

УДК 004.272

doi 10.26089/NumMet.v19r434

ИССЛЕДОВАНИЕ МЕТОДОВ РЕАЛИЗАЦИИ ФИЛЬТРА СОБЕЛЯ НА РЕКОНФИГУРИРУЕМЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ

Е. Е. Таратута¹

Основной мотивацией для данной статьи послужила растущая популярность реконфигурируемых ПЛИС (программируемых логических интегральных схем) в области высокопроизводительных вычислений, а также необходимость исследовать особенности написания эффективных программ с использованием инструмента Altera SDK for OpenCL, позволяющего писать приложения для ПЛИС с помощью языка программирования Си++ и стандарта OpenCL. В статье рассмотрены особенности программной модели OpenCL и представлено исследование производительности и особенностей различных реализаций фильтра Собеля с использованием указанного инструмента.

Ключевые слова: ПЛИС, FPGA, OpenCL, фильтр Собеля, Altera OpenCL SDK.

1. Введение. Значительная гибкость программируемых логических интегральных схем (ПЛИС) и их потенциал производительности сделали их привлекательным выбором в широком диапазоне вычислительных областей, от быстрого прототипирования схем до высокопроизводительных вычислений. Увеличение количества транзисторов на чипе позволило создавать их с большим количеством вычислительных ресурсов и топологий межсоединений, что привело к созданию систем с широким спектром вариантов реализации. Раньше, учитывая ограниченную емкость устройств ПЛИС, они использовались, как правило, в качестве интерфейсных логических схем (glue-logic), теперь же возможность конфигурировать эти устройства позволяет инженерам-разработчикам решать множество различных задач.

ПЛИС, или FPGA (Field Programmable Gate Arrays) представляют собой цифровые интегральные микросхемы, состоящие из программируемых логических блоков и программируемых соединений между этими блоками.

Огромная популярность ПЛИС продиктована также возможностью их программирования на месте. Другими словами, в отличие от устройств, внутренняя функциональность которых жестко прописана производителем, ПЛИС представляют собой своего рода “конструктор” — набор деталей, из которых можно построить любые цифровые электронные устройства неограниченное число раз. Примером может служить вычислительное устройство, способное выполнять одну и только одну необходимую нам вычислительную процедуру, после чего можно перепрограммировать устройство на выполнение другой вычислительной процедуры. ПЛИС-устройства могут программироваться как непосредственно, так и как часть некоторой уже работающей электронной системы. Во втором случае они называются внутрисистемно программируемыми.

Например, большинство процессоров не имеют специальной инструкции, которая может выполнять следующий код на языке Си: $((A + B) \& D) \gg 2$. Без специальной инструкции для этого кода процессоры CPU или GPU должны выполнить несколько инструкций по выполнению операции. Однако можно использовать ПЛИС как аппаратную платформу, реализующую любой набор команд, который требует создаваемый программный алгоритм. Можно настроить ПЛИС на реализацию приведенного выше кода за один такт, построив соответствующий конвейер. Однако за эту возможность приходится платить отсутствием свойственной классическим вычислителям возможности использовать однажды прошитую плату для различных задач. Конкретный длинный конвейер можно построить только для реализации одной конкретной, известной заранее вычислительной процедуры, поэтому технология длинных конвейеров и является основной именно в вычислителях на базе ПЛИС.

До недавнего времени, чтобы использовать потенциал реконфигурируемых архитектур ПЛИС, программисты вынуждены были отображать свои приложения, написанные обычно на языках программирования высокого уровня, таких как C/C++ или MATLAB, на аппаратно-ориентированные языки, такие как Verilog или VHDL. В этом процессе они должны были взять на себя роль разработчиков аппаратных средств и системных программистов и перемещаться по лабиринту программных преобразований для

¹ Московский государственный университет им. М. В. Ломоносова, факультет вычислительной математики и кибернетики, Ленинские горы, 119991, Москва; аспирант, e-mail: taratutakaty92@gmail.com

создания эффективных реконфигурируемых вычислительных реализаций. Необходимо было выделить комбинационную и последовательную логику приложений [5], сформировать определенные операции в параллельных и последовательных, конвейерных и неконвейерных частях устройства, построить операционный и управляющий автомат (автомат Мура и автомат Мили соответственно) [5], описать регистр и регистровый файл, и в конечном итоге все функции устройства описать терминами RTL (уровня регистровых передач) на языке Verilog/VHDL.

Программирование ПЛИС с помощью языков описания схем для использования его в качестве ускорителя вычислений представляет собой достаточно трудоемкую и нудную работу. Например, программа, в которой несколько процессоров совершают несколько итераций цикла всего лишь для увеличения значения одной и той же переменной x , на языке Verilog состоит из порядка 600 строк. И хотя большинство из этих строк однотипные и отличаются только названиями того или иного провода или регистра, их все же необходимо явно прописать, не совершив при этом случайных ошибок или описок.

Увеличение количества программируемых блоков ПЛИС, увеличение размера программного кода, богатство и изощренность любого из описанных выше шагов делали вычисления с этими архитектурами все более сложным процессом.

