Оглавление

[Введение. 3](#_Toc453161530)

[Постановка задачи. 4](#_Toc453161531)

[Теоретическая часть. 5](#_Toc453161532)

[Программируемая пользователем вентильная матрица (FPGA) 5](#_Toc453161533)

[GPGPU 5](#_Toc453161534)

[OpenCL. 6](#_Toc453161535)

[Определение. 6](#_Toc453161536)

[Особенности языка OpenCL. 6](#_Toc453161537)

[История появления OpenCL. 7](#_Toc453161538)

[Подробности технологии. 9](#_Toc453161539)

[Медианная фильтрация 16](#_Toc453161540)

[Практическая часть. 20](#_Toc453161541)

[Описание алгоритма фильтрации изображения 20](#_Toc453161542)

[Приведение результатов работы программы 23](#_Toc453161543)

[Выводы. 25](#_Toc453161544)

[Список литературы. 26](#_Toc453161545)

# Введение.

# Постановка задачи.

# Теоретическая часть.

## Программируемая пользователем вентильная матрица (FPGA)

Программируемая пользователем вентильная матрица (ППВМ, англ. Field-Programmable Gate Array, FPGA) - полупроводниковое устройство, которое может быть сконфигурировано производителем или разработчиком после изготовления; отсюда название: «программируемая пользователем». ППВМ программируются путём изменения логики работы принципиальной схемы, например с помощью исходного кода на языке проектирования (типа VHDL), на котором можно описать эту логику работы микросхемы. ППВМ является одной из архитектурных разновидностей программируемых логических интегральных схем (ПЛИС).

ППВМ могут быть модифицированы практически в любой момент в процессе их использования. Они состоят из конфигурируемых логических блоков, подобных переключателям с множеством входов и одним выходом (логические вентили или gates). В цифровых схемах такие переключатели реализуют базовые двоичные операции AND, NAND, OR, NOR и XOR. В большинстве современных микропроцессоров функции логических блоков фиксированы и не могут модифицироваться. Принципиальное отличие ППВМ состоит в том, что и функции блоков, и конфигурация соединений между ними могут меняться с помощью специальных сигналов, посылаемых схеме. В некоторых специализированных интегральных схемах (ASIC) используются логические матрицы, аналогичные ППВМ по структуре, однако они конфигурируются один раз в процессе производства, в то время как ППВМ могут постоянно перепрограммироваться и менять топологию соединений в процессе использования. Однако такая гибкость требует существенного увеличения количества транзисторов микросхемы.

## GPGPU

GPGPU (также GPGP, GP²U, англ. General-purpose computing for graphics processing units, неспециализированные вычисления на графических процессорах) -техника использования графического процессора видеокарты, который обычно имеет дело с вычислениями только для компьютерной графики, чтобы выполнять расчёты в приложениях для общих вычислений, которые обычно проводит центральный процессор. Это стало возможным благодаря добавлению программируемых шейдерных блоков и более высокой арифметической точности растровых конвейеров, что позволяет разработчикам ПО использовать потоковые процессоры для не-графических данных.

## OpenCL.

### Определение.

OpenCL (от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Open Computing Language - открытый язык вычислений) - [фреймворк](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA) для написания [компьютерных программ](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0), связанных с [параллельными вычислениями](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) на различных [графических](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) GPU) и [центральных процессорах](https://ru.wikipedia.org/wiki/%D0%A6%D0%B5%D0%BD%D1%82%D1%80%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) CPU), а также [FPGA](https://ru.wikipedia.org/wiki/FPGA). Вo фреймворк OpenCL входят [язык программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), который базируется на стандарте [C99](https://ru.wikipedia.org/wiki/C99), и [интерфейс программирования приложений](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) API). OpenCL обеспечивает параллелизм на уровне инструкций и на уровне данных и является реализацией техники [GPGPU](https://ru.wikipedia.org/wiki/GPGPU). OpenCL является полностью [открытым стандартом](https://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BA%D1%80%D1%8B%D1%82%D1%8B%D0%B9_%D1%81%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82), его использование не облагается лицензионными отчислениями.

Цель OpenCL состоит в том, чтобы дополнить [OpenGL](https://ru.wikipedia.org/wiki/OpenGL" \o "OpenGL) и [OpenAL](https://ru.wikipedia.org/wiki/OpenAL" \o "OpenAL), которые являются открытыми отраслевыми стандартами для [трёхмерной компьютерной графики](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D1%91%D1%85%D0%BC%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D0%BA%D0%B0" \o "Трёхмерная графика) и звука, пользуясь возможностями [GPU](https://ru.wikipedia.org/wiki/GPU). OpenCL разрабатывается и поддерживается некоммерческим консорциумом [Khronos Group](https://ru.wikipedia.org/wiki/Khronos_Group" \o "Khronos Group), в который входят много крупных компаний, включая [AMD](https://ru.wikipedia.org/wiki/AMD" \o "AMD), [Apple](https://ru.wikipedia.org/wiki/Apple), [ARM](https://ru.wikipedia.org/wiki/ARM_%28%D0%BA%D0%BE%D0%BC%D0%BF%D0%B0%D0%BD%D0%B8%D1%8F%29), [Intel](https://ru.wikipedia.org/wiki/Intel), [Nvidia](https://ru.wikipedia.org/wiki/Nvidia), [Sony Computer Entertainment](https://ru.wikipedia.org/wiki/Sony_Computer_Entertainment), [Sun Microsystems](https://ru.wikipedia.org/wiki/Sun_Microsystems) и другие.

