставим для дальнейших исследований и не будем рассматривать в этой статье.

Подход к распараллеливанию расчета статистики битового изображения спокойно расширяется на общий случай полноцветного изображения. Единственное отличие будет в том, что на устройство будет передаваться битовый массив изображения, и сами рабочие элементы (которых уже понадобится 8 * количество бит в изображении) смогут извлекать нужные биты, используя свой рабочий номер.

Теперь проанализируем два подхода к реализации подготовки контекста для алгоритма развёртки битового изображения: «классический», который выполняется на CPU без лишних накладных расходов на инициализацию OpenCL, передачу данных и компиляцию исходного кода, и подход, использующий возможности OpenCL для распараллеливания. Разумно предположить, что для маленьких изображений классический подход будет работать быстрее. Проверим это утверждение, измерив время выполнения каждой реализации и накладные расходы алгоритма, в которые отнесем все остальные задержки, которые возникают в программе. Измерять мы будем время работы классической реализации с помощью методики, основанной на использовании функций ОС Windows QueryPerformanceCounter / QueryPerformanceFrequency с учетом замечаний из [1] и [6]. Чтобы померить время выполнения алгоритма на устройстве,

OpenCL предоставляет возможность профилирования выполнения программ-ядер с помощью функции clGetEventProfilingInfo(). Эта функция позволяет узнать, время начала пересылки запроса на выполнения программы-ядра, время запуска программы-ядра, время завершения и время конца пересылки ответа от устройства до хоста. Точность этой функции зависит от аппаратных датчиков, и ее также можно узнать средствами OpenCL, а именно, функцией-запросом clGetDeviceInfo (CL_DEVICE_PROFILING_TIMER_RESOLUTION) к нашему устройству.

Характеристики устройства, на котором исполнялись реализации алгоритмов, приведены в табл. 1. Характеристики компьютера-хоста, на котором проводились измерения задержек передачи данных и «классическая» реализация, приведены ниже:

- ОС Windows 8.1
- 6 Gb RAM
- CPU Intel Core i3 (Haswell) @2.0 GHz;

Таблица 1

Характеристики исполняющего устройства[1]
(Тестовый код собирался с использованием Microsoft Visual Studio 2013
в режиме максимальной оптимизации скорости исполнения кода)

Тип	GPU
Профиль устройства OpenCL	Полный профиль[2] (FULL_PROFILE)
Название	NVIDIA GeForce 820M
Версия драйвера OpenCL	368.81
Версия интерфейса OpenCL	OpenCL 1.1 CUDA
Количество вычислительных устройств	2
Тактовая частота	1250 МГц
Максимальное количество рабочих элементов	67108864
Максимальное количество глобальной памяти	2048Мб
Точность таймера профилирования	1000 нс

Примечания: 1) так как предлагаемые реализации использует только глобальную память для входного битового изображения и выходного набора переходов, то мы не приводим размер локальной и приватной памяти, а приводим только те характеристики, от которых зависит исполнение рассматриваемых реализаций);

2) также есть EMBEDDED_PROFILE, поддерживаемый различными портативными (embedded) устройствами. В этом профиле часть возможностей ограничена, в частности, набор математических функций, точные вычисления с числами с плавающей точкой, отсутствует требование поддержки 64-битных целых и др. Также большинство устройств поддерживает загрузку бинарного кода в своем формате, которое определяется спецификацией расширения OpenCL, примеру, cl_altera_compiling_mode).

В качестве входных данных были использованы наборы тестовых изображений размерами 5*5 пикселей, 1 бит и 24 бит – для тестирования задержек передачи данных и константных задержек инициализации, 128*128 пикселей, 1 и 24 бит – для тестирования небольших изображений, 512*512 пикселей, 1 и 24 бит – для тестирования средних изображений и 4096*4096 пикселей – для тестирования больших (4k) изображений.

Результаты измерения времени исполнения в микросекундах (среднее арифметическое десяти запусков) приведены в табл. 2.

Результаты измерения задержек при исполнении алгоритма (тоже среднее арифметическое десяти запусков) на GPU приведены в табл. 3.

Сравниваемые реализации стадии подготовки контекста для алгоритма генерации путей:

- CPU – использующая классический подход, и выполняющаяся на CPU;
- GPU 1 – использующая OpenCL, но без параллелизма (на 1 рабочем элементе);

• GPU 8 – использующая OpenCL, с параллелизмом по направлениям (для каждого столбца статистики использовался свой рабочий элемент).

Таблица 2

Результаты измерений временных задержек алгоритма, использующего CPU, использующего GPU (px – пиксели)

A / B	5, 1	5, 24	128, 1	128, 24	512,1	512, 24	4096, 1	4096, 24
CPU	3.78137	72.93	1540.10	38704.39	29897.61	833327.88	2131223.50	43580156.000012
GPU 1	421743,28	418645,28	418411,18	419565,88	419069,58	419447,00	420622,14	421404,86
GPU 8	418351,09	419524,05	418759,71	419071,07	419156,78	418427,74	424128,70	422875,74

Условные обозначения: A – Разрешение входного изображения, его ширина в пикселях, битность;
B – тип реализации и ее время выполнения, в мкс.