Производители ПЛИС весьма разумно задумались о том, что нужно сокращать время, необходимое на программирование и отладку программ под ПЛИС: позволить программистам писать их легко и быстро.

Одним из вариантов написания программы для параллельных вычислений на ПЛИС, появившимся в 2012 г., является OpenCL. OpenCL является стандартом для написания параллельных программ для гетерогенных вычислительных систем. С помощью инструмента Altera SDK for OpenCL, представляющего собой набор библиотек и приложений, который позволяет компилировать код на C/C++ и OpenCL в прошивку для ПЛИС фирмы Altera. Конструкции OpenCL синтезируются в пользовательскую логику с целью получения конфигурационного файла для устройства ПЛИС, что дает возможность программисту, незнакомому с языками описания схем, использовать ПЛИС как ускоритель.

2. Стандарт OpenCL. OpenCL (Open Computing Language, открытый язык вычислений) — это открытый стандарт для параллельного программирования, предлагающий эффективный и переносимый способ использования возможностей разнородных вычислительных многоядерных платформ (CPU, GPU, FPGA и др.). Он включает в себя программный интерфейс для координации параллельных вычислений в среде разнородных процессоров и кроссплатформенный язык, используемый в определенном вычислительном окружении.

На конкретной системной платформе стандарт реализуется в виде исполняемых модулей (библиотек), достаточных для запуска пользовательских OpenCL-приложений. Разработка подобных приложений предполагает наличие также C/C++-компилятора для целевой платформы и набора необходимых заголовочных файлов.

OpenCL-приложение состоит из программного кода, исполняемого на хосте (host) и OpenCL-устройствах (device). Под хостом обычно понимается центральный процессор вычислительного устройства (компьютера, планшета, телефона и т.п.), а устройства OpenCL — некоторый набор вычислительных единиц, соответствующий, например, графическому процессору (GPU), многоядерному CPU, программируемой логической интегральной схеме или другим процессорам с параллельной архитектурой, доступным из хост-программы. Исполнение OpenCL-приложения, таким образом, включает в себя исполнение хост-программы (на хосте) и исполнение специального кода на одном или нескольких OpenCL-устройствах под управлением хост-программы. На рис. 1 представлена соответствующая модель программирования OpenCL.

Программы, исполняемые на OpenCL-устройствах, содержат одно или несколько ядер (kernels — функции, помеченные специальным ключевым словом `__kernel` и являющиеся “точкой входа” в исполняемый на устройствах код), при необходимости — вспомогательные функции, вызываемые в тексте ядер, а также, возможно, константные данные.

Программная модель OpenCL-приложения — SIMD (Single Instruction Multiple Data) или SIMT (Single Instruction Multiple Threads): код ядра может быть исполнен одновременно на многих вычислительных единицах, каждая из которых работает со “своими” данными.

Выполнение команд в ядре происходит в объектах, называемых рабочими единицами (work-item). Каждая рабочая единица не зависит от других и может исполнять код параллельно с остальными. Если же процесс из одной рабочей единицы хочет получить данные, используемые или уже обработанные любой другой рабочей единицей, и/или обменяться данными с хостом, то он может это сделать через общую память (константную, глобальную, global memory; виды памяти OpenCL подробнее описаны ниже). Кроме того, в OpenCL у каждой рабочей единицы может быть локальная память (local memory), но не любой

процесс может получить к ней доступ. К локальной памяти могут обращаться только рабочие единицы одной рабочей группы (compute unit или workgroup). В зависимости от устройства в каждой из этих групп может быть различное (фиксированное или нет) количество рабочих единиц. Существует также закрытая память (приватная, private memory), к которой рабочая единица может обращаться единолично. После запуска на сопроцессоре все процессы равнозначны и исполняют равнозначный код.

В терминологии OpenCL каждая исполняемая копия кода ядра (рабочая единица) характеризуется своими уникальными индексами: глобальным идентификатором (global ID) и локальным идентификатором (local ID). Этим индексам два, поскольку локальный индекс характеризует конкретную рабочую единицу в рамках рабочей группы, а глобальный индекс уникально характеризует рабочую единицу независимо от принадлежности к какой-либо рабочей группе.

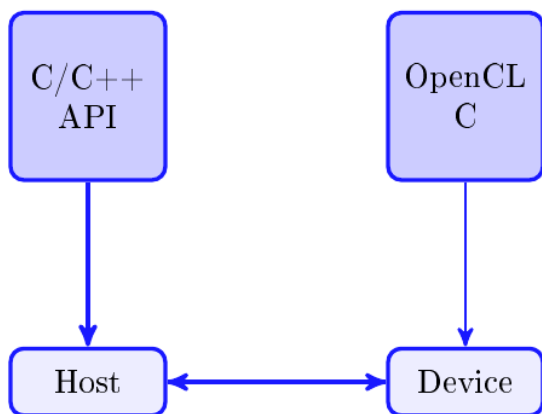


Рис. 1. Модель программирования OpenCL

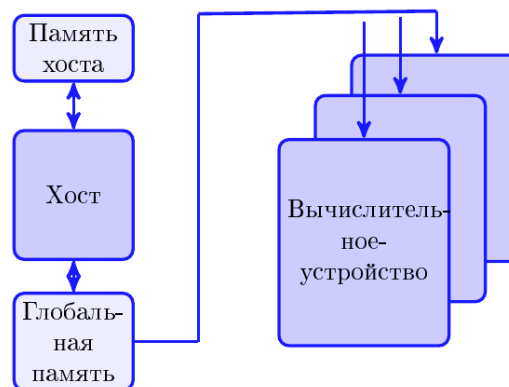


Рис. 2. Модель памяти OpenCL

Модель памяти OpenCL представлена на рис. 2.

Ядрам в OpenCL доступно четыре вида памяти: глобальная, константная, локальная и приватная. Все рабочие единицы имеют доступ к глобальной памяти на чтение и на запись. Входная информация переносится в глобальную память с хоста, а результаты вычислений из глобальной памяти возвращаются обратно на хост.

Константная память доступна всем рабочим единицам, но только для чтения. Выделяется и инициализируется этот вид памяти на хосте.

Локальная память — это область памяти, общая для всех рабочих единиц в рамках одной рабочей группы. Посредством нее рабочие единицы группы могут обмениваться информацией друг с другом.

Приватная память — область памяти для локальных переменных экземпляра ядра.

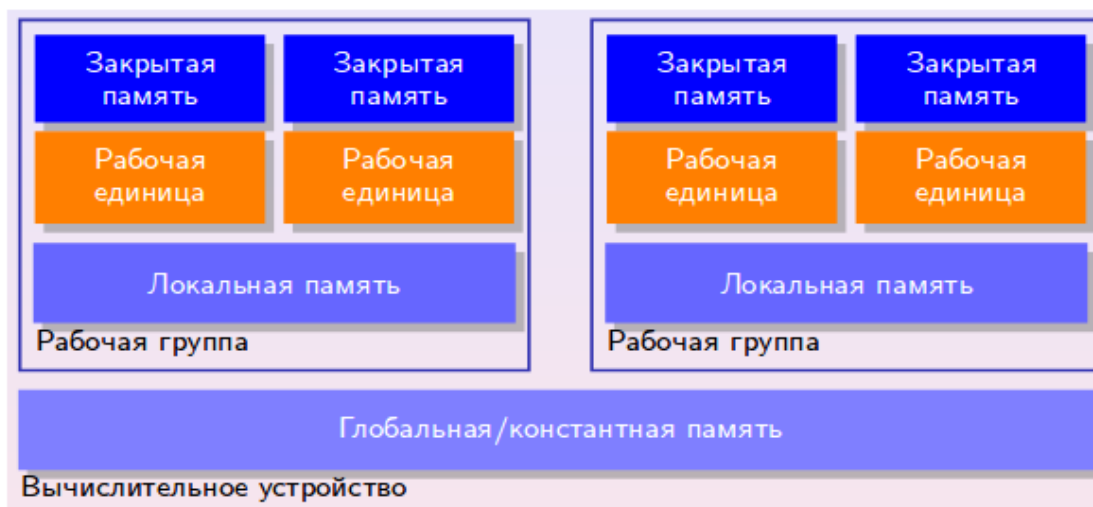


Рис. 3. Иерархия памяти в OpenCL

Иерархия памяти OpenCL представлена на рис. 3.

Любая рабочая единица имеет свою копию каждой локальной переменной, которая доступна только ей, но не другим рабочим единицам.

3. Исследование реализаций фильтра Собеля. В данном разделе представлены описание фильтра Собеля, варианты его реализации для ПЛИС и исследование их производительности.

Для того чтобы писать приложения для Altera FPGA с помощью OpenCL, необходимы:

- FPGA: Altera Stratix IV и старше, семейство Altera Cyclon;
- Altera Quartus 2 и старше — САПР (система автоматизированного проектирования) для проектирования и отладки проектов. Эта система позволяет проектировать логику работы микросхем схематически и на языках программирования Verilog, VHDL, AHDL и других языках описания схем. Среда программирования Altera Quartus II так же позволяет производить симуляцию проектов, программировать микросхемы и многое другое;
- Altera OpenCL SDK for FPGA — это набор библиотек и приложений, который позволяет компилировать код, написанный на Си и OpenCL, в прошивку для ПЛИС фирмы Altera. Это дает возможность программисту использовать ПЛИС как ускоритель высокопроизводительных вычислений без знания языков описания схем;
- Microsoft Visual Studio 2010 Pro и старше — используется для написания хост-программы и интеграции с Altera OpenCL SDK (это, вообще говоря, не единственный подход, но, вероятно, самый удобный и используемый).

3.1. Фильтр Собеля. Фильтр Собеля — это оператор, вычисляющий приблизительный градиент яркости на основе градиентов изображений вдоль осей x и y . Наиболее распространенным примером практического использования является определение границ объектов на изображении, т.е. точек резкого изменения яркости.

Этот оператор использует ядра 3×3 , с которыми и происходит свертка рассматриваемого изображения по вертикали и по горизонтали. Пусть A — исходное изображение, представленное матрицей, элементами которой являются пиксели изображения, а G_x и G_y — два изображения, на которых каждая точка содержит приближенные производные по x и по y . В классическом варианте они вычисляются следующим образом:

$$G_x = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * A, \quad G_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A,$$

где $*$ — операция свертки изображения, т.е. каждый пиксель a_{ij} изображения, кроме граничных, изменяется по правилу

$$\begin{aligned} G_x(ij) &= a_{i-1j-1} * (-1) + a_{i-1j} * (-2) + a_{i-1j+1} * (-1) + a_{i+1j-1} * 1 + a_{i+1j} * 2 + a_{i+1j+1} * 1, \\ G_y(ij) &= a_{i-1j-1} * (-1) + a_{i-1j+1} * 1 + a_{ij-1} * (-2) + a_{ij+1} * 2 + a_{i+1j-1} * (-1) + a_{i+1j+1} * 1, \end{aligned}$$

а для граничных элементов $G_x(ij) = G_y(ij) = a_{ij}$.

Величина градиента вычисляется по формуле $G = \sqrt{G_x^2 + G_y^2}$ либо $G = |x| + |y|$. Здесь G — результат применения оператора Собеля в горизонтальном и вертикальном направлениях.

Каждый пиксель изображения может обрабатываться параллельно с другими, однако он зависит от восьми соседних, что в общем случае ведет к постоянным обращениям к одним и тем же фрагментам памяти.

3.2. Различные реализации. Начнем с первой, “наивной” реализации оператора Собеля: каждый пиксель изображения обрабатывается отдельной рабочей единицей. Всего получается 1280×720 рабочих единиц (по размеру изображения). Каждая рабочая единица получает из глобальной памяти обрабатываемый элемент: 6 соседних элементов для вычисления элемента матрицы G_x и 6 соседних элементов для вычисления элемента матрицы G_y (всего 13 обращений к памяти). Кроме того, рабочая единица вычисляет операцию свертки и градиента параллельно. Получаем 45 секунд. Полученное время связано с тем, что все рабочие единицы пытаются одновременно получить данные из медленной глобальной памяти, из-за чего возникает много конфликтов доступа в память.

Исправим немного первую реализацию: будем сохранять в локальных переменных 10 элементов входного массива (1 обрабатываемый пиксель, а у соседних элементов, необходимых для вычисления G_x и G_y , есть пересекающиеся), участвующих в преобразовании, чтобы избежать двукратного доступа в память.

Элементы, к которым происходит лишь однократный доступ, тоже заранее переносим в локальную память (так как ПЛИС для каждой последовательности операций способна выстроить собственный конвейер, а периодические доступы обратно в глобальную память могут его приостанавливать). Получаем 37 секунд.

Теперь рассмотрим вариант обработки каждой рабочей единицей отдельной строки (группы строк). При этом данные предварительно переносим в локальную память. Это позволит не обращаться постоянно в глобальную память, а копировать за одно обращение группы строк на устройство, а также значительно снизит нагрузку на пропускную способность памяти за счет значительного уменьшения количества работающих рабочих единиц (теперь их $1280 \times 720 / \text{кол-во обрабатываемых строк каждой рабочей единицей}$).

В таблице представлены полученные результаты в зависимости от количества строк, обрабатываемых отдельной рабочей единицей: 1, 5 и 25 строк соответственно.

Вычисление алгоритма Собеля для 1280×720 .
Каждая рабочая единица обрабатывает 1, 5 или 25 строк

Кол-во изображений	1 строка, время (сек.)	5 строк, время (сек.)	25 строк, время (сек.)
1	0.071	0.029	0.018
100	6.06	2.78	2.04
500	31.05	14.76	11.14
1000	61.28	27.39	20.92
1500	92.54	44.14	31.63

Наиболее оптимальный вариант: обрабатывать по 25 строк каждой рабочей единицей. При обработке меньшего количества строк получаем достаточно много рабочих единиц, которые часто конфликтуют при доступе в память, из-за чего время работы увеличивается. При дальнейшем увеличении количества строк, обрабатываемых одной рабочей единицей (30, 50 и т.д. вплоть до 400), получим время обработки изображений такое же, как и при обработке 25 строк. Иными словами, несмотря на то что происходит уменьшение количества параллельно выполняющихся рабочих единиц, а следовательно и конфликтов доступа в глобальную память, это преимущество нивелируется с увеличением вычислительной нагрузки на одну рабочую единицу. При дальнейшем увеличении числа обрабатываемых строк одной рабочей единицей время обработки изображений начнет ухудшаться, так как произойдет значительное преобладание вычислительной нагрузки на одну рабочую единицу над доступами в память.

Последнее описанное решение показывает, что без знания многих особенностей ПЛИС и основ схемотехники можно написать код, в целом достаточно близкий по своей логике к коду для традиционных вычислителей, и получить результаты, сопоставимые с результатами, получаемыми на GPU [11].

В первых трех реализациях узким местом оказался обмен данными с глобальной памятью (не единственным, но основным). Эта же проблема доступа в память существует и для классических фоннеймановских архитектур. Рассмотрим теперь принципиально иной подход.

Последнее решение — это так называемый метод скользящего окна (sliding window). Пример скользящего окна на рис. 4.

Предыдущие OpenCL-реализации фильтра Собеля работают, создавая сотни потоков, все работают параллельно, но в конечном итоге ограничены доступной глобальной пропускной способностью. Компилятор Altera OpenCL предлагает альтернативную модель программирования OpenCL, которая создает одно или несколько конвейерных ядер, где параллелизм исходит из сложности конвейера. Чем сложнее конвейер, тем больше операций выполняется параллельно. ПЛИС имеют значительные ресурсы локальной памяти, которые могут быть настроены различными способами: от больших одиночных буферов до сотен небольших буферов. Эта гибкость позволяет Altera OpenCL Compiler (АОС) создавать топологии памяти, специально разработанные для алгоритма, который требует ускорения. Следствием этого является значительное снижение требований к пропускной способности глобальной памяти в алгоритме. В фильтре Собеля доступ к глобальной памяти осуществляется линейно и не нуждается в оптимизации, однако для расчета каждого пикселя требуются данные из соседних точек решетки. Доставка данных может быть оптимизирована с использованием их кэшированной копии. В ПЛИС нет кэш-памяти, однако можно создавать свой кэш. Для его реализации мы и будем использовать метод скользящего окна (sliding

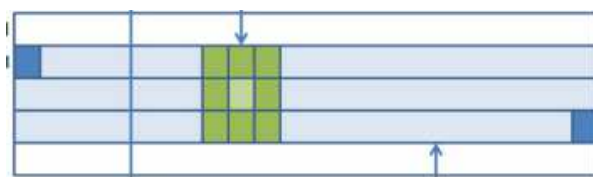


Рис. 4. Скользящее окно 3×3 , обрабатывающее 2-ю строку и использующее для этого элементы соседних строк

window). Вообще говоря, название не совсем точное, так как с точки зрения алгоритма окно 3×3 статично, а изображение проходит сквозь него. Это позволяет считывать данные линейно и буферизировать в локальной памяти, из которой данные могут считываться так часто, как требуется, а данные, которые нам уже не понадобятся, будут заменяться на новые. Таким образом, мы не будем хранить в памяти ничего лишнего, а весь алгоритм может быть конвейерным, чтобы генерировать результат за каждый такт. Более того, несколько конвейеров могут быть каскадированы для сбора с глобальным доступом к памяти, которые необходимы только для ввода в первый этап и выхода из заключительной стадии. Следовательно, количество пикселей, вычисляемых в секунду, линейно возрастает на один этап конвейера без увеличения требований к глобальной памяти.

Для хранения строки входной матрицы будем использовать буфер строк — массив размера одной строки. Если в данный момент обрабатываются элементы k -й строки, то в нижнем буфере строк находится $(k - 1)$ -я строка, в верхнем буфере строк k -я строка, а в трех дополнительных элементах соответствующие элементы $(k + 1)$ -й строки. После обработки данного элемента один элемент покидает наш кэш (последний из нижнего буфера строки), остальные сдвигаются на 1 и еще один новый элемент подгружается. Буферы строк, идущие через окно 3×3 , представлены на рис. 5. Попробуем написать последнее в двух вариантах:

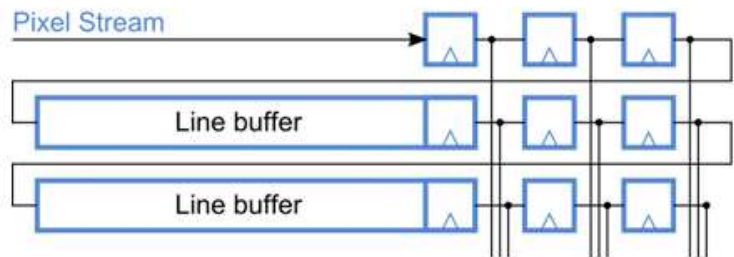


Рис. 5. Буферы строк, или linebuffer и окно 3×3

- работает одна-единственная рабочая единица, реализующая данный метод;
- работает несколько рабочих единиц, каждая из которых обрабатывает данным методом различные куски изображения.

Вариант реализации окна 3×3 на OpenCL представлен ниже. Массив *cache* как раз представляет кэш-память — 2 буфера строки и еще 3 элемента, цикл *for* осуществляет сдвиг на каждой итерации.

3.2.1. Скользящее окно на OpenCL.

```

...some works...
int cache[2*cols+3];
...some works...
while (count != iter) {
    #pragma unroll
    for (int i = 2*cols+2; i > 0; -i)
        cache[i] = cache[i-1];
    if (count >= 0) cache[0] = input_image[count];
    else cache[0] = 0;
    Sobel...
}

```

Для сравнения код на Verilog, реализующий функционал того же окна 3×3 , представлен далее.

3.2.2. Скользящее окно на Verilog.

```

module shift_fsm
    #(parameter Width = 8)
    (input clk, reset,
     output reg [7:0] x0, y0, z0,
     output reg [7:0] x1, y1, z1,
     output reg [7:0] x2, y2, z2)
always (posedge clk or negedge reset)
    if (!reset)
        begin

```

```

        x0 <= 8'd0; y0 <= 8'd0; z0 <= 8'd0;
        x1 <= 8'd0; y1 <= 8'd0; z1 <= 8'd0;
        x2 <= 8'd0; y2 <= 8'd0; z2 <= 8'd0;
    end
else
    begin
        x0 <= line1_data;
        y0 <= line2_data;
        z0 <= line3_data;
        x1 <= x0;
        y1 <= y0;
        z1 <= z0;
        x2 <= x1;
        y2 <= y1;
        z2 <= z1;
    end
endmodule

```

Сравнив эти две реализации, можно убедиться, насколько актуален вопрос поиска высокоуровневых решений для ПЛИС и, как частный случай, освоения инструментов, позволяющих программировать с использованием OpenCL.

Для оптимизации работы с памятью вместо классического *malloc* при выделении памяти для массива, который будет передаваться в ПЛИС, полезно использовать *alignedMalloc* для выравнивания памяти. Выравнивание ускоряет доступ к памяти за счет генерации кода, в котором на чтение и запись ячейки памяти требуется по одной инструкции. Выравнивание памяти на стороне хоста позволяет также передавать данные с хоста в ПЛИС и обратно, используя прямой доступ к памяти (DMA), а также повышает эффективность переноса буфера [6]. Реализовать сказанное можно, например, так.

3.2.3. Aligned Malloc для хоста, чтобы оптимизировать обмен данными с сопроцессором.

```

const unsigned AOCL_ALIGNMENT = 64;
...
void *alignedMalloc (size_t size) {
    return _aligned_malloc (size, AOCL_ALIGNMENT);
}
void alignedFree (void *ptr) {
    _aligned_free (ptr);
}

```

Возвращаясь к коду на Си и OpenCL: если запустим его на одной-единственной рабочей единице, то получим время обработки одного изображения, равное 5.21 секундам. Запустим на двух рабочих единицах — 4.87 секунд, на 8 — 4.94 секунд, на 16 — 10.74.

Основная проблема заключается в долгом обмене с памятью. Несмотря на то что был создан кэш, сдвиг в кэше осуществляется быстро, но добавление нового элемента долгое и происходит 1280*720 раз — по размеру массива.

Для ускорения этого процесса используем каналы (channel) OpenCL. В каналы можно записывать некоторые данные, и они будут там находиться, пока мы их оттуда не вытащим (либо завершится выполнение приложения). Эти каналы используются для обмена информацией между парой ядер, минуя глобальную память.

Теперь в приложении будет всего 3 ядра: Producer, Consumer и Sobel. Работать они будут в форме конвейера (наглядно представлено на рис. 6):

GlobalMem → *Producer* → *Sobel* → *Consumer* → *GlobalMem*

либо

GlobalMem → *Producer* → *Sobel1...k* → *Consumer* → *GlobalMem*

Код ядер Producer и Consumer представлен далее.

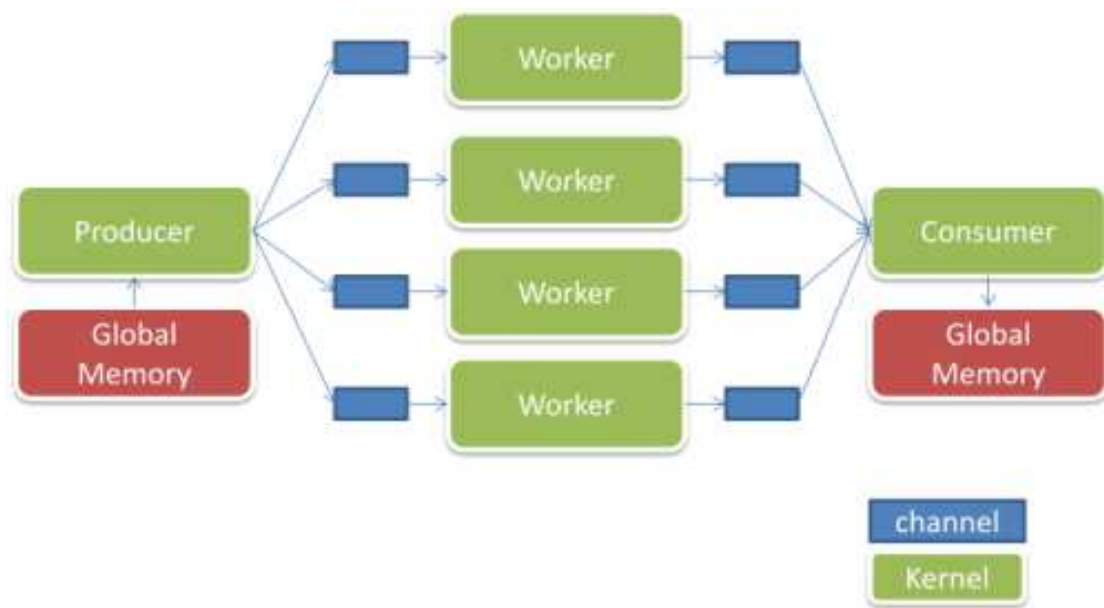


Рис. 6. Конвейер из ядер, работающих на ускорителе для оптимизации доступа в глобальную память

3.2.4. Код ядер **Producer** и **Consumer**.

```

__kernel void Producer (unsigned int __global
    * restrict in_img)
{
    int it = cols*rows/works;
    #pragma unroll works
    for (int k = 0; k < works; ++k)
        for (int i = 0; i < it; ++i)
            write_channel_altera (c1[k], in_img[i+k*it]);
}

-----

__kernel void Consumer (unsigned int __global
    * restrict out_img) {
    int it = cols*rows/works;
    #pragma unroll works
    for (int k = 0; k < works; ++k)
        for (int i = 0; i < it; ++i)
            out_img[i+k*it] = read_channel_altera (c2[k]);
}

```

Здесь опять же, несмотря на кажущийся вложенный цикл, все внутренние циклы будут выполняться параллельно (вернее, максимально параллельно в зависимости от значения `works` и доступных ресурсов).

Код ядра Sobel останется таким же за исключением того, что данные теперь будут поступать из `c1` и отправляться в канал `c2`.

3.2.5. Скользящее окно на OpenCL с использованием каналов OpenCL.

```

...some works...
int cashe[2*cols+3];
...some works...
while (count != iter) {
    #pragma unroll
    for (int i = 2*cols+2; i > 0; -i)
        cashe[i] = cashe[i-1];
}

```



```

    if (count >= 0)
        cashe[0] = read_channel_altera (c1[idx]);
    else
        cashe[0] = 0;
    ...some works...
    write_channel_altera (c2[idx], G)
}

```

Теперь три ядра работают одновременно, данные непрерывно поступают в канал, обрабатываются фильтром и отправляются в глобальную память. Лучшее время работы получается при использовании одного ядра Producer, двух ядер Sobel и одного ядра Consumer. Получим порядка 120 кадров в секунду (FPS). При использовании одного ядра Sobel ядро Producer обрабатывает быстрее и простаивает, а при использовании трех и более ядер Sobel ядро Producer не успевает заполнять все каналы и ядра Sobel не все работают параллельно.

Причины такого ускорения за счет конвейеризации доступа в память и вычислительных операций кроются в устройстве самого ПЛИС. Большинство ПЛИС используют для хранения конфигурации ячейки памяти статического ОЗУ, которые могут быть многократно перепрограммируемыми. Одно из главных преимуществ этой технологии — она позволяет достаточно легко и быстро реализовать и протестировать все новые идеи, относительно легко подстраиваясь под новые стандарты и протоколы. Однако принцип работы статического ОЗУ таков, что при случайных обращениях к той или иной ячейке результат вы получите только на следующем такте. Иными словами, в момент, когда вы заносите в ячейку памяти новое значение и из другой команды пытаетесь его считать, на текущем такте вы получите лишь старое, хранившееся там ранее значение, а новое получите только на следующем такте. Однако, если данные через эту ячейку пойдут в форме конвейера, то можно получать очередное значение на каждом такте. Подробнее об устройстве статического ОЗУ см. в [1] или [5].

Конвейер хорош еще и тем, что он сам для себя является памятью для хранения промежуточных результатов, а это приводит еще и к снижению нагрузки на пропускную способность памяти, и к уменьшению простоя функциональных устройств. Современные кристаллы ПЛИС позволяют строить конвейеры из сотен арифметических операций, что и обеспечивает беспрецедентный объем полезной работы в среднем за такт.

4. Заключение. Если алгоритм не содержит сложной логики и/или требует небольшого вычислительного ресурса, однако требует достаточно большой глобальной пропускной способности памяти, то наиболее оптимальный подход к программированию для ПЛИС для этого алгоритма следующий: в ядре выполняется конвейеризированное однопоточное приложение. Подобный подход уменьшает влияние на пропускную способность глобальной памяти, так как позволяет хранить ранее вычисленные строки в локальной памяти, устраняя необходимость постоянного обращения к глобальной памяти. При таком подходе осуществляется только одно чтение из глобальной памяти и одна запись в глобальную память на пиксель. На ПЛИС можно создавать много ядер для обработки, однако при внедрении большого количества ядер логика управления памятью будет занимать большое количество ресурсов на устройстве. Чтобы избежать копирования схемы памяти, следует использовать два дополнительных ядра, предназначенных для обработки доступа к глобальной памяти: один на чтение, другой на запись. Это предотвратит ненужное копирование логики работы с памятью и позволит реализовать больше параллельных путей: каждое ядро, реализующее алгоритм, получает данные из своего собственного канала и записывает их обратно в свой собственный канал.

Статья рекомендована к публикации Программным комитетом Международной научной конференции “Суперкомпьютерные дни в России 2018” (<http://russianscdays.org>).

СПИСОК ЛИТЕРАТУРЫ

1. *Maxfield C.* The design warrior’s guide to FPGAs. Devices, tools and flows. Burlington: Mentor Graphics Corporation and Xilinx, 2004.
2. *Cardoso J.M.P., Diniz P.C.* Compilation techniques for reconfigurable architectures. New York: Springer, 2009.
3. *Shi K., Boland D., Constantinides G.A.* Efficient FPGA implementation of digit Parallel online arithmetic operators // Proc. Int. Conf. on Field-Programmable Technology. New York: IEEE Press, 2014. doi 10.1109/FPT.2014.7082763.
4. *Thomas D.E., Moorby P.R.* The Verilog hardware description language. Dordrecht: Kluwer, 2002.
5. *Harris D.M., Harris S.L.* Digital design and computer architecture. Waltham: Morgan Kaufman, 2013.

6. Intel FPGA SDK for OpenCL Programming Guide. UG-OCL003 | 2018.05.04.
<https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407.html>.
7. Altera SDK for OpenCL. Best Practices Guide.
https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf
8. The OpenCL Specification. Khronos OpenCL Working Group. Version: 2.2-7.
https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf.
9. *Tompson J., Schlachter K.* An introduction to the OpenCL programming model. <https://hgpu.org/?p=7588>.
10. *Kilts S.* Advanced FPGA design: architecture, implementation, and optimization. New York: Wiley, 2007.
11. *Кондратьев А. А.* Параллельная обработка и кластеризация изображений на основе самоорганизующихся карт Кохонена с использованием кластерных и графических вычислителей.
<http://docplayer.ru/29195533-Parallelnaya-obrabotka-i-klasterizaciya-izobrazheniy-na-osnove-samoorganizuyushchih-sya-kart-kohonena-s-ispolzovaniem-klasternyh-i-graficheskikh-vychisliteley.html>.
12. *Sanny A., Prasanna V.K.* Energy-efficient median filter on FPGA // Proc. 2013 International Conference on Reconfigurable Computing and FPGAs. New York: IEEE Press, 2013. doi 10.1109/ReConFig.2013.6732283.

Поступила в редакцию
22.08.2018

A Study of Implementation Methods for the Sobel Filter on Reconfigurable Computing Systems

E. E. Taratuta ¹

¹ *Lomonosov Moscow State University, Faculty of Computational Mathematics and Cybernetics;
Leninskie Gory, Moscow, 119991, Russia; Graduate Student, e-mail: taratutakatya92@gmail.com*

Received August 22, 2018

Abstract: The main motivation is the growing popularity of reconfigurable FPGAs in the field of high-performance computing as well as the need to study the features of writing efficient software using the Altera SDK for OpenCL tool, which allows one to implement applications for FPGA using the C++ programming language and the OpenCL standard. The paper discusses the features of the OpenCL software model and presents a study of the performance and features of various Sobel filter implementations using the mentioned tool.

Keywords: FPGA, OpenCL, Sobel filter, Altera OpenCL SDK.

References

1. C. Maxfield, *The Design Warrior's Guide to FPGAs. Devices, Tools and Flows* (Mentor Graphics Corporation and Xilinx, Burlington, 2004).
2. J. M. P. Cardoso and P. C. Diniz, *Compilation Techniques for Reconfigurable Architectures* (Springer, New York, 2009).
3. K. Shi, D. Boland, and G. A. Constantinides, "Efficient FPGA Implementation of Digit Parallel Online Arithmetic Operators," in *Proc. Int. Conf. on Field-Programmable Technology, Shanghai, China, December 10–12, 2014* (IEEE Press, New York, 2014), doi 10.1109/FPT.2014.7082763
4. D. E. Thomas and P. R. Moorby, *The Verilog Hardware Description Language* (Kluwer, Dordrecht, 2002).
5. D. M. Harris and S. L. Harris, *Digital Design and Computer Architecture* (Morgan Kaufman, Waltham, 2013).
6. Intel FPGA SDK for OpenCL Programming Guide. UG-OCL003 | 2018.05.04.
<https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407.html>. Cited September 10, 2018.
7. Altera SDK for OpenCL. Best Practices Guide.
https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf. Cited September 10, 2018.
8. The OpenCL Specification. Khronos OpenCL Working Group. Version: 2.2-7.
https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf. Cited September 10, 2018.
9. J. Tompson and K. Schlachter, "An Introduction to the OpenCL Programming Model," <https://hgpu.org/?p=7588>. Cited September 10, 2018.

10. S. Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization* (Wiley, New York, 2007).
11. A. A. Kondratyev, "The Parallel Image Processing and Clustering Based on Kohonen Maps by Using Clusters and Graphics Processing Units," <http://docplayer.ru/29195533-Parallelnaya-obrabotka-i-klasterizaciya-izobrazheniy-na-osnove-samoorganizuyushchihsya-kart-kohonena-s-ispolzovaniem-klasternyh-i-graficheskikh-vychisliteley.html>. Cited September 10, 2018.
12. A. Sanny and V. K. Prasanna, "Energy-Efficient Median Filter on FPGA," in *Proc. 2013 International Conference on Reconfigurable Computing and FPGAs, Cancun, Mexico, December 9–11, 2013* (IEEE Press, New York, 2013), doi 10.1109/ReConFig.2013.6732283