### Особенности языка OpenCL.

Ключевыми отличиями используемого языка от Си (стандарт ISO 1999 года) являются:

* Отсутствие поддержки указателей на [функции](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29), [рекурсии](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BA%D1%83%D1%80%D1%81%D0%B8%D1%8F), [битовых полей](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82%D0%BE%D0%B2%D0%BE%D0%B5_%D0%BF%D0%BE%D0%BB%D0%B5), массивов переменной длины (VLA), стандартных заголовочных файлов
* Расширения языка для параллелизма: векторные типы, синхронизация, функции для Work-items/Work-Groups
* Квалификаторы типов памяти: \_\_global, \_\_local, \_\_constant, \_\_private
* Иной набор встроенных функций

OpenCL новый стандарт для разработки приложений для гетерогенных систем. Изначально OpenCL задумывался как нечто большее: единый стандарт для написания приложений, которые должны исполняться в системе, где установлены различные по архитектуре процессоры, ускорители и платы расширения.

### История появления OpenCL.

Гетерогенные вычислительные системы в основном используются для высокопроизводительных вычислений, таких как моделирование физических процессов, кодирование видео и т.д. Ранее подобные задачи решались с помощью суперкомпьютера либо очень мощной настольной системы. С появлением технологий NVidia CUDA/AMD Stream стало возможным относительно просто писать программы, использующие вычислительные возможности GPU.

Подобные программы создавались и раньше, но именно NVidia CUDA обеспечила рост популярности GPGPU за счет облегчения процесса создания приложений, выполняемых графическим процессором. Первые GPGPU приложения в качестве ядер (kernel в CUDA и OpenCL) использовали шейдеры, а данные запаковывались в текстуры. Таким образом, необходимо было хорошо разбираться в OpenGL или DirectX. Чуть позже появился язык Brook, который упрощал процесс написание программы (на основе этого языка создавалась AMD Stream (в ней используется Brook+)).

Компьютер, на котором производятся вычисления, на аппаратном уровне может иметь процессоры x86, x86-64, Itanium, SpursEngine (Cell), NVidia GPU, AMD GPU, VIA (S3 Graphics) GPU и т.д. Для каждого из этих типов процессов существует свой SDK, свой язык программирования и программная модель. То есть если нужно, чтобы программа расчета какого-либо физического процесса работала на простой рабочей станции, суперкомпьютере BlueGene, или компьютере, оборудованном двумя ускорителями NVidia Tesla - будет необходимо переписывать достаточно большую часть программы, так как каждая из платформ в силу своей архитектуры имеет набор жестких ограничений. Написание одной и той же программы под разные платформы является очень дорогой и трудоемкой задачей.

Для решения этой проблемы было решено создать некий единый стандарт для программ, исполняющихся в гетерогенной среде. Это означает, что программа, вообще говоря, должна быть способна исполняться на компьютере, в котором установлены одновременно GPU NVidia и AMD, Toshiba SpursEngine и т.д.

Для разработки открытого стандарта решено было привлечь Khronos Group, которые разрабатывали такие стандарты как OpenGL и OpenML. В разработке и финансировании так же участвовали AMD, IBM, Activision Blizzard, Intel, NVidia и т.д.

Компания NVidia не афишировала свое участие в проекте, и быстрыми темпами наращивала функциональность и производительность CUDA. В то время как несколько ведущих инженеров NVidia участвовали в создании OpenCL. Вероятно, именно участие NVidia в большой мере определило синтаксическую и идеологическую схожесть OpenCL и CUDA, что облегчает для программиста переход от одного языка к другому.

Первая версия стандарта была опубликована в конце 2008 года.

Драйвер для OpenCL был выпущен NVidia и прошел проверку на совместимость со стандартом, но доступен только для ограниченного круга людей - зарегистрированных разработчиков.

Реализация OpenCL для NVidia была достаточно легкой задачей, так как основные идеи сходны. CUDA и OpenCL - некоторое расширение языка С, со сходным синтаксисом, используют одинаковую программную модель в качестве основной - Data Parallel (SIMD). Так же OpenCL поддерживает Task Parallel programming model - модель, когда одновременно могут выполняться различные kernel (work-group содержит один элемент). О схожести двух технологий говорит так же то, что NVidia опубликовала специальный документ, в котором описано как писать для CUDA так, чтобы потом легко перейти на OpenCL.

Основной проблемой реализации OpenCL от NVidia является низкая производительность по сравнению с CUDA, но с каждым новым релизом драйверов производительность OpenCL под управлением CUDA все ближе подбирается к производительности CUDA приложений

Летом 2009 года компания AMD сделала заявление о поддержке и соответствии стандарту OpenCL в новой версии Stream SDK. На деле же оказалось, что поддержка была реализована только для CPU. OpenCL стандарт для гетерогенных систем и ничего не мешает запустить kernel на CPU, более того - это очень удобно в случае если в системе нет другого OpenCL устройства. В таком случае программа будет продолжать работать, только медленнее. Или же можно задействовать все вычислительные мощности, которые есть в компьютере - как GPU так и CPU, хотя на практике это не имеет особого смысла, так как время исполнения kernel’ов которые исполняются на CPU будет намного больше тех что исполняются на GPU. Зато для отладки приложений это очень удобно.

Поддержка OpenCL для графических адаптеров AMD так же не заставила себя долго ждать - по последним сообщениям компании версия для графических чипов сейчас доступна всем желающим.

Так как OpenCL должен работать поверх некоторой специфической для железа оболочки, а значит для того чтобы можно этот стандарт действительно стал единым для различных гетерогенных систем – необходимо чтобы соответствующие оболочки (драйверы) были выпущены и для IBM Cell и для Intel Larrabie. Драйверов под эти процессоры пока нет. Таким образом OpenCL остается еще одним средством разработки для GPU на ряду с CUDA, Stream и DirectX Compute.

Apple также заявляет о поддержке OpenCL, которая обеспечивается за счет NVidia CUDA.

Также в настоящее время сторонними разработчиками предлагается:

* OpenTK - библиотека-обертка над OpenGL, OpenAL и OpenCL для .Net.
* PyOpenCL - обертка над OpenCL для Pyton.
* Java обертка для OpenCL.

### Подробности технологии.

OpenCL задумывался как технология для создания приложений, которые могли бы исполняться в гетерогенной среде. Более того, он разработан так, чтобы обеспечивать комфортную работу с такими устройствами, которые сейчас находятся только в планах и даже с теми, которые еще никто не придумал. Для координации работы всех этих устройств гетерогенной системе всегда есть одно «главное» устройство, который взаимодействует со всеми остальным посредствами OpenCL API. Такое устройство называется «хост», он определяется вне OpenCL.

Поэтому OpenCL исходит из наиболее общих предпосылок, дающих представление об устройстве с поддержкой OpenCL: так как это устройство предполагается использовать для вычислений – в нем есть некий «процессор» в общем смысле этого слова. Нечто, что может исполнять команды. Так как OpenCL создан для параллельных вычислений, то такой процессор может, иметь средства параллелизма внутри себя (например, несколько ядер одного CPU, несколько SPE процессоров в Cell). Также элементарным способом наращивания производительности параллельных вычислений является установка нескольких таких процессоров на устройстве (к примеру, многопроцессорные материнские платы PC итд.). И естественно в гетерогенной системе может быть несколько таких OpenCL-устройств (вообще говоря, с различной архитектурой).

Кроме вычислительных ресурсов устройство имеет какой-то объем памяти. Причем никаких требований к этой памяти не предъявляется, она может быть как на устройстве, так и вообще быть размечена на ОЗУ хоста (как например, это сделано у встроенных видеокарт).

Собственно все. Больше об устройстве никаких предположений не делается.

Такое широкое понятие об устройстве позволяет не накладывать каких-либо ограничений на программы, разработанные для OpenCL. Эта технология позволит Вам разрабатывать как приложения, сильно оптимизированные под конкретную архитектуру специфического устройства, поддерживающего OpenCL, так и те, которые будут демонстрировать стабильную производительность на всех типах устройств (при условии эквивалентной производительности этих устройств).

OpenCL предоставляет программисту низкоуровневый API, через который он взаимодействует с ресурсами устройства. OpenCL API может либо напрямую поддерживаться устройством, либо работать через промежуточный API (как в случае NVidia: OpenCL работает поверх CUDA Driver API, поддерживаемый устройствами), это зависит от конкретной реализации не описывается стандартом.

Рассмотрим как же OpenCL обеспечивает такую универсальность, сохраняя при этом низкоуровневую природу.

Далее я приведу вольный перевод части спецификации OpenCL 1.0 с некоторыми комментариями и дополнениями.

Для описания основных идей OpenCL воспользуемся иерархией из 4х моделей:

* Модель платформы (Platform Model);
* Модель памяти (Memory Model);
* Модель исполнения (Execution Model);
* Программная модель (Programming Model);

#### Модель платформы (Platform Model).

Платформа OpenCL состоит из хоста соединенного с устройствами, поддерживающими OpenCL. Каждое OpenCL-устройство состоит из вычислительных блоков (Compute Unit), которые далее разделяются на один или более элементы-обработчики (Processing Elements, далее PE).

OpenCL-приложение исполняется на хосте в соответствии с нативными моделями его платформы. OpenCL-приложение отправляет с хоста команды устройствам на выполнение вычислений на PE. PE в рамках вычислительного блока выполняют один поток команд как SIMD блоки (одна инструкция выполняется всеми одновременно, обработка следующей инструкции не начнется, пока все PE не завершат исполнение текущей инструкции), либо как SPMD блоки (у каждого PE собственный счетчик инструкций (program counter)).

То есть OpenCL обрабатывает некие команды, поступающие от хоста. Таким образом приложение не связано жестко с OpenCL, а значит всегда можно подменить реализацию OpenCL, не нарушив работоспособность программы. Даже если будет создано такое устройство, которое не укладывается в модель «OpenCL-устройства», для него можно будет создать реализацию OpenCL, транслирующую команды хоста в более удобный для устройства вид.

#### Модель исполнения (Execution Model).

Выполение OpenCL-программы состоит из двух частей: хостовая часть программы и kernels (ядра; с Вашего позволения я далее буду употреблять английский термин, как более привычный большинству из нас) исполняющиеся на OpenCL-устройстве. Хостовая часть программы определяет контекст, в котором исполняются kernel'ы, и управляет их исполнением.

Основная часть модели исполнения OpenCL описывает исполнение kernel’ов. Когда kernel ставится в очередь на исполнение, определяется пространство индексов (NDRange, определение будет дано ниже). Копия (instanse) kernel'а выполнятся для каждого индекса из этого пространства. Копия kernel'а выполняющаяся для конкретного индекса называется «Work-Item» (рабочей единицей) и определяется точкой в пространстве индексов, то есть каждой «единице» предоставляется глобальный ID. Каждый Work-Item выполняет один и тот же код, но конкретный путь исполнения (ветвления итп.) и данные, с которыми он работает, могут быть различными.

Work-Item'ы организуются в группы (Work-Groups). Группы предоставляют более крупное разбиение в пространстве индексов. Каждой группе приписывается групповой ID с такой же размерностью, которая использовалась для адресации отдельных элементов. Каждому элементу сопоставляется уникальный, в рамках группы, локальный ID. Таким образом, Work-Item'ы могут быть адресованы как по глобальному ID, так и по комбинации группового и локального ID.

Work-Item'ы в группе исполняются конкурентно (параллельно) на PE одного вычислительного блока.

Здесь хорошо видна унифицированная модель устройства: несколько PE -> CU, несколько CU -> устройство, несколько устройств -> гетерогенная система.

Пространство индексов в OpenCL 1.0 называется NDRange и может быть 1-, 2- и 3-мерным. NDRange – массив целых чисел (integer) длины N, указывающий размерность в каждом из направлений.

Выбор размерности NDRange определяется удобством для конкретного алгоритма: в случае работы с трехмерными моделями удобно индексировать по трехмерным координатам, в случае работы с изображениями или двумерными сетками – удобнее, когда размерность индексов – 2. 4х-мерные объекты в нашем мире большая редкость, поэтому размерность ограничена 3. Кроме того, как бы там ни было, но в данный момент основная цель OpenCL – это GPU. GPU Nvidia сейчас нативно поддерживают размерность индексов до 3, соответственно, чтобы реализовать большую размерность, пришлось бы прибегать к хитростям и усложнению либо CUDA Driver API, либо реализации OpenCL.

##### Контекст исполнения и очереди команд в модели исполнения OpenCL.

Хост определяет контекст исполнения kernel'ов. Контекст включает в себя следующие ресурсы:

* **Устройства:** набор OpenCL-устройств, которые использует хост.
* **Kernel'ы:** OpenCL функции, которые исполняются на устройствах.
* **Объекты программ (Program Objects):** исходные коды и исполняемые файлы kernel’ов.
* **Объекты памяти (Memory Objects):** набор объектов в памяти, видимых как хосту, так и OpenCL устройству. Объекты памяти содержат значения, с которыми могут работать kernel'ы.

Контекст создается и управляется по средствам функций из API OpenCL. Хост создает структуру данных, называемую «очередь команд» (command-queue), чтобы управлять исполнением kernel’ов на устройствах. Хост отправляет команды в очередь, после чего они устанавливаются планировщиком для исполнения на устройствах в нужном контексте.

Команды могут быть следующих типов:

* **Команда исполнения ядра:** исполнить ядро на PEs устройства.
* **Команды памяти:** переместить данные в объекты памяти, из них или между ними.
* **Команды синхронизации:** управление порядком исполнения команд.

Очередь команд планирует команды для исполнения на устройстве. Они исполняются асинхронно между хостом и устройством. Команды могут исполняться друг относительно друга двумя способами:

* **Исполнение по порядку:** команды запускаются на исполнение в том порядке, в котором они расположены в очереди и завершаются так же по порядку. То есть команды выполняются последовательно.
* **Непоследовательное исполнение:** команды отправляются на исполнение по порядку, но не ждут завершения предыдущей команды перед началом исполнения. В этом случае программист должен явно использовать команды синхронизации.

С одним контекстом можно связать несколько очередей команд. Эти очереди исполняются, конкурируя между собой, и независимо без каких-либо явных способов синхронизации между ними.

Использование очереди команд, позволяет добиться большой универсальности и гибкости при использовании OpenCL. Современные GPU имеют собственный планировщик, который решает, что и когда и на каких вычислительных блоках исполнять. Использование очереди не стесняет работу планировщика, который имеет собственную очередь команд.

##### Модель исполнения: категории kernel.

В OpenCL kernel может быть двух категорий:

* **OpenCL kernel:** написаны на OpenCL C и компилируется компилятором OpenCL. Все реализации OpenCL должны поддерживать OpenCl-kernel. Реализации могут предоставлять другие механизмы создания OpenCL-kernel.
* **Naitive kernel:** доступ к ним осуществляется через хостовые указатели на функцию. Нативные kernel ставится в очередь на исполнение, так же как и OpenCL-kernel и использует те же объекты памяти, что и OpenCL-kernel. К примеру, такие kernel'ы могут быть функциями, определенными в коде приложения или экспортированными из библиотеки. Отметим, что возможность исполнять нативные kernel'ы является опциональной и их семантика не определяется стандартом. API OpenCL включает функции для опроса возможностей устройства на предмет поддержки таких kernel'ов.

#### Модель памяти (Memory Model).

Work-Item, исполняющий kernel может использовать четыре различных типа памяти:

* **Глобальная память.** Эта память предоставляет доступ на чтение и запись элементам всех групп. Каждый Work-Item может писать и читать из любой части объекта памяти. Запись и чтение глобальной памяти может кэшироваться в зависимости от возможностей устройства.
* **Константная память.** Область глобальной памяти, которая остается постоянной во время исполнения kernel'а. Хост аллоцирует и инициализирует объекты памяти, расположенные в константной памяти.
* **Локальная память.** Область памяти, локальная для группы. Эта область памяти может использоваться, чтобы создавать переменные, разделяемые всей группой. Она может быть реализована как отдельная память на OpenCL-устройстве. Альтернативно эта память может быть размечена как область в глобальной памяти.
* **Частная (private) память.** Область памяти, принадлежащая Work-Item. Переменные, определенные в частной памяти одного Work-Item’а, не видны другим.

Спецификация определяет 4 типа памяти, но снова не накладывает никаких требований на реализацию памяти в железе. Все 4 типа памяти могут находиться в глобальной памяти, и разделение типов может осуществляться на уровне драйвера и напротив, может существовать жесткое разделение типов памяти, продиктованное архитектурой устройства.

Существование именно этих типов памяти достаточно логично: у процессорного ядра есть свой кэш, у процессора есть общий кэш и у всего устройства есть некоторый объем памяти.

#### Программная модель. (Programming Model)

Модель исполнения OpenCL поддерживает две программные модели: параллелизм данных (Data Parallel) и параллелизм заданий (Task Parallel), так же поддерживаются гибридные модели. Основная модель, определяющая дизайн OpenCL, – параллелизм данных.

##### Программная модель с параллелизмом данных.

Эта модель определяет вычисления как последовательность инструкций, применяемых к множеству элементов объекта памяти. Пространство индексов, ассоциированное с моделью исполнения OpenCL, определяет Work-Item'ы и как данные распределяются между ними. В строгой модели параллелизма данных существует строгое соответствие один к одному между Work-Item и элементом в объекте памяти, с которым kernel может работать параллельно. OpenCL реализует более мягкую модель параллелизма данных, где строгое соответствие один к одному не требуется.

OpenCL предоставляет иерархическую модель параллелизма данных. Существует два способа определить иерархическое деление. В явной модели программист определяет общее число элементов, которые должны исполняться параллельно и так же каким образом эти элементы будут распределены по группам. В неявной модели программист только определяет общее число элементов, которые должны исполняться параллельно, а разделение по рабочим группам выполняется автоматически.

##### Программная модель с параллелизмом заданий.

В этой модели каждая копия kernel'а исполняется независимо от какого-либо пространства индексов. Логически это эквивалентно исполнению kernel'а на вычислительном блоке (CU) с группой, состоящей из одного элемента. В такой модели пользователи выражают параллелизм следующими способами:

* используют векторные типы данных, реализованные в устройстве;
* устанавливают в очередь множество заданий;
* устанавливают в очередь нативные kernel'ы, использующие программную модель, ортогональную к OpenCL;

Существование двух моделей программирования – также дань универсальности. Для современных GPU и Cell хорошо подходит первая модель. Но не все алгоритмы можно эффективно реализовать в рамках такой модели, а так же есть вероятность появления устройства, архитектура которого будет неудобна для использования первой модели. В таком случае вторая модель позволяет писать специфичные для другой архитектуры приложения.

#### Из чего состоит платформа OpenCL

Платформа OpenCL позволяет приложениям использовать хост и одно или несколько OpenCL-устройств как одну гетерогенную параллельную компьютерную систему. Платформа состоит из следующих компонент:

* **OpenCL Platform Layer:** позволяет хосту обнаруживать OpenCL-устройства, опрашивать их свойства и создавать контекст.
* **OpenCL Runtime:** среда исполнения позволяет программе на хосте управлять контекстами после того как они были созданы.
* **Компилятор OpenCL:** компилятор OpenCL создает исполняемые файлы, содержащие OpenCL–kernel. Язык программирования OpenCL-C реализуется компилятором, который поддерживает подмножество стандарта языка ISO C99 с расширениями для параллелизма.

В итоге модель OpenCL получилась весьма универсальной, при этом она остается низкоуровневой, позволяя оптимизировать приложения под конкретную архитектуру. Так же она обеспечивает кроссплатформенность при переходе от одного типа OpenCL-устройств к другому. Поставщик реализации OpenCL имеет возможность всячески оптимизировать взаимодействие своего устройства с OpenCL API, добиваясь повышения эффективности распределения ресурсов устройства. Кроме того, правильно написанное OpenCL приложение будет оставаться эффективным при смене поколений устройств.

## Медианная фильтрация

Все линейные алгоритмы фильтрации приводят к сглаживанию резких перепадов яркости изображений, прошедших обработку. Этот недостаток, особенно существенный, если потребителем информации является человек, принципиально не может быть исключен в рамках линейной обработки. Дело в том, что линейные процедуры являются оптимальными при гауссовском распределении сигналов, помех и наблюдаемых данных. Реальные изображения, строго говоря, не подчиняются данному распределению вероятностей. Причем, одна из основных причин этого состоит в наличии у изображений разнообразных границ, перепадов яркости, переходов от одной текстуры к другой и т. п. Поддаваясь локальному гауссовскому описанию в пределах ограниченных участков, многие реальные изображения в этой связи плохо представляются как глобально гауссовские объекты. Именно это и служит причиной плохой передачи границ при линейной фильтрации.

Вторая особенность линейной фильтрации - ее оптимальность, как только что упоминалось, при гауссовском характере помех. Обычно этому условию отвечают шумовые помехи на изображениях, поэтому при их подавлении линейные алгоритмы имеют высокие показатели. Однако, часто приходится иметь дело с изображениями, искаженными помехами других типов. Одной из них является импульсная помеха. При ее воздействии на изображении наблюдаются белые или (и) черные точки, хаотически разбросанные по кадру. Применение линейной фильтрации в этом случае неэффективно - каждый из входных импульсов (по сути - дельта-функция) дает отклик в виде импульсной характеристики фильтра, а их совокупность способствует распространению помехи на всю площадь кадра.

Удачным решением перечисленных проблем является применение медианной фильтрации, предложенной Дж. Тьюки в 1971 г. для анализа экономических процессов. Отметим, что медианная фильтрация представляет собой эвристический метод обработки, ее алгоритм не является математическим решением строго сформулированной задачи. Поэтому исследователями уделяется большое внимание анализу эффективности обработки изображений на ее основе и сопоставлению с другими методами.

При применении медианного фильтра (МФ) происходит последовательная обработка каждой точки кадра, в результате чего образуется последовательность оценок. В идейном отношении обработка в различных точках независима (этим МФ похож на масочный фильтр), но в целях ее ускорения целесообразно алгоритмически на каждом шаге использовать ранее выполненные вычисления.

При медианной фильтрации используется двумерное окно (апертура фильтра), обычно имеющее центральную симметрию, при этом его центр располагается в текущей точке фильтрации. На рис. 1 показаны два примера наиболее часто применяемых вариантов окон в виде креста и в виде квадрата. Размеры апертуры принадлежат к числу параметров, оптимизируемых в процессе анализа эффективности алгоритма. Отсчеты изображения, оказавшиеся в пределах окна, образуют рабочую выборку текущего шага.

|  |  |
| --- | --- |
| http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image001.gif | http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image002.gif |
| а) Окно в виде креста | б) Окно в виде квадрата |
| Рис. 1. Примеры окон при медианной фильтрации | |

Двумерный характер окна позволяет выполнять, по существу, двумерную фильтрацию, поскольку для образования оценки привлекаются данные как из текущих строки и столбца, так и из соседних. Обозначим рабочую выборку в виде одномерного массива http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image003.gif; число его элементов равняется размеру окна, а их расположение произвольно. Обычно применяют окна с нечетным числом точек http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image004.gif (это автоматически обеспечивается при центральной симметрии апертуры и при вхождении самой центральной точки в ее состав). Если упорядочить последовательность http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image005.gif по возрастанию, то ее медианой будет тот элемент выборки, который занимает центральное положение в этой упорядоченной последовательности. Полученное таким образом число и является продуктом фильтрации для текущей точки кадра. Понятно, что результат такой обработки в самом деле не зависит от того, в какой последовательности представлены элементы изображения в рабочей выборке http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image006.gif. Введем формальное обозначение описанной процедуры в виде:

http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image007.gif. (1)

Рассмотрим пример. Предположим, что выборка имеет вид: http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image008.gif, а элемент 250, расположенный в ее центре, соответствует текущей точке фильтрации http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image009.gif (рис. 1). Большое значение яркости в этой точке кадра может быть результатом воздействия импульсной (точечной) помехи. Упорядоченная по возрастанию выборка имеет при этом вид http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image010.gif{45,55,75,99,104,110,136,158,250}, следовательно, в соответствии с процедурой (1), получаем http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image011.gif. Видим, что влияние “соседей” на результат фильтрации в текущей точке привело к “игнорированию” импульсного выброса яркости, что следует рассматривать как эффект фильтрации. Если импульсная помеха не является точечной, а покрывает некоторую локальную область, то она также может быть подавлена. Это произойдет, если размер этой локальной области будет меньше, чем половина размера апертуры МФ. Поэтому для подавления импульсных помех, поражающих локальные участки изображения, следует увеличивать размеры апертуры МФ.

Из (1) следует, что действие МФ состоит в “игнорировании” экстремальных значений входной выборки - как положительных, так и отрицательных выбросов. Такой принцип подавления помехи может быть применен и для ослабления шума на изображении. Однако исследование подавления шума при помощи медианной фильтрации показывает, что ее эффективность при решении этой задачи ниже, чем у линейной фильтрации.

Результаты экспериментов, иллюстрирующие работу МФ, приведены на рис. 3.11. В экспериментах применялся МФ, имеющий квадратную апертуру со стороной равной 3. В левом ряду представлены изображения, искаженные помехой, в правом - результаты их медианной фильтрации. На рис. 3.11.а и рис. 3.11.в показано исходное изображение, искаженное импульсной помехой. При ее наложении использовался датчик случайных чисел с равномерным на интервале [0, 1] законом распределения, вырабатывающий во всех точках кадра независимые случайные числа. Интенсивность помехи задавалась вероятностью http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image012.gif ее возникновения в каждой точке. Если для случайного числа http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image013.gif, сформированного в точке http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image009.gif, выполнялось условие http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image014.gif, то яркость изображения http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image010.gifhttp://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image015.gif в этой точке замещалась числом 255, соответствующим максимальной яркости (уровню белого). На рис. 3.11.а действием импульсной помехи искажено 5 % (http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image012.gif=0.05), а на рис. 3.11.в - 10 % элементов изображения. Результаты обработки говорят о практически полном подавлении помехи в первом случае и о ее значительном ослаблении во втором.

// TODO добавить результаты фильтрации

Рис. 3.11.д показывает изображение, искаженное независимым гауссовским шумом при отношении сигнал/шум http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image022.gifдБ, а рис. 3.11.е - результат его фильтрации медианным фильтром. Условия данного эксперимента позволяют сравнивать его результаты с результатами рассмотренной выше линейной фильтрации. В таблице 3.1 приведены данные, дающие возможность такого сравнения. Для различных методов фильтрации в этой таблице приводятся значения относительного среднего квадрата ошибок http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image023.gif и коэффициента ослабления шума http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image024.gif для случая, когда отношение сигнал/шум на входе фильтра составляет -5 дБ.

Табл.3.1. Сравнение эффективности подавления шума при фильтрации изображений, http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image022.gif дБ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | масочный фильтр с оптимальн. КИХ | масочный фильтр с равномерн. КИХ | двумерный рекуррентн. фильтр | двумерный фильтр Винера | Медианный фильтр |
| http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image023.gif | 0.309 | 0.395 | 0.29 | 0.186 | 0.539 |
| http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image024.gif | 10.2 | 8.0 | 10.9 | 17.0 | 5.86 |

Наибольшей эффективностью обладает двумерный фильтр Винера, уменьшающий средний квадрат ошибок в 17 раз. Медианный фильтр имеет наименьшую из всех рассмотренных фильтров эффективность, ему соответствует http://www.sernam.ru/archive/arch.php?path=../htm/book_kir/files.book&file=kir_25.files/image024.gif=5.86. Тем не менее, это число свидетельствует о том, что и при его помощи удается значительно снизить уровень шума на изображении.

Вместе с тем, как говорилось выше, и что демонстрирует рис. 3.11.е, медианная фильтрация в меньшей степени сглаживает границы изображения, чем любая линейная фильтрация. Механизм этого явления очень прост и заключается в следующем. Предположим, что апертура фильтра находится вблизи границы, разделяющей светлый и темный участки изображения, при этом ее центр располагается в области темного участка. Тогда, вероятнее всего, рабочая выборка будет содержать большее количество элементов с малыми значениями яркости, и, следовательно, медиана будет находиться среди тех элементов рабочей выборки, которые соответствуют этой области изображения. Ситуация меняется на противоположную, если центр апертуры смещен в область более высокой яркости. Но это и означает наличие чувствительности у МФ к перепадам яркости.

# Практическая часть.

Была написана программа фильтрации изображения медианным алгоритмом. Алгоритм был реализован с использованием технологии OpenCL, а так же с использованием стандартных средств C++.

### Описание алгоритма фильтрации изображения

Для каждого пикселя берем окно размером “Глубина фильтрации”\*“Глубина фильтрации”. В стандартном изображении каждый пиксель кодируется тремя цветами RGB. Для качественной фильтрации шумов необходимо фильтровать пиксели по всем трем цветам. Т.е. в действительности нам необходимо провести медианную фильтрацию 3 раза для каждого пикселя.

1. Сначала берем окно для красной компоненты.
2. Расставляем все интенсивности по возрастанию.
3. Величина интенсивности красной компоненты обрабатываемого пикселя будет равна центральному элементу отсортированного окна.
4. Повторяем эту процедуру для зеленого и синего цвета

После этого мы получаем готовый пиксель. Для сортировки был использован алгоритм …, который достаточно быстрый и простой в реализации, что не маловажно, т.к. графические устройства не поддерживают рекурсию и динамическое выделение памяти.

Листинг написанной программы, которая выполняется ядром устройства OpenCL:

\_\_kernel void Filter(

\_\_global unsigned int \*in,

\_\_global unsigned int \*out,

int edge)

{

const int x = get\_global\_id(0); // Получаем индекс в 0 измерение

const int y = get\_global\_id(1); // Получаем индекс в 1 измерение

const int width = get\_global\_size(0); // Получаем размерность 0 измерения

const int height = get\_global\_size(1); // Получаем размерность 1 измерения

if ((x >= width) || (y >= height)) return; // Проверяем что индексы не вышли за диаппазон

unsigned int tmp[1000]; // Создадим массив для фильтрующего окна

unsigned char colorTmp[1000]; // Массив для цветов

int tmpSize = edge \* edge;

unsigned int pixel = 0x000000;

// Берем окно размером edge x edge

for(int l = -edge/2; l < edge/2; l++)

{

int line = l;

if(l + y >= height)

{

line = height - (l + y);

}

else if(y + l < 0)

{

line = -(y + l);

}

for(int r = -edge/2; r < edge/2; r++)

{

int raw = r;

if(r + x >= width)

{

raw = width - (r + x);

}

else if(r + x < 0)

{

raw = -(r + x);

}

tmp[(l + edge/2) \* edge + (r + edge/2)] = in[(width \* (y + line)) + (x + raw)];

}

}

// Красный

for(int i = 0; i < tmpSize; i++)

{

colorTmp[i] = RED(tmp[i]);

}

sort(&colorTmp, tmpSize);

pixel = pixel + OUTRED(colorTmp[(edge \* edge - 1) / 2]);

// Зеленый

for(int i = 0; i < tmpSize; i++)

{

colorTmp[i] = GREEN(tmp[i]);

}

sort(&colorTmp, tmpSize);

pixel = pixel + OUTGREEN(colorTmp[(edge \* edge - 1) / 2]);

// Синий

for(int i = 0; i < tmpSize; i++)

{

colorTmp[i] = BLUE(tmp[i]);

}

sort(&colorTmp, tmpSize);

pixel = pixel + OUTBLUE(colorTmp[(edge \* edge - 1) / 2]);

// Записываем в пиксель медиану (центральный пиксель)

out[width \* y + x] = pixel;

}

Листинг алгоритма с использованием стандартных средств C++:

void COpenCLImageFilterDlg::LAFilter(unsigned int\* in, unsigned int\* out, int width, int height, int edge)

{

for(int y = 0; y < height; y++)

{

for(int x = 0; x < width; x++)

{

int tmpSize = edge \* edge;

unsigned char \*colorTmp = new unsigned char[tmpSize]; // Массив для цветов

unsigned int \*tmp = new unsigned int[tmpSize]; // Создадим массив для фильтрующего окна

unsigned int pixel = 0x000000;

// Берем окно размером edge x edge

for(int l = -edge/2; l < edge/2; l++)

{

int line = l;

if(l + y >= height)

{

line = height - (l + y);

}

else if(y + l < 0)

{

line = -(y + l);

}

for(int r = -edge/2; r < edge/2; r++)

{

int raw = r;

if(r + x >= width)

{

raw = width - (r + x);

}

else if(r + x < 0)

{

raw = -(r + x);

}

tmp[(l + edge/2) \* edge + (r + edge/2)] = in[(width \* (y + line)) + (x + raw)];

}

}

// Красный

for(int i = 0; i < tmpSize; i++)

{

colorTmp[i] = RED(tmp[i]);

}

sort(colorTmp, tmpSize);

pixel = pixel + OUTRED(colorTmp[(edge \* edge - 1) / 2]);

// Зеленый

for(int i = 0; i < tmpSize; i++)

{

colorTmp[i] = GREEN(tmp[i]);

}

sort(colorTmp, tmpSize);

pixel = pixel + OUTGREEN(colorTmp[(edge \* edge - 1) / 2]);

// Синий

for(int i = 0; i < tmpSize; i++)

{

colorTmp[i] = BLUE(tmp[i]);

}

sort(colorTmp, tmpSize);

pixel = pixel + OUTBLUE(colorTmp[(edge \* edge - 1) / 2]);

// Записываем в пиксель медиану (центральный пиксель)

out[width \* y + x] = pixel;

delete [] tmp;

delete [] colorTmp;

}

}

}

### Приведение результатов работы программы

Для демонстрации целесообразности использования данной технологии сравним скорость выполнения одного и того же алгоритма с использованием OpenCL на разных платформах/устройствах и с использованием стандартных средств C++.

Конфигурация компьютера, на котором выполнялось сравнение:

CPU Intel Core i5-2450M

GPU AMD Radeon HD 7400M

ОЗУ 6Гб

Устройства поддерживающие OpenCL: CPU, GPU

Результаты фильтрации изображения размером 584x329 px с разным уровнем зашумленности представлены в таблице 1.

|  |  |  |  |
| --- | --- | --- | --- |
| Уровень шума, % | Глубина фильтрации, px | Устройство | Время, мс |
| 15% | 4 | CPU | 146.899 |
| 15% | 4 | GPU | 273.904 |
| 15% | 4 | CPU & GPU | 162.921 |
| 15% | 4 | Линейный алгоритм | 766.343 |
| 50% | 8 | CPU | 1186.602 |
| 50% | 8 | GPU | 3544.097 |
| 50% | 8 | CPU & GPU | 1773.141 |
| 50% | 8 | Линейный алгоритм | 5128.284 |

Таблица 1. Результаты фильтрации изображения размером 584x329 px

Результаты фильтрации изображения размером 128x697 px с разным уровнем зашумленности представлены в таблице 2.

|  |  |  |  |
| --- | --- | --- | --- |
| Уровень шума, % | Глубина фильтрации, px | Устройство | Время, мс |
| 15% | 4 | CPU | 639.554 |
| 15% | 4 | GPU | 932.504 |
| 15% | 4 | CPU & GPU | 492.247 |
| 15% | 4 | Линейный алгоритм | 3052.037 |
| 50% | 8 | CPU | 8412.208 |
| 50% | 8 | GPU | 14822.143 |
| 50% | 8 | CPU & GPU | 7448.263 |
| 50% | 8 | Линейный алгоритм | 30735.625 |

Таблица 2. Результаты фильтрации изображения размером 128x697 px

Как видно из таблиц, фильтрация изображения с использованием технологии OpenCL проходит до 5-6 раз быстрее, чем с использованием линейных алгоритмов С++.

# Выводы.

# Список литературы.

1. OpenCL: <https://ru.wikipedia.org/wiki/OpenCL> (Дата обращения: 07.06.2016 г.).

2. Программируемая пользователем вентильная матрица: <https://ru.wikipedia.org/w/index.php?title=FPGA&redirect=no> (Дата обращения: 07.06.2016 г.).

3. GPGPU: <https://ru.wikipedia.org/wiki/GPGPU> (Дата обращения: 07.06.2016 г.).

4.