Также отдельно выделим измеренные задержки выполнения следующих частей алгоритмов GPU 1 и GPU 8:

1. Инициализации OpenCL и присоединения к устройству (далее init);
2. Компиляции программы-ядра (compile);
3. Ожидания в очереди исполнения программы-ядра (exec_queue);
4. Пересылка двоичных данных программы-ядра на устройство (exec_tx_dev);
5. Выполнение алгоритма на устройстве (exec);
6. Задержка при получении результата с устройства (exec_rx_dev).

Результаты приведены в табл. 3.

Таблица 3

Измеренные задержки при исполнении алгоритмов GPU 1 и GPU 8

A / B	5, 1	5, 24	128, 1	128, 24	512,1	512, 24	4096, 1	4096, 24
GPU1 init	416537,50	416537,50	416537,50	416537,50	416537,50	416537,50	416537,50	416537,50
GPU1 compile	3934,78	1802,63	1546,04	2698,81	1932,82	2408,73	1666,50	2430,33
GPU1 exec_queue	31,55	58,82	43,26	32,38	38,14	36,51	34,91	26,21
GPU1 exec_tx_dev	433,34	116,16	105,34	102,88	103,87	61,79	74,34	64,42
GPU1 exec	23,74	12,03	71,84	72,26	271,78	273,34	2140,64	2149,44
GPU1 exec_rx_dev	782,37	118,14	107,20	122,05	185,47	129,12	168,26	196,96
GPU8 init	416537,50	416537,50	416537,50	416537,50	416537,50	416537,50	416537,50	416537,50
GPU8 compile	1446,10	2626,97	1852,87	2080,83	1818,83	1148,99	2898,68	1611,94
GPU8 exec_queue	32,48	89,47	37,73	68,22	34,46	24,61	27,71	25,25
GPU8 exec_tx_dev	63,94	173,89	94,21	113,50	93,22	51,14	75,17	63,65
GPU8 exec	46,46	12,90	144,93	145,09	563,62	565,06	4470,78	4473,34
GPU8 exec_rx_dev	224,61	83,33	92,48	125,92	109,15	100,45	118,85	164,06

Условные обозначения: A – Разрешение входного изображения, его ширина в пикселях, битность;
B – тип реализации и ее время выполнения, в мкс.

Как видно из усредненных данных табл. 2, реализация на центральном процессоре действительно работает быстрее на небольших изображениях (до 512*512 пикс.), чем реализации на GPU. Однако для бо́льших по объему данных даже вариант алгоритма, исполняющегося на GPU без параллелизма, выполняется быстрее, чем вариант на CPU, а накладные расходы на передачу данных не сильно влияют на скорость. Время инициализации GPU для работы с библиотекой OpenCL, занимает значительное время по сравнению с временем выполнения алгоритма. При выполнении алго-

ритма на GPU действительно происходят задержки на стадиях инициализации, компиляции, передаче и приеме информации, и при маленьком объеме данных выигрыш от использования графического чипа незначителен. После первого запуска также видно, что время компиляции сократилось примерно вдвое, что можно объяснить оптимизацией компилятора библиотеки OpenCL, который при анализе входного исходного кода программы может повторно использовать предыдущий вариант компиляции [4]. Колебания в цифрах результатов можно объяснить тем, что мы ждали результата синхронно, а GPU работает асинхронно, и драйверу взаимодействия с частью библиотеки OpenCL режима пользователя приходится ждать ответа от GPU, чтобы вернуть результат. Также драйвер старается уменьшить количество переключений контекста с режима пользователя в режим ядра и для этого может структурировать запросы на взаимодействие с GPU в очередь, чтобы при очередном переключении контекста сразу выполнить несколько операций. Однако синхронные запросы заставляют его выполнять отложенные операции, игнорируя его группировку в очередь. Из всего этого для увеличения общей производительности на большом наборе изображений рекомендуется однократная инициализация GPU, явное повторное использование результатов компиляции (а если реализация позволяет, то использовать бинарный код вычислительных устройств), загрузка по возможности большего количества вычислительных элементов GPU, асинхронная загрузка изображений и получения результатов.

СПИСОК ЛИТЕРАТУРЫ

1. *Горгуров А.Д., Челейкин М.Ю.* исследование вопросов оптимизации и распараллеливания алгоритмов обработки изображений // Материалы научно-технической конференции ИНТЕРМАТИК-2014. – М., МИРЭА, 2014, с. 186-188.
2. *Горгуров А.Д.* Использование современных возможностей интегрированных графических процессоров в алгоритмах обработки изображений // Материалы научно-технической конференции ПТСПИ. Владимир, 2015.
3. NVIDIA OpenCL Programming Guide [Электронный ресурс] // NVIDIA OpenCL SDK. URL: http://developer.nvidia.com/ (дата обращения: 06.08.2016) – 61 с.
4. The OpenCL 1.0 Specification [Электронный ресурс] // Khronos Inc. URL: http://khronos.org/registry/cl/sdk/ (Дата обращения: 06.08.2016) – 316 с.
5. *Батчер П.* 7 моделей конкуренции и параллелизма за 7 недель. / пер. с англ. Киселев А.Н. – М.: ДМК Пресс, 2015, - 360 с.: ил.
6. *Карпов А.* Вечный вопрос измерения времени [Электронный ресурс] // Static Code Analyzer for C/C++/C++11 and 64-bit migration. URL: http://www.viva64.com/ru/b/0097/ (дата обращения: 20.08.15).