В.А.Антонюк

# OpenCL

## Открытый язык для параллельных программ

Москва
Физический факультет МГУ им.М.В.Ломоносова
2017

**Антонюк Валерий Алексеевич**

**OpenCL. Открытый язык для параллельных программ.** –
М.: Физический факультет МГУ им. М.В. Ломоносова, 2017. – 88 с.

Пособие написано по материалам спецкурса, впервые прочитанного студентам отделения прикладной математики (ОПМ) физического факультета осенью 2016 года (3-й семестр магистратуры) о стандарте OpenCL для параллельных вычислений. В нём подробно анализируется содержимое интерфейсных заголовочных файлов OpenCL, разбираются основные функции программного C-интерфейса и его C++-вариант. Читатели ознакомятся с расширениями OpenCL-устройств (device extensions), их типами и схемой их поименования, с информацией о них в заголовочных файлах, а также с конкретными расширениями Khronos и производителей.

Обсуждаются сосуществование различных реализаций OpenCL в рамках одной системы (Windows, Linux, Android) и способы реализации такого сосуществования (т.н. ICD Loader от Khronos и альтернативный: ocl-icd). Изучаются имеющиеся в OpenCL объекты памяти (буферы и изображения) и работа с этими объектами: создание, чтение/запись, копирование, а также другие общие функции работы с ними. Рассматриваются описатели изображений, поддерживаемые форматы, сэмплеры.

Важное место уделяется взаимодействию в OpenCL-программе вычислительных частей с визуализирующими (т.н. OpenCL-OpenGL Interoperability), разъясняются правила совместного владения общими объектами памяти и раздельного их использования. Уточняются способы создания контекста для работы с подобными совместно используемыми объектами и прорабатываются практические примеры программ с координацией работы OpenCL и OpenGL.

Читатели узнают о существующих реализациях OpenCL: «настольных» вариантах (AMD, Intel, NVidia), вариантах для мобильных устройств и реализациях с открытым кодом (Beignet и POCL). Освещаются некоторые новые возможности OpenCL версии 2.0.

Пособие рассчитано на студентов старших курсов и аспирантов.

Автор — сотрудник кафедры математического моделирования и информатики физического факультета МГУ.

Рецензенты: доцент И.К.Гайнуллин, к.ф.-м.н. Д.А.Бикулов.

Подписано в печать 20.09.2017. Объем 5,5 п.л. Тираж 30 экз. Заказ №151. Физический факультет им. М.В.Ломоносова, 119991 Москва, ГСП-1, Ленинские горы, д.1, стр. 2.
Отпечатано в отделе оперативной печати физического факультета МГУ.

# Оглавление

# Предисловие

Перед тем, как приступить к изложению весьма интересной попытки создания стандарта параллельного программирования *OpenCL*, хотелось бы сделать несколько замечаний.

Это пособие было бы правильно считать авторским конспектом некоторых, кажущихся наиболее существенными и важными, сведений из вышеупомянутой области: кое-где — чуть более подробным, кое-где — совсем кратким. Почему автор почувствовал необходимость в создании такого конспекта — объяснить легко: осознанное желание иметь переносимые программы, многочисленные уже имеющиеся реализации *OpenCL*, новые возможности для распространённых вокруг устройств. Понятно, что любой заинтересованный может найти необходимую информацию в спецификациях. Однако, следует признать: всё-таки текст спецификаций — хотя и является исчерпывающим, — мало пригоден для первоначального чтения и тем более — лёгкого усвоения. Возможно, что этим объясняется довольно медленный рост популярности такого переносимого решения как *OpenCL*. Данное пособие призвано помочь желающим войти в курс дела.

### Это не справочное пособие

Несмотря на кажущееся подробным изложение отдельных тематических частей, пособие следует рассматривать не как небольшой справочник по *OpenCL*, а лишь как вводный курс в предметную область, тем более, что большая часть материала ограничивается (увы!) изложением аспектов версии *OpenCL 1.1*.

### Для удобочитаемости

Шрифтовые выделения в тексте пособия следуют весьма простым и интуитивно понятным правилам:

- имена типов, констант, переменных, функций, фрагменты кода, названия файлов, интернет-ссылки и т.п. используют `моноширинный шрифт`;
- понятия, имеющие значение для излагаемого курса, написаны *курсивом*, а если они вводятся или поясняются в какой-то части текста, то выделены ***жирным курсивом***; это относится и к написанию аналогов русскоязычных терминов на английском языке, а также к именам/фамилиям и некоторым названиям.

Вкрапления в основной текст, объясняющие или уточняющие какие-то детали, используют мелкий шрифт и при первом чтении могут быть пропущены без ущерба для понимания текста. Обращать на них внимание имеет смысл лишь при повторном обращении к тексту.

Функции программного интерфейса представлены, как правило, в виде объявлений (прототипов). Для имён фиктивных параметров в них используется *`моноширинный курсив`*. Если же приводится пример (или фрагмент) исполняемого кода, то параметры у функций курсивом не выделяются и имеют реальные имена.

Аналогично выделяются поясняющие кусочки (часто — на русском языке) в условных схемах командных строк, названиях отдельных групп родственных функций и др.

### Об индексировании понятий

В предметный указатель (индекс) включены далеко не все понятия, обсуждаемые в этом пособии, но это связано только с тем, что в *OpenCL* вводится огромное число понятий, типов, функций, величин и других идентификаторов, а потому охватить всё — трудно, да и не обязательно для введения в данную предметную область; для разного рода справок и уточнений лучше обращаться непосредственно к первоисточникам: текстам спецификаций различных версий. Те же понятия, что включены в индекс, имеют такие же шрифтовые выделения, что и в тексте пособия — чтобы сделать проще пользование им.

# 1. Введение

В настоящее время мы наблюдаем значимые изменения в принципах построения вычислительной техники, поскольку центральные процессоры теперь содержат несколько ядер, а графические процессоры (*GPU*) превратились из узко специализированных устройств в программируемые параллельные процессоры общего назначения.

Создание программ для подобных многоядерных *CPU* и *GPU* очевидным образом требует новых подходов и новых программных средств. Одним из таких средств является программная платформа *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* и др.). Он включает в себя программный интерфейс (*API*) для координирования параллельных вычислений в среде разнородных процессоров и кроссплатформенный язык, используемый в определённом 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 системной платформы и набора необходимых заголовочных файлов.

## 1.1. Используемая терминология

*OpenCL*-приложение — это совокупность программного кода, исполняемого на хосте (***host***) и *OpenCL*-устройствах (***device***). Под хостом обычно понимается центральный процессор (*CPU*) вычислительного устройства (компьютера, планшета, телефона и т.п.), а устройства *OpenCL* (*devices*) — некоторый набор вычислительных единиц, соответствующий графическому процессору (*GPU*), многоядерному *CPU* или другим процессорам с параллельной архитектурой, доступным из хост-программы.

Исполнение *OpenCL*-приложения, таким образом, включает исполнение хост-программы (на хосте) и исполнение специального кода на одном или нескольких *OpenCL*-устройствах под управлением хост-программы.

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.

Исполняемые на *OpenCL*-устройствах ядра пишутся на языке *OpenCL C*, который основывается на спецификации языка *C* 1999 года (т.н. *C99*) со специфическими расширениями и ограничениями.

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.

В терминологии *OpenCL* каждая исполняемая копия кода ядра именуется ***work-item*** (мы будем использовать далее название «рабочая единица»), она характеризуется своими уникальными индексами: глобальным идентификатором (***global ID***) и локальным идентификатором (***local ID***). Этих индексов — два, поскольку локальный индекс характеризует конкретную рабочую единицу в рамках так называемой рабочей группы (***work-group***), а глобальный индекс уникально характеризует рабочую единицу независимо от принадлежности к какой-либо рабочей группе.

> В способе исполнения ядер в *OpenCL* можно усмотреть сходство с архитектурой *CUDA*, тем более, что различия кажутся чисто терминологическими, однако, здесь есть и много отличий.

## 1.2. Память

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 неё рабочие единицы группы могут обмениваться информацией друг с другом.

*Приватная память* — область памяти для локальных переменных экземпляра ядра. Любая рабочая единица имеет свою копию каждой локальной переменной, которая доступна только ей, но не другим рабочим единицам.

## 1.3. Работа *OpenCL*-приложения

Первоначально *OpenCL*-приложение опрашивает имеющиеся *OpenCL*-платформы. Из списка найденных платформ приложение выбирает какую-то одну — нужного типа (*CPU*, *GPU* и т.п.) — и создаёт контекст: некое окружение, в котором будут запускаться на исполнение ядра. Контекст включает в себя информацию о наборе устройств, существующих в рамках платформы, памяти, доступной устройствам, а также очереди команд (***command queues***), используемые для организации исполнения ядер или операций над объектами памяти.

Взаимодействие хоста и устройств осуществляется с помощью команд, а для доставки этих команд устройствам и используются очереди команд. Одновременно с командой можно создать объект события (***event***). Такие объекты позволяют приложению проверять завершение исполнения команд, а потому могут использоваться для синхронизации.

Для выделения памяти на устройствах создаются объекты памяти; свойства их (например, возможность чтения/записи) устанавливаются приложением.

Программные объекты (***program objects***) создаются загрузкой исходного или бинарного представления одного или нескольких ядер и последующей процедурой построения (***build***) исполняемого кода.

В результате возникают объекты ядер (***kernel objects***), которые — после указания параметров для ядер — могут быть отправлены на устройства для исполнения.

Исполнение (после передачи ядрам необходимых параметров) осуществляется в рамках ***NDRange*** — одно-, двух- или трёхмерного индексного пространства рабочих единиц (*work-item*); его размерность обычно соответствует размерности входных или выходных данных. Это индексное пространство подразделяется на рабочие группы (*work-group*) одинакового размера. В *OpenCL* предполагается, что размеры индексного пространства нацело делятся на соответствующие размеры рабочих групп. Если это не так, то индексное пространство дополняется по каждому измерению рабочими единицами — до выполнения этого условия. При этом в коде ядер дополнительных рабочих единиц достаточно просто не производить никаких действий.

Размер рабочей группы обычно определяется возможностями вычислительных единиц (см. далее функцию `clGetDeviceInfo()` с параметрами `CL_DEVICE_MAX_WORK_GROUP_SIZE`, `CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS` и `CL_DEVICE_MAX_WORK_ITEM_SIZES`).

> Надо отметить, что имеется и другое ограничение на размер рабочей группы; это существенно для реализаций *OpenCL* с ограниченными ресурсами (например, на мобильных устройствах). Там реальный размер рабочей группы зависит от размера кода ядер и может быть получен вызовом функции `clGetKernelWorkGroupInfo()` с параметром `CL_KERNEL_WORK_GROUP_SIZE`. Поэтому в программах при запуске ядер параметр т.н. локального размера часто вообще не указывают (`NULL`), оставляя его на усмотрение реализации.

Финальным шагом любого *OpenCL*-приложения — после завершения вычислений и возвращения результатов на хост — должно быть освобождение всех созданных объектов.

# 2. Определение наличия платформ и устройств

Для опроса имеющихся *OpenCL*-платформ используется функция `clGetPlatformIDs()`:

```
cl_int clGetPlatformIDs(
            cl_uint num_entries,
            cl_platform_id *platforms,
            cl_uint *num_platforms);
```

Часто она вызывается приложением дважды. Первый вызов использует лишь указатель *num_platforms* на переменную, куда возвращается количество обнаруженных платформ; параметры *num_entries* и *platforms* при этом принимают «неопределённые» значения `0` и `NULL` соответственно.

```
cl_uint nP;
cl_uint status = clGetPlatformIDs(0, NULL, &nP);
```

> Если обратиться к описанию функции `clGetPlatformIDs()` в спецификации *OpenCL* `https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clGetclGetPlatformIDs.html`, то можно заметить, что возможно возвращение ошибки `CL_INVALID_VALUE` в случае передаваемых нулевого значения *num_ entries* и при этом ненулевого значения *platforms*, что выглядит спорно, поскольку по приводимому там же описанию первый параметр задаёт количество идентификаторов для возвращения в массив, заданный вторым параметром, а, значит, пользователь вполне может счесть допустимым подготовить массив для идентификаторов, но (скажем, временно) ничего не считывать туда.

Как только количество платформ становится известно, выделяется память под заданное число идентификаторов

```
cl_platform_id *pfs = new cl_platform_id[nP];
```

и функция вызывается повторно, при этом передаётся указатель на массив величин типа `cl_platform_id` и его размер (в этих величинах); указатель *num_platforms* при этом уже не нужен и его значение передаётся равным `NULL`:

```
status = clGetPlatformIDs(nP, pfs, NULL);
```

После этого функция `clGetPlatformInfo()` позволяет получить более детальную информацию об *OpenCL*-платформе (профиль, версия, имя, изготовитель и список расширений).

```
cl_int clGetPlatformInfo(
            cl_platform_id platform,
            cl_platform_info prm_name,
            size_t prm_value_size,
            void *prm_value,
            size_t *prm_value_size_ret);
```

Для заданной платформы *platform* указывается тип необходимой дополнительной информации *prm_name* (с помощью заранее предопределённых констант `CL_PLATFORM_PROFILE`, `CL_PLATFORM_VERSION`, `CL_PLATFORM_NAME`, `CL_PLATFORM_VENDOR`, `CL_PLATFORM_EXTENSIONS`); остальные параметры используются в описанном выше стиле «двойного вызова»:

```
size_t size;
char *str;

clGetPlatformInfo(pfs[i], ..., 0, NULL, &size);
str = new char [size];
clGetPlatformInfo(pfs[i], ..., size, str, NULL);
```

Когда доступные *OpenCL*-платформы известны, можно проанализировать, какие в них имеются *OpenCL*-устройства. Для этого используется функция `clGetDeviceIDs()`, аналогичная функции `clGetPlatformIDs()`, но принимающая дополнительные параметры: конкретную платформу *`platform`* и желательный тип устройства (`CL_DEVICE_TYPE_GPU`, `CL_DEVICE_TYPE_CPU` и др. либо `CL_DEVICE_TYPE_ALL`).

```
cl_int clGetDeviceIDs(
            cl_platform_id platform,
            cl_device_type device_type,
            cl_uint num_entries,
            cl_device_id *devices,
            cl_uint *num_devices);
```

Для каждого устройства эту функцию обычно тоже вызывают дважды, например:

```
status = clGetDeviceIDs(pfs[i], CL_DEVICE_TYPE_ALL, 0, NULL, &nD);
devs = new cl_device_id[nD];
status = clGetDeviceIDs(pfs[i], CL_DEVICE_TYPE_ALL, nD, devs, NULL);
```

Далее можно воспользоваться функцией `clGetDeviceInfo()` — чтобы получить сведения об имени, типе и изготовителе для каждого устройства, а также их многочисленных дополнительных параметрах и возможностях — аналогично функции `clGetPlatformInfo()`: `https://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clGetDeviceInfo.html`.

```
cl_int clGetDeviceInfo(
            cl_device_id device,
            cl_device_info param_name,
            size_t prm_value_size,
            void *prm_value,
            size_t *param_value_size_ret);
```

Полную программу, содержащую все эти вызовы, демонстрирует файл `printinfo.cpp`, упоминаемый в курсе по параллельному программированию в ***Oregon State University*** (автор — ***Mike Bailey***).

> Эта программа — один из не очень многочисленных *OpenCL*-примеров в Сети, куда не пришлось вносить поправки, чтобы откомпилировать и на *Windows*-компьютерах, и на компьютерах с *Linux*, — вероятно, потому, что автору сразу была нужна переносимость между этими системными платформами, а сама программа использовала не так уж и много функций из *OpenCL*.
>
> Тем не менее, в ней имеется досадная неточность, не очень заметная на первый взгляд, особенно если пробовать её на компьютере с малым числом установленных платформ *OpenCL*. Дело в том, что после просмотра всех платформ и устройств на каждой платформе там вдруг зачем-то выводятся сведения о расширениях, причём первого же устройства (с индексом 0) последней из просмотренных платформ. Видимо, это дополнение было внесено в программу уже после её написания и ошибочность положения этого фрагмента не была замечена автором из-за единственности *OpenCL*-платформы и *OpenCL*-устройства на его компьютере.
>
> Следует исправить её, внеся вывод расширений для устройств, расположенный в конце вне всех циклов, в цикл по устройствам — с заменой нулевого индекса устройства в оригинале на переменную `j` с номером текущего устройства.

Более профессиональной программой аналогичного назначения является `clinfo.c`[1]. Она написана (как утверждается) на *C99*, что позволяет откомпилировать её практически для любой системной платформы. Кроме того, в ней предприняты меры для «устойчивой» работы в условиях недоступности каких-либо из опрашиваемых свойств. Следует также отметить, что её название совпадает с названием стандартной программы (`clinfo`) такого же назначения для систем *Linux*.

[1] За неимением точных сведений об авторе, говоря далее об этой программе, будем использовать его псевдоним (***Oblomov***) вместо имени.

# 3. Информация о возможностях устройств

Как уже говорилось, для получения сведений о возможностях конкретного устройства *OpenCL* используется функция `clGetDeviceInfo()`, которой должны быть переданы: само устройство *`device`* (то, что возвращено вызванной ранее функцией `clGetDeviceIDs()`), вид информации об устройстве *`prm_name`* (см. список ниже), указатель *`prm_value`* на место в памяти, куда может быть помещена такая информация, размер памяти в байтах *`prm_value_size`*, который должен быть не меньше, чем размер возвращаемой информации, и реальный размер *`prm_value_size_ret`* информации в байтах.

```
cl_int clGetDeviceInfo(
            cl_device_id device,
            cl_device_info prm_name,
            size_t prm_value_size,
            void *prm_value,
            size_t *prm_value_size_ret);
```

Для тех сведений, возвращаемый размер которых заранее известен, можно вызывать эту функцию один раз, передавая размер переменной и её адрес, а оставшийся последний параметр полагая равным `NULL`:

```
clGetDeviceInfo(device, <ВидИнформации>, sizeof(aValue), &aValue, NULL);
```

Для параметров заранее неизвестного размера (скажем, `CL_DEVICE_NAME`, `CL_DEVICE_VENDOR`, `CL_DRIVER_VERSION`, `CL_DEVICE_PROFILE`, `CL_DEVICE_VERSION`, `CL_DEVICE_OPENCL_C_VERSION`, `CL_DEVICE_EXTENSIONS`) следует вызывать эту функцию дважды, первый раз определяя необходимый размер памяти, затем выделять эту необходимую память и повторно вызывать функцию для получения значения параметра:

```
clGetDeviceInfo(device, <ВидИнформации>, 0, NULL, &infoSize);
char *info = new char [infoSize];
clGetDeviceInfo(device, <ВидИнформации>, infoSize, info, NULL);
```

Здесь в коде уже учтено, что возвращаемая информация — это просто последовательность символов.

## 3.1. Список возможных видов информации

В нижеследующем списке для каждого вида информации сначала указано условное имя, затем (на следующей строке) — возвращаемый тип данных, далее следует краткое описание.

`CL_DEVICE_TYPE`
: `cl_device_type`
Тип *OpenCL*-устройства (*CPU*, *GPU*, акселератор или их комбинация).

`CL_DEVICE_VENDOR_ID`
: `cl_uint`
Уникальный идентификатор производителя.

`CL_DEVICE_MAX_COMPUTE_UNITS`
: `cl_uint`
Количество параллельных вычислительных единиц; рабочая группа исполняется на одной такой единице, поэтому минимальное значение параметра — 1.

`CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS`
: `cl_uint`
Максимальное количество измерений, используемых в модели исполнения; минимальное значение — 3.

`CL_DEVICE_MAX_WORK_ITEM_SIZES`

`size_t []`

Максимальное число рабочих единиц, которое может быть указано по каждому измерению рабочей группы при вызове функции `clEnqueueNDRangeKernel()`. Возвращается $n$ значений типа `size_t`, где $n$ — величина, получаемая запросом `CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS`. Минимальным значением является тройка $(1, 1, 1)$.

`CL_DEVICE_MAX_WORK_GROUP_SIZE`

`size_t`

Максимальное число рабочих единиц (*work-items*) в рабочей группе (*work-group*), исполняющих код ядра на одной вычислительной единице одновременно.

`CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR`
`CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT`
`CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT`
`CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG`
`CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT`
`CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE`
`CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF`

`cl_uint`

Предпочтительный размер вектора для встроенных скалярных типов; если расширения `cl_khr_fp64`, `cl_khr_fp16` не поддерживаются, то соответствующие им параметры должны возвращать нулевое значение.

`CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR`
`CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT`
`CL_DEVICE_NATIVE_VECTOR_WIDTH_INT`
`CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG`
`CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT`
`CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE`
`CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF`

`cl_uint`

Реальный размер вектора встроенных скалярных типов; если расширения `cl_khr_fp64`, `cl_khr_fp16` не поддерживаются, то соответствующие им параметры должны возвращать нулевое значение.

`CL_DEVICE_MAX_CLOCK_FREQUENCY`

`cl_uint`

Максимальная частота устройства в мегагерцах (*MHz*).

`CL_DEVICE_ADDRESS_BITS`

`cl_uint`

Размер адресного пространства устройства в битах как беззнаковое число. Поддерживаемые сейчас величины — 32 или 64 бита.

`CL_DEVICE_MAX_MEM_ALLOC_SIZE`

`cl_ulong`

Максимальный размер аллоцируемого объекта памяти в байтах. Минимальное значение — наибольшая из величин: четверть `CL_DEVICE_GLOBAL_MEM_SIZE` и $128 \times 1024 \times 1024$.

`CL_DEVICE_IMAGE_SUPPORT`

`cl_bool`

`CL_TRUE`, если изображения поддерживаются устройством, иначе — `CL_FALSE`.

`CL_DEVICE_MAX_READ_IMAGE_ARGS`

`cl_uint`

Максимальное количество объектов изображений, которые могут одновременно читаться ядром. Минимальное значение — 128, если `CL_DEVICE_IMAGE_SUPPORT` имеет значение `CL_TRUE`.

CL_DEVICE_MAX_WRITE_IMAGE_ARGS
: cl_uint
Максимальное количество объектов изображений, которые могут одновременно записываться ядром. Минимальное значение — 8, если CL_DEVICE_IMAGE_SUPPORT имеет значение CL_TRUE.

CL_DEVICE_IMAGE2D_MAX_WIDTH
: size_t
Максимальная ширина 2D-изображения в пикселах (не менее 8192).

CL_DEVICE_IMAGE2D_MAX_HEIGHT
: size_t
Максимальная высота 2D-изображения в пикселах (не менее 8192).

CL_DEVICE_IMAGE3D_MAX_WIDTH
: size_t
Максимальная ширина 3D-изображения в пикселах (не менее 2048).

CL_DEVICE_IMAGE3D_MAX_HEIGHT
: size_t
Максимальная высота 3D-изображения в пикселах (не менее 2048).

CL_DEVICE_IMAGE3D_MAX_DEPTH
: size_t
Максимальная глубина 3D-изображения в пикселах (не менее 2048).

CL_DEVICE_MAX_SAMPLERS
: cl_uint
Максимальное число сэмплеров, которое может быть использовано в ядре. Минимальное значение — 16, если CL_DEVICE_IMAGE_SUPPORT имеет значение CL_TRUE.

CL_DEVICE_MAX_PARAMETER_SIZE
: size_t
Максимальный суммарный размер в байтах параметров, передаваемых ядру (не менее 1024, при этом число передаваемых параметров также не должно быть больше 128).

CL_DEVICE_MEM_BASE_ADDR_ALIGN
: cl_uint
Минимальное значение — размер в битах наибольшего встроенного типа данных *OpenCL*, поддерживаемого устройством (это тип long16 в профиле FULL и тип long16 или int16 — в профиле EMBEDDED).

CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE
: cl_uint
Минимальное значение — размер в байтах наибольшего встроенного типа данных *OpenCL*, поддерживаемого устройством (это тип long16 в профиле FULL и тип long16 или int16 — в профиле EMBEDDED).

CL_DEVICE_SINGLE_FP_CONFIG
: cl_device_fp_config
Описывает возможности устройства в части работы с вещественными величинами. Это битовое поле, содержащее какой-то набор из следующих величин: CL_FP_DENORM (поддерживаются денормализованные величины), CL_FP_INF_NAN (поддерживаются *INF* и *quiet NANs*), CL_FP_ROUND_TO_NEAREST (поддерживается режим округления к ближайшему чётному числу), CL_FP_ROUND_TO_ZERO (поддерживается режим округления к нулю), CL_FP_ROUND_TO_INF (поддерживается режим округления к бесконечности), CL_FP_FMA (поддерживается совмещённое умножение–сложение по *IEEE754-2008*), CL_FP_SOFT_FLOAT (базовые операции с вещественными величинами — сложение, вычитание, умножение — реализованы программно). Минимально допустимые возможности здесь — CL_FP_ROUND_TO_NEAREST | CL_FP_INF_NAN.

`CL_DEVICE_GLOBAL_MEM_CACHE_TYPE`
: `cl_device_mem_cache_type`
Поддерживаемый тип кэша глобальной памяти. Допустимыми величинами будут: `CL_NONE`, `CL_READ_ONLY_CACHE` и `CL_READ_WRITE_CACHE`.

`CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE`
: `cl_uint`
Размер строки кэша глобальной памяти в байтах.

`CL_DEVICE_GLOBAL_MEM_CACHE_SIZE`
: `cl_ulong`
Размер кэша глобальной памяти в байтах.

`CL_DEVICE_GLOBAL_MEM_SIZE`
: `cl_ulong`
Размер глобальной памяти устройства в байтах.

`CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE`
: `cl_ulong`
Максимальный размер аллокации (в байтах) буфера констант. Минимальное значение — 64 килобайта.

`CL_DEVICE_MAX_CONSTANT_ARGS`
: `cl_uint`
Максимальное число параметров с квалификатором `__constant` в ядре (не менее 8).

`CL_DEVICE_LOCAL_MEM_TYPE`
: `cl_device_local_mem_type`
Поддерживаемый тип локальной памяти. Может иметь значение `CL_LOCAL` (что подразумевает наличие выделенной локальной памяти на устройстве) или значение `CL_GLOBAL`.

`CL_DEVICE_LOCAL_MEM_SIZE`
: `cl_ulong`
Размер области локальной памяти устройства в байтах.

`CL_DEVICE_ERROR_CORRECTION_SUPPORT`
: `cl_bool`
`CL_TRUE`, если устройство реализует коррекцию ошибок для доступа к памяти устройства (глобальной и константной), и `CL_FALSE` — в противном случае.

`CL_DEVICE_HOST_UNIFIED_MEMORY`
: `cl_bool`
`CL_TRUE`, если устройство и хост имеют подсистему разделяемой (объединённой, *unified*) памяти, и `CL_FALSE` — в противном случае.

`CL_DEVICE_PROFILING_TIMER_RESOLUTION`
: `size_t`
Разрешение таймера устройства в наносекундах.

`CL_DEVICE_ENDIAN_LITTLE`
: `cl_bool`
`CL_TRUE`, если устройство *OpenCL* является *little-endian*-устройством, иначе — `CL_FALSE`.

`CL_DEVICE_AVAILABLE`
: `cl_bool`
`CL_TRUE`, если устройство доступно. `CL_FALSE`, если устройство недоступно.

`CL_DEVICE_COMPILER_AVAILABLE`
: `cl_bool`
`CL_FALSE`, если реализация не имеет компилятора для компиляции исходного кода ядер. `CL_TRUE`, если такой компилятор имеется. Значение параметра может быть `CL_FALSE` только для платформы с профилем `EMBEDDED`.

`CL_DEVICE_EXECUTION_CAPABILITIES`

`cl_device_exec_capabilities`

Описывает возможности исполнения на устройстве. Это битовое поле, объединяющее не менее одной из следующих величин: `CL_EXEC_KERNEL` (устройство может исполнять ядра *OpenCL*), `CL_EXEC_NATIVE_KERNEL` (устройство может исполнять т.н. *«native»*-ядра). Необходимый минимум возможностей — исполнение ядер *OpenCL*.

`CL_DEVICE_QUEUE_PROPERTIES`

`cl_command_queue_properties`

Описывает свойства очереди команд, поддерживаемые устройством. Это битовое поле, объединяющее величины: `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` (возможность параллельного исполнения) и `CL_QUEUE_PROFILING_ENABLE` (возможность профилирования). Необходимый минимум — наличие свойства `CL_QUEUE_PROFILING_ENABLE`.

`CL_DEVICE_PLATFORM`

`cl_platform_id`

Платформа, связанная с этим устройством.

`CL_DEVICE_NAME`

`char[]`

Возвращается строка с именем устройства.

`CL_DEVICE_VENDOR`

`char[]`

Возвращается строка с наименованием изготовителя.

`CL_DRIVER_VERSION`

`char[]`

Возвращается строка с версией драйвера в виде *<Major>.<Minor>*.

`CL_DEVICE_PROFILE`

`char[]`

Возвращается имя профиля, поддерживаемого устройством (возможные значения здесь — `FULL_PROFILE` или `EMBEDDED_PROFILE`).

`CL_DEVICE_VERSION`

`char[]`

Возвращается версия *OpenCL* (в виде *<Major>.<Minor>*), поддерживаемая данным устройством. Может сопровождаться дополнительной информацией от изготовителя.

`CL_DEVICE_OPENCL_C_VERSION`

`char[]`

Возвращается максимальная поддерживаемая компилятором версия *OpenCL C* (тоже в виде *<Major>.<Minor>*) для данного устройства. Также может сопровождаться дополнительной информацией от изготовителя.

> Версия, возвращаемая здесь, не может быть ниже, чем версия *OpenCL*, поддерживаемая устройством. Версия *OpenCL*, поддерживаемая устройством, может быть ниже, чем версия *OpenCL C*, поддерживаемая компилятором.

`CL_DEVICE_EXTENSIONS`

`char[]`

Возвращается разделённый пробелами список поддерживаемых расширений: как самого производителя, так и из числа одобренных *Khronos*.

> Согласно версии *1.1* спецификации *OpenCL* поддерживающие её устройства должны возвращать такие имена одобренных *Khronos* расширений: `cl_khr_byte_addressable_store`, `cl_khr_global_int32_base_atomics`, `cl_khr_global_int32_extended_atomics`, `cl_khr_local_int32_base_atomics`, `cl_khr_local_int32_extended_atomics`.
> Возможна также поддержка расширений `cl_khr_fp64`, `cl_khr_int64_base_atomics`, `cl_khr_int64_extended_atomics`, `cl_khr_fp16`, `cl_khr_gl_sharing`, `cl_khr_gl_event`, `cl_khr_d3d10_sharing`.

# 4. Контекст

Как уже говорилось выше, контекст в *OpenCL* — это некоторое окружение, в котором исполняются специальные программные объекты *OpenCL* (т.н. ядра). Контекст включает в себя набор *OpenCL*-устройств, на которых будет производиться исполнение ядер, сами программные объекты с их исходными кодами и содержащимися там *OpenCL*-функциями, а также некоторый набор объектов памяти, видимых и хосту, и *OpenCL*-устройствам, — там содержатся величины, с которыми могут работать ядра.

Контекст *OpenCL* создаётся с помощью одной из двух функций: `clCreateContext()` или `clCreateContextFromType()`:

```
cl_context clCreateContext (
            const cl_context_properties *properties,
            cl_uint num_devices,
            const cl_device_id *devices,
            void (CL_CALLBACK *pfn_notify)(
                  const char *errinfo,
                  const void *private_info, size_t cb,
                  void *user_data),
            void *user_data,
            cl_int *errcode_ret);

cl_context clCreateContextFromType (
            const cl_context_properties *properties,
            cl_device_type device_type,
            void (CL_CALLBACK *pfn_notify)(
                  const char *errinfo,
                  const void *private_info, size_t cb,
                  void *user_data),
            void *user_data,
            cl_int *errcode_ret);
```

Помимо имени эти функции различаются только тем, что первая создаёт контекст для указанной последовательности устройств *devices*, используя заданное число устройств *num_devices*, а вторая — контекст для устройств определённого типа *device_type* (*CPU*, *GPU*, ускоритель; возможно также указание всех устройств или устройства по умолчанию). В остальном они принимают совершенно одинаковые параметры: указатель *properties* на массив свойств контекста, указатель *pfn_notify* на функцию обратного вызова (со своим списком параметров), указатель *user_data* на пользовательские данные, если они будут необходимы для функции обратного вызова. Возможные ошибочные ситуации при вызове любой из функций создания контекста фиксируются в переменной типа `cl_int`, указатель на которую передаётся через последний параметр *errcode_ret*.

Несмотря на кажущееся довольно большим число параметров в этих функциях, реально их вызовы чаще всего не используют ни массив свойств контекста, ни функцию обратного вызова вместе с пользовательскими данными для неё, поэтому, скажем, создание контекста для конкретного устройства с уже известным дескриптором `device` может выглядеть так:

```
cl_context Ctx = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
```

Здесь `err` — это переменная, куда в случае неудачи при создании контекста попадёт код ошибки (возможны: общие для обеих функций `CL_INVALID_PLATFORM`, `CL_INVALID_VALUE`, `CL_DEVICE_NOT_AVAILABLE`, `CL_OUT_OF_HOST_MEMORY`, а также `CL_INVALID_DEVICE` — для первой функции и `CL_DEVICE_NOT_FOUND`, `CL_INVALID_DEVICE_TYPE` — для второй).

**Замечание.** Реализация *OpenCL* от *NVidia* очень «не любит» вызов создания контекста по типу, если первый параметр в вызове функции `clCreateContextFromType()` отсутствует:

```
cl_context Ctx = clCreateContextFromType(NULL, <Tun>, NULL, NULL, &err);
```

в этом случае возвращается ошибка -32 (CL_INVALID_PLATFORM). И формально это справедливо: ничего про платформу в этом вызове не указано, а потому контекст не создаётся... Хотя другие реализации *OpenCL* в данной ситуации используют ранее выбранную платформу — и код кажется вполне работоспособным!

Кстати, утилита clinfo от ***Oblomov*** тестирует поведение некоторых функций *OpenCL API* в «неблагоприятных» условиях (в отсутствие сведений о платформе), в том числе и функций получения контекста.

Правильный путь в таком случае — создать минимальный массив свойств контекста, где содержится только идентификатор платформы cpPlatform (он определяется ранее), и только потом вызывать создание контекста по типу, причём со свойствами контекста:

```
cl_context_properties props[] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties)cpPlatform,
    0
};
cl_context Ctx = clCreateContextFromType(props, <Tun>, NULL, NULL, &err);
```

По завершении работы с *OpenCL*-устройствами контекст — как и любой другой ресурс, для которого выделяется память, — полагается «освободить». Для этой цели используется функция clReleaseContext(), которой следует передать идентификатор освобождаемого контекста.

```
cl_int clReleaseContext (cl_context context);
```

Реально действия, осуществляемые при вызове этой функции, таковы: уменьшается на единицу счётчик использования контекста и — если его значение стало равным нулю (т.е., контекст уже никем не используется), а все объекты, связанные с этим контекстом (объекты памяти, очереди команд), ранее были «освобождены», — контекст удаляется. Возвращает функция CL_SUCCESS в случае успешного завершения либо CL_INVALID_CONTEXT в случае недействительного идентификатора контекста.

Начиная с *OpenCL 1.2* появляется такая функция, как clRetainContext(), которую, как оказывается, неявно будут вызывать обе функции создания контекста.

```
cl_int clRetainContext (cl_context context);
```

Делает она не так много: увеличивает на единицу счётчик использования контекста. Необходимость в ней возникает из-за сторонних библиотек, которым контекст может быть передан как параметр. При этом, если основное приложение удаляет контекст, у сторонней библиотеки нет никакого способа узнать, что контекстом пользоваться уже нельзя. С помощью же этой функции указанная проблема легко решается: сторонняя библиотека, получая идентификатор контекста, вызывает функцию clRetainContext(), завершая работу с ним, — функцию clReleaseContext(). И теперь не имеет значения, откуда функция освобождения контекста вызвана в последний раз, контекст освобождается тогда, когда уже никем не используется.

Для получения сведений о контексте используется функция clGetContextInfo():

```
cl_int clGetContextInfo (
            cl_context context,
            cl_context_info param_name,
            size_t param_value_size,
            void *param_value,
            size_t *param_value_size_ret);
```

По заданному контексту *context* и выбранному виду информации *param_name* можно получить необходимое значение через указатель *param_value*; однако, поскольку здесь — как и в случае функции clGetDeviceInfo() — разные виды информации являются различными типами данных (со своими размерами), эту функцию тоже следует вызывать дважды для каждого вида информации (числа устройств, их списка и т.п.), см. стр.10.

# 5. Очереди команд

Для управления исполнением ядер на устройствах хост-программа создаёт специальные структуры данных, именуемые *очередями команд* (***command queue***). Примеры команд, направляемых в очередь: команды исполнения ядер, команды памяти (для перемещения данных в объекты памяти, из них или между ними), а также команды синхронизации, управляющие порядком исполнения команд.

Создаётся очередь команд путём вызова функции `clCreateCommandQueue()` с указанием нужного контекста *context*, конкретного устройства *device* и свойств очереди *properties*:

```
cl_command_queue clCreateCommandQueue (
            cl_context context,
            cl_device_id device,
            cl_command_queue_properties properties,
            cl_int *errcode_ret);
```

Возвращает функция в случае успешного исполнения дескриптор очереди команд и статус исполнения `CL_SUCCESS` — через указатель *errcode_ret*; в противном случае через этот указатель возвращается код ошибки (либо недействительности какого-либо из переданных параметров — `CL_INVALID_CONTEXT`, `CL_INVALID_DEVICE`, `CL_INVALID_QUEUE_PROPERTIES`, `CL_INVALID_VALUE`, либо нехватки памяти на хосте — `CL_OUT_OF_HOST_MEMORY`). Часто при вызове свойства очереди отсутствуют вообще, поскольку они нужны лишь в специальных случаях (и могут поддерживаться не всеми устройствами):

```
cl_command_queue que = clCreateCommandQueue(ctx, dev_id, 0, &ret);
```

Возможные значения свойств очереди: `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` — разрешить непоследовательное (***out-of-order***) исполнение — и `CL_QUEUE_PROFILING_ENABLE` — разрешить профилирование (замер времени исполнения) команд в очереди (см. стр.14).

> Команды исполняются между хостом и устройствами асинхронно: хост-программа получает управление после вызова функции помещения команды в очередь, не дожидаясь завершения исполнения команды. Относительно же друг друга команды могут исполняться по порядку (запускаются и завершаются в порядке расположения в очереди) или непоследовательно (запускаются в порядке следования, но не ждут завершения предыдущей команды). В последнем случае должны явно использоваться команды синхронизации.
>
> Начиная с *OpenCL 2.0* функция `clCreateCommandQueue()` объявлена «нежелательной», вместо неё предлагается использовать `clCreateCommandQueueWithProperties()`.

После использования очереди команд должны освобождаться с помощью функции

```
cl_int clReleaseCommandQueue (cl_command_queue command_queue);
```

ей при вызове передаётся как параметр уже ненужная очередь команд *command_queue*.

> Начиная с версии *OpenCL 1.2* появляется также функция `clRetainCommandQueue()`:
>
> ```
> cl_int clRetainCommandQueue (cl_command_queue command_queue);
> ```

Получить информацию об очереди *command_queue* можно с помощью функции

```
cl_int clGetCommandQueueInfo (
            cl_command_queue command_queue,
            cl_command_queue_info param_name,
            size_t param_value_size,
            void *param_value,
            size_t *param_value_size_ret);
```

указывая вид информации *param_name*; в остальном всё аналогично рассмотренной выше функции `clGetContextInfo()`: её тоже следует вызывать дважды для каждого *param_name*.

# 6. Работа с кодом ядра

Ядра в *OpenCL*-программах пока присутствуют в виде исходного текста: либо как отдельные файлы с расширением `.cl`, либо как текстовые строки в рамках самой программы. Процесс подготовки такого исходного текста к виду, в котором ядро может быть исполнено на устройстве, целиком лежит на программисте (в отличие от ситуации в *CUDA*, где ядра хотя и компилируются другим компилятором, но происходит это невидимым для пользователя образом).

Первым делом исходный текст ядра размещается в памяти, поскольку функция *OpenCL* `clCreateProgramWithSource()`, начинающая весь процесс, ожидает в качестве одного из своих параметров указание на *C*-строки с текстом; если исходный текст находится в файле, его содержимое считывается в память. Другие параметры, которые понадобится указать при вызове — это контекст, параметры представления текста в виде массива указателей на строки (размер этого массива и сопутствующий массив длин строк) и указатель для возвращения кода ошибки.

Результатом вызова будет программный объект типа `cl_program`:

```
cl_program clCreateProgramWithSource (
            cl_context context,
            cl_uint count,
            const char **strings,
            const size_t *lengths,
            cl_int *errcode_ret);
```

Здесь *context* — контекст, *strings* — массив указателей на символьные строки (в количестве *count*), которые могут быть как завершены нулевым символом, так и не использовать его; если «завершителя» у строк нет, их длина должна быть указана в массиве длин *lengths*. Возможные ошибочные ситуации при вызове фиксируются в переменной типа `cl_int`, указатель на которую передаётся через последний параметр *errcode_ret*.

Применяемый тут способ указания строк является достаточно гибким: можно поместить исходный текст в одну строчку, либо прочесть (или сформировать) исходный текст из нескольких строк. Если, например, используется первый вариант, то параметры вызова будут особенно просты:

```
cl_program prg = clCreateProgramWithSource(ctx, 1, &src, NULL, &err);
```

Здесь `ctx` — контекст, `1` — это количество *C*-строк текста (одна), `src` — указывает на единственный указатель на эту *C*-строку, а сама строка завершается нулевым символом, поскольку массив длин не указан (значение параметра `NULL`).

Далее программный объект подвергается «построению» с помощью вызова функции `clBuildProgram()`: компилируется и линкуется для получения исполняемого на устройстве кода.

```
cl_int clBuildProgram (
            cl_program prog,
            cl_uint num_devices,
            const cl_device_id *device_list,
            const char *options,
            void (CL_CALLBACK *pfn_notify)(cl_program prog, void *udata),
            void *udata);
```

Параметров, которые здесь передаются, довольно много, однако в простейшем случае можно обойтись построением программы по умолчанию, т.е., построить для всех устройств, без дополнительных параметров компиляции и линковки, а также без функции обратного вызова и её данных:

```
err = clBuildProgram(prg, 0, NULL, NULL, NULL, NULL);
```

Смысл этих параметров таков. Для построения исполняемого кода ядер необходимо указать программный объект *prog*, полученный на предыдущем шаге (здесь различаются и по-разному выделены имя фиктивного параметра *prog* в прототипе функции и реально используемая при вызове величина prg); задать — если это необходимо — список устройств *device_list*, для которых должно быть осуществлено построение, и количество этих устройств *num_devices* (иначе производится построение для всех устройств, связанных с программой); перечислить дополнительные опции *options* компиляции и линковки (фактически — параметры командной строки компилятора и линкера); задать — если требуется — адрес нотифицирующей функции, которая будет вызвана после завершения построения с передаваемыми ей значениями *prog* и *udata* в качестве необходимых параметров.

Если адрес такой нотифицирующей функции не указывается и вместо него передаётся NULL, возврат из функции производится только после завершения построения.

В случае возникновения каких-либо ошибок (возвращаемых здесь в переменную с именем err) необходимо дополнительно вызвать функцию clGetProgramBuildInfo() со специальным флагом CL_PROGRAM_BUILD_LOG, причём лучше всего дважды: сначала для определения размеров возвращаемого лога:

```
clGetProgramBuildInfo(prg, d_id, CL_PROGRAM_BUILD_LOG, 0, NULL, &l_size);
```

а затем — после выделения для него необходимой памяти — для получения собственно лога:

```
clGetProgramBuildInfo(prg, d_id, CL_PROGRAM_BUILD_LOG, l_size, log, NULL);
```

Вообще же эта функция используется для получения информации о процессе построения для конкретного устройства (статуса процесса, опций построения и содержимого лога).

```
cl_int clGetProgramBuildInfo (
            cl_program program,
            cl_device_id device,
            cl_program_build_info prm_name,
            size_t prm_value_size,
            void *prm_value,
            size_t *prm_value_size_ret);
```

Нужный тип информации для программного объекта *program* и устройства *device* задаётся значением параметра *prm_name* (CL_PROGRAM_BUILD_STATUS, CL_PROGRAM_BUILD_OPTIONS или CL_PROGRAM_BUILD_LOG), *prm_value* — указатель на участок памяти, куда возвращается информация, *prm_value_size* — размер информации в байтах, *prm_value_size_ret* — указатель на переменную, в которую помещается размер информации в байтах при запросе, когда *prm_value* равен NULL; если размер уже известен, параметр *prm_value_size_ret* может иметь значение NULL.

> В случае использования реализации *OpenCL* от *Apple* можно вывести лог построения на консоль путём установки переменной окружения CL_LOG_ERRORS=stdout или добавления этой информации в комадной строке запуска *OpenCL*-приложения.
>
> При работе в системе *Windows* можно также использовать специальную утилиту clcc.exe, позволяющую компилировать ядра отдельно от самого приложения прямо с командной строки (подробности её использования можно будет найти далее, см. стр.21); сообщения об ошибках будут выведены на консоль.

Важным шагом всей процедуры работы с ядрами программы будет создание объектов ядра для каждого присутствующего в исходном коде ядра с помощью clCreateKernel():

```
cl_kernel clCreateKernel (
            cl_program program,
            const char *kernel_name,
            cl_int *errcode_ret);
```

Здесь *program* — програмный объект с успешно построенным исполняемым кодом ядра, *kernel_name* — имя функции со спецификатором `__kernel`, *errcode_ret* — указатель на переменную для кода ошибки. В результате возвращается объект ядра, готовый для последующего запуска на рабочих единицах (*work-item*) соответствующего устройства.

Если ядро только одно, понадобится один вызов, где надо указать имя ядра (`kname`).

```
cl_kernel krnl = clCreateKernel(prg, kname, &err);
```

В случае нескольких ядер в программе делается столько вызовов этой функции, сколько имеется ядер в программе, — каждый со своим значением параметра имени ядра.

Поскольку ядрам почти всегда нужна дополнительная информация (это те параметры, которые перечислены при определении как параметры функции ядра), следует перед запуском каждого ядра передать необходимые значения этих параметров. Осуществляется это с помощью функции `clSetKernelArg()`, вызываемой требуемое число раз (возвращаемым значением является код ошибки или признак успешного выполнения):

```
cl_int clSetKernelArg (
            cl_kernel kernel,
            cl_uint arg_index,
            size_t arg_size,
            const void *arg_value);
```

Здесь *kernel* — объект ядра, для которого передаётся параметр, *arg_index* — индекс параметра (отсчитывается с нуля, начиная с самого левого параметра функции ядра), *arg_value* — указатель на данные, которые должны быть использованы для установки значения параметра (данные будут скопированы, поэтому значение указателя после вызова может быть изменено), *arg_size* — размер данных (в байтах).

> **Ограничения.** Если какие-то параметры сопровождаются квалификатором `__constant`, то следует позаботиться о том, чтобы размеры отдельных объектов памяти не превышали значения `CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE`, а количество таких параметров — величины `CL_DEVICE_MAX_CONSTANT_ARGS` (чтобы не произошло выхода за пределы имеющейся константной памяти).

Запуск ядер на исполнение производится функцией `clEnqueueNDRangeKernel()`, которой в качестве одного из параметров указывается «размер задачи»:

```
cl_int clEnqueueNDRangeKernel (
            cl_command_queue command_queue,
            cl_kernel kernel,
            cl_uint work_dim,
            const size_t *global_work_offset,
            const size_t *global_work_size,
            const size_t *local_work_size,
            cl_uint num_events_in_wait_list,
            const cl_event *event_wait_list,
            cl_event *event);
```

Минимально необходимыми здесь параметрами являются: очередь команд *command_queue*, объект ядра *kernel*, количество измерений *work_dim* рабочего размера и сам глобальный рабочий размер *global_work_size* с таким числом измерений. Поэтому довольно часто вызов этой функции выглядит примерно так (одномерный размер здесь выбран вполне произвольно):

```
size_t gsize[1] = { 1000000 };
err = clEnqueueNDRangeKernel(queue,krnl,1,NULL,gsize,NULL,0,NULL,NULL);
```

После использования объектов-ядер их полагается «отпустить» (реально уменьшается счётчик ссылок на каждый объект ядра) с помощью функции

```
cl_int clReleaseKernel (cl_kernel kernel);
```

# 7. Отдельная компиляция ядер

Из-за того, что ядра компилируются после запуска хост-программы — прямо в процессе её работы, — а ошибки такой компиляции должны проверяться отдельно (и без дополнительных действий о них ничего не будет известно!), существует опасность, что работа «плохо» написанной хост-программы не будет сопровождаться запуском необходимых ядер. В результате возможна ситуация, когда *OpenCL*-программа вроде бы успешно запускается, но получаемые с её помошью результаты кажутся как минимум странными. А иногда неправильность работы подобной программы можно вообще не заметить...

Чтобы быть абсолютно уверенными в том, что ядра откомпилировались без ошибок, следует обязательно проверять результаты компиляции — так, как это рекомендовано в предыдущем разделе. Кроме того, во время разработки кода самих ядер имеет смысл компилировать их отдельно, чтобы возможные синтаксические ошибки обнаруживались быстрее.

Для отдельной компиляции ядер удобно использовать специальную программу, которая даёт доступ к скрытому от нас компилятору ядер через упомянутые ранее фунции `clCreateProgramWithSource()`, `clBuildProgram()`, `clGetProgramBuildInfo()`.

Если ограничиться системой *Windows*, то роль такой программы вполне может исполнять `clcc` — ***OpenCL Kernel Compiler*** (`http://sourceforge.net/projects/clcc/`), поскольку даже её давно доступная версия `0.3` имеет минимально необходимые в таких случаях возможности: выдаёт информацию об *OpenCL*-устройствах и позволяет указать для компиляции конкретную платформу и конкретное устройство.

```
clcc -p <Платформа> -d <Устройство> <ИмяФайлаСЯдром>
```

Необходимые для этой строки значения идентификаторов платформы и устройства можно узнать с помощью «справочного» запуска этого программы (`clcc -i`). Подсказка по имеющимся возможностям — `clcc -h`.

Единственный её недостаток — она использует для анализа параметров командной строки функции библиотеки *Boost*, а потому её компиляция несколько осложнена подобной зависимостью. Существуют и более легковесные программы такого типа с открытым кодом:

1. ***oclc — Simple OpenCL offline compiler***
   `https://github.com/lighttransport/oclc`

2. oclc.c
   ***Simple command-line OpenCL C compiler***
   `https://gist.github.com/jrprice/abf644ef6032538655bf`

   Для того, чтобы эти программы можно было отличать друг от друга (поскольку авторами они названы одинаково), имена исполняемых файлов для них были выбраны, исходя из условных обозначений авторов (`lt-oclc` — для первой и `jrprice-oclc` — для второй).

Первая программа тоже позволяет указать, какое устройство и какой платформы использовать для компиляции ядер, а также даёт возможность получить сведения о платформах и устройствах. Дополнительно могут быть заданы опции компилятора *OpenCL* и специальные заголовочные файлы.

Более конкретно, она имеет такие опции командной строки:

| | |
|---|---|
| `-verbose` | *выдача информации о платформах/устройствах* |
| `-platform=N` | *указание конкретной платформы* |
| `-device=N` | *указание конкретного устройства* |
| `-clopt=STRING` | *опции компилятора OpenCL* |
| `-header=FILENAME` | *дополнительный заголовочный файл* |
| `-c` | *создание бинарного модуля ядра* |

Однако, надо сказать, что в случае наличия нескольких платформ *OpenCL* выдача информации выглядит не так хорошо, как в случае одной платформы. Во-первых, тогда платформы выглядят «сваленными в кучу», т.е., перечисляются без чёткого указания, где каждая начинается и заканчивается. Во-вторых, самое неприятное — это то, что устройства перечисляются только из самой первой платформы, а выбранное для компиляции устройство не сопровождается явно указанным номером выбранной платформы.

Командная строка для компиляции программы под *Windows* отдельно установленным компилятором `g++` с использованием *CUDA*-реализации *OpenCL*:

```
g++ main.cc muda_device_ocl.cc OptionParser.cpp -o lt-oclc \
    -DHAVE_OPENCL -I"%CUDA_INC_PATH%" -L"%CUDA_LIB_PATH%" -lOpenCL
```

Командная строка для компиляции программы под *Linux* может выглядеть ещё проще, учитывая тот факт, что заголовочные файлы и библиотека *OpenCL* обычно находятся в «стандартном» месте:

```
g++ main.cc muda_device_ocl.cc OptionParser.cpp -o lt-oclc \
    -DHAVE_OPENCL -lOpenCL
```

Обратите внимание на то, что для нормальной компиляции программы должна быть определена величина `HAVE_OPENCL`. Помимо этого, возможны предупреждения при компиляции на 64-разрядных системах об используемых форматах при выводе некоторых величин с помощью `printf()`. Для нормальной работы программы также имеет смысл исправить мелкие неточности в файле `muda_device_ocl.cc` в вызовах функции `clGetDeviceInfo()`: во-первых, при получении значения параметра `CL_DEVICE_VENDOR` последний параметр должен быть равен `&size_ret`, во-вторых, при запросе размера глобальной памяти с помощью `CL_DEVICE_GLOBAL_MEM_SIZE` надо использовать `sizeof(cl_ulong)` вместо `sizeof(cl_uint)`, поскольку именно таков тип переменной `uval`.

> Для того, чтобы воспользоваться опцией `-c` (создание бинарного модуля ядра), надо исправить ошибку в методе `getModule()` класса `MUDADeviceOCL`: он должен возвращать булевское значение, но `true` не возвращается никогда (!), потому что в конце метода не стоит оператор возврата (`return true;`).

Вторая программа на фоне первой выглядит ещё проще, так как у неё — помимо исходного `.cl`-файла — можно указать только номер устройства для первой попавшейся платформы. Для того, чтобы она откомпилировалась, надо в ней добавить включение заголовочного файла `cl_ext.h` (иначе не найдётся определение `CL_DEVICE_DOUBLE_FP_CONFIG`) и операции приведения типа указателей `(char *)` перед именем функции `malloc()` в двух местах. Кроме того, по её тексту можно заметить, что размер исходного файла ограничен примерно 64 килобайтами, а способ открытия этого файла таков, что исходные тексты с *Windows*-окончаниями строк нормально компилироваться не будут (надо исправить режим открытия файла на `"rb"`).

Командная строка для компиляции программы под *Windows* отдельно установленным компилятором `g++` с использованием *CUDA*-реализации *OpenCL*:

```
g++ oclc.c -o jrprice-oclc \
    -I"%CUDA_INC_PATH%" -L"%CUDA_LIB_PATH%" -lOpenCL
```

Командная строка для компиляции программы под *Linux*:

```
g++ oclc.c -o jrprice-oclc -lOpenCL
```

Сообщения об ошибках в ядрах обе программы выдают практически одинаково, что позволяет использовать их для отладочной компиляции с выдачей найденных ошибок при ограниченном наборе устройств *OpenCL*.

Более профессиональные программы для отдельной компиляции ядер присутствуют в средствах разработки от *Intel* (***ioc/ioc64***), *AMD* (***CLOC***), *ARM* (***malisc***), ...

# 8. Язык для написания ядер

Исполняемые на *OpenCL*-устройствах ядра пишутся на языке программирования ***OpenCL C*** (*OpenCL C Programming Language*), который основывается на спецификации языка *C* 1999 года (т.н. *C99*) со специфическими расширениями и ограничениями. Расширения включают новые типы данных (в том числе векторные), ключевые слова, дополнительные квалификаторы, а также набор функций (математические, геометрические, отношения, чтения/записи для векторов и др.). Ограничения касаются: работы с указателями (из-за наличия разных адресных пространств), отсутствия возможности работать с битовыми полями, массивами переменной длины, рекурсией и спецификаторами класса хранения (`extern`, `static`, `auto`, `register`). Также недоступны предопределённые идентификаторы и ставшие привычными стандартные заголовочные файлы а, стало быть, и объявляемые в них функции.

> Излагаемое далее соответствует спецификации *OpenCL 1.1* (если явно не указано иное), поскольку именно эта версия доступна для экспериментов в дисплейном классе 5-42.

## 8.1. Встроенные скалярные типы данных

Список встроенных типов скалярных данных содержит как привычные в рамках *C* типы (`bool`, `char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `void`), так и «сокращённые» варианты для беззнаковых типов (`uchar`, `ushort`, `uint`, `ulong`), а также дополнительно: `half` («облегчённый» вариант вещественных значений, занимающий 16 бит), `size_t` (целое без знака для результата операции `sizeof`), `ptrdiff_t` (целое со знаком для результата вычитания двух указателей), `intptr_t` (тип целое со знаком, такой, что указатель на `void` может быть преобразован к нему и обратно без изменений в значении указателя), `uintptr_t` (тип целое без знака, аналогичный по свойствам предыдущему). Все типы с суффиксом `_t` в именах имеют размер, определяемый значением свойства `CL_DEVICE_ADDRESS_BITS` у конкретного *OpenCL*-устройства (32 бита или 64 бита).

> Использование в коде ядер синонимов для беззнаковых типов, вероятно, объясняется тем, что для подобных имён типов из одного слова легко образовать векторные варианты этих типов: `uint2`, `uint3`, `uint4`, `uint8`, `uint16`, — аналогично `float2`, `float3`, `float4` и т.д.

Большая часть встроенных скалярных типов имеет аналоги, определённые в заголовочных файлах *OpenCL* и используемые в качестве параметров функций *API*: `cl_char`, `cl_uchar`, `cl_short`, `cl_ushort`, `cl_int`, `cl_uint`, `cl_long`, `cl_ulong`, `cl_float`, `cl_half`.

> Тип данных `half` должен соответствовать стандарту *IEEE 754-2008*. Числа этого типа используют один знаковый бит, 5 бит экспоненты и 10 бит мантиссы. Смещение экспоненты равно 15. В этом типе данных могут быть представлены обычные вещественные числа, а также денормализованные числа, бесконечности и не-числа (*NaN*).
>
> Тип данных `half` может быть использован только для объявления указателя на буфер с такими величинами. Загрузка и выгрузка этих величин по указателю должна выполняться с использованием функций `vload_half`, `vload_half`***n***, `vloada_half`***n*** и `vstore_half`, `vstore_half`***n***, `vstorea_half`***n*** (здесь ***n*** обозначает одно из чисел 2,3,4,8,16).
>
> Следует отметить, что тип `half` (`half`***n***) определён только тогда, когда поддерживается расширение `cl_khr_fp16` (подробности излагаются далее).

## 8.2. Встроенные векторные типы данных

В коде ядер поддерживаются векторные типы `char`***n***, `uchar`***n***, `short`***n***, `ushort`***n***, `int`***n***, `uint`***n***, `long`***n***, `ulong`***n***, `float`***n***, где ***n*** обозначает одно из чисел 2,3,4,8,16. Соответствующие им типы для приложений называются `cl_char`***n***, `cl_uchar`***n***, `cl_short`***n***, `cl_ushort`***n***, `cl_int`***n***, `cl_uint`***n***, `cl_long`***n***, `cl_ulong`***n***, `cl_float`***n*** (завершающее ***n*** по-прежнему обозначает 2,3,4,8 или 16).

## 8.3. Другие встроенные типы данных

Список дополнительных типов данных в коде ядер: `image2d_t` (двумерное изображение), `image3d_t` (трёхмерное изображение), `sampler_t` (т.н. сэмплер), `event_t` (событие). Подробнее типы изображения и сэмплеры обсуждаются ниже.

> **Замечание.** Первые три типа определены, если устройство поддерживает объекты изображения, т.е., значение свойства `CL_DEVICE_IMAGE_SUPPORT` равно `CL_TRUE`.

Некоторые имена типов данных пока не используются, но зарезервированы на будущее, а потому не могут использоваться как имена типов (например, приводимые выше векторные типы, для которых $n$ не есть 2,3,4,8,16, а также некоторые другие; полный список имеет смысл посмотреть в спецификации *OpenCL* в разделе *Reserved Data Types*).

> Данные, располагающиеся в памяти всегда являются выравненными на размер своего типа; для трёхкомпонентных векторов их размер и выравнивание совпадают с таковыми для четырёхкомпонентных.

## 8.4. Векторные литералы

Они могут быть использованы для создания векторов из списка скаляров, векторов или их смеси. Векторный литерал может быть инициализатором вектора или просто выражением, но не может использоваться как *l-value*.

Векторный литерал записывается как векторный тип с круглыми скобками, за которым следует список параметров в круглых скобках. Это чем-то напоминает перегруженную функцию, только варианты, допустимые здесь, — набор аргументов, в котором все аргументы имеют один и тот же тип элемента, а общее число элементов равно количеству элементов результирующего вектора. Например, для `float4` допустимы такие варианты:

```
(float4)(float, float, float, float)
(float4)(float2, float, float)
(float4)(float, float2, float)
(float4)(float, float, float2)
(float4)(float2, float2)
(float4)(float3, float)
(float4)(float, float3)
(float4)(float)
```

В последнем случае показан вариант с одним скалярным параметром и этот параметр реплицируется на весь вектор.

Примеры:

```
float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
uint4 u = (uint4)(1); // u будет равно (1, 1, 1, 1)
float4 g = (float4)((float2)(1.0f, 2.0f),(float2)(3.0f, 4.0f));
float4 h = (float4)(1.0f, (float2)(2.0f, 3.0f), 4.0f);
```

## 8.5. Компоненты векторов

Компоненты векторных типов с двумя, тремя и четырьмя компонентами достижимы с помощью имён компонент `.x`, `.y`, `.z` и `.w`. Чем больше у вектора компонент, тем больше имён используется, начиная с `.x`.

Синтаксис выбора компонент позволяет быть выбранным сразу многим компонентам — просто перечислением их после точки — даже в случае присваивания:

```
float4 c;
c.xyzw = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
c.z = 1.0f;
c.xy = (float2)(3.0f, 4.0f);
c.xyz = (float3)(3.0f, 4.0f, 5.0f);
```

Также синтаксис выбора компонент позволяет им быть переставленными или продублированными:

```
float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
float4 swiz = pos.wzyx; // swiz = (4.0f, 3.0f, 2.0f, 1.0f)
float4 dup = pos.xxyy;  // dup = (1.0f, 1.0f, 2.0f, 2.0f)
```

В случае использования обозначения группы компонент в качестве *l-value* не должно быть дублирования компонент, неправильной размерности векторов и несоответствия между числом компонент в группе и размерностью вектора справа:

```
pos.xx = (float2)(3.0f, 4.0f); // компонента 'x' использована дважды
// Недопустимо --- компонента a.xxxxxxx не является допустимым типом
x = (float16)(a.xxxxxxx, b.xyz, c.xyz, d.xyz);
// Недопустимо, т.к. имеется рассогласование между float2 и float4
pos.xy = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
```

Элементы вектора могут также адресоваться с помощью числового индекса (например, для 16-компонентного вектора можно применять в качестве индексов шестнадцатиричные цифры: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `a`, `A`, `b`, `B`, `c`, `C`, `d`, `D`, `e`, `E`, `f`, `F`), при этом числовому индексу должен предшествовать символ `s` или `S`. Перемешивать имена компонент с числовыми индексами при этом нельзя, должно использоваться что-то одно.

Векторные типы данных также могут использовать суффиксы `.lo`, `.even`, `.hi`, `.odd` для считывания частей величин векторных типов или комбинирования частей в «больший» тип. Суффикс `.lo` относится к младшей половине вектора, суффикс `.hi` — к старшей. Суффиксы `.even` и `.odd` относятся к чётным или нечётным элементам векторов соответственно.

Иллюстрирующие сказанное примеры:

```
float4 vf;
float2 low = vf.lo;      // считывается vf.xy
float2 high = vf.hi;     // считывается vf.zw
float2 even = vf.even;   // считывается vf.xz
float2 odd = vf.odd;     // считывается vf.yw
```

Ещё один пример из спецификации *OpenCL*, иллюстрирующий реализацию транспонирования матрицы $4 \times 4$ с помощью индексов и суффиксов:

```
// транспонировать матрицу 4x4
void transpose(float4 m[4])
{
    // матрица считывается в вектор float16
    float16 x = (float16)(m[0], m[1], m[2], m[3]);
    float16 t;
    // транспонирование
    t.even = x.lo; t.odd = x.hi;
    x.even = t.lo; x.odd = t.hi;
    // запись в исходную матрицу
    m[0] = x.lo.lo; //  m[0][0], m[1][0], m[2][0], m[3][0]
    m[1] = x.lo.hi; //  m[0][1], m[1][1], m[2][1], m[3][1]
```

```
    m[2] = x.hi.lo; //  m[0][2], m[1][2], m[2][2], m[3][2]
    m[3] = x.hi.hi; //  m[0][3], m[1][3], m[2][3], m[3][3]
}
```

Ну, и напоследок надо сказать, что операция взятия адреса элемента вектора приведёт к возникновению ошибки компиляции.

## 8.6. Ключевые слова

Имена, зарезервированные для использования в качестве ключевых слов в *OpenCL C*:

- ключевые слова, зарезервированные по стандарту *C99*;
- типы данных, определённые в *OpenCL C*, а также зарезервированные на будущее;
- квалификаторы адресного пространства `__global`, `__local`, `__constant`, `__private` и их варианты без подчёркиваний;
- квалификаторы функций ядра `__kernel` и `kernel`;
- квалификаторы доступа `__read_only`, `__write_only`, `__read_write` и их варианты без подчёркиваний впереди.

## 8.7. Преобразования и приведение типов

В языке *OpenCL C* между встроенными типами поддерживаются неявные преобразования типов (исключением являются типы `void` и `half` — если последний не поддерживается). Неявные преобразования между встроенными векторными типами не разрешены, а для указателей применяются правила, описанные в спецификации *C99*.

Приведение типа для встроенных скалярных типов выполняет соответствующее преобразование (кроме случаев `void` и `half`, если последний не поддерживается). Явные приведение типов между векторными типами невозможно. Преобразование скаляра к вектору может быть выполнено приведением типа скаляра к желательному векторному типу. При этом выполняются и необходимые арифметические преобразования. Для преобразования к встроенному целочисленному векторному типу будет использоваться округление к нулю, для преобразования к вещественным векторным типам — текущий режим округления. Приведение булевской величины к целочисленному вектору создаст вектор с компонентами, равными $-1$ (все биты установлены), если булевская величина имеет истинное значение, или 0 — в противном случае.

```
float f = 1.0f;
float4 va = (float4)f;
// va -- вектор типа float4 с элементами (f, f, f, f)
uchar u = 0xFF;
float4 vb = (float4)u;
// vb -- такой же вектор с элементами ((float)u,(float)u,(float)u,(float)u)
float f = 2.0f;
int2 vc = (int2)f;
// vc -- вектор типа int2 с элементами ((int)f, (int)f)
uchar4 vtrue = (uchar4)true;
// vtrue -- вектор типа uchar4 с элементами (0xff, 0xff, 0xff, 0xff)
```

Явные преобразования типов осуществляются набором функций с именами, подобными `convert_<`*ТипРезультата*`>(<`*Тип*`>)` для поддерживаемых типов — за исключением `bool`, `half`, `size_t`, `ptrdiff_t`, `intptr_t`, `uintptr_t` и `void`. При этом количество элементов в исходном и результирующем векторах должно быть одинаковым.

```
uchar4 u;
int4 c = convert_int4(u);
float f;
int i = convert_int(f);
```

Поведение преобразования типов может быть модифицировано одним или двумя (необязательными) модификаторами, указывающими насыщение для выходящих за границы диапазона величин и округляющее поведение. Полная форма скалярной функции преобразования:

<*ТипРезультата*> convert_<*ТипРезультата*>[_sat][<*_РежимОкругления*>](<*Тип*>)

Полная форма векторной функции преобразования:

<*ТипРезультата*> convert_<*ТипРезультата*>n[_sat][<*_РежимОкругления*>](<*Тип*>)

Модификаторы режимов округления: `_rte` (к ближайшему чётному), `_rtz` (к нулю), `_rtp` (к плюс бесконечности), `_rtn` (к минус бесконечности).

По умолчанию (при отсутствии модификатора округления) применяется округление к нулю при преобразовании к целому значению и текущий режим округления — при преобразовании к вещественному значению. В настоящее время, правда, поддерживается только один режим округления к вещественным значениям — округление к ближайшему чётному.

Преобразование к целому в режиме насыщения ограничивает значения выходящих за разрешённый диапазон величин ближайшими представимыми из диапазона. Не-числа (*NaNs*) превращаются в 0.

```
short4 s;
// отрицательные величины "схлопываются" в 0
ushort4 u = convert_ushort4_sat(s);
// величины, большие CHAR_MAX, становятся CHAR_MAX
// величины, меньшие CHAR_MIN, становятся CHAR_MIN
char4 c = convert_char4_sat(s);
```

## 8.8. Операторы

Арифметические операции сложения, вычитания, умножения и деления применимы к целым и вещественным скалярным величинам, а также к векторным. Взятие остатка (%) применимо только к целым скалярным и векторным величинам. Все арифметические операции дают результат того же встроенного типа, что и типы операндов — возможно, после преобразования типа. Если оба операнда — скаляры, результат операции — скаляр. Если один из операндов скаляр, а другой — вектор, то скаляр «расширяется» к вектору и операция осуществляется покомпонентно, давая вектор того же размера. Если оба операнда — векторы (должны быть одного типа), то результат получается покомпонентной операцией и даёт вектор такого же размера.

Деление целых, приводящее к величинам за пределами представимого диапазона, не генерирует исключительную ситуацию и даёт неопределённый результат; это же справедливо и для деления на 0 для целых. Для вещественных величин деление на 0 даёт либо плюс или минус бесконечность, либо *NaN* — как это предписывается стандартом *IEEE-754*.

Следует учитывать, что результаты операций над операндами со знаком и без знака приводят к результатам со знаком; операции с логическим результатом для скаляра дают 0 в случае `false`-результата и 1, если результат — `true`. Для векторных типов получается 0, если результат — `false`, и $-1$ (т.е., все биты установлены), если результат — `true`.

Оператор `sizeof` выдаёт величину операнда в байтах — включая дополнительные байты, необходимые для правильного выравнивания. Результат определяется типом операнда и имеет специальный тип `size_t`.

Поведение `sizeof` применительно к величинам типов `bool`, `image2d_t`, `image3d_t`, `sampler_t` и `event_t` не стандартизовано и зависит от реализации.

К типу `half` никакие операции кроме `sizeof` неприменимы.

Векторные операции осуществляются покомпонентно; скалярные операнды приводятся к векторам.

## 8.9. Квалификаторы

### Квалификаторы адресного пространства

Поскольку *OpenCL* определяет раздельные адресные пространства, обозначаемые квалификаторами `__global`, `__local`, `__constant`, `__private`, указатели из одного адресного пространства не могут быть приведены к указателям из другого адресного пространства. Локальные переменные и аргументы функций располагаются в адресном пространстве `__private`. Параметры функции-ядра, являющиеся указателями, могут указывать только на адресные пространства `__global`, `__local` или `__constant`. Аргументы функций, имеющие тип `image2d_t` или `image3d_t`, ссылаются на объекты памяти в адресном пространстве `__global`. Все переменные из области действия программы-ядра должны быть объявлены в адресном пространстве `__constant`.

Квалификатор `__local` используется для описания переменных, разделяемых всеми рабочими единицами в рабочей группе. Указатели на адресное пространство `__local` допустимы в качестве параметров функций, включая функции-ядра. Переменные, объявляемые в адресном пространстве `__local`, должны появляться в блоке функции-ядра, но не во вложенных блоках. Они выделяются для каждой рабочей группы, исполняющей ядро, и существуют только во время исполнения этого ядра рабочей группой; инициализировать их нельзя.

Адресное пространство `__constant` используется для описания переменных в глобальной памяти, доступных из ядра всех рабочим единицам во время его исполнения, — но только для чтения. Такие переменные должны инициализироваться и величины, используемые для инициализации, должны быть константами времени компиляции.

Переменные в функции ядра, не помеченные никаким квалификатором адресного пространства, считаются находящимися в адресном пространстве `__private`. Переменные, объявленные как указатели, считаются указывающими на пространство `__private`, если при них нет квалификатора адресного пространства.

### Квалификаторы доступа

Объекты изображений, указываемые в качестве параметров функции ядра, могут быть объявлены доступными только для чтения (`__read_only`, `read_only`) или только для записи (`__write_only`, `write_only`). По умолчанию объекты изображений являются доступными только для чтения.

### Квалификатор функций `__kernel`

Этот квалификатор объявляет функцию ядром, которое исполняется приложением на *OpenCL*-устройствах. Ядро может быть вызвано также другим ядром как обыкновенная функция. Ядра, имеющие переменные из адресного пространства `__local`, могут быть вызваны с хоста с помощью функций `clEnqueueNDRangeKernel()` и `clEnqueueTask()`.

Исходя из вводного характера данного курса, в нём не рассматриваются дополнительные квалификаторы атрибутов и атрибуты вообще, а также препроцессорные директивы и макроопределения...

Кроме того, функции работы с изображениями – хотя по смыслу и должны быть упомянуты при обсуждении языка описания ядер – здесь тоже не упоминаются, а частично рассматриваются в разделе 13.Объекты памяти и работа с ними в *OpenCL 1.1* (см. стр.52).

## 8.10. Встроенные функции

### Функции для работы с рабочими единицами

Напомним, что каждая рабочая единица в *OpenCL* является частью 1-, 2-, 3-мерной рабочей группы, которая, в свою очередь, — часть 1-, 2-, 3-мерного набора рабочих групп (*NDRange*). Для идентификации рабочих единиц в исходном коде ядер используются различные функции, позволяющие получать их индексы, а также размеры как рабочих групп, так и всего их набора по различным имеющимся измерениям.

`uint get_work_dim ()`
: Используемое число измерений, величина параметра `work_dim` при вызове функции `clEnqueueNDRangeKernel()`.

`size_t get_global_size (uint dimindx)`
: Количество глобальных рабочих единиц по заданному измерению `dimindx`.

`size_t get_global_id (uint dimindx)`
: Уникальный глобальный индекс рабочей единицы по заданному измерению `dimindx`.

`size_t get_local_size (uint dimindx)`
: Количество локальных рабочих единиц по заданному измерению `dimindx`.

`size_t get_local_id (uint dimindx)`
: уникальный локальный индекс рабочей единицы по заданному измерению `dimindx`.

`size_t get_num_groups (uint dimindx)`
: Количество рабочих групп, исполняющих ядро, по заданному измерению `dimindx`.

`size_t get_group_id (uint dimindx)`
: Идентификатор рабочей группы по заданному измерению `dimindx`.

`size_t get_global_offset (uint dimindx)`
: Величина смещения, указанная в параметре `global_work_offset`, — по заданному измерению `dimindx` — при вызове функции `clEnqueueNDRangeKernel()`.

### Математические и некоторые другие функции

Несмотря на то, что в *OpenCL C* нет заголовочного файла `math.h`, большая часть функций, упоминаемых в нём (плюс ещё некоторые), доступны в коде ядер, причём они принимают в качестве параметров часто не только скалярные (`float`), но и векторные величины (`float`$n$, где $n$ = 2,3,4,8,16). Кроме того, имеются варианты некоторых из этих функций с префиксами `half_` и `native_`: первые имеют ограниченную точность (16 бит) и малый диапазон (но работают быстрее); вызовы вторых отображаются в инструкции конкретного устройства, давая максимальный выигрыш в скорости — возможно, за счёт точности.

Среди других функций: ***целые*** (`abs`, `abs_diff`, `add_sat`, `clz`, `hadd`, `mad_hi`, `mad_sat`, `max`, `min`, `mul_hi`, `rhadd`, `rotate`, `sub_sat`, `upsample` и две «быстрые» — `mad24` и `mul24`); ***общие*** (`clamp`, `degrees`, `max`, `min`, `mix`, `radians`, `sign`, `step`, `smoothstep`); ***геометрические*** (`cross`, `dot`, `distance`, `length`, `normalize` и `fast_`-варианты трёх последних); функции ***отношения*** (работают со скалярными и векторными встроенными типами и формируют скалярный или векторный целочисленный результат со знаком); функция ***синхронизации*** `barrier` (все рабочие единицы в группе должны вызвать эту функцию — прежде чем любой из них будет разрешено продолжить исполнение кода после этой функции); функции ***загрузки*** `vload`$n$ и ***выгрузки*** `vstore`$n$ (позволяют читать и записывать векторные типы данных по указателю памяти), а также их варианты с суффиксом `_half` и с дополнительными суффиксами режима округления и некоторые другие.

Более детальную информацию о них имеет смысл смотреть в спецификациях *OpenCL* (*1.0*, *1.1*, *1.2*, *2.0*, *2.1*, *2.2*).

# 9. Заголовочные файлы *OpenCL*

Комплект заголовочных файлов, необходимых для написания программ с использованием *OpenCL*, можно найти на сайте `https://www.khronos.org/registry/OpenCL/` консорциума *Khronos*. Указанная страница содержит ссылки на спецификации: функций программного интерфейса; языка, используемого для написания ядер (*OpenCL C*); *OpenCL*-расширений (как одобренных производителями, так и самим консорциумом), — а также ссылки на различные версии заголовочных файлов. В системах *Ubuntu* или *Debian* заголовочные файлы *OpenCL* можно также установить с помощью команды

```
sudo apt-get install opencl-headers
```

Если на компьютере установлен *CUDA SDK*, то заголовочные файлы *OpenCL* там уже есть.

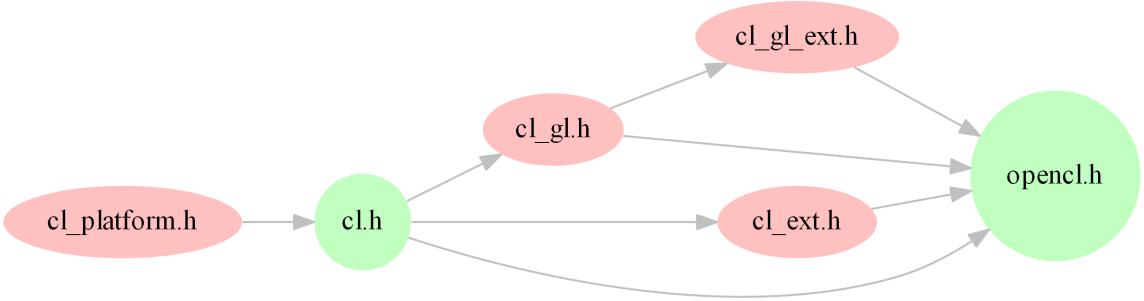
В зависимости от версии *OpenCL* в комплект заголовочных файлов входит от 7 до 10 файлов. Мы оставим в стороне те из них, что имеют отношение к *Direct3D*: `cl_d3d10.h`, `cl_d3d11.h`, `cl_dx9_media_sharing.h` — здесь они вообще не рассматриваются. Из оставшихся наиболее часто используется один — залоговочный файл `cl.h`, поскольку именно в нём описано подавляющее большинство функций программного интерфейса (см. таблицу). Остальные файлы понадобятся, если работать в программе с расширениями *OpenCL* (*extensions*, см. стр.42) и/или организовывать взаимодействие с *OpenGL* (см. стр.63) для воспроизведения графической информации. Об этом чуть подробнее мы будем говорить позднее — в соответствующих разделах.

| **Версия *OpenCL*** | ***1.0*** | ***1.1*** | ***1.2*** | ***2.0*** | ***2.1*** | ***2.2*** |
|---|---|---|---|---|---|---|
| Файлы | Количество функций API | | | | | |
| `cl.h` | 66 | 74 | 88 | 101 | 108 | 110 |
| `cl_egl.h` | – | – | 4 | 4 | 4 | 4 |
| `cl_ext.h` | 5 | 11 | 24 | 25 | 25 | 25 |
| `cl_gl.h` | 9 | 9 | 10 | 10 | 10 | 10 |
| `cl_gl_ext.h` | 0 | 1 | 1 | 1 | 1 | 1 |
| `cl_platform.h` | 0 | 0 | 0 | 0 | 0 | 0 |
| `opencl.h` | 0 | 0 | 0 | 0 | 0 | 0 |

Дополняет этот комплект файл `cl.hpp` (в последних версиях *OpenCL* он постепенно заменяется файлом `cl2.hpp`), поскольку в нём реализуется *C++*-интерфейс к функциям *API*; особенности работы с ним рассматриваются в следующем разделе (см. стр.35).

Стандартным местом расположения этих файлов является подкаталог `CL` в каталоге, где располагаются заголовочные файлы вообще (в *Mac OS* подкаталог имеет немного другое название: `OpenCL`); для реализации *OpenCL* в рамках *CUDA* — это местоположение заголовочных файлов *CUDA*, поэтому при компиляции *OpenCL*-программы (под *Windows*) там достаточно включить в путь поиска заголовочных файлов путь `%CUDA_INC_PATH%` (при компиляции с командной строки) или `$(CUDA_INC_PATH)` (при компиляции из среды *Visual Studio*). При компиляции под *Linux* путь к заголовочным файлам часто можно не указывать вообще, поскольку они почти всегда расположены там, где компилятор ищет их по умолчанию (каталоги `/usr/include`, `/usr/local/include`). Отметим, что подкаталог `CL` (или `OpenCL` для *Mac OS*) указывается прямо в тексте программы в директиве препроцессора.

Взаимосвязь рассматриваемых заголовочных файлов `cl.h`, `cl_platform.h`, `cl_ext.h`, `cl_gl.h`, `cl_gl_ext.h`, `opencl.h` (здесь они перечислены в порядке важности для нас) иллюстрируется диаграммой



и взаимозависимость файлов показана рёбрами направленного графа, а сами файлы — узлами графа. Направление ребра в этом графе означает включение (в смысле директивы `#include`) файла в другой файл. Видно, что самый часто используемый заголовочный файл `cl.h` включает в себя только файл `cl_platform.h` (т.е., зависит только от него). Все остальные заголовочные файлы всегда подключают `cl.h`, а файл `opencl.h` вообще подключает все остальные файлы. Таким образом, резюмируя, можно сказать, что для программ, которые не используют никаких расширений и взаимодействия с *OpenGL*, необходимыми являются только два файла: `cl.h` и `cl_platform.h`, причём директивы препроцессора (`#include <CL/cl.h>`) достаточно одной, поскольку `cl_platform.h` подключается автоматически. Если же программы используют явную работу с расширениями и/или должны взаимодействовать с *OpenGL*, то проще сразу пользоваться заголовочным файлом `opencl.h`; его одного будет достаточно для компиляции программ.

## 9.1. Содержимое файла `cl.h`

За время существования *OpenCL* заголовочные файлы (и особенно `cl.h`) подвергались многочисленным изменениям: это были как дополнения (в каждой версии появлялись какие-то новые функции интерфейса), так и уточнения или исправления выявленных ошибок. Поэтому, приступая к программированию в рамках *OpenCL*, надо чётко представлять себе, какие заголовочные файлы имеются в используемой системе и не конфликтуют ли они в случае непреднамеренного дублирования — а такое возможно, если одновременно в системе имеется несколько реализаций *OpenCL* (о реализациях — см. стр.48). Можно было бы различать версии заголовочных файлов, если бы их ревизия была указана прямо в них, но, к сожалению, именно в случае `cl.h` это и не работает, т.к. в более новых его версиях на сайте консорциума никаких сведений о ревизии уже нет (хотя во многих других файлах — есть).

Пример весьма «живучей» ошибки в рассматриваемом файле (проверено на момент завершения пособия в сентябре 2017 года) — отсутствие пометки `CL_CALLBACK` при указателе на функцию обратного вызова в прототипе довольно редко используемой интерфейсной функции `clEnqueueNativeKernel()`: в файле `cl.h` для версии *OpenCL 1.0* она имеется до сих пор![2].

Излагаемое здесь опирается на содержимое файла `cl.h` для *OpenCL 1.1* (ревизия 11985).

Начинается файл со стандартного способа «избегания повторного включения» этого файла, когда при первом включении файла препроцессором определяется специальное уникальное для этого файла символьное имя, проверяемое на наличие при повторных включениях и исключающее повторное включение содержимого, если это символьное имя уже определено.

В самом начале файла осуществляется также подключение файла `cl_platform.h` — по-разному, в зависимости от системного окружения (*Mac OS* или другие системы), а всё последующее помечено как `extern "C"` — не подвергаемое используемому в языке *C++* изменению имён (т.н. ***mangling***, в переводе — искажение), когда, например, в имена функций включаются и типы их параметров.

С помощью `typedef` в файле определены синонимы типов для используемых в *OpenCL* объектов: для идентификаторов платформ и устройств (`cl_platform_id`, `cl_device_id`), для контекстов (`cl_context`), командных очередей (`cl_command_queue`), объектов памяти (`cl_mem`), объектов программ (`cl_program`), объектов ядер (`cl_kernel`), событий (`cl_event`) и сэмплеров (`cl_sampler`). Видно, что всё перечисленное — это указатели на соответствующие структуры данных, без конкретизации их строения.

Также с помощью `typedef` определены более конкретные типы величин, основанные на общих типах, определяемых в рамках *OpenCL* — в файле `cl_platform.h`.

Стоит обратить внимание на замечание, располагающееся там в комментарии: типу `cl_bool` не гарантируется тот же размер, что и `bool`, используемому в тексте ядер!

Кроме того, определены две структуры данных, описывающие форматы изображений в *OpenCL* (`cl_image_format`; состоит из двух `cl_int`-значений, определяющих каналы и

[2] Вероятно, потому, что этим файлом уже давно никто не пользуется.

порядок их следования в изображении) и область буфера (`cl_buffer_region`; состоит из двух значений типа `size_t`, задающих начало и размер области в буфере).

Далее следует довольно длинный список символических имён констант, соответствующих возможным типам ошибок, возвращаемых различными функциями программного интерфейса в разных ситуациях; все эти константы — не очень большие отрицательные числа (код успешного завершения `CL_SUCCESS` равен нулю).

Важными символьными именами являются имена вида `CL_VERSION_X_Y` (где `X` — старший номер версии, а `Y` — младший): чем выше версия *OpenCL*, тем больше символьных имён такого вида определено в заголовочном файле; тем самым они как бы показывают, какие версии поддерживаются этим заголовочным файлом. Например, в файле `cl.h` для версии *1.1* определены имена `CL_VERSION_1_0` и `CL_VERSION_1_1`, а для версии *2.1* — ещё и `CL_VERSION_1_2`, `CL_VERSION_2_0` и `CL_VERSION_2_1`.

Среди прочего, там также определены константы, задающие различные виды информации: о платформе, об устройстве, об очередях команд, об объектах памяти, об изображениях, о сэмплерах, о программах и их построении и статусе, о ядрах и их параметрах, о квалификаторах адресов и доступа для этих параметров, о рабочих группах и подгруппах, о событиях, о командах профилирования; битовые поля для типов устройств, возможностей устройств с плавающей точкой, для исполнительных возможностей, свойств очереди команд, флагов памяти и отображения; константы возможных типов каналов в изображениях и их порядков следования, типов кэша и памяти, режимов фильтрации и адресации и т.д.

Остаток файла содержит прототипы поддерживаемых в рамках соответствующей версии *OpenCL* функций программного интерфейса. Каждый прототип функции — помимо типов параметров — содержит краткие комментарии о смысле этих параметров, а также дополнительную информацию в сопровождающих прототип «пометках», часть из которых — это ключевые слова языка, а часть — макроопределения, имеющие в разных случаях различные (иногда — пустые) значения.

Пометка `extern`, как обычно, говорит о том, что функции не присутствуют в нашей (компилируемой с помощью данного заголовочного файла) программе (все эти функции располагаются в динамических библиотеках, реализующих *OpenCL*). Можно также заметить, что, помимо типа возвращаемого значения функций (как правило, это `cl_int` — за исключением функций, создающих объекты *OpenCL* и возвращающих их), все прототипы функций снабжены в начале пометками `CL_API_ENTRY` и `CL_API_CALL`, а в конце — пометками `CL_API_SUFFIX__VERSION_1_0` или `CL_API_SUFFIX__VERSION_1_1` (для заголовочного файла *OpenCL* версии *1.1*). Первая из начальных пометок, как можно увидеть по содержимому файла `cl_platform.h`, всегда пуста, а потому не играет никакой особой роли; вторая пометка имеет значение лишь при компиляции в рамках системы *Windows*, когда она задаёт конкретное соглашение о способе вызова функций и типе передачи параметров (`__stdcall`).

> `__stdcall` (а также `__cdecl`) — это соглашения о вызовах, принятые в рамках *Win32*. Соглашение `__stdcall` предполагает, что вызываемая сторона ответственна за очистку стека от переданных параметров, и традиционно используется в вызовах *Windows API*. Соглашение `__cdecl` используется многими *C*-компиляторами для архитектуры x86; в нём очищение стека от параметров осуществляется вызывающей стороной. Кроме того, эти соглашения используют противоположный друг другу порядок расположения параметров в стеке. Поэтому вызываемая и вызывающая сторона должны согласованно подходить к вызову функций между собой. Функции программных интерфейсов и обратного вызова вызываются системой, а она ожидает при вызове, что будет использоваться соглашение `__stdcall`, в то время как компилятор по умолчанию реализует соглашение `__cdecl`, из-за чего без пометки `__stdcall` ни функции в динамических библиотеках, ни функции обратного вызова (***callback***) не могут быть правильно вызваны.

Пометки, располагающиеся в конце прототипа (`CL_API_SUFFIX__...`), как оказывается (см. содержимое файла `cl_platform.h`), имеют какой-то смысл лишь в случае системы *Mac OS*; для нас они носят скорее справочный характер, позволяя понять, в какой

версии *OpenCL* появилась соответствующая функция программного интерфейса. Кроме того, там же возможна пометка `CL_EXT_SUFFIX__VERSION_1_0_DEPRECATED`, «не рекомендующая» функцию к использованию в последующих версиях.

> В заголовочных файлах последних версий можно найти пометки `CL_API_SUFFIX__VERSION_1_2`, `CL_API_SUFFIX__VERSION_2_0` и `CL_API_SUFFIX__VERSION_2_1`, а также и совершенно новые: `CL_EXT_SUFFIX__VERSION_1_1_DEPRECATED` и `CL_EXT_SUFFIX__VERSION_1_2_DEPRECATED`, помечающие функции, не рекомендуемые к использованию.

Обратите внимание, что в самом конце файла `cl.h` имеется прототип весьма важной функции `clGetExtensionFunctionAddress()`: она используется для получения адреса функции расширения по её имени.

## 9.2. Содержимое файла `cl_platform.h`

Опять можно видеть стандартный способ «избегания повторного включения», а также пометку всего содержимого файла с помощью `extern "C"`, что предотвращает искажения имён, которые возможны при использовании *C++*-компилятора. Кроме этого, в начале файла даны определения величин, используемых в других заголовочных файлах при прототипах интерфейсных функций.

Видно также, кстати, что `cl_platform.h` всё-таки не является самодостаточным файлом, потому что включает в себя и другие — но системные — заголовочные файлы, а потому нам не нужно заботиться об их наличии. Имеет место подобное включение в случае конкретной системы (для *Mac OS* определено значение `__APPLE__`, для *Windows* и компилятора *Microsoft* имеются, соответственно, значения `_WIN32` и `_MSC_VER`), либо специфических возможностей системы команд основного процессора (`__SSE__`, `__SSE2__`, `__MMX__`, `__AVX__`; подобные варианты могут быть заданы должным образом в командной строке компилятора), или при использовании определённого компилятора (`__MINGW64__`, `__GNUC__`, `__STRICT_ANSI__`). В каждом конкретном случае используются свои встроенные типы данных и свои способы их выравнивания в памяти (если предусмотрены), а на их основе определяются сначала встроенные скалярные типы данных (`cl_char`, `cl_uchar`, `cl_short`, `cl_ushort`, `cl_int`, `cl_uint`, `cl_long`, `cl_ulong`, `cl_half`, `cl_float`, `cl_double`), а потом и векторные. Каждый векторный тип данных задан с помощью объединения, содержащего не только исходные мельчайшие компоненты типа, но и представления этого типа в виде более «крупных» векторных величин. Типичный пример — определение векторного типа `cl_long8`:

```
typedef union
{
    cl_long   CL_ALIGNED(64) s[8];
#if defined( __GNUC__) && ! defined( __STRICT_ANSI__ )
   __extension__ struct{ cl_long  x, y, z, w; };
   __extension__ struct{ cl_long  s0, s1, s2, s3, s4, s5, s6, s7; };
   __extension__ struct{ cl_long4 lo, hi; };
#endif
#if defined( __CL_LONG2__)
    __cl_long2     v2[4];
#endif
#if defined( __CL_LONG4__)
    __cl_long4     v4[2];
#endif
#if defined( __CL_LONG8__ )
    __cl_long8     v8;
#endif
} cl_long8;
```

Этот тип (напомним: в хост-программе!) изначально представляет из себя массив из восьми величин типа `cl_long`, выравненный на границу 64 байт. У этого массива даже есть имя (`s`), поэтому каждый элемент массива может адресоваться (помимо способов, аналогичных адресации компонент в коде ядер из спецификации *OpenCL*; см. далее) также как `.s[i]`, где `i` – значение из диапазона 0..7. Можно представлять себе величину такого типа как структуру из восьми элементов с именами `s0`, `s1`, `s2`, `s3`, `s4`, `s5`, `s6`, `s7`, а, стало быть, адресовать эти элементы как `.s0`, `.s1`, `.s2`, `.s3`, `.s4`, `.s5`, `.s6`, `.s7` соответственно; можно именовать первые четыре элемента и как `.x`, `.y`, `.z`, `.w`; можно представлять такую величину состоящей из двух «половинок» типа `cl_long4` с именами `lo`, `hi`.

В случае наличия у типа данных «родного» (*native*) представления его «частей» (в данном примере — будут определены имена `__CL_LONG2__`, `__CL_LONG4__`, `__CL_LONG8__`), обращение к этим частям возможно: как к целому — через имя `v8`, к «половинкам» — как к массиву `v4` из двух компонент, к «четвертинкам» — как к массиву `v2` из четырёх компонент.

> Предваряющее анонимные структуры ключевое слово `__extension__` — это способ избежать предупреждений компилятора *GNU C*; никакого другого — особого — смысла оно не имеет. Одновременно можно заметить, что только для этого компилятора может использоваться адресация компонент векторных типов через имена `.xyzw`, `.s0123...{f|F}` и `.hi/.lo` (и возможность подобной адресации может быть проверена во время компиляции программы по наличию макро-величин `CL_HAS_NAMED_VECTOR_FIELDS` и `CL_HAS_HI_LO_VECTOR_FIELDS`) — напомним, что речь идёт о содержимом хост-программы, а не коде ядра, где подобная возможность гарантирована спецификацией.

И ещё. Векторные типы с шестнадцатью компонентами демонстрируют способ достижения вариативности имён старших компонент — путём параллельно объявленных структур с разными именами полей в рамках одного объединения. А малозаметные строки в тексте файла с определениями вида

```
typedef cl_<СкалярныйТип>4 cl_<СкалярныйТип>3;
```

способны многое разъяснить по поводу типов данных из трёх компонент...

## 9.3. Содержимое файлов `cl_ext.h`, `cl_gl.h`, `cl_gl_ext.h`

Говоря коротко, перечисленные файлы описывают, соответственно: расширения без внешних зависимостей, одобренные *Khronos* расширения, имеющие зависимости от *OpenGL*, а также расширения производителей, имеющие зависимости от *OpenGL*. Там определены необходимые величины и типы данных, а также функции интерфейса. Подробнее к содержимому этих файлов имеет смысл вернуться при рассмотрении взаимодействия *OpenCL* и *OpenGL* (см. стр.63).

> В зависимости от «источника» заголовочных файлов *OpenCL* (т.е., установлены ли эти файлы вместе со средствами разработки от изготовителя или взяты с сайта консорциума *Khronos*) возможны некоторые различия между ними. Это касается в первую очередь файлов, описывающих расширения. Поэтому при работе с несколькими реализациями *OpenCL* на компьютере важно не пользоваться для компиляции программ «чужими» заголовочными файлами, а стараться применять файлы консорциума (если, конечно, в них есть информация о необходимых расширениях), либо употреблять «родные» заголовочные файлы от изготовителя.

## 9.4. Содержимое файла `opencl.h`

Оно весьма тривиально, поскольку кроме уже рассмотренных выше обязательных фрагментов там содержатся лишь подключения всех основных заголовочных файлов (осуществляемые по-разному — в зависимости от используемой платформы). Тем не менее, там имеется также и ревизия файла — неоправданно «высокая» для столь незамысловатого содержимого...

# 10. Содержимое заголовочного файла `cl.hpp` версии *1.1*

Сразу после пространных комментариев к файлу (включающих, помимо прочего, даже текст примерной программы с использованием *C++*-интерфейса) можно увидеть обычные для любого заголовочного файла фрагменты: избегание повторного включения, а также избирательное подключение других необходимых в каких-то ситуациях заголовочных файлов. Стоит обратить внимание на проверки наличия двух (явно задаваемых внешне) величин: `__NO_STD_VECTOR` и `__NO_STD_STRING`. Если какая-то из величин не определена, подключаются стандартные определения векторов или строк соответственно (из библиотеки *STL*); если же их определить, то будут задействованы присутствующие в файле простые классы для их замены.

Всё, что определяется дальше, включено в пространство имён `cl`, поэтому при написании программ либо понадобится декларация погружения этого пространства имён в глобальную область видимости (с помощью `using namespace cl;`), либо придётся добавлять префиксы для всех объектов/имён/функций из рассматриваемого заголовочного файла.

Если в программе должны быть разрешены исключения (это достигается одной строкой с директивой препроцессора, употребляемой до включения файла `cl.hpp`):

```
#define __CL_ENABLE_EXCEPTIONS
#include <CL/cl.hpp>
. . .
```

то добавляется определение класса *OpenCL*-ошибок, основанного на стандартных исключениях `std::exception`; каждый его объект дополнительно хранит код *OpenCL*-ошибки и сопутствующую строку сообщения. В случае использования исключений применяется «стрингизация» имён функций с помощью макроопределения (`#define __ERR_STR(x) #x`).

Далее следует «собственное» для рассматриваемого файла определение класса строк `string`. Обобщённое имя, ссылающееся на используемое определение типа для строк — `STRING_CLASS`, оно определено с помощью `typedef`. Затем следует «собственное» определение класса `vector`, содержащее, как и полагается, внутренний класс `iterator`; обобщённое имя типа для ссылки на используемый вид вектора — `VECTOR_CLASS`, которое задано на сей раз с помощью директивы препроцессора (`#define`).

Если используется «собственное» определение `vector`, то все векторы будут одного из статически выбранных размеров или иметь размер `__MAX_DEFAULT_VECTOR_SIZE` по умолчанию. На основе этого класса определён шаблонный тип «размеров» — для интерфейса между вызовами *C++* и *OpenCL C*, где нужны массивы величин фиксированного размера:

```
template <int N>
struct size_t : public cl::vector< ::size_t, N> { };
```

### Вложенное пространство имён `detail`

Следующая далее часть заголовочного файла помещена во вложенное пространство имён `detail`, в рамках которого определены некоторые вспомогательные шаблонные классы и довольно внушительные (и замысловатые) макроопределения для описания разнообразных параметров *OpenCL*-объектов.

Объекты типа `GetInfoHelper<>` (на основе задаваемого функтора и имени типа) будут обеспечивать возвращение значения поименованного параметра с помощью статического метода `get()`, который, по сути, сведётся к вызову соответствующей функции *OpenCL*. Имеются специализации шаблонного типа `GetInfoHelper<>` для типов `VECTOR_CLASS<T>`, `STRING_CLASS`, `VECTOR_CLASS<char *>` (причём для первых двух используются двойные вызовы функций *OpenCL*: сначала для определения размера, затем — после выделения необходимой памяти — для получения значения); их дополняет макроопределение `__GET_INFO_HELPER_WITH_RETAIN()`, разворачиваемое в разные специализации (тоже в пространстве имён `detail`) с заданным *C++*-типом данных. Отличие этих специализаций от явно перечисленных выше в том, что

после вызова функции *OpenCL* вызывается ещё метод `retain()` обработчика ссылок на тип-синоним передаваемого макроопределению параметра `CPP_TYPE` в строке

```
return ReferenceHandler<CPP_TYPE::cl_type>::retain((*param)());
```

и методу передаётся величина дескриптора (т.е., `(*param)()`), а результат этого вызова является возвращаемым значением метода `get()` (если же при вызове функции *OpenCL* произошла ошибка, то её код будет возвращаемым значением метода `get()`).

Макроопределение `__PARAM_NAME_INFO_1_0` (и дополнения к нему: `__PARAM_NAME_INFO_1_1` и `__PARAM_NAME_DEVICE_FISSION`) имеют всего по одному параметру, но везде это — имя ещё одного макроопределения (`__DECLARE_PARAM_TRAITS`), в котором из трёх передаваемых ему параметров `token`, `param_name`, `T` препроцессор формирует предварительное объявление структуры `token` и специализацию шаблонного типа `param_traits<detail::token, param_name>`, где в результате возникают локальные определения некоторого перечислимого типа с именем `value` и значением `param_name` (а это — некоторая препроцессорная константа из этого же файла), а также типа-синонима для типа `T` с именем `param_type`. В конце концов все эти макроопределения будут развёрнуты в многочисленные специализации `param_traits<>`, соответствующие перечисленным для каждого параметра *OpenCL* данным: групповому наименованию, условному идентификатору значения параметра и его типу (который может быть как встроенным, так и объявленным в данном файле — вроде `cl::Memory`).

Шаблонные функции `getInfo<>()` (со вспомогательными функторами[3] `GetInfoFunctor0` и `GetInfoFunctor1`) дают унифицированный интерфейс для извлечения самых разнообразных параметров *OpenCL*-объектов и в свою очередь вызывают — помимо специфических действий в разных специализациях — статический метод `get()` из конкретной специализации шаблонного класса `GetInfoHelper<>`; далее они нужны в *C++*-реализациях объектов *OpenCL*.

Шаблонный класс «ОбработчикСсылки» (`ReferenceHandler<>`) определён «пустым», поскольку вся его функциональность сосредоточена в специализациях, содержащих пару статических методов: `retain()` и `release()`. Такие специализации имеются для типов ссылок на устройства (`cl_device_id`), на платформы (`cl_platform_id`), на контекст (`cl_context`), на командную очередь (`cl_command_queue`), на объект памяти (`cl_mem`), на сэмплер (`cl_sampler`), на объект программы (`cl_program`), на объект ядра (`cl_kernel`), на объект события (`cl_event`). Но для первых двух методы `retain()` и `release()` возвращают ошибки (`CL_INVALID_DEVICE` и `CL_INVALID_PLATFORM`, соответственно, — поскольку эти ресурсы не освобождаются, а потому и не могут быть «удержаны»), а для всех остальных — результаты вызовов соответствующих функций *OpenCL* (`clRetain`*`Объект`*`()` и `clRelease`*`Объект`*`()`).

Шаблонный тип-«обёртка» `Wrapper<T>` вокруг заданного типа `T` разворачивается в определение класса вместе с определением синонима `cl_type` для типа `T` и (защищённым) содержимым объекта этого класса в виде дескриптора `object_` заданного типа `cl_type`. Помимо конструкторов/деструктора в этом классе есть публичные оператор присваивания и оператор вызова и защищённые методы `retain()` и `release()`, вызывающие далее статический метод `retain()` или `release()` соответствующего класса `ReferenceHandler<cl_type>` с параметром, равным «своему» дескриптору. А защищённые методы класса вызываются: в конструкторе при инициализации дескриптором — `retain()`, в деструкторе — `release()`, в методе реализации оператора присваивания — сначала `release()`, затем `retain()`. Оператор вызова просто возвращает дескриптор.

Говоря проще, такой «обёртывающий» тип добавляет к хранению дескриптора (когда это возможно) необходимые вызовы *OpenCL*-функций удержания/освобождения объектов *OpenCL* с таким дескриптором.

Завершается пространство имён `detail` одним из определений статической `inline`-функции `errHandler()`: в случае, если разрешены исключения (`__CL_ENABLE_EXCEPTIONS`), возбуждается исключительная ситуация (с кодом ошибки и строкой описания — если таковая передаётся), если нет, то просто возвращается переданный код ошибки.

### Продолжение пространства имён `cl` в файле `cl.hpp`

Остальное содержимое заголовочного файла `cl.hpp` — почти исключительно описания классов для основных объектов *OpenCL* (перечислены в порядке появления): `ImageFormat`, `Device`, `Platform`, `Context`, `Event`, `UserEvent`, `Memory`, `Buffer`, `BufferD3D10`, `BufferGL`,

[3] Функторы — это объекты, у которых определён метод `operator ()`, т.е., они могут быть вызваны — так же, как и функции.

`BufferRenderGL`, `Image`, `Image2D`, `Image2DGL`, `Image3D`, `Image3DGL`, `Sampler`, `NDRange`, `Kernel`, `Program`, `CommandQueue`, `KernerFunctor`.

Часть функциональности классов добавляется лишь в случае компиляции для версии *OpenCL 1.1* (когда определена величина `CL_VERSION_1_1`), при наличии взаимодействия с *Direct3D* (если определена `USE_DX_INTEROP`) или при использовании разделения устройств (задано макроопределение `USE_CL_DEVICE_FISSION`).

> Кое-где попадаются ни к чему не привязанные определения: статические `inline`-функции `UnloadCompiler()`, `WaitForEvents()`, статическая константа `NullRange` типа `NDRange`, структура `LocalSpaceArg` и связанная с ней `inline`-функция `__local()`, а также дополнение к пространству имён `detail` — вспомогательный шаблонный класс `KernelArgumentHandler<>` со специализацией для типа `LocalSpaceArg`.
>
> Иерархия классов такова: часть имеет родителем `detail::Wrapper<`*`ОбъектOpenCL`*`>` (`Device`, `Platform`, `Context`, `Event`, `Memory`, `Sampler`, `Kernel`, `Program`, `CommandQueue` — «обёртки» вокруг `cl_device_id`, `cl_platform_id`, `cl_context`, `cl_event`, `cl_mem`, `cl_sampler`, `cl_kernel`, `cl_program`, `cl_command_queue`), причём все они — кроме первых двух — сопровождаются вызовом макроопределения `__GET_INFO_HELPER_WITH_RETAIN` с параметром соответствующего *C++*-объекта из пространства имён `cl`; класс `KernerFunctor` вообще не имеет родителя; класс `ImageFormat` основан на структуре `cl_image_format`; остальные классы являются дочерними уже определённых в файле классов (`UserEvent` порождён от `Event`, `Buffer` и `Image` — от `Memory`, `BufferGL` и `BufferRenderGL` — от `Buffer`, `Image2D` и `Image3D` — от `Image` и т.д.).
>
> Размеры описаний классов весьма различаются, но класс `CommandQueue` здесь является рекордсменом — ввиду обилия методов размещения команд в очереди (`enqueue<`*`Команда`*`>()`), хотя все они по сути — вызов соответствующей функции *OpenCL*, «обёрнутой» обработчиком ошибок `detail::errHandler()`.
>
> Класс `KernerFunctor` имеет многочисленные шаблонные варианты определения метода вызова (`operator ()`) с различным количеством параметров (до 15), в рамках почти всех этих вариантов объединены вместе установка параметров с помощью `clSetKernelArg()` и запуск ядер с помощью `clEnqueueNDRangeKernel()`. В последующих версиях рассматриваемого заголовочного файла этот класс уже отсутствует (вместо него используется класс `KernelFunctorGlobal`).

В заключение производится (с помощью `#undef`) «разопределение» некоторых внутренних макроопределений (с именами, начинающимися с двух подчёркиваний).

## 10.1. Примеры программ с использованием *C++*-интерфейса

> Замечание. Если текст программы (там, откуда он был взят) не имел явного названия, имя программе давалось вполне произвольно.

### `khronos_cl-hpp_test.cpp`

Простейший пример, с которым можно столкнуться, приведён в самом заголовочном файле `cl.hpp`, в его первоначальных комментариях. Этот пример использует исключения (`exceptions`), поэтому код сразу оказывается свободен от большинства проверок кодов ошибок, возвращаемых при вызовах функций *OpenCL*, — поскольку при любой ошибке генерируется исключительная ситуация. Весь содержательный код функции `main()` заключён в блок `try {}`; любая возникающая проблема сопровождается прекращением работы программы с выводом минимальных сведений о месте и типе проблемы (имя функции, которая завершилась неудачно, и сопутствующий цифровой код ошибки).

В программе создаётся пустой *STL*-вектор из объектов `cl::Platform`, описывающих платформу *OpenCL*, после чего вызывается статический метод `get()` класса `Platform` для получения списка существующих *OpenCL*-платформ. То, что этот метод должен быть статическим, — понятно, поскольку первоначально не существует объектов типа `Platform` и вызвать метод объекта никак не получится.

Результирующий вектор платформ `platforms` может оказаться пустым, если никаких *OpenCL*-платформ не будет обнаружено, поэтому далее проверяется размер этого вектора; в случае его нулевой длины работа программы завершается (с возвращаемым кодом $-1$).

Затем делается попытка создания *OpenCL*-контекста в первой же обнаруженной платформе, причём желательный тип устройств указан значением `CL_DEVICE_TYPE_CPU` — это значит, что ищется реализация *OpenCL* для *CPU*. Если имеется реализация только для *GPU*, эту константу надо сменить на `CL_DEVICE_TYPE_GPU`, иначе программа завершится досрочно с ошибкой создания контекста.

Следующий этап — получение списка устройств `devices` в созданном контексте; список представляется *STL*-вектором из объектов `cl::Device` (далее он понадобится при построении программного объекта).

Исходный код программы-ядра в текстовой строке `helloStr` описывает тривиальную «пустую» функцию ядра без параметров. Из этой строки создаётся объект `source`, а на его основе — программный объект `program_`, который подвергается построению для заданного списка устройств `devices`. Теперь из построенного программного объекта можно получить объект ядра `kernel`, использующий определённую в тексте ядра функцию `hello`. Здесь, кстати, впервые появляется возможность проверить возвращаемую ошибку `err`, но она здесь и далее не проверяется, поскольку используется механизм исключительных ситуаций.

Перед запуском ядра на исполнение (тут — на первом же устройстве из списка) необходимо создать очередь команд `queue`, в которую и попадает команда запуска ядра. Помимо очереди надо подготовить объект события `event`, с помощью которого можно дождаться окончания работы всех копий ядра после запуска — поскольку исполнение происходит асинхронно и не связано с завершением вызова метода `enqueueNDRangeKernel`. Конфигурация запуска такова: без смещения в глобальном двумерном пространстве рабочих единиц с размерами $4 \times 4$ и размером рабочей группы на усмотрение реализации.

### `opencl_example.cc`

Некоторую вариацию рассмотренной выше программы можно обнаружить также по ссылке `https://gist.github.com/jirihnidek/3f23b7451c1744d71dbf` (файл `opencl_example.cc`).

В ней нет директив `using` ни для одного пространства имён, а потому для всех объектов пространство имён указывается явным префиксом; векторы при этом используются стандартные (т.е., *STL*, а не определённые в `cl.hpp`). Помимо этого в программе выводятся все обнаруженные платформы, а также все устройства самой первой обнаруженной платформы. В остальном программа не отличается от предыдущей.

### `dhruba.cpp`

Этот пример создан автором учебного пособия по *OpenCL*, поэтому хорошо комментирован (см. `dhruba.name/2012/10/06/opencl-cookbook-hello-world-using-cpp-host-binding/`). Опрос платформ, выбор устройств из первой же платформы, создание контекста с этими устройствами, затем очереди команд в контексте, связанной с первым устройством, считывание файла с кодом ядра, создание объекта программы и построение её для устройств, выделение памяти для буфера, куда помещается сообщение, передаваемое ядру в качестве параметра, задание параметров для ядра и его вызов с помощью метода очереди `enqueueTask()` с ожиданием результата с помощью `queue.finish()` — вот все используемые в ней этапы.

### `simpleopencl.cpp`

Файл `simpleopencl.cpp` (название файла — условное) взят со страницы блога по адресу `simpleopencl.blogspot.ru/2013/06/tutorial-simple-start-with-opencl-and-c.html`.

> Компиляция с заголовочным файлом версии *1.2* показывает, что в файле уже не находится идентификатор `KernelFunctor`, однако, переходя к альтернативному способу запуска ядра (закомментированному там же рядом), можно добиться компиляции этой программы.

### smallest-opencl-program.cpp

Ещё один пример программы с использованием *C++*-интерфейса, его имеет смысл рассмотреть, поскольку это заявляемый как пример минимальной по размеру *OpenCL*-программы с сайта *ArrayFire* (`http://arrayfire.com/quest-for-the-smallest-opencl-program/`) — разработчика программного обеспечения. В тексте заметки код никак не назван, поэтому здесь используется условное название файла `smallest-opencl-program.cpp`.

Для минимизации размера кода используются исключения, задействуемые, как обычно, с помощью определения специальной величины (`#define __CL_ENABLE_EXCEPTIONS`).

В программе создаётся объект *OpenCL*-контекста `Context` с устройством по умолчанию (это в дальнейшем позволяет вызывать функции *OpenCL* вообще без указания контекста), вектор вещественных величин `data` — заданной длины и заполненный одной и той же величиной, а также на его основе — два копируемых объекта `Buffer`, и один пустой объект `Buffer` — все одинакового размера, но для результата — с возможностью чтения и записи. Специальные конструкторы объекта `Buffer` принимают в качестве параметров итераторы и это позволяет осуществлять выделение памяти нужного размера и перенос данных в неё одним вызовом.

Текст программы ядра средствами *C++11* «обёрнут» в символьную строку, на основе которой создаётся объект программы (класса `Program`), т.е., пример иллюстрирует использование так называемой необработанной (*raw*) строки для представления кода ядра.

С помощью вызова шаблонной функции `make_kernel<...>()` создаётся функтор, использующий в качестве параметров объект `EnqueueArgs` и три объекта `Buffer` и предназначенный для действия при помощи функции ядра с именем `add()` на соответствующие компоненты этих буферов (суммирования первых двух в третью). Объект `EnqueueArgs` нужен для установки конфигурации запуска ядра. И снова — для краткости — используется командная очередь по умолчанию, хотя её, конечно, можно установить с помощью объекта `EnqueueArgs`.

> Так как тип переменной, которую возвращает `make_kernel<...>()`, объявлен как `auto` (стандарт *C++11* это позволяет), не сразу может быть понятно, что это именно функтор, но таковы издержки нового подхода к *C++*-программированию: простота при написании программ выливается потом в затруднения при их чтении...

Для размещения результата в хост-программе создаётся вектор вещественных величин `result` нужного размера, куда с устройства копируется содержимое результирующего буфера `c`; после чего полученный результат выводится в стандартный вывод через запятую при помощи итератора потока вывода вещественных значений.

> Из-за того, что в программе явно не указано, какой тип вектор используется, заголовочные файлы `cl.hpp` разных версий могут давать разные результаты при компиляции программы: имеет место неоднозначность объявления `vector<float> data(elements, 5);` — класс `vector` определён как пространстве имён `std`, так и пространстве имён `cl`, вводимом в заголовочном файле `cl.hpp`, а в программе используются сразу оба:
>
> ```
> using namespace cl;
> using namespace std;
> ```
>
> поэтому — при неправильном «использовании» — конфликт имён неизбежен.

Правильно компилируется эта маленькая программа только компилятором с поддержкой *C++11* (опция командной строки для `g++`: `-std=c++11`) и при использовании заголовочного файла версии не ниже *1.2*.

### ErikSmistad-CPP-Test.cpp

Программа в статье `https://www.eriksmistad.no/using-the-cpp-bindings-for-opencl/` никак не названа, поэтому здесь ей дано название по имени автора. Она использует встроенное определение вектора (задаётся директивой препроцессора `#define __NO_STD_VECTOR`). В остальном эта тестовая программа написана просто и традиционно.

## 10.2. Полезные фрагменты кода – *C++*-интерфейс

Резюмируем увиденное в разобранных примерах. Для получения набора устройств (а это — необходимый параметр для построения программы ядра), можно создать вектор платформ, заполнить его реальной информацией, получить свойство контекста платформы и потом извлечь информацию об устройствах:

```
vector<cl::Platform> platforms;
cl::Platform::get(&platforms);
cl_context_properties properties[] =
{
    CL_CONTEXT_PLATFORM, (cl_context_properties)(platforms[0])(),
    0
};
cl::Context context(CL_DEVICE_TYPE_ALL, properties);
vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();
```

Здесь в массиве свойств контекста устройств располагаются пары свойство–значение, завершаемые нулевым значением.

Получение значения свойства выглядит немного необычно, а потому требует пояснения:

```
(cl_context_properties)(platforms[0])()
```

Приведение типа к `cl_context_properties` необходимо, поскольку сведения о платформе имеют совсем другой тип, а так как тип `cl::Platform` — это всего лишь «обёртка», то к значению этого типа надо применить метод вызова (`operator()`), чтобы получить идентификатор платформы, который затем привести к нужному типу. Вероятно, это же самое можно записать и так:

```
(cl_context_properties)(platforms[0]())
```

Можно сделать чуть по-другому: создать векторы платформ и устройств, получить список платформ и у какой-то из них (например, самой первой) опросить устройства, относящиеся к заданному классу устройств:

```
vector<cl::Platform> platforms;
vector<cl::Device> devices;
cl::Platform::get(&platforms);
platforms[0].getDevices(CL_DEVICE_TYPE_GPU, &devices);// CL_DEVICE_TYPE_ALL
cl::Context context(devices);
```

Устройства должны указываться при создании контекста списком, а потому при выборе конкретного (например, самого первого), надо создать из него список (требует *C++ 11*!):

```
cl::Context context({devices[0]});
```

Можно также — как это продемонстрировано в «минимальной» *C++*-программе *ArrayFire* — создать так называемый контекст по умолчанию, а потом использовать конструкторы для нужных объектов или методы, не содержащие контекста в качестве параметра. Разница будет только в том, что будут использоваться все устройства, заданные типом, выбранные или какие попало, что, впрочем, для случая единственной реализации *OpenCL* даст один и тот же результат.

### Получение объекта ядра из исходного кода

Для создания ядра из исходного кода надо прежде всего получить этот код в памяти. Если он располагается в отдельном файле, его надо прочесть из файла в память (не надо забывать при этом о бинарном режиме открытия файла, иначе в системе *Windows* могут быть проблемы с компиляцией кода!).

```
std::ifstream file("<ИмяФайлаЯдра>",ios_base::binary);
std::string code(
    std::istreambuf_iterator<char>(file),
    std::istreambuf_iterator<char>());
cl::Program::Sources src(1, std::make_pair(code.c_str(), code.length()+1));
cl::Program program = cl::Program(context, src);
program.build(devices);
cl::Kernel kernel(program, "<ИмяФункцииЯдра>");
```

Здесь положение кода указано двумя итераторами: начала и конца потока (последний является результатом создания объекта итератора потока при вызове конструктора по умолчанию). Преобразованный к *C*-строке код вместе с его длиной (увеличенной на единицу) образует единственную пару, составляющую текст программы ядра. Из него создаётся объект программы, на основе которого получается ядро с соответствующим именем функции.

Если код уже находится в тексте программы, скажем в строке `codeStr`, то считывать из файла, понятное дело, ничего не нужно:

```
cl::Program::Sources src(1, std::make_pair(codeStr, strlen(codeStr)));
cl::Program program = cl::Program(context, src);
program.build(devices);
cl::Kernel kernel(program, "<ИмяФункцииЯдра>", &err);
```

Если компилятор поддерживает новые возможности языка *C++*, то можно сделать и так:

```
cl::Program::Sources src;
src.push_back(codeStr.c_str(), codeStr.length());
cl::Program program(context, src);
cl::Kernel kernel = cl::Kernel(program, "<ИмяФункцииЯдра>");
```

или даже вот так (здесь предполагается, что пространство имён `cl` погружено в глобальную область видимости):

```
Program addProg(R"d(
    kernel
    void add(global const float * restrict const a,
             global const float * restrict const b,
             global       float * restrict const c) {
        unsigned idx = get_global_id(0);
        c[idx] = a[idx] + b[idx];
    }
)d", true);

auto add = make_kernel<Buffer, Buffer, Buffer>(addProg, "add");
```

Тут — как уже писалось ранее — получается функтор с именем `add`, задействуемый для обработки векторов `a`, `b`, `c` вот такой строкой кода:

```
add(EnqueueArgs(elements), a, b, c);  // elements -- это размер вектора
```

### Запуск ядра на исполнение

Более традиционным образом ядро попадает в очередь на исполнение с помощью вызова одного из двух методов класса командной очереди (`CommandQueue`): `enqueueNDRangeKernel()` или `enqueueTask()`. Последняя возможность, впрочем, является специальным случаем первой, когда параметры смещения, глобального и локального размера равны, соответственно, 0, 1 и 1.

# 11. Расширения (*Device Extensions*)

Под расширениями *OpenCL* (***extensions***) понимаются такие свойства устройств, которые могут быть доступны только на некоторых устройствах каких-то производителей, — в отличие от обязательных (*core*) свойств, предусмотренных спецификацией.

Расширения *OpenCL* обеспечивают коду ядер дополнительные возможности — такие как арифметика двойной (или половинной) точности, атомарные операции, запись в объекты трёхмерных изображений, совместное использование объектов в *OpenCL* и в *OpenGL*, разделение устройств и др. Конечно, применение расширений ухудшает переносимость кода программ, однако, позволяет полнее использовать возможности конкретных устройств.

Расширения *OpenCL* могут быть определены самим изготовителем, частью рабочей группы *OpenCL* или всей рабочей группой консорциума *Khronos*. Самыми переносимыми (т.е., работающими на разных аппаратных платформах без внесения изменений в программу) являются расширения, одобренные всей рабочей группой, в то время как расширения изготовителя наименее переносимы и привязаны либо к конкретным устройствам, либо к линейке продуктов. Но независимо от «источника» расширения всё равно нет никакой гарантии, что расширение будет доступно на любой платформе.

Часть расширений относится ко всей платформе, часть — только к конкретным устройствам. Узнать, какие расширения поддерживаются платформой `platform`, можно с помощью вызова функции `clGetPlatformInfo(platform, CL_PLATFORM_EXTENSIONS, ...)`, возвращающего список имён расширений; аналогичная информация о расширениях устройства `device` может быть получена с помощью вызова функции `clGetDeviceInfo(device, CL_DEVICE_EXTENSIONS, ...)`.

## 11.1. Типы и имена расширений

Имеется три типа расширений и соглашений о их поименовании:

- KHR-расширение формально ратифицировано рабочей группой *OpenCL* и обеспечивается на некоторых, но не на всех устройствах. Такое расширение имеет уникальное имя вида `"cl_khr_<`*`ИмяРасширения`*`>"`.

- EXT-расширение разработано частью рабочей группы *OpenCL* и его следует воспринимать как «незавершённую работу». Такое расширение имеет уникальное имя вида `"cl_ext_<`*`ИмяРасширения`*`>"`.

- Расширение производителя. Такие расширения не являются переносимыми и имеют уникальные имена вида `"cl_<`*`Изготовитель`*`>_<`*`ИмяРасширения`*`>"`, где `<`*`Изготовитель`*`>` может принимать значения `nv`, `amd`, `intel`, `arm` и т.п.

  На сайте *Khronos* имеются описания некоторых расширений изготовителей и консорциума (`https://www.khronos.org/registry/cl/extensions/`), откуда можно заключить, что используемые сейчас «имена» изготовителей весьма немногочисленны: `altera` (*Altera*), `amd` (*AMD*), `arm` (*ARM*), `img` (*Imagination Technologies*), `intel` (*Intel*), `nv` (*NVidia*), `qcom` (*QualComm*). Имеются также другие изготовители со своими реализациями *OpenCL*, (например, фирма *Texas Instruments*), однако неясно, используются ли для расширений ещё какие-то «имена» изготовителей. Вполне вероятно, что роль такого «имени» может играть значение параметра *Platform Extensions function suffix* из свойств платформы (добавлен в версии *1.2* спецификации и называется `CL_PLATFORM_ICD_SUFFIX_KHR`).

## 11.2. Компиляция при использовании расширений

Директива препроцессора `#pragma OPENCL EXTENSION` управляет поведением компилятора *OpenCL*, позволяя или запрещая расширения при компиляции ядер. Её синтаксис:

```
#pragma OPENCL EXTENSION <ИмяРасширения> : <Поведение>
```

<*Поведение*> может иметь одно из двух значений: `enable` или `disable`.

По умолчанию все расширения запрещены и должны быть явно разрешены — как если бы предварительно была употреблена такая директива:

```
#pragma OPENCL EXTENSION all : disable
```

Если расширение разрешено упомянутой выше директивой, то в коде ядра будет определена величина с именем расширения (т.е., для ратифицированных *Khronos* расширений — величина с именем "cl_khr_<*ИмяРасширения*>"). Кроме того, расширение может определять собственные интерфейсные функции (которые будут доступны в коде *host*-приложения — см. далее); они будут именоваться "cl<*ИмяФункции*>KHR". Перечислимые величины, определяемые расширением, имеют имена вида "CL_<*ИмяПеречисления*>_KHR". Для незавершённых расширений или расширений производителя `KHR` будет заменено на `EXT` или `AMD`, `INTEL`, `NV`, `ARM` и т.п. соответственно.

Например, расширение `cl_khr_3d_image_writes` добавляет препроцессорное определение вида `#define cl_khr_3d_image_writes` при компиляции ядра, поэтому в тексте ядра можно пользоваться этим, предусматривая разные действия в случае наличия или отсутствия расширения:

```
#ifdef cl_khr_3d_image_writes
// действия при наличии расширения
#else
// действия при отсутствии расширения
#endif
```

Если у расширения имеются интерфейсные функции, к ним можно получить доступ с помощью функции `clGetExtensionFunctionAddressForPlatform()` из заголовочного файла `cl_ext.h`:

```
void* clGetExtensionFunctionAddressForPlatform(
    cl_platform_id platform,
    const char *fnname
);
```

Она даёт адрес функции расширения с именем *fnname* для заданной платформы *platform*. Возвращаемое значение `NULL` показывает, что указанная функция не существует в данной реализации либо платформа не является действительной. Результат, отличный от `NULL`, не гарантирует, что функция расширения поддерживается платформой, следует сделать запросы с помощью `clGetPlatformInfo(platform, CL_PLATFORM_EXTENSIONS, ...)` и `clGetDeviceInfo(device, CL_DEVICE_EXTENSIONS, ...)`, чтобы определить, поддерживается ли расширение реализацией *OpenCL*.

## 11.3. Информация о расширениях в файле `cl_ext.h` (`cl_gl_ext.h`)

Описания расширений в заголовочном файле `cl_ext.h` (или `cl_gl_ext.h`) построены таким образом: определяется символьное имя расширения, определяются — если это необходимо — синонимы новых типов, символьные константы, прототипы функций и синонимы для типов указателей на дополнительные функции. Вот как выглядит, например, фрагмент заголовочного файла `cl_ext.h`, соответствующий описанию расширения `cl_khr_icd` (подробнее само расширение будет разобрано позднее):

```
/***********************
* cl_khr_icd extension *
***********************/
#define cl_khr_icd 1

/* cl_platform_info                                                  */
```

```
#define CL_PLATFORM_ICD_SUFFIX_KHR                  0x0920

/* Additional Error Codes                                          */
#define CL_PLATFORM_NOT_FOUND_KHR                   -1001

extern CL_API_ENTRY cl_int CL_API_CALL
clIcdGetPlatformIDsKHR(cl_uint          /* num_entries */,
                       cl_platform_id * /* platforms */,
                       cl_uint *        /* num_platforms */);

typedef CL_API_ENTRY cl_int (CL_API_CALL *clIcdGetPlatformIDsKHR_fn)(
    cl_uint          /* num_entries */,
    cl_platform_id * /* platforms */,
    cl_uint *        /* num_platforms */);
```

Синонимы соответствующих типов указателей на функции должны быть объявлены с помощью `typedef` для всех расширений с добавленными интерфейсными функциями. Такие синонимы — часть интерфейса, определяемого в файле `cl_ext.h` (для *OpenCL*-расширений) или в файле `cl_gl_ext.h` (для расширений, использующих взаимодействие между *OpenCL* и *OpenGL*).

Если расширения являются более простыми, то добавляться могут только символьные имена, как, например, для расширения `cl_arm_printf` (работающего на *GPU* марки ***Mali***  и дающего возможность применять в коде ядер старую добрую функцию `printf()`):

```
/*********************************
* cl_arm_printf extension
*********************************/
#define CL_PRINTF_CALLBACK_ARM                      0x40B0
#define CL_PRINTF_BUFFERSIZE_ARM                    0x40B1
```

## 11.4. Наиболее распространённые расширения консорциума

`cl_khr_global_int32_base_atomics`
`cl_khr_local_int32_base_atomics`
`cl_khr_global_int32_extended_atomics`
`cl_khr_local_int32_extended_atomics`
: Реализуют атомарные операции из базового и расширенного набора над 32-битными целыми (со знаком и без знака) в локальной и глобальной памяти.

`cl_khr_byte_addressable_store`
: Расширение снимает ограничения на встроенные типы `char`, `uchar`, `char2`, `uchar2`, `short`, `ushort` и `half`, позволяя записывать такие значения в память по указателю.

`cl_khr_int64_base_atomics`
`cl_khr_int64_extended_atomics`
: Реализуют атомарные операции, соответственно, базового и расширенного набора над 64-битными целыми (со знаком и без знака) в локальной и глобальной памяти.

`cl_khr_3d_image_writes`
: Позволяет осуществлять запись в объекты трёхмерных изображений.

`cl_khr_fp16`
: Добавляет поддержку для так называемых вещественных значений «половинной» точности и соответствующих типов: математические функции, функции преобразования, функции общего назначения, геометрические функции, функции сравнения,

а также функции чтения из памяти и записи в память (включая и такие объекты памяти, как изображения).

`cl_khr_gl_sharing`
: Осуществляет ассоциирование контекстов *OpenGL* и *OpenCL* или позволяет контекстам *OpenGL* и *OpenCL* совместно использовать объекты (реализация *OpenGL* предполагается).

`cl_khr_spir`
: Добавляет поддержку создания программного объекта из стандартного переносимого промежуточного представления (*Standard Portable Intermediate Representation*, сокращённо — ***SPIR***). Такое представление является бинарным и не зависит ни от одного производителя.

`cl_khr_icd`
: Позволяет одновременное использование нескольких реализаций *OpenCL* на основе простого механизма: загрузчика инсталлируемых клиентских драйверов (*ICD Loader*) для инсталлируемых клиентских драйверов разных производителей (*Vendor ICDs*).

## 11.5. Расширения в коде ядра — на примере `cl_khr_fp64`

Для любого расширения хост-программа должна сначала проверять, что это расширение поддерживается, вызывая `clGetDeviceInfo()` с параметром `CL_DEVICE_EXTENSIONS`; возвращаемая строка должна содержать имя расширения (в данном случае — `"cl_khr_fp64"`), если такая поддержка есть. Тогда при компиляции кода ядра (скажем, при помощи функции `clBuildProgram()`) будет определена величина `cl_khr_fp64` и можно будет добавить к коду ядра директиву для разрешения расширения:

```
#ifdef cl_khr_fp64
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#endif
```

а в коде — использовать разные необходимые действия в зависимости от наличия или отсутствия расширения:

```
#ifdef cl_khr_fp64
      // действия при наличии расширения
#else
      // действия при отсутствии или #error
#endif
```

Всё это справедливо для подавляющего большинства расширений, однако, именно с данным расширением (`cl_khr_fp64`) нужно действовать намного аккуратнее, поскольку, начиная с версии *OpenCL 1.2* это уже не расширение, а свойство, поэтому строка с директивой разрешения этого расширения (см. выше) в коде ядра для *OpenCL 1.2* больше не нужна!

Таким образом, реальный код, учитывающий всё сказанное, должен быть примерно таким (пример адаптирован из текста, приводимого в файле `summarization_kernel.cl` проекта (`https://github.com/Milkyway-at-home/milkywayathome_client/`):

```
#if DOUBLEPRECISION
/* В версии 1.2 cl_khr_fp64 -- уже не расширение */
  #if __OPENCL_VERSION__ < 120
    #if cl_khr_fp64
      #pragma OPENCL EXTENSION cl_khr_fp64 : enable
    #elif cl_amd_fp64
      #pragma OPENCL EXTENSION cl_amd_fp64 : enable
```

```
    #else
      #error Расширения с двойной точностью нет!
    #endif
  #endif
#endif /* DOUBLEPRECISION */

#if DOUBLEPRECISION
typedef double  real;
typedef double2 real2;
typedef double4 real4;
#else
typedef float  real;
typedef float2 real2;
typedef float4 real4;
#endif /* DOUBLEPRECISION */
```

Здесь использование в коде определяемой ранее величины `DOUBLEPRECISION` позволяет задавать применение величин двойной точности, если они поддерживаются, либо не применять их вообще, ограничивая операции вещественными значениями одинарной точности.

Для более простых расширений часто достаточно просто директивы разрешения расширения.

```
#pragma OPENCL EXTENSION cl_khr_int64_base_atomics : enable
```

Надо только не забывать, что расширения нередко становятся частью спецификации, а потому с некоторого момента для их использования не нужно будет и директивы разрешения (как, скажем, для атомарных операций с 32-битными величинами, которые с версии *1.1* уже вошли в разряд обязательных: базовые `cl_khr_global_int32_base_atomics` и `cl_khr_local_int32_base_atomics`, расширенные `cl_khr_global_int32_extended_atomics` и `cl_khr_local_int32_extended_atomics`).

## 11.6. Некоторые расширения производителей

Несложно понять, что работа с любым конкретным расширением будет весьма индивидуальной. Так, иногда у производителей имеются полезные и сходные по функциональности расширения, но им для правильной работы могут понадобиться разные дополнительные действия. Например, для расширения `cl_amd_printf` достаточно просто разрешить его в коде ядра:

```
#pragma OPENCL EXTENSION cl_amd_printf : enable
```

и далее в ядре можно использовать вызовы функции `printf()`, а вот для `cl_arm_printf` — помимо аналогичного разрешения в коде ядра — необходимо также добавить в хост-код, во-первых, функцию обратного вызова (*callback*):

```
void printf_callback(const char *buffer, size_t len,
                     size_t complete, void *user_data)
{
  printf( "%.*s", len, buffer );
}
```

а, во-вторых, специальным образом создать контекст с этой функцией и заданным размером буфера для неё (обратите внимание на упоминавшиеся ранее константы расширения):

```
cl_context_properties props[] =
{
```

```
    CL_PRINTF_CALLBACK_ARM,   (cl_context_properties) printf_callback,
    CL_PRINTF_BUFFERSIZE_ARM, (cl_context_properties) 0x100000,
    CL_CONTEXT_PLATFORM,      (cl_context_properties) platform,
    0
};
cl_context context = clCreateContext(props, 1, &device, NULL, NULL, &err);
```

Нередко производители предлагают специальные расширения, наглядно демонстрирующие какие-то уникальные свойства своих устройств. В качестве примера рассмотрим одно такое расширение от *Intel*: `cl_intel_motion_estimation` (расширение для оценивания движения; подробности см. в документе `motion-estimation-intro.pdf`).

Оценивание движения — процесс определения векторов движения, описывающих переход от одного двумерного изображения к другому, как правило — соседним кадрам видео.

Интерфейс для этого расширения содержится в файле `cl_ext.h` из *Intel OpenCL SDK*. Никаких интерфейсных хост-функций у этого расширения нет, но имеются т.н. встроенные ядра (появились в *OpenCL 1.2*), выполняющие основную работу; они посылаются в очередь команд на исполнение стандартным образом — с помощью `NDRange`, параметры которого позволяют выбирать подобласть во входящих кадрах. Пример сигнатуры ядра:

```
__kernel void
block_motion_estimate_intel
(
    accelerator_intel_t accelerator,
    __read_only image2d_t src_image,
    __read_only image2d_t ref_image,
    __global short2 *prediction_motion_vector_buffer,
    __global short2 *motion_vector_buffer,
    __global ushort *residuals
);
```

## 11.7. Дополнительные ссылки

`http://www.codeproject.com/Articles/330174/Part-OpenCL-Extensions-and-Device-Fission`
Одна из статей цикла об *OpenCL* на сайте **CodeProject**, посвящённая расширениям, а также разбиениям устройств на подустройства: ***«OpenCL Extensions and Device Fission»***.

`https://www.khronos.org/registry/cl/extensions/arm/cl_arm_printf.txt`
Описание расширения `cl_arm_printf` на сайте консорциума *Khronos*.

`https://software.intel.com/en-us/articles/intro-to-motion-estimation-extension...`
`https://software.intel.com/sites/default/files/motion-estimation-intro.pdf`
Вводная статья с сайта фирмы *Intel* о расширении для оценивания движения и её PDF-вариант.

`https://software.intel.com/en-us/articles/video-motion-estimation-using-opencl`
Страница с краткой характеристикой обучающего материала фирмы *Intel* по оцениванию движения с помощью *OpenCL*. Содержит ссылки на документацию (`intel_ocl_motion_estimation.pdf`) и исходный код (`intel_ocl_motion_estimation.zip`).

`https://www.khronos.org/registry/cl/extensions/intel/cl_intel_accelerator.txt`
`https://www.khronos.org/registry/cl/extensions/intel/cl_intel_motion_estimation.txt`
`https://www.khronos.org/registry/cl/extensions/intel/...advanced_motion_estimation.txt`
Описания некоторых расширений фирмы *Intel*: `cl_intel_accelerator`, `cl_intel_motion_estimation`, `cl_intel_advanced_motion_estimation` — на сайте консорциума *Khronos*.

# 12. Сосуществование реализаций *OpenCL*

Расширение `cl_khr_icd` определяет простой механизм использования нескольких реализаций *OpenCL* с возможностью выбора одной из них для работы пользовательской программы. Вместо конкретной реализации, к которой может получить доступ программа, используется специальная промежуточная библиотека (т.н. ***ICD Loader***), которая далее взаимодействует с конкретными реализациями *OpenCL*. Каждая такая реализация в терминологиии этого расширения — это инсталлируемый клиентский драйвер производителя (***Vendor ICD*** или просто ***ICD***), для которого *ICD Loader* — координирующая «инстанция». Именно загрузчик клиентского драйвера позволяет любой программе получить доступ к программному интерфейсу *OpenCL*, — но только после выбора этой программой (или пользователем программы) конкретного драйвера производителя (или, что то же самое — конкретной реализации *OpenCL*). Выбор осуществляется на основе полученной от *ICD* (драйверов производителя) информации в процессе первоначального опроса имеющихся платформ и устройств *OpenCL*, поддерживающих расширение `cl_khr_icd`.

Загрузчик *ICD* в системе *Windows* называется `OpenCL.dll` и располагается в системном каталоге; в различных вариациях *Linux* — это разделяемая библиотека `libOpenCL.so`, также находящаяся в пути поиска. Прелесть выбранного подхода состоит в том, что для пользовательской программы процесс её построения никак не меняется: происходит линковка со статической библиотекой-«обёрткой» вокруг такой динамической библиотеки:

```
gcc . . . -lOpenCL . . .
```

точно так же, как это было бы в случае «одинокой» *OpenCL*-реализации с таким именем. Просто в случае, когда вместо *ICD* используется *ICD Loader*, количество платформ, доступных программе, становится большим единицы и появляется необходимость выбора платформы (а затем — и конкретного *OpenCL*-устройства).

> Таким образом, любая платформа *OpenCL* может иметь одно или несколько устройств. Одно и то же устройство может принадлежать одной или нескольким платформам. Однако, версии платформы и устройств не обязательно совпадают.

Для *Windows* и *Linux* загрузчик драйверов (*ICD Loader*) был доступен, начиная с версии *OpenCL 1.0*. Напротив, система *OSX* не имеет такого загрузчика вообще. Фирма *Apple* выбрала другой путь: все *OpenCL*-устройства принадлежат единственной платформе.

## 12.1. Перебор реализаций *OpenCL*

### Перебор реализаций в системе *Windows*

Предполагается, что все реализации с поддержкой расширения `cl_khr_icd` регистрируются в ветви реестра

```
HKEY_LOCAL_MACHINE\SOFTWARE\Khronos\OpenCL\Vendors
```

где каждая величина с нулевым `DWORD`-значением имеет в качестве имени либо полный путь, либо просто имя динамической библиотеки, которую надо подгрузить в память с помощью функции `LoadLibraryA()` для получения доступа к функциям *OpenCL*-интерфейса изготовителя реализации. Вот что, например, имеется на моём старом компьютере, где установлена *CUDA 4.1* с обновлёнными позднее драйверами:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Khronos\OpenCL\Vendors]
"nvcuda.dll"=dword:00000000
"C:\\WINDOWS\\System32\\nvopencl.dll"=dword:00000000
```

Абсолютно непонятно, что здесь делает файл `nvcuda.dll`, ибо в нём нет нужных функций. В библиотеке `nvopencl.dll` экспортированы только три функции: `clGetExportTable()`, `clGetExtensionFunctionAddress()`, `clGetPlatformInfo()`, — в противовес тому, что требуется от *ICD* сейчас (см. ниже). Вероятно, это связано с тем, что используется самый древний загрузчик версии 1.0.

Ещё один пример (для 32-битных программ в 64-битной системе; взят из Сети, но вопрос о файле `nvcuda.dll` — остаётся):

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Khronos\OpenCL\Vendors]
"nvcuda.dll"=dword:00000000
"intelocl.dll"=dword:00000000
```

#### Перебор реализаций в системе *Linux*

В рамках системы *Linux* реализации с поддержкой расширения `cl_khr_icd` регистрируются в каталоге `/etc/OpenCL/vendors/`, который просматривается загрузчиком клиентского драйвера. Каждый файл в этом каталоге (обычно они имеют расширение `.icd`) открывается как текстовый и анализируется его первая строка; она может содержать как полное имя файла, так и просто его имя. Затем загрузчик драйвера пытается открыть этот файл как разделяемый объект (***shared object***, `.so`), используя функцию `dlopen()`.

#### Перебор реализаций в системе *Android*

Отличается от перебора в системе *Linux* только местоположением каталога для файлов `.icd`; в системе *Android* — это каталог `/system/vendors/Khronos/OpenCL/vendors`.

## 12.2. Работа с *ICD*-библиотекой изготовителя

После подгрузки найденной библиотеки (клиентского драйвера *ICD*; *Vendor ICD*) в память загрузчик клиентского драйвера (*ICD Loader*) пытается получить доступ к функциям: `clIcdGetPlatformIDsKHR()`, `clGetPlatformInfo()` и `clGetExtensionFunctionAddress()`. Отсутствие хотя бы одной из них означает невозможность нормальной работы с драйвером (реализацией *OpenCL*) и тогда библиотека клиентского драйвера выгружается из памяти. Если всё нормально, загрузчик опрашивает платформы, поддерживающие инсталлируемые драйверы с помощью функции `clIcdGetPlatformIDsKHR()`. Для каждой такой платформы запрашиваются строки с перечислением поддерживаемых расширений — чтобы удостовериться, что расширение `cl_khr_icd` действительно поддерживается, — и с помощью вызова функции `clGetPlatformInfo()` с параметром `CL_PLATFORM_ICD_SUFFIX_KHR` извлекается строка суффикса, употребляемого в расширениях изготовителя платформы. Неудача на любом из этих шагов ведёт к прекращению работы с рассматриваемой платформой и переходу к следующей платформе. Если платформ с поддержкой `cl_khr_icd` не найдено, будет возвращена специальная ошибка `CL_PLATFORM_NOT_FOUND_KHR` с численной величиной $-1001$.

Для интерфейсных функций *OpenCL*, поддерживаемых загрузчиком клиентского драйвера, функция `clGetExtensionFunctionAddress()` возвращает указатель на функцию, реализуемую самим загрузчиком; для неизвестных ему функций отыскивается реализация изготовителя с присущим ему суффиксом, что позволяет изготовителю задействовать появляющиеся новые функции сразу после их включения в библиотеку-драйвер.

## 12.3. Стандартный *ICD Loader* от *Khronos*

Загрузчик консорциума (`https://github.com/KhronosGroup/OpenCL-ICD-Loader`) может быть откомпилирован для *Windows* или *Linux* самостоятельно, однако часто является частью установки реализации *OpenCL*, и, значит, уже наверняка присутствует на компьютере.

## 12.4. Альтернативный *ICD Loader* (ocl-icd)

Этот вариант загрузчика клиентских драйверов (реализаций *OpenCL*) написал ***Vincent Danjean*** (https://forge.imag.fr/projects/ocl-icd/). Он доступен также во многих репозиториях *Linux* под именем ocl-icd (*Ubuntu*) или аналогичным. В отличие от стандартного, он поддерживает некоторые переменные окружения, которые могут влиять на его работу. Вот эти переменные:

OPENCL_VENDOR_PATH
: Позволяет изменить обычно используемый путь /etc/OpenCL/vendors/ для .icd-файлов

OCL_ICD_VENDORS
: Позволяет изменить способ поиска *ICD* в системе. Если значение переменной окружения OCL_ICD_VENDORS — каталог, тогда он замещает обычно используемый путь /etc/OpenCL/vendors/. Если $OCL_ICD_VENDORS содержит .icd-файл, подгружается только *ICD*, указанный в этом файле. В остальных случаях будет сделана попытка загрузить $OCL_ICD_VENDORS как саму разделяемую библиотеку (с помощью dlopen()).

OCL_ICD_ASSUME_ICD_EXTENSION
: Если переменная установлена, загрузчик не будет проверять, что загружаемые *ICD* указывают расширение cl_khr_icd как поддерживаемое. Это может понадобиться при использовании *Intel ICD* вместе с optirun (утилита запуска программ на дискретной видеокарте), т.к. в противном случае — из-за ошибки в *Intel ICD* — программа прекратит работу.

OCL_ICD_PLATFORM_SORT
: Позволяет выбрать способ сортировки платформ при опросе с помощью функции clGetPlatformIDs(). Сейчас реализованы два алгоритма: 1) devices — сначала выводятся платформы с поддержкой *GPU*, затем — с поддержкой *CPU*, и, наконец, акселераторы; 2) none — никакой сортировки не выполняется и порядок может изменяться от вызова к вызову. Если переменная не установлена (имеет какое-то неизвестное значение), используется первый алгоритм.

OCL_ICD_DEFAULT_PLATFORM
: Номер платформы, выбираемой в качестве платформы по умолчанию. Без указания способа сортировки платформ смысла не имеет.

OCL_ICD_DEBUG
: Если ocl-icd откомпилирована с поддержкой отладки, можно установить эту переменную, чтобы воздействовать битовыми значениями на отображение различной информации: 1 — предупреждения; 2 — информативные сообщения; 4 — вход/выход для некоторых *OpenCL* функций; 8 — вывод внутренней структуры загруженных драйверов (*ICD*). Наиболее полезна эта переменная при разработке самого загрузчика или драйверов.

Применение указанных переменных позволяет задавать исполняемой *OpenCL*-программе, какой реализацией *OpenCL* ей надлежит пользоваться.

## 12.5. Варианты построения *OpenCL*-программ

Несмотря на то, что компьютер может иметь несколько *OpenCL*-платформ со своими версиями, загрузчик драйверов (*ICD Loader*) для программы всегда один. Он координирует взаимодействие со всеми платформами и в зависимости от выбранной платформы перенаправляет вызовы функций *OpenCL* нужному драйверу (т.е., реализации). Таким образом,

можно компилировать программы, считая загрузчик единственной реализацией *OpenCL*: во время исполнения программа подгрузит его, а он, в свою очередь сможет отыскать все зарегистрированные драйверы *OpenCL*.

Компиляция *OpenCL* хост-программы для использования загрузчика `ocl-icd`[4]:

```
gcc example.c -o example ‘pkg-config --cflags --libs OpenCL‘
```

Компиляция *OpenCL* хост-программы для использования загрузчика от *AMD*:

```
gcc example.c -o example -lOpenCL
```

Можно также компилировать хост-программу для использования конкретной реализации *OpenCL*, например, ***POCL*** (`http://portablecl.org`)[5] — тоже с помощью `pkg-config`:

```
gcc example.c -o example ‘pkg-config --cflags --libs pocl‘
```

Разумеется, тогда программе для работы понадобится *OpenCL*-библиотека *POCL* и без неё она (без перекомпиляции) работать не будет.

## 12.6. Некоторые ссылки по теме

`http://www.khronos.org/registry/cl/extensions/khr/cl_khr_icd.txt`
Описание расширения `cl_khr_icd` на сайте консорциума *Khronos*.

`https://github.com/KhronosGroup/OpenCL-ICD-Loader`
«Каноническая» реализация *ICD*-загрузчика, предложенная консорциумом *Khronos*.

`https://forge.imag.fr/projects/ocl-icd/`
Реализация *ICD*-загрузчика `ocl-icd` с лицензией *BSD* (автор — ***Vincent Danjean***). В ней поддерживаются переменные окружения для управления работой загрузчика. Является стандартным пакетом в *Ubuntu* и *Debian*.

`http://manpages.org/libopencl/7`
Описание *ICD*-загрузчика `ocl-icd` для *Ubuntu* и *Debian*, реализованного в виде подгружаемой динамически библиотеки `libOpenCL.so`. Перечисляются имена всех переменных окружения, которые способны изменять поведение этого загрузчика, а также их возможные значения и соответствующие им действия. Стоит отметить, что в «канонической» реализации загрузчика от *Khronos* никакие переменные окружения не опрашиваются.

`http://blog.cyez.nl/opencl-icd-stub/`
Статья, в которой обсуждается «заглушка» *OpenCL* (***OpenCL ICD Stub***), т.е., по сути «пустая» «заготовка», позволяющая в дальнейшем реализовать *ICD*-загрузчик. Обсуждаются некоторые тонкости создания загрузчика.

`http://www.zuzuf.net/FreeOCL/`
`https://github.com/zuzuf/freeocl`
Статья об одной из ранних реализаций *OpenCL* для *CPU* и её исходный код. Эта открытая реализация называлась ***FreeOCL*** (автор — ***Roland Brochard***). Несмотря на то, что она в целом устарела и давно не поддерживается, для обсуждаемой здесь темы она интересна тем, что в ней приводится работоспособная версия т.н. диспетчеризирующей таблицы *ICD*-загрузчика.

[4] Здесь в командной строке употребляются символы ***backticks***, позволяющие вставить в командную строку результат выполнения программы `pkg-config` с необходимыми параметрами; есть и альтернативный вариант — заключить вызов программы `pkg-config` с параметрами в круглые скобки, предварённые символом доллара (см. пример командной строки на стр.75).

[5] Подробнее об этой реализации *OpenCL* будет идти речь далее (см. стр.72).

# 13. Объекты памяти и работа с ними в *OpenCL 1.1*

В модели памяти *OpenCL* имеются два типа объектов памяти: объекты-буферы (***buffer objects***) и объекты-изображения (***image objects***). Буфер — непрерывный участок памяти, к которому можно обращаться из всех исполняемых копий ядра. Это позволяет для работы с такой памятью использовать указатели. Изображение — некий «непрозрачный» объект, доступ к которому производится только с помощью функций, что, видимо, связано с особенностями его размещения в памяти.

Использование буферов предполагает: выделение памяти (функция `clCreateBuffer()`), перенос данных (функция `clEnqueueWriteBuffer()` — копирование данных с хоста на устройство, функция `clEnqueueReadBuffer()` — копирование данных с устройства на хост), а также последующее освобождение памяти `clReleaseMemObject()`.

Для работы с изображениями *OpenCL* в коде ядер необходимы также объекты типа сэмплер (***sampler objects***). Такие объекты определяют необходимую координатную систему для доступа к изображениям (либо с использованием целочисленных координат, либо с использованием нормализованных вещественных координат) и варианты обработки ситуаций выхода за пределы границ изображений (реализованы аппаратно: скажем, возвращается 0 или «цвет» ближайшего пиксела), а также способы интерполяции при извлечении значений «между» отсчётами. Сэмплеры также могут передаваться ядрам как обычные параметры.

В зависимости от типа пиксела в каждом канале изображения для считывания значений в ядрах применяются варианты функций `read_imagei()`, `read_imageui()`, `read_imagef()`. Они различаются типом изображения, из которого производится чтение, сэмплером и типом, задающим координаты считываемого пиксела, но все возвращают вектор из четырёх компонент.

Для записи в изображения используются, соответственно, функции `write_imagei()`, `write_imageui()`, `write_imagef()`, которым указываются объект изображения, координаты пиксела и его «цвет». Следует иметь в виду, что одновременные чтение и запись изображения в одном и том же ядре первоначально (*OpenCL 1.0*, *OpenCL 1.1*) были невозможны.

## 13.1. *Buffer Objects*

Объект буфера (*buffer object*) хранит одномерный набор элементов: скалярных типов (`int`, `float`, `...`), векторных или пользовательских структур. Создаётся объект буфера функцией `clCreateBuffer()`:

```
cl_mem clCreateBuffer (
            cl_context context,
            cl_mem_flags flags,
            size_t size,
            void *host_ptr,
            cl_int *errcode_ret);
```

где *`context`* — это *OpenCL*-контекст, в котором создаётся объект буфера, *`flags`* — битовые поля для указания информации о выделении памяти и о её использовании (возможные значения флагов перечисляются ниже; если значение параметра 0, то по умолчанию используется `CL_MEM_READ_WRITE`), *`size`* — размер памяти в байтах, *`host_ptr`* — указатель на уже выделенную приложением память (размер её не должен быть меньше, чем *`size`*), *`errcode_ret`* — указатель для возвращения кода ошибки (если он будет равен `NULL`, никакие ошибки не возвращаются).

Функция создания буфера `clCreateBuffer()` возвращает соответствующий объект и устанавливает значение по указателю *`errcode_ret`* в `CL_SUCCESS` в случае успеха. В противном случае она возвращает значение `NULL` и какую-то из возможных ошибок через указатель *`errcode_ret`*. Часть кодов ошибок указывает на недопустимые значения передаваемых параметров (`CL_INVALID_CONTEXT`, `CL_INVALID_VALUE`, `CL_INVALID_BUFFER_SIZE`,

CL_INVALID_HOST_PTR), часть — на невозможность выделения памяти или других ресурсов (CL_MEM_OBJECT_ALLOCATION_FAILURE, CL_OUT_OF_RESOURCES, CL_OUT_OF_HOST_MEMORY).

### Допустимые флаги при выделении памяти (cl_mem_flags)

CL_MEM_READ_WRITE
: Объект памяти может читаться и записываться ядром (значение по умолчанию).

CL_MEM_WRITE_ONLY
: Объект памяти может записываться в ядре, но не может считываться в нём.

CL_MEM_READ_ONLY
: Этот флаг показывает, что объект памяти в ядре может только читаться. Флаги CL_MEM_READ_WRITE, CL_MEM_WRITE_ONLY, CL_MEM_READ_ONLY взаимоисключают друг друга.

CL_MEM_USE_HOST_PTR
: Флаг действителен только если *host_ptr* имеет значение, отличное от NULL. Если он указан, имеется в виду, что реализация *OpenCL* будет использовать память по этому указателю для хранения объекта памяти. Содержимое буфера может кэшироваться в памяти устройства и тогда ядра при исполнении будут использовать эту кэшированную копию.

CL_MEM_ALLOC_HOST_PTR
: Этот флаг показывает, что программа ожидает от реализации *OpenCL* выделения памяти, доступной хосту. Флаги CL_MEM_ALLOC_HOST_PTR и CL_MEM_USE_HOST_PTR взаимоисключают друг друга.

CL_MEM_COPY_HOST_PTR
: Флаг действителен только если *host_ptr* не имеет значение NULL и показывает, что программа ожидает от реализации *OpenCL* выделения памяти для объекта и копирования данных из области памяти, указанной значением параметра *host_ptr*. Флаги CL_MEM_COPY_HOST_PTR и CL_MEM_USE_HOST_PTR взаимоисключают друг друга. Два последних флага (CL_MEM_COPY_HOST_PTR и CL_MEM_ALLOC_HOST_PTR) могут быть использованы вместе для инициализации содержимого объекта типа cl_mem, выделенного с использованием памяти, доступной на хосте.

### Чтение и запись буферных объектов

Чтение содержимого буферного объекта в память хост-программы выполняется функцией

```
cl_int clEnqueueReadBuffer (
            cl_command_queue command_queue,
            cl_mem buffer, cl_bool blocking_read,
            size_t offset, size_t cb, void *ptr,
            cl_uint num_events_in_wait_list,
            const cl_event *event_wait_list,
            cl_event *event);
```

Её параметры мало отличаются от параметров аналогичной функции записи: необходим дескриптор очереди *command_queue* (т.к. действие не осуществляется сразу, а попадает в очередь команд); должен быть указан объект буфера *buffer* и способ осуществления действия в виде булевской величины: блокирующий или неблокирующий (при неблокирующем вызове возврат из функции происходит без ожидания завершения действия); нужны также область буфера («откуда» — *offset* и «сколько» — *cb*) и место сохранения *ptr*. Остальные параметры в простых случаях могут иметь «неопределённые» значения 0 и NULL.

Стоит обратить внимание на то, что многие функции с префиксом `clEnqueue...` возвращают объекты событий (***event objects***) через указатель *event*, передаваемый в качестве последнего параметра при вызове; из таких событий формируются т.н. списки ожидания (***wait lists***). Большинство этих функций принимают в качестве параметров подобный список ожидания и соответствующее ему число событий, так что основное действие, производимое функцией (для функции `clEnqueueReadBuffer()` — чтение буфера), не осуществляется, пока не завершатся все события из переданного ей списка ожидания. При этом контекст, ассоциированный с событиями в списке и очередью команд, разумеется, должен быть одним и тем же.

Прототип функции `clEnqueueWriteBuffer()` записи содержимого в буферный объект из хост-памяти отличается от разобранной функции чтения лишь тем, что указатель *ptr* в ней будет константным (при записи в буфер по указателю осуществляется только чтение).

## 13.2. Объекты изображений

Объект изображения (***image object***) используется для хранения одно-, двух- и трёхмерных текстур, буферов кадров или изображений. Элементы таких объектов могут быть только из списка предопределённых форматов.

Создаётся объект двумерного изображения с помощью функции `clCreateImage2D()`:

```
cl_mem clCreateImage2D (
            cl_context context,  cl_mem_flags flags,
            const cl_image_format *image_format,
            size_t image_width, size_t image_height,
            size_t image_row_pitch,
            void *host_ptr, cl_int *errcode_ret);
```

*context* — *OpenCL*-контекст, в котором создаётся объект изображения, *flags* — битовое поле для указания информации об изображении как объекте памяти (если величина поля — нулевая, то величиной по умолчанию будет `CL_MEM_READ_WRITE`), *image_format* — указатель на структуру, которая описывает свойства изображения (см. описание дескриптора формата изображения далее), *image_width* и *image_height* — ширина и высота изображения в пикселах (они должны быть не менее единицы), *host_ptr* – указатель на данные в изображении.

Возвращает функция `clCreateImage2D()` отличный от нуля идентификатор объекта изображения и через *errcode_ret* ошибка устанавливается в `CL_SUCCESS`, если объект успешно создан. В противном случае функция возвращает значение `NULL` и один из кодов ошибки: `CL_INVALID_CONTEXT`, `CL_INVALID_VALUE`, `CL_INVALID_IMAGE_FORMAT_DESCRIPTOR`, `CL_INVALID_IMAGE_SIZE`, `CL_INVALID_HOST_PTR` — если какой-то из передаваемых параметров имеет недействительное значение, `CL_IMAGE_FORMAT_NOT_SUPPORTED` — если указанный формат изображения не поддерживается, `CL_INVALID_OPERATION` — если ни одно из устройств не поддерживает работу с изображениями, `CL_OUT_OF_HOST_MEMORY` или `CL_OUT_OF_RESOURCES`, — если невозможно выделить память или другие необходимые ресурсы.

Объект трёхмерного изображения создаётся функцией `clCreateImage3D()`:

```
cl_mem clCreateImage3D (
            cl_context context, cl_mem_flags flags,
            const cl_image_format *image_format,
            size_t image_width, size_t image_height,
            size_t image_depth,
            size_t image_row_pitch,
            size_t image_slice_pitch,
            void *host_ptr, cl_int *errcode_ret);
```

В дополнение к параметрам предыдущей функции здесь появляются ещё три параметра типа `size_t`: *image_depth* — «глубина» изображения, *image_row_pitch* — размер одной строки изображения, *image_slice_pitch* — размер одного «среза» изображения.

Возвращает функция `clCreateImage3D()` отличный от нуля идентификатор объекта изображения и через *errcode_ret* ошибка устанавливается в `CL_SUCCESS`, если объект успешно создан. В противном случае функция возвращает значение `NULL` и один кодов ошибки предыдущей функции `clCreateImage2D()`, причём ошибка `CL_INVALID_IMAGE_SIZE` может относиться здесь и к дополнительным трём параметрам рассматриваемой функции.

Начиная с версии *OpenCL 1.2* обе функции объявлены «нежелательными», вместо них предлагается использовать одну новую функцию `clCreateImage()`, совмещающую в себе создание 1D-изображения или 1D-буфера изображения или 1D-массива изображений, 2D-изображения или 2D-массива изображений, а также 3D-изображения.

```
cl_mem clCreateImage (
            cl_context context, cl_mem_flags flags,
            const cl_image_format *image_format,
            const cl_image_desc *image_desc,
            void *host_ptr, cl_int *errcode_ret);
```

Появившийся здесь новый параметр *image_desc* является указателем на такую структуру:

```
typedef struct _cl_image_desc {
  cl_mem_object_type image_type;
  size_t image_width;
  size_t image_height;
  size_t image_depth;
  size_t image_array_size;
  size_t image_row_pitch;
  size_t image_slice_pitch;
  cl_uint num_mip_levels;
  cl_uint num_samples;
  cl_mem buffer;
} cl_image_desc;
```

Тип изображения может быть выбран из поддерживаемых выбранной реализацией *OpenCL*, `image_array_size` — количество изображений в массиве (если это массив изображений), параметры `num_mip_levels` и `num_samples` никак не объясняются и должны быть равны `0`; `buffer` используется лишь если тип `image_type` изображения есть `CL_MEM_OBJECT_IMAGE1D_BUFFER`, в остальных ситуациях он должен быть равен `NULL`.

### Описатели изображений (*Image Format Descriptor*)

Структура описателя формата изображения весьма проста и состоит из порядка каналов в изображении и типа данных одного канала:

```
typedef struct _cl_image_format {
  cl_channel_order image_channel_order;
  cl_channel_type image_channel_data_type;
} cl_image_format;
```

Заметим, что количество каналов указывается здесь неявно — с помощью порядка каналов. Типы данных в канале изображения допускаются такие: `CL_SNORM_INT8`, `CL_SNORM_INT16`, `CL_UNORM_INT8`, `CL_UNORM_INT16` — нормализованные вещественные значения со знаком или без знака, хранимые как 8- или 16-битные целые, диапазон — [0.0...1.0] для типов без знака и [−1.0...1.0] — для типов со знаком; `CL_UNORM_SHORT_565`, `CL_UNORM_SHORT_555`, `CL_UNORM_INT_101010` — т.н. упакованные, они предполагают вполне определённый порядок каналов (`CL_RGB` или `CL_RGBx`); `CL_SIGNED_INT8`, `CL_SIGNED_INT16`, `CL_SIGNED_INT32`, `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16`, `CL_UNSIGNED_INT32` — целые ненормализованные со знаком или без знака, занимающие 1, 2 или 4 байта; `CL_HALF_FLOAT`, `CL_FLOAT` — каждая компонента канала вещественная, половинной или одинарной точности соответственно.

Значения, которые может принимать величина типа `cl_channel_order`: `CL_R`, `CL_Rx`, `CL_A` (одноканальные изображения любого типа, кроме т.н. упакованных), `CL_INTENSITY`, `CL_LUMINANCE` (могут использоваться только с вещественными типами), `CL_RG`, `CL_RGx`, `CL_RA` (двухканальные изображения), `CL_RGBA`, `CL_ARGB`, `CL_BGRA` (четырёхканальные изображения), `CL_RGB`, `CL_RGBx` (упакованные изображения, порядок каналов — традиционный, используются только типами данных `CL_UNORM_SHORT_565`, `CL_UNORM_SHORT_555` или `CL_UNORM_INT_101010`).

В зависимости от значения величины, определяющей порядок каналов в изображении, функции чтения из изображений возвращают вполне определённые компоненты изображения:

| **Порядок каналов** | **Компоненты** |
|---|---|
| `CL_R`, `CL_Rx` | `(R,0,0,1)` |
| `CL_A` | `(0,0,0,A)` |
| `CL_RG`, `CL_RGx` | `(R,G,0,1)` |
| `CL_RA` | `(R,0,0,A)` |
| `CL_RGB`, `CL_RGBx` | `(R,G,B,1)` |
| `CL_RGBA`, `CL_ARGB`, `CL_BGRA` | `(R,G,B,A)` |
| `CL_INTENSITY` | `(I,I,I,I)` |
| `CL_LUMINANCE` | `(L,L,L,1)` |

### Список поддерживаемых форматов изображений

Функция `clGetSupportedImageFormats()` с таким прототипом

```
cl_int clGetSupportedImageFormats (
            cl_context context,
            cl_mem_flags flags,
            cl_mem_object_type image_type,
            cl_uint num_entries,
            cl_image_format *image_formats,
            cl_uint *num_image_formats);
```

может быть использована для получения списка форматов изображений, поддерживаемых реализацией *OpenCL*, если заданы контекст, тип изображения и информация о его выделении (т.е., три первых параметра функции).

### Минимальный список поддерживаемых форматов

Для двумерных и трёхмерных изображений должен поддерживаться некоторый минимальный список форматов изображений (как для чтения, так и для записи): четырёхканальные изображения с порядком `CL_RGBA` и типами `CL_UNORM_INT8`, `CL_UNORM_INT16`, `CL_SIGNED_INT8`, `CL_SIGNED_INT16`, `CL_SIGNED_INT32`, `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16`, `CL_UNSIGNED_INT32`, а также четырёхканальные изображения с порядком `CL_BGRA` и типом `CL_UNORM_INT8`.

### Чтение и запись объектов изображений

`clEnqueueReadImage()` и `clEnqueueWriteImage()` — функции чтения и записи объектов изображений — похожи на аналогичные функции для буферных объектов, с той лишь разницей, что область изображения указывается чуть сложнее, чем участок (одномерного) буфера: вместо двух параметров типа `size_t` (*`offset`* и *`cb`*) имеется восемь параметров этого типа, из которых первые шесть организованы в два трёхмерных массива: *`origin`*`[3]`, *`region`*`[3]`, *`row_pitch`*, *`slice_pitch`*. Смысл массивов — тот же: смещения и размеры, но теперь по отдельным измерениям изображения. Если изображение не трёхмерное, а двумерное, третье значение для смещения равно `0`, а для размера — `1`. Последние два параметра — размер в байтах строки и слоя (если изображение — трёхмерное).

### От изображений к буферным объектам и обратно

Обмен информацией между изображениями и буферными объектами тоже возможен, для этого имеются функции `clEnqueueCopyImageToBuffer()` и `clEnqueueCopyBufferToImage()`. Они объединяют в своих прототипах многие из уже обсуждавшихся параметров: дескрипторы очереди и объектов памяти, «положение» начала в изображении и в буфере, размеры (фрагмента) изображения (но не буфера), информацию о списке ожидания, указатель для возвращения объекта события — и, вероятно, являются неблокирующими, хотя явно это вроде бы нигде не указано.

```
cl_int clEnqueueCopyImageToBuffer (
            cl_command_queue command_queue,
            cl_mem src_image,
            cl_mem dst_buffer,
            const size_t src_origin[3],
            const size_t region[3],
            size_t dst_offset,
            cl_uint num_events_in_wait_list,
            const cl_event *event_wait_list,
            cl_event *event);

cl_int clEnqueueCopyBufferToImage (
            cl_command_queue command_queue,
            cl_mem src_buffer,
            cl_mem dst_image,
            size_t src_offset,
            const size_t dst_origin[3],
            const size_t region[3],
            cl_uint num_events_in_wait_list,
            const cl_event *event_wait_list,
            cl_event *event);
```

### Получение информации об изображениях

Узнать параметры изображения *image* можно с помощью функции `clGetImageInfo()`:

```
cl_int clGetImageInfo (
            cl_mem image,
            cl_image_info param_name,
            size_t param_value_size,
            void *param_value,
            size_t *param_value_size_ret);
```

Задавая в качестве *param_name* значение `CL_IMAGE_FORMAT`, можно узнать его формат, `CL_IMAGE_ELEMENT_SIZE` — размер элемента, `CL_IMAGE_WIDTH` или `CL_IMAGE_HEIGHT` или `CL_IMAGE_DEPTH` — размеры по нужному измерению и т.д.

## 13.3. Общие функции для работы с объектами памяти

Поскольку и буферы, и изображения в *OpenCL* имеют один и тот же тип `cl_mem`, т.е., являются объектами памяти, к ним одинаково применимы некоторые общие функции, скажем, функция освобождения объекта памяти:

```
cl_int clReleaseMemObject (cl_mem memobj);
```

В *OpenCL 1.2* появилась также функция удержания объекта памяти:

```
cl_int clRetainMemObject (cl_mem memobj);
```

Такова же функция получения информации об объекте памяти *memobj*:

```
cl_int clGetMemObjectInfo (
            cl_mem memobj,
            cl_mem_info param_name,
            size_t param_value_size,
            void *param_value,
            size_t *param_value_size_ret);
```

Можно узнать: тип объекта памяти (CL_MEM_TYPE) — буфер (CL_MEM_OBJECT_BUFFER), двумерное (CL_MEM_OBJECT_IMAGE2D) или трёхмерное (CL_MEM_OBJECT_IMAGE3D) изображение, флаги (CL_MEM_FLAGS), размер (CL_MEM_SIZE), счётчик ссылок (CL_MEM_REFERENCE_COUNT) и другие характеристики.

## 13.4. Сэмплеры (*Sampler Objects*)

Объект сэмплер описывает, как извлекаются элементы изображения, когда оно читается в коде ядра: какая используется координатная система (целочисленные или нормализованные координаты), как обрабатывается ситуация выхода координат за пределы изображения (обнуление или установка «цвета» ближайшего элемента), а также производится ли интерполяция при извлечении значений, расположенных «между» элементами изображения.

> Следует иметь в виду, что типы изображений и сэмплеров — не совсем «настоящие»: они определены, только если поддерживаются устройством; нельзя объявлять массивы таких величин и указатели на них; наконец, они не могут быть частью структуры.

Встроенные функции чтения элементов изображений (read_imageui(), read_imagef() и др.) принимают сэмплеры в качестве параметра.

> Вместе с изображениями могут быть указаны квалификаторы доступа, показывающие, является ли изображение доступным только для чтения (__read_only) или только для записи (__write_only). Связано это с ограничениями некоторых современных *GPU*, которые не позволяют из ядра одновременно и читать, и записывать изображение. Причина — кэширование операций чтения, из-за чего запись в изображение никак не меняет содержимое кэша, из которого изображение потом будет читаться.

Разумеется, у сэмплеров, как и у многих объектов *OpenCL*, имеются функции создания/удержания/освобождения, а также функция получения информации о них. Однако для большинства программ достаточно статического создания сэмплера на время работы конкретного ядра (см. пример на стр.61), поэтому эти функции здесь не рассматриваются.

## 13.5. Практический пример работы с буферами и изображениями

В брошюре **«Introduction to GPU Computing with CUDA and OpenCL»** (***J.Shearer***, ***W.Chen***; http://vidi.cs.ucdavis.edu/Programming/OpenCL/Shearer_gpgpu.pdf) в четвёртой главе под названием ***«OpenCL in Practice»*** (стр.16 – 26) разобран поучительный пример программы, формирующей по заданному изображению сначала чёрно-белое изображение со значениями яркостей исходного изображения, а затем на его основе фильтрующей исходное изображение так, чтобы вместо каждого элемента осталось максимальное значение по некоторой его окрестности.

Более того, рассматриваются два варианта этой программы: вариант, использующий объекты-буферы, и вариант, использующий объекты-изображения. И хотя полного текста программы найти не удалось, её вполне можно восстановить, поскольку в брошюре приведены исходные тексты ключевых фрагментов. Рассмотрим далее эти фрагменты (исходный текст немного подкорректирован и в несущественных деталях сокращён; функции error_and_exit() и textFromFile() считаем реализованными).

### Ядра вычисления яркости и фильтрации — вариант для буферов

Параметрами ядра `luminance` являются два указателя на глобальную память, на места, где должны располагаться элементы первого (цветного) изображения (тип — `uchar4`) и элементы второго (чёрно-белого) изображения (тип — `uchar`).

```
__kernel void luminance(__global uchar4 *orig, __global uchar *lum)
{
//  index = y * width + x
  uint gid = get_global_id(1)*get_global_size(0) + get_global_id(0);
  uchar4 pixel = orig[gid]; lum[gid] = pixel.x*0.3 + pixel.y*0.59 + pixel.z*0.11;
}
```

Параметрами ядра `max_filter` являются: целое без знака (размер области), указатель на элементы исходного цветного изображения (тип — `uchar4`), указатель на элементы чёрно-белого изображения со значениями яркости (тип — `uchar`) и указатель на результат фильтрации (изображение с элементами типа `uchar4`).

```
__kernel void max_filter(const uint width,
                         __global uchar4 *orig, __global uchar *lumi,
                         __global uchar4 *result)
{
  int i, j, index, maxindex = 0;
  uchar lum, maxlum = 0;
  int4 range;

  int x = get_global_id(0);
  int y = get_global_id(1);
  int xdim = get_global_size(0);
  int ydim = get_global_size(1);

  uint gid = mad24(y, xdim, x);

  int xlo = x - (width>>1); int xhi = x + (width>>1);
  int ylo = y - (width>>1); int yhi = y + (width>>1);

  xlo = xlo < 0 ? 0 : xlo; xhi = xhi > xdim - 1 ? xdim - 1 : xhi;
  ylo = ylo < 0 ? 0 : ylo; yhi = yhi > ydim - 1 ? ydim - 1 : yhi;

  for (i = xlo; i <= xhi; i++) {
    for (j = ylo; j <= yhi; j++) {
      index = mad24(j, xdim, i);

      lum = lumi[index];
      maxindex = lum > maxlum ? index : maxindex;
      maxlum   = lum > maxlum ? lum : maxlum;
    }
  }

  result[gid] = orig[maxindex];
}
```

### Выделение памяти, инициализация — вариант для буферов

```
  cl_int err;
  int size = IMAGE_WIDTH*IMAGE_HEIGHT*4;

  unsigned char *data = (unsigned char *) calloc(sizeof(unsigned char), size);
  FILE* orig = fopen(IMAGE_FILE, "rb");
  fread((void*) data, sizeof(unsigned char), size, orig); fflush(orig); fclose(orig);

  original = clCreateBuffer(context, CL_MEM_READ_ONLY,
                            sizeof(unsigned char)*size, NULL, &err);
```

```
err = clEnqueueWriteBuffer(commands, original, CL_TRUE, 0, size, data, 0, NULL, NULL);
if (err != CL_SUCCESS) error_and_exit("WriteBuffer", err);

luminance = clCreateBuffer(context, CL_MEM_READ_WRITE,
                           sizeof(unsigned char)*IMAGE_WIDTH*IMAGE_HEIGHT, NULL, &err);
if (!luminance) error_and_exit("clCreateBuffer", err);

result = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                        sizeof(unsigned char)*size, NULL, &err);
if (!result) error_and_exit("clCreateBuffer", err);

free(data);
```

### Компиляция ядер и построение объекта программы

Здесь предполагается, что оба ядра размещены в одном исходном файле `kernels.cl` и называются `luminance` и `max_filter`.

```
cl_int err;
char *source = NULL;
size_t length = 0, len;

textFromFile("kernels.cl", &source, &length);

program = clCreateProgramWithSource(context, 1, (const char**)&source, NULL, &err);
if (!program)  error_and_exit("clCreateProgramWithSource", err);

err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS) {
  char buffer[2048];

  fprintf(stderr, "Failed to build CL source.  Error log:\n");
  clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG,
                        sizeof(buffer), buffer, &len) ;

  fprintf(stdout, "%s\n", buffer);
  return EXIT_FAILURE;
}

klum = clCreateKernel(program, "luminance", &err);
if (!klum || err != CL_SUCCESS) error_and_exit("clCreateKernel", err);

kmaxf = clCreateKernel(program, "max_filter", &err);
if (!kmaxf || err != CL_SUCCESS) error_and_exit("clCreateKernel", err);
```

**!!!** Неудачным здесь является использование статического буфера для лога компиляции ядер.

Этот фрагмент кода нет необходимости модифицировать при смене варианта реализации ядер, если имя файла с ядрами (`kernels.cl`), а также имена функций-ядер (`luminance`, `max_filter`) остаются неизменными.

### Запуск ядер на исполнение

Обратите внимание на то, что в этом фрагменте кода при смене варианта реализации ядер тоже можно ничего не менять: количество параметров в ядрах и их порядок — такие же, а типы параметров в функциях-ядрах хотя и меняются при смене варианта, с точки зрения хост-программы остаются неизменными, это просто объекты памяти типа `cl_mem`.

```
unsigned int i, t, width = FILTER_WIDTH;
cl_event event = NULL;
cl_int err = CL_SUCCESS;

unsigned char* result_host = (unsigned char*) calloc(sizeof(unsigned char),
```

```
                                IMAGE_WIDTH*IMAGE_HEIGHT*4);

    unsigned int arg = 0;
    err |= clSetKernelArg(klum, arg++, sizeof(cl_mem), &original);
    err |= clSetKernelArg(klum, arg++, sizeof(cl_mem), &luminance);

    arg = 0;
    err |= clSetKernelArg(kmaxf, arg++, sizeof(unsigned int), &width);
    err |= clSetKernelArg(kmaxf, arg++, sizeof(cl_mem), &original);
    err |= clSetKernelArg(kmaxf, arg++, sizeof(cl_mem), &luminance);
    err |= clSetKernelArg(kmaxf, arg++, sizeof(cl_mem), &result);

    if (err != CL_SUCCESS) error_and_exit("clSetKernelArg", err);

    size_t global_dim[2] = { IMAGE_WIDTH, IMAGE_HEIGHT };

    err  = clEnqueueNDRangeKernel(commands, klum,  2, NULL, global_dim,
                                  NULL, 0, NULL, NULL);
    err |= clEnqueueNDRangeKernel(commands, kmaxf, 2, NULL, global_dim,
                                  NULL, 0, NULL, &event);

    err |= clEnqueueReadBuffer(commands, result, CL_TRUE, 0, sizeof(unsigned char)
                               *IMAGE_WIDTH*IMAGE_HEIGHT*4, result_host, 1, &event, NULL);

// . . . . . .

    free(result_host);
```

Обработка «рабочего пространства» производится последовательно: сначала с помощью объекта ядра `klum` (ядро `luminance`), затем — с помощью объекта ядра `kmaxf` (ядро `max_filter`); ошибки запуска ядер и чтения результата обработки здесь не проверяются!

### Ядра вычисления яркости и фильтрации — вариант для изображений

Тут появляются сэмплеры для работы с объектами изображений и функции чтения/записи.

В этом варианте параметрами ядра `luminance` являются дескрипторы изображений: первого (цветного) изображения (тип — `image2d_t`) и второго (чёрно-белого; тип — тоже `image2d_t`), а параметрами ядра `max_filter` — целое без знака (размер области) и три дескриптора (тип — `image2d_t`): исходного цветного изображения, чёрно-белого изображения со значениями яркости и результата фильтрации.

```
const sampler_t sampler_plain = CLK_NORMALIZED_COORDS_FALSE |
                                CLK_ADDRESS_NONE |
                                CLK_FILTER_NEAREST;
const sampler_t sampler_clamp = CLK_NORMALIZED_COORDS_FALSE |
                                CLK_ADDRESS_CLAMP_TO_EDGE |
                                CLK_FILTER_NEAREST;

__kernel void luminance(__read_only image2d_t input, __write_only image2d_t luminance)
{
  int2 coord = (int2)(get_global_id(0), get_global_id(1));

  uint4 pixel = read_imageui(input, sampler_plain, coord);
  uint4 lum = (uint4)(pixel.x*0.30 + pixel.y*0.59 + pixel.z*0.11);

  write_imageui(luminance, coord, lum);
}

__kernel void max_filter(const uint width, __read_only image2d_t input,
                         __read_only image2d_t luminance, __write_only image2d_t output)
{
  uint maxlum = 0;
```

```
  uint4 lum;
  int i, j;

  int2 coord = (int2)(get_global_id(0),get_global_id(1));
  int2 maxcoord;

  int xlo = coord.x - (width>>1); int xhi = coord.x + (width>>1);
  int ylo = coord.y - (width>>1); int yhi = coord.y + (width>>1);

  for (i = xlo; i <= xhi; i++) {
    for (j = ylo; j <= yhi; j++) {

      lum = read_imageui(luminance, sampler_clamp, (int2)(i,j));
      if (lum.x > maxlum) {
        maxlum = lum.x;
        maxcoord = (int2)(i,j);
      }
    }
  }

  uint4 value = read_imageui(input, sampler_clamp, maxcoord);
  write_imageui(output, coord, value);
}
```

**Замечание.** Несмотря на то, что ядра в двух рассматриваемых вариантах обрабатывают вроде бы разные типы объектов (`__global uchar4*` и `__global uchar*` в первом и `image2d_t` — во втором), при смене варианта ядер можно ничего не менять в тексте хост-программы: ядра в обоих случаях называются одинаково, а различия в типах передаваемых им величин будут несущественны, т.к. все эти типы для хост-программы — просто `cl_mem`.

### Изменения в работе с памятью — вариант для изображений

Здесь используются определённые ранее (в варианте для буферов) переменные `err` и `size`, а также заполненный из файла динамический массив `data`.

```
// cl_int err;
// int size = IMAGE_WIDTH*IMAGE_HEIGHT;

  cl_image_format format;
  format.image_channel_order = CL_RGBA;
  format.image_channel_data_type = CL_UNSIGNED_INT8;

  original = clCreateImage2D(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, &format,
                IMAGE_WIDTH, IMAGE_HEIGHT, IMAGE_WIDTH*4, data, &err);

  luminance = clCreateImage2D(context, CL_MEM_READ_WRITE, &format,
                IMAGE_WIDTH, IMAGE_HEIGHT, 0, NULL, &err);

  result = clCreateImage2D(context, CL_MEM_WRITE_ONLY, &format,
                IMAGE_WIDTH, IMAGE_HEIGHT, 0, NULL, &err);

  if (!original || !luminance || !result || err != CL_SUCCESS)
    error_and_exit("clCreateImage2d", err);
```

По сравнению с вариантом для буферов здесь код чуть проще ещё из-за неявного копирования данных (флаг `CL_MEM_COPY_HOST_PTR` при создании изображения `original`) из динамического массива `data` (и «совмещённой» проверки ошибок в конце при создании трёх изображений).

## 13.6. Задание для самостоятельной работы

Попробуйте создать из приводимых фрагментов работоспособную программу.

# 14. *OpenCL-OpenGL* Interoperability

Совместная работа *OpenCL* и *OpenGL* предполагает, что данные должны передаваться от *OpenCL* к *OpenGL* и обратно, причём довольно часто, скажем, от кадра к кадру. Лучше всего, если бы при этом не происходило никакого копирования данных вообще (поскольку в случае применения *GPU* и вычисления, и отображение используют одну и ту же память графического процессора), а перемещалось лишь «право владения» ими, в то время как сами данные находились бы в каком-то совместном (и поочерёдном) владении. Но т.к. ни вызовы *OpenCL*, ни вызовы *OpenGL* не исполняются немедленно, а лишь помещаются в свои отдельные очереди команд, требуется определённая координация их совместной работы со стороны хоста. При этом не следует забывать, что объекты *OpenCL* создаются на основе объектов *OpenGL*, но не наоборот.

Совместная работа *OpenCL* и *OpenGL* реализуется как расширение *Khronos* с именем `cl_khr_gl_sharing`, поэтому прежде всего надо проверить, что данное расширение поддерживается реализацией. Это так для большинства современных *GPU*-реализаций *OpenCL*, а потому там объекты *OpenGL* (буферы и текстуры) могут использоваться как объекты *OpenCL* (буферы и изображения).

> **Замечание.** Для *Apple* (*OSX*/*iOS*) проверка поддержки расширения немного отличается: вместо проверки наличия строки `"cl_khr_gl_sharing"` в списке поддерживаемых расширений устройства (т.е., информации, полученной при вызове функции `clGetDeviceInfo()` со значением параметра `CL_DEVICE_EXTENSIONS`) проверяется наличие строки `"cl_APPLE_gl_sharing"` (см., например, файл презентации `sa10-dg-opencl-gl-interop.pdf` в ссылках далее).

Программный интерфейс этого расширения находится в заголовочном файле `cl_gl.h`. Как и в случае других расширений, для получения указателей на реальные функции следует применять вызов функции `clGetExtensionFunctionAddressForPlatform()`. Вот какие новые функции доступны при использовании рассматриваемого расширения:

`clGetGLContextInfoKHR()`
: Опрос устройств, связанных с контекстом *OpenGL*.

`clCreateFromGLBuffer()`
: Создание объекта буфера *OpenCL* из объекта буфера *OpenGL*.

`clCreateFromGLTexture2D()`
`clCreateFromGLTexture3D()`
`clCreateFromGLTexture()`
: Создание объекта изображения *OpenCL* из текстурного объекта *OpenGL*. Первые две функции использовались в версиях *OpenCL 1.0* и *1.1*. Начиная с версии *1.2* вместо них предлагается использовать третью функцию.

`clCreateFromGLRenderbuffer()`
: Создание двумерного изображения *OpenCL* из буфера отрисовки *OpenGL*.

`clGetGLObjectInfo()`
: Получение тип и имени объекта *OpenGL*, использованного для создания объекта памяти *OpenCL*.

`clGetGLTextureInfo()`
: Получение дополнительной информации о текстурном объекте *OpenGL*, связанном с объектом памяти *OpenCL*.

`clEnqueueAcquireGLObjects()`
: Получение (во временное распоряжение) объектов памяти *OpenCL* от *OpenGL*.

`clEnqueueReleaseGLObjects()`
: Возвращение объектов памяти *OpenCL* в распоряжение *OpenGL*.

## 14.1. Совместное владение и синхронизация — общие соображения

Как уже отмечалось выше, приложение должно правильно синхронизировать доступ к совместно используемым *OpenCL*/*OpenGL* объектам — это необходимо для сохранения целостности данных. Скажем, до вызова функции `clEnqueueAcquireGLObjects()` нужно убедиться, что любые *OpenGL* операции с объектами в контексте уже завершены. Это достигается вызовом функции `glFinish()`. Аналогично, до выполнения последующих команд *OpenGL*, затрагивающих «освобождённые» объекты (после того, как была вызвана функция `clEnqueueReleaseGLObjects()`), приложение должно быть уверено, что операции *OpenCL* над объектами закончены. Это может быть сделано, например, с помощью вызова функции `clFinish()`, и этот способ используется далее.

## 14.2. Создание контекста *OpenCL* для совместной работы

Поскольку объекты памяти *OpenCL* при совместной работе *OpenCL* и *OpenGL* создаются из объектов *OpenGL*, необходим контекст, который совместно разделяют и *OpenCL*, и *OpenGL*. Чтобы избежать явного копирования, важно создавать контекст *OpenCL* для того же устройства, что управляет контекстом *OpenGL*; это можно сделать, если указать флаг `CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR` при вызове `clGetGLContextInfoKHR()`.

Вообще же устройств, которые могут разделять данные с контекстом *OpenGL*, может быть больше, правда, при этом такое разделение, скорее всего, будет использовать копирование данных. Вот как, например, можно получить список всех устройств, для которых в принципе возможно разделение данных между *OpenCL* и *OpenGL* (в этом способе используется функция `clGetGLContextInfoKHR()` с флагом `CL_DEVICES_FOR_GL_CONTEXT_KHR`):

```
size_t bytes = 0;

// Определение необходимого места
clGetGLContextInfoKHR(props, CL_DEVICES_FOR_GL_CONTEXT_KHR, 0, NULL, &bytes);
// Реальное количество устройств
size_t devNum = bytes/sizeof(cl_device_id);
// Выделение памяти на весь список
std::vector<cl_device_id> devs(devNum);
// Получение всего списка устройств
clGetGLContextInfoKHR(props, CL_DEVICES_FOR_GL_CONTEXT_KHR, bytes, devs, NULL));
// Просмотр свойств этих устройств
for (size_t i = 0; i < devNum; i++)
{
    // Получение сведений о каждом устройстве
    clGetDeviceInfo(devs[i], CL_DEVICE_TYPE, ...);
    ...
    clGetDeviceInfo(devs[i], CL_DEVICE_EXTENSIONS, ...);
    ...
}
```

Получение подобного контекста предполагает сначала создание набора свойств контекста (обратите внимание, что здесь и в нескольких фрагментах далее для экономии места закрывающие фигурные скобки в коде расположены нестандартно),

```
cl_context_properties props[] = {
. . .
0};
```

который затем передаётся в качестве параметра функции `clCreateContext()`. Эти наборы состоят из пар имя–значение для каждого необходимого свойства, завершаемых в конце нулевым значением, и, к сожалению, специфичны для каждой конкретной системной платформы, хотя часть свойств для отдельных платформ может совпадать.

Например, обязательно понадобится указать текущий контекст *OpenGL*; делается это с помощью функции `wglGetCurrentContext()` (для *Win32*) или `glXGetCurrentContext()` (для систем *X Window*).

```
HGLRC glCtx = wglGetCurrentContext();      // Win32

GLXContext glCtx = glXGetCurrentContext(); // X Window
```

Нужен также и контекст дисплея, который получается путём вызова, соответственно, функций `wglGetCurrentDC()` или `glXGetCurrentDisplay()`.

Таким образом, набор свойств, на основе которых создаётся контекст *OpenCL* в случае *Win32*, может выглядеть так:

```
cl_context_properties props[] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties) platform,
    CL_WGL_HDC_KHR, (cl_context_properties) wglGetCurrentDC(),
    CL_GL_CONTEXT_KHR, (cl_context_properties) glCtx,
    0};
```

Для случая *X Window* системы — например, так:

```
cl_context_properties props[] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties) platform,
    CL_GLX_DISPLAY_KHR, (cl_context_properties) glXGetCurrentDisplay(),
    CL_GL_CONTEXT_KHR, (cl_context_properties) glCtx,
    0};
```

Можно также вот так объединить код инициализации массива свойств контекста *OpenCL* с помощью директивы условной компиляции — если контексты, получаемые с помощью функций, больше нигде в программе не используются:

```
cl_context_properties props[] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties) platform,
#ifdef _WIN32
    CL_WGL_HDC_KHR, (cl_context_properties) wglGetCurrentDC(),
    CL_GL_CONTEXT_KHR, (cl_context_properties) wglGetCurrentContext(),
#else //!_WIN32
    CL_GLX_DISPLAY_KHR, (cl_context_properties) glXGetCurrentDisplay(),
    CL_GL_CONTEXT_KHR, (cl_context_properties) glXGetCurrentContext(),
#endif
    0};
```

Аналогичные действия для систем *Apple* будут немного другими; соответствующий код встречается не так часто, но всё же в некоторых примерах его удалось найти (см., например, вышеупомянутый файл презентации `sa10-dg-opencl-gl-interop.pdf` или проект , автор — ***Arthur Nishimoto***). В последнем проекте исходный код написан так, что позволяет компиляцию на совершенно разных системных платформах (приводимый ниже фрагмент кода немного подправлен для сохранения совместимости с тем, что изложено выше; обратите также внимание на *C++*-стиль приведения типа в случае платформы *Win32*):

```
#if defined (__APPLE__)
  CGLContextObj kCGLContext = CGLGetCurrentContext();
  CGLShareGroupObj kCGLShareGroup = CGLGetShareGroup(kCGLContext);
  cl_context_properties props[] = {
      CL_CONTEXT_PROPERTY_USE_CGL_SHAREGROUP_APPLE, (cl_context_properties)kCGLShareGroup,
      0};
#else
  #ifdef _WIN32
    cl_context_properties props[] = {
      CL_CONTEXT_PLATFORM, reinterpret_cast<cl_context_properties>(platform),
      CL_GL_CONTEXT_KHR, reinterpret_cast<cl_context_properties>(wglGetCurrentContext()),
      CL_WGL_HDC_KHR, reinterpret_cast<cl_context_properties>(wglGetCurrentDC()),
      0};
```

```
  #else
    cl_context_properties props[] = {
      CL_GL_CONTEXT_KHR, (cl_context_properties)glXGetCurrentContext(),
      CL_GLX_DISPLAY_KHR, (cl_context_properties)glXGetCurrentDisplay(),
      CL_CONTEXT_PLATFORM, (cl_context_properties)platform,
      0};
  #endif
#endif
```

После этого можно создавать контекст (вместе с очередью команд) для какого-то выбранного устройства:

```
ctx = clCreateContext(props, 1, &dev, NULL, NULL, NULL);
queue = clCreateCommandQueue(ctx, dev, 0, NULL);
```

или, если требуется профилирование исполнения команд:

```
queue = clCreateCommandQueue(ctx, dev, CL_QUEUE_PROFILING_ENABLE, NULL);
```

**Замечание.** В случае создания разделяемого контекста *Apple* вызов будет несколько проще:

```
ctx = clCreateContext(props, 0, 0, NULL, NULL, NULL);
```

## 14.3. Создание совместно используемых объектов *OpenCL*

Следующий этап — создание объектов *OpenCL* из существующих объектов *OpenGL*. Из объекта буфера *OpenGL* можно создать объект буфера *OpenCL* (предполагается, что оба уже объявлены ранее):

```
bufferCL = clCreateFromGLBuffer(ctx, CL_MEM_READ_WRITE, bufferGL, &err);
```

Либо из существующего объекта текстуры *OpenGL* можно создать изображение *OpenCL*:

```
imageCL = clCreateFromGLTexture2D(ctx, CL_MEM_WRITE_ONLY,
                                  GL_TEXTURE_2D, 0, textureGL, &err);
```

## 14.4. Координирование совместной работы *OpenCL* и *OpenGL*

Далее остаётся запустить ядра для работы с этими разделяемыми объектами. Некоторые дополнительные команды понадобятся, чтобы «объяснить» реализации, что необходимо передавать владение объектами к или от *OpenCL* — в зависимости от того, следует ли производить обработку объектов или же их отображение:

```
// Очистить очередь команд GL
glFinish();
// Получить разделяемые объекты в распоряжение OpenCL
err = clEnqueueAcquireGLObjects(queue, 1, &bufferCL, 0, NULL, NULL);
. . .
// Передать в очередь команды для осуществления операций
// (запуска ядер на обработку, команды чтения/записи и т.п.)
. . .
// Освободить разделяемые объекты
err = clEnqueueReleaseGLObjects(queue, 1, &bufferCL, 0, NULL, NULL);
. . .
// Очистить очередь команд CL
clFinish(queue);
```

## 14.5. Практические примеры программ

Данные примеры — из набора программ к книге **«OpenCL Programming Guide»**, 2011, авторы — ***A.Munshi***, ***B.Gaster***, ***T.Mattson***, ***J.Fung***, ***D.Ginsburg***, ISBN 9780321749642.

### Демонстрационная программа `GLinterop` из главы 10

В проект необходимо включить исходный файл `GLinterop.cpp` и — для простоты — библиотеки `freeglut.lib` и `glew32.lib`; в каталог проекта надо также поместить файл с ядрами `GLinterop.cl` и динамические библиотеки `freeglut.dll` и `glew32.dll` (для упрощения запуска программы из среды *Visual Studio*). Если впоследствии понадобится запускать программу автономно, последние три файла должны быть в каталоге исполняемого файла.

Установки для компиляции и сборки программы: пути поиска заголовочных файлов — `$(CUDA_INC_PATH);$(NVSDKCOMPUTE_ROOT)\C\common\inc;` пути для поиска библиотек — `$(CUDA_LIB_PATH)` (две библиотеки искать не нужно, поскольку они включены в проект, хотя их можно было также взять из *NVidia SDK*, как и заголовочные файлы к ним); дополнительные библиотеки — `OpenCL.lib OpenGL32.lib` (остальные — включены).

Обратите внимание на то, что для работы на платформе *Apple* программа не завершена, т.к. нет формирования свойств, необходимых для создания контекста; впрочем, это легко исправить, используя код, приводимый выше.

### Демонстрационная программа `sinewave` из главы 12

Пример из главы 12 (использует файлы `sinewave.cpp`, `sinewave.cl`) можно построить аналогично предыдущему, но гораздо интереснее поработать с его вариацией (автор ***Rob Farber***), опубликованной в одной из статей цикла, посвященного *OpenCL* (статья называется ***«Part 6: Primitive Restart and OpenGL Interoperability»***).

В статье полностью приведены тексты обоих необходимых файлов (`gltest.cpp` и ядра `sinewave.cl`), а также подробно поясняются все значимые части исходного кода. Легко заметить, что по отношению к оригиналу автор усовершенствовал программу, добавив в неё изменение массива цветов вершин с течением времени в коде ядра, из-за чего у ядра появился ещё один параметр — указатель на массив цветов вершин. При этом программа стала немного «веселее», т.к. теперь отображаемая поверхность изменяется не только интерактивно по углу зрения и автоматически — по форме, но и автоматически — по цвету.

В ядро `sinewave` может понадобиться внести небольшие исправления: добавить явное приведение типов для аргументов функций `sin()` и `cos()` к вещественным, поскольку изначально эти аргументы — целочисленные без знака. В файл `gltest.cpp` — для компиляции под *WinXP* — пришлось также вместо `GL/glxew.h` включить `windows.h` и `stdlib.h`.

Для сборки проекта необходимо добавить в проект исходный файл `gltest.cpp` и — для простоты — библиотеки `glut32.lib` и `glew32.lib`; в каталог проекта надо также поместить файл ядра `sinewave.cl` и динамические библиотеки `glut32.dll` и `glew32.dll`.

Установки для компиляции и сборки программы: пути поиска заголовочных файлов — `$(CUDA_INC_PATH);$(NVSDKCOMPUTE_ROOT)\C\common\inc;` пути для поиска библиотек — `$(CUDA_LIB_PATH)` (две библиотеки искать не нужно, поскольку они включены в проект, хотя их можно было также взять из *NVidia SDK*, как и заголовочные файлы к ним); дополнительные библиотеки — `OpenCL.lib OpenGL32.lib` (остальные — включены).

В фунции отображения, вызываемой всякий раз, когда экран должен быть перерисован, можно выделить две разнородные части: одна связана с вычислениями, которые должны быть проделаны (до процесса отрисовки), т.е., предполагает обращение к функциям *OpenCL*, вторая связана с отображением результата вычислений и предполагает вызовы функций *OpenGL*.

Для перехода к вычислениям (и использованию совместных данных вообще) отрисовка должна быть прекращена (с помощью вызова функции `glFinish()`); после вычислительной части работы с общими данными можно приступать к отображению, завершая обращения к

этим данным со стороны *OpenCL* — с помошью вызова функции `clFinish()`, использующей идентификатор очереди в качестве параметра (фактически — освобождение очереди от *OpenCL*-команд).

Стоит обратить внимание на параметры, которые передаются всем копиям ядер: четыре параметра остаются постоянными (и при этом не перезаписываются), а один — перед каждым запуском ядер на исполнение формируется заново, это параметр, ответственный за временну**ю** эволюцию отображаемого объекта.

> Наличие специального индекса для каждого передаваемого ядру параметра указывает, скорее всего, на то, что из передаваемых параметров формируется массив, которым пользуется функция запуска ядер на исполнение. В отличие от передачи параметров функциям через стек, когда порядок заполнения параметров важен, а сами параметры «исчезают» после вызова, здесь вполне можно формировать параметры в любом порядке, изменяя их все (или только те, что надо) перед очередным запуском ядер на исполнение.

## 14.6. Варианты реализации совместного владения *OpenCL-OpenGL*

### Совместное владение текстурой с помощью `clCreateFromGLTexture()`

Наиболее эффективный метод по производительности; также позволяет модифицировать текстуру «на месте». В статье специалиста *Intel* (см. ссылку в конце) почему-то упоминается в связи с этим реализация *OpenCL* для *Intel HD Graphics*. Надо полагать, что это самый лучший метод совместного владения для этой реализации.

### Совместное владение текстурой посредством *PBO* и `clCreateFromGLBuffer()`

Этот метод требует промежуточного ***Pixel-Buffer-Object*** (***PBO***) для текстуры *OpenGL*, создаваемого с помощью `clCreateFromGLBuffer()`, последующего обновления буфера с помощью *OpenCL* и потом копирования результата обратно в текстуру. Он немного менее эффективен, чем предыдущее прямое совместное владение, поскольку требует копирования данных, однако имеет меньше ограничений на форматы текстуры.

### Совместное владение текстурой с помощью `glMapBuffer()`

Этот метод похож на предыдущий, но вместо использования `clCreateFromGLBuffer()` для совместного владения *PBO* вместе с *OpenCL*, здесь осуществляется прямолинейное отображение *PBO* в хост-память, так что даже не требуется специального расширения. Однако, это ещё медленнее, т.к. всякий раз для отображения создаётся/освобождается буфер *OpenCL*.

## 14.7. Ссылки по теме занятия

`https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/cl_khr_gl_sharing.html`
Описание расширения `cl_khr_gl_sharing` в версии *1.2* с сайта консорциума *Khronos*.

`https://software.intel.com/en-us/articles/opencl-and-opengl-interoperability-tutorial`
Очень обстоятельная статья по теме, автор — ***Maxim Shevtsov*** (*Intel*).

`http://stackoverflow.com/questions/26802905/getting-opencl-buffers-using-opencl`
Интересен подробный ответ на поставленный вопрос по теме занятия — включая формирование набора свойств контекста для работы под *Android*.

`http://sa10.idav.ucdavis.edu/docs/sa10-dg-opencl-gl-interop.pdf`
Насыщенная и полезная презентация по теме занятия. Один из немногочисленных источников, где удалось найти параметры, необходимые для создания контекста для совместной работы *OpenCL* и *OpenGL* на компьютерах *Apple*.

`https://github.com/bgaster/opencl-book-samples`
Практические примеры к книге **«OpenCL Programming Guide»** (авторы ***A.Munshi***, ***B.Gaster***, ***T.Mattson***, ***J.Fung***, ***D.Ginsburg***, 2011. ISBN 978-0-321-74964-2. По тематике занятия интересно посмотреть: `ImageFilter2D.cpp`, `ImageFilter2D.cl` (глава 8); `GLinterop.cpp`, `GLinterop.cl` (глава 10); можно также взглянуть на оригинал той программы, что привёл в своей статье ***Rob Farber***: `sinewave.cpp`, `sinewave.cl` (глава 12).

`http://www.codeproject.com/Articles/201263/Part-6-Primitive-Restart-and-OpenGL...`
Статья из цикла про *OpenCL* (автор — ***Rob Farber***). Часть утверждений автора — спорна, но приводимый пример (развитие программы `sinewave`) интереснее оригинала. Из текста статьи имеет смысл скопировать `gltest.cpp` и `sinewave.cl`, а также внести в них упомянутые выше изменения.

`http://www.eriksmistad.no/marching-cubes-implementation-using-opencl-and-opengl/`
Статья об алгоритме построения поверхностей по трёхмерным наборам точек (т.н. ***Marching Cubes***) и реализации этого алгоритма в рамках *OpenCL* (автор — ***Erik Smistad***). Исходный код к статье находится на *GitHub* (`https://github.com/smistad/GPU-Marching-Cubes`).

> Надо сказать, что почему-то в русскоязычной литературе принят довольно прямолинейный перевод выражения *«marching cubes»* как *«шагающие кубики»* (или *«марширующие кубики»*), хотя у использованного глагола *march* есть не только значение *маршировать*, но и значение *граничить*, что даёт перевод *«граничащие кубики»* (или чуть более благозвучный вариант: *«граничные кубики»*), гораздо точнее отражающий суть самого алгоритма...

## 14.8. Задание для самостоятельной работы

Откомпилировать и запустить код проекта **GPU-Marching-Cubes** из упомянутой выше статьи на реализации *OpenCL*, поддерживающей взаимодействие между *OpenCL* и *OpenGL*.

Проект состоит из нескольких файлов (кроме того, понадобятся файлы данных `*.raw`):

- исходные файлы в каталоге проекта (`main.cpp`, `gpu-mc.cpp`, `rawUtilities.cpp`);
- файлы с ядрами (`gpu-mc.cl` и `gpu-mc-morton.cl`);
- сопутствующие заголовочные файлы (`gpu-mc.hpp`, `rawUtilities.hpp`);
- файлы в подкаталоге `OpenCLUtilities` (`openCLGLUtilities.cpp`, `openCLUtilities.cpp`, `openCLGLUtilities.hpp`, `openCLUtilities.hpp`).

Для компиляции в системе *Windows* в каталог проекта следует добавить библиотеки:

- статические (добавляются также и в проект): `glew32.lib`, `glut32.lib`;
- динамические: `glew32.dll`, `glut32.dll`.

> Возможно, придётся внести исправления в пару файлов этой программы для её успешной компиляции. В файле `gpu-mc.cpp` может понадобиться явное приведение типа для функции `pow()`, а в файле `openCLGLUtilities.cpp` (для компиляции под *Windows*) надо будет добавить включение файла `windows.h`, чтобы нашлись объявления функций `wglGetCurrentContext()` и `wglGetCurrentDC()`.

Для запуска программы `GPU-Marching-Cubes` нужны параметры командной строки (имя файла с отображаемым объектом и размеры объекта по трём измерениям), например:

```
GPU-Marching-Cubes skull.raw 256 256 256
```

или:

```
GPU-Marching-Cubes hydrogenAtom.raw 128 128 128
```

# 15. Некоторые реализации *OpenCL* и их особенности

В настоящее время реализаций *OpenCL* существует уже не так мало и число их даже продолжает понемногу увеличиваться. Каждый из производителей *GPU* имеет свою реализацию *OpenCL*, поскольку создание её — в интересах самого производителя. Кроме того, эти производители являются также и членами консорциума *Khronos*...

## 15.1. Целевые устройства реализаций *OpenCL*

Целевыми устройствами реализаций производителей обычно являются те устройства, которые они разрабатывают или производят, поэтому в большинстве случаев это — *GPU* (графические процессоры). Иногда — когда производитель специализируется не только на *GPU*, но выпускает также и *CPU* (как в случае *AMD* и *Intel*), — их реализации поддерживают в качестве целевых устройств и *CPU*. При этом такая *CPU*-реализация всё равно имеет некие скоростные преимущества — за счёт того, что для параллельного выполнения операций широко используются т.н. векторные инструкции современных *CPU*.

В случае открытых реализаций целевыми устройствами могут стать те устройства, до которых — по тем или иным причинам — «не доходят руки» основных производителей.

Например, для того, чтобы компенсировать некоторый «пробел» в реализации *OpenCL* для *GPU* фирмы *Intel* (не охвачены *GPU*, входящие, в частности, в процессоры *Intel Core* младших поколений), создана открытая реализация, именуемая ***Beignet*** (следует читать, по мысли авторов, как [bɛnˈjeɪ]).

Кроме того, имеется также открытая реализация, называемая ***POCL*** (расшифровывается как ***Portable Computing Language***), и для неё целевыми устройствами являются *CPU* различных архитектур (**x86_64**, **ARM v7**, **PowerPC** и других).

> Кстати, возможность применения в *POCL* в качестве целевого устройства **ARM v7** позволяет экспериментировать с *OpenCL* на такой экзотической «игрушке», как *Raspberry Pi 2* — несмотря на то, что её *GPU* для этого не приспособлен.

## 15.2. «Настольные» варианты от *AMD*, *Intel*, *NVidia*

«Большая тройка» производителей (*AMD*, *Intel*, *NVidia*) занимается *GPU* уже давно, а потому они традиционно ориентируются на графический процессор (*GPU*) в составе дискретной или встроенной видеокарты как на целевое устройство.

Поскольку для работы программ *OpenCL* достаточно наличия в системе драйвера для целевого устройства, далее приводятся ссылки именно на страницы таких драйверов, хотя понятно, что драйвер также появится и в случае установки соответствующего *SDK* или других (более сложных) пакетов от производителя.

### AMD

*AMD OpenCL™ 2.0 Driver* — страница сайта *AMD*, посвящённая новой версии драйвера.
`http://support.amd.com/en-us/kb-articles/Pages/OpenCL2-Driver.aspx`

Ссылки на версии драйверов:

- `http://www2.ati.com/drivers/amd-14.41rc1-win8.1-64bit-opencl2-sep19.exe`
- `http://www2.ati.com/drivers/linux-amd-14.41rc1-opencl2-sep19.zip`

Поддержаны 64-разрядные версии *Windows 8.1*, *Red Hat Enterprise 6.5*, *Ubuntu 14.04*. Ссылка на полную платформу разработки параллельных программ находится на странице `http://developer.amd.com/tools-and-sdks/opencl-zone/`.

### Intel

OpenCL™ Drivers: `https://software.intel.com/en-us/articles/opencl-drivers`
Intel® SDK for OpenCL™ Applications: `https://software.intel.com/en-us/intel-opencl`

> Стоит обратить внимание на то, что *OpenCL™ 2.0* драйверы для *GPU/CPU* от *Intel* рассчитаны на относительно недавние устройства: процессоры *Intel® Core™* поколений 5,6,7, *Pentium J4000*, *Celeron J3000*, *Intel® Xeon® v4, v5*. Поддержки *GPU* в качестве целевого устройства для процессоров *Intel® Core™* поколений *1–4* и *Intel® Xeon®* поколений *1–4* — не предусмотрено!

### NVidia

Несмотря на то, что *NVidia* является членом консорциума *Khronos Group* и вовлечена в разработку спецификации *OpenCL*, о реализации версии *2.0* от неё сведений нет. Её реализация *OpenCL* не очень давно «закрепилась» на версии *1.2*, в то время как у *AMD* и *Intel* давно анонсированы и уже выпущены реализации версии *2.0*.

Поддержка *OpenCL* включена в драйверы *GPU* от *NVidia* (которые доступны по адресу `http://www.nvidia.com/drivers`), поэтому для работы *OpenCL*-программ достаточно этих дисплейных драйверов. Аналогично поддержка *OpenCL* присутствует в рамках *CUDA SDK*, поэтому появится после установки *CUDA*.

По адресу `https://developer.nvidia.com/opencl` можно найти дополнительные сведения о реализации; там же есть и ссылки на примеры программ из *NVidia OpenCL SDK*.

## 15.3. Реализации *OpenCL* для мобильных устройств

Производители *GPU* для мобильных устройств, из которых наиболее известными можно считать *Imagination* (графические процессоры **PowerVR**), *ARM* (графические процессоры **Mali**), *Qualcomm* (графические процессоры **Adreno**), а также *Vivante*, тоже заявили о поддержке *OpenCL* некоторыми своими устройствами, однако, предлагаемые реализации пока трудно рассматривать совсем серьёзно из-за отсутствия или труднодоступности драйверов для различных систем, хотя они, несомненно, есть, да и время от времени интерес к ним «подогревается» производителями. Иногда необходимые драйверы являются частью средств разработки (*SDK*); актуальные ссылки на *SDK* приведены ниже.

**PowerVR SDK** (2016 R1; iOS, Android, Linux; эмуляция – Windows, OS X, Linux)
`http://community.imgtec.com/developers/powervr/graphics-sdk/`

**Mali OpenCL SDK** (v1.1; Windows, Linux)
`http://malideveloper.arm.com/resources/sdks/mali-opencl-sdk/`

**Adreno GPU SDK** (v5.0; Windows, OS X, Linux)
`https://developer.qualcomm.com/software/adreno-gpu-sdk`

**Vivante Vega GPGPU Technology**
`http://www.vivantecorp.com/en/technology/gpgpu.html`

## 15.4. Реализации *OpenCL* с открытым кодом

Наиболее известными в настоящее время являются две упомянутые выше реализации. Первая — ***Beignet*** (целевые устройства — *GPU* фирмы *Intel*, входящие в состав их центральных процессоров: *Intel HD*, *Intel Iris/Iris Pro*); разрабатывается она группой энтузиастов из фирмы **Intel China OTC**, начиная с 2013 года (проект создал ***Ben Segovia***). У проекта два сайта: `https://www.freedesktop.org/wiki/Software/Beignet/` и `01.org/beignet`. Вторая — ***POCL*** (*Portable Computing Language*); разработчики — группа из **Tampere University of Technology** (Финляндия). И у этого проекта два сайта — `portablecl.org` и `pocl.sourceforge.net` ; исходный код есть и на *GitHub* (`https://github.com/pocl/pocl`).

## 15.5. *Beignet* (текущая версия *1.2.1*)

Открытая реализация спецификации *OpenCL*, содержит код для исполнения программ *OpenCL* на графических процессорах фирмы *Intel*, входящих в состав процессоров. Включает в себя также соответствующий компилятор (имеется в каталоге `backend`). Последний доступный выпуск пакета — *Beignet 1.2.1*, размещён 4 ноября 2016 г. Утверждается, что последние версии частично поддерживают возможности *OpenCL* версии *2.0*: *SVM*, вызовы *API*, атомарные 64-разрядные инструкции и т.д.

## 15.6. Установка *Beignet* (на примере *Ubuntu 14.04* и *16.04*)

Будет весьма предусмотрительно сначала «обновиться» (`sudo apt-get update` и затем — `sudo apt-get upgrade`), после чего нужно установить необходимые пакеты — в том случае, если они не установлены: `cmake`, `pkg-config`, `python`, `ocl-icd-dev`, `ocl-icd-opencl-dev`, `libdrm-dev`, `libxfixes-dev`, `libxext-dev`, `llvm-3.5-dev`, `clang-3.5`, `libclang-3.5-dev`, `libtinfo-dev`, `libedit-dev`, `zlib1g-dev`. Если по какой-либо причине отсутствует компилятор *C++*, то понадобится ещё пакет `build-essential`. После этого архив с исходным кодом надо куда-то распаковать (`tar -xvf beignet-1.2.1-source.tar.gz -C <`*`Mecmo`*`>`) и, перейдя туда, создать каталог `build` (`mkdir build`) и из него (`cd build`) запустить `cmake` (`cmake ..`), а затем `make`. Окончательная установка пакета должна производиться с правами суперпользователя (`sudo make install`).

В результате в каталог `/etc/OpenCL/vendors` скопируется необходимый `.icd`-файл, в каталог `/usr/local/include` — заголовочные файлы *OpenCL*, а все библиотеки (`libcl.so`, `libgbe.so`, `libgbeinterp.so`) и вспомогательные файлы (`beignet.bc`, `beignet.pch`) попадут в каталог `/usr/local/lib/beignet`, где также возникает подкаталог `include` с заголовочными файлами, используемыми при компиляции ядер (`ocl*.h`). Полный список всего, что установлено, содержится в файле `install_mainfest.txt` каталога построения пакета (`build`).

## 15.7. Особенности пакета *Beignet*

Применяя утилиту `clinfo` от ***Oblomov***, можно узнать, что версии *Beignet 1.1.2* и *1.2* имеют так называемые встроенные ядра (т.е., готовые программы ядер) в количестве 28 штук. Эти ядра предназначены (как можно понять из их названий) для копирования или заполнения заданной области буфера/изображения. Используются они фирмой *Intel*, надо полагать, для того, чтобы защитить свою интеллектуальную собственность в тот переходный период, когда ещё не вполне возможно сокрытие исходного кода ядер.

Расширения устройства в *Beignet* — атомарные операции с целыми `int32`, возможность байтовой адресации памяти, запись в трёхмерные изображения, создание двумерных изображений из буфера, поддержка *SPIR* (стандарта для переносимого промежуточного представления кода), а также специфические расширения *Intel*: `cl_intel_accelerator`, `cl_intel_motion_estimation`, `cl_intel_subgroups`.

## 15.8. *POCL* (текущая версия *0.13*)

***POCL*** (***Portable Computing Language***) — открытая реализация *OpenCL* с лицензией *MIT* (т.е., код может использоваться в составе закрытых программ, ***proprietary software***). *POCL* легко адаптировать для новых целевых устройств: *CPU*, а также *GPU* и других акселераторов. Он использует *Clang* как *OpenCL C* фронтенд и *LLVM* — для реализации компилятора ядер, что гарантирует переносимость этой реализации *OpenCL* на другие архитектуры.

Дополнительной целью проекта провозглашена возможность применять её в качестве исследовательской платформы для решения проблем параллельного программирования на

разнородных платформах.

*POCL* обычно использует два разных компилятора (хотя может быть откомпилирован одним). Один используется для компиляции *C* и *C++* файлов, обычно это «системный» компилятор. Второй компилятор используется для работы с *OpenCL*-файлами и это всегда *Clang+LLVM*.

Так как в *POCL* используется компилятор *LLVM*, то выход очередной версии пакета привязывается к появлению нового выпуска этого компилятора, так, скажем, версия *POCL 0.10* (`pocl-0.10.tar.gz`) поддерживает *LLVM 3.5*, а последняя версия *0.13 — LLVM 3.8*, используемый, кстати, по умолчанию в *Ubuntu 16.04*. Все вышедшие версии *POCL* можно найти также по адресу `http://sourceforge.net/projects/pocl/files/`.

> Для *Ubuntu 16.04* стандартным является *Clang/LLVM 3.8* (т.е., установлены пакеты `clang-3.8` и `llvm-3.8`), к ним надо лишь добавить `libclang-3.8-dev`.

В составе версии для настольных компьютеров эта реализация *OpenCL* поддерживает только одно устройство (*CPU*), но зато это устройство может иметь архитектуру из весьма широкого спектра (**x86_64** (64-bit), **ARM v7**, **PowerPC64**, **PowerPC32**, **Cell SPU**).

В настоящее время *POCL* совместима с широким кругом приложений *OpenCL*, включая примеры библиотеки *ViennaCL*, примеры к книге *«OpenCL Programming Guide»*, примеры из *AMD APP SDK* и прочие.

## 15.9. Установка *POCL* (на примере *Ubuntu 14.04* и *16.04*)

Так как пакет распространяется в виде исходного кода, его установка начинается с построения всего пакета, для чего потребуются: *LLVM* и *Clang*, `gcc` или совместимый с ним компилятор, утилита `make`, а также пакеты: подгрузки библиотечных файлов (он может называться `libltdl-dev` или `libltdl3-dev`), `pthread` (вероятно, он уже имеется) и `hwloc` *v1.0* или новее (скорее всего, он будет называться `libhwloc-dev`). Полезно иметь установленными `ocl-icd` и `opencl`.

Если всё это имеется в наличии, то (после создания каталога `build` и перехода туда) можно попробовать запустить скрипт конфигурации (`../configure`) — как это было можно для версии *POCL 0.10* — или команду `cmake ..` (если есть `CMakeLists.txt`, как в последующих версиях *POCL*; при этом должен быть установлен *CMake*). Затем, как обычно, запускается `make` — для сборки исходных текстов в соответствии с конфигурационными установками, — после чего идёт установка (`sudo make install`).

## 15.10. Особенности пакета *POCL*

Запуск утилиты `clinfo` (стандартной или от ***Oblomov***) при установленной платформе *POCL* показывает, что единственным расширением платформы является `cl_khr_icd`, т.е. эта платформа может быть выбрана для работы через стандартный *ICD*-загрузчик. Реализуемая версия *OpenCL — 1.2*, а начиная с версии *POCL 0.13 — OpenCL 2.0*. Расширения устройства (*POCL 0.13*): возможность работы с вещественными значениями двойной точности (`cl_khr_fp64`, `cl_amd_fp64`), атомарные операции с целыми `int32` и `int64`, а также обычные для всех реализаций возможность байтовой адресации памяти и поддержка ***SPIR*** — стандарта для переносимого промежуточного представления кода.

## 15.11. Компиляция и установка пакетов из исходного кода

В отличие от инсталляционных пакетов закрытых реализаций *OpenCL*, пакеты открытых реализаций распространяются в виде исходного кода, а потому полезно знать, как следует осуществлять его компиляцию, и хотя бы примерно понимать, в каком виде это реализуется в настоящее время.

Раньше, когда программные пакеты были невелики, а компиляторов было не так много, достаточно было применить к файлам исходного кода так называемый `Makefile` — и компиляция пакета сводилась тогда лишь к запуску команды `make` в нужном каталоге.

Сейчас программные пакеты зачастую просто огромны, а потому распространяются как архивы, содержащие всю иерархию каталогов с программными файлами. Но помимо этого существует и большое число различных вариантов и версий систем, для которых может собираться программный пакет, вдобавок, он ещё зависит от других программ и путей их установки (это — так называемые зависимости).

Поэтому один из вариантов построения пакета (после его распаковки и перехода в его каталог) — это вызов скрипта конфигурирования (`./configure`), который как бы «изучает» систему, проверяет наличие всех зависимостей, установки дополнительных опций сборки, а на основе этого создаёт `Makefile`. Изначально пакет не содержит этого файла, вместо него имеется «заготовка» — файл `Makefile.in`, и `configure` создаёт специализированную версию `Makefile`, основываясь на сведениях о системе. (В свою очередь файл `Makefile.in` создаётся другим набором программ под названием `autotools`, куда входят `autoconf`, `automake` и др., но «углубляться» в эти тонкости пока не будем).

Скрипт оболочки `configure` пытается найти правильные значения разных переменных, меняющихся от системы к системе и используемых в процессе компиляции. Они нужны для создания `Makefile` в каждом каталоге пакета, а также заголовочных файлов, содержащих зависящие от системы определения. В конце работы скрипта создаётся — в числе других файлов — файл `config.log` с подробностями процесса. Если `configure` сообщил об отсутствии каких-то пакетов, их надо установить и снова перезапустить скрипт конфигурации. Если же команда `./configure` завершилась без ошибок, то и компиляция тоже должна пройти успешно. Таким образом, стандартный набор команд установки любого пакета (здесь он условно называется *`Package`*) выглядит примерно так:

```
tar xvzf Package.tar.gz
cd Package
./configure
make
sudo make install
```

Другой вариант — использование утилиты для автоматической сборки программы из исходного кода под названием ***CMake***; эта утилита является кроссплатформенной, поэтому её можно встретить и при сборке пакетов под *Windows*, и при сборке их в системах *Linux*. Она пользуется в своей работе файлами с именами `CMakeLists.txt`, создавая в результате проект нужного нам типа, скажем, тот же `Makefile`, поэтому, если в корневом каталоге пакета имеется файл `CMakeLists.txt`, мы можем запустить генерацию всего проекта, например, из созданного для построения пакета каталога `build`, — одной командой `cmake ..` (или `cmake <ОпцииСборки> ..`).

Сборка пакета и его установка проводятся аналогично (`make && sudo make install`).

## 15.12. Последствия наличия множественных реализаций

Если на компьютере имеется несколько реализаций *OpenCL*, то самой спецификацией предусмотрена возможность выбора исполняемой программой какой-то одной из них, что осуществляется (как уже ранее говорилось) с помощью промежуточной библиотеки *ICD Loader*. Это, конечно, положительный момент для программы, работающей в столь «насыщенной» среде. Однако, при компиляции самой программы в подобной ситуации возможен и некоторый отрицательный момент: каждая из реализаций могла установить свои варианты (или просто другие версии) заголовочных файлов (хотя, по идее, они все должны совпадать с образцами на сайте консорциума *Khronos*), из-за чего надо внимательно подходить к указанию местоположения заголовочных файлов и аккуратно задавать параметры проектов или опции для командных строк компиляции.

Для поиска труднонаходимых ошибок, вызванных использованием различных версий заголовочных файлов из непредсказуемых мест, полезно, например, при компиляции с помощью `g++`/`gcc` использовать ключ командной строки `-v`, позволяющий узнать, кроме всего прочего, порядок и места поиска заголовочных файлов. Аналогично, удобно полагаться на программу `pkg-config` и её файлы `.pc`, когда необходимо указывать эти параметры для сборки конкретной программы, скажем, вот так:

```
$ gcc -o <Программа> <Программа>.c $(pkg-config --cflags --libs libpng)
```

вместо явного указания каталогов заголовочных файлов и библиотек. Здесь `libpng` — имя конфигурационного файла, необходимого для собираемой программы (полное имя этого файла — `libpng.pc`).

Кроме того, если иметь в виду исполнение программы в рамках одной из имеющихся реализаций *OpenCL*, разумно было бы написать её так, чтобы конкретную реализацию можно было выбрать непосредственно при исполнении программы. Для этого надо или опрашивать пользователя программы во время её исполнения, анализируя далее его выбор, или иметь возможность указать исполняющейся программе желательные платформу и устройство каким-то «внешним» способом, например, через переменную окружения.

Кстати, последний способ применяется в библиотеке ***PyOpenCL***, использующей имеющиеся реализации *OpenCL* и написанной на языке *Python*: можно либо установить должным образом специальную переменную окружения `PYOPENCL_CTX` для программы, либо указать её прямо во время запуска программы с командной строки.

```
PYOPENCL_CTX=´1:0´ python <Программа>.py.
```

Первый из указанных способов («правильное» написание программы) демонстрируется примерами с сайта `http://raytracey.blogspot.com`, предлагаемыми для ознакомления (и самостоятельного корректирования).

## 15.13. Практические примеры-задания: Ray Tracing

Примеры, приводимые в статьях ***«OpenCL path tracing tutorial»*** (см. ссылки ниже), удобны тем, что они весьма близки к работоспособному состоянию в «устаревающем» аудиторном окружении 5-42, написаны с применением *C++* и формируют (простейшими средствами) красивые графические результаты. Для показа получаемых изображений может понадобиться программа ***IrfanView*** или аналогичная, «понимающая» файлы `.ppm`.

Работа программ построена вполне разумно: они сначала запрашивают пользователя, какую платформу и какое устройство *OpenCL* надо использовать. Правда, нумерация при этом ведётся с единицы, что может немного дезориентировать, да и в тексте выглядит противоестественно, ибо из номеров приходится всегда вычитать единицу.

При компиляции этих программ — если их не подправлять — потребуется указывать опцию «продвинутого» *C++* (`-std=c++11`, `-std=gnu++11` или аналогичную им), но если возможности *C++* более скромные (как у *Visual Studio 2008*), проще внести в текст небольшие изменения. Исправления, которые могут понадобиться в программе, включают приведение типа строки `string` к *C*-строке в функции `printErrorLog()`, а также получение вектора устройств из одного-единственного устройства при вызове метода `.build()` объекта `program`. Возможно, что придётся изменить и возвращаемое значение у функции `main()`...

Стоит также обратить внимание на ещё два момента в этих программах. Первый (положительный) — это способ чтения файла с текстом ядра: он «устойчив» к текстовому режиму открытия файла, поскольку чтение производится построчно, а длина файла не определяется. Второй (отрицательный) состоит в том, что практически никакие (кроме построения ядра) ошибочные ситуации в программах не обрабатываются, из-за чего даже довелось наблюдать BSOD в дисплейном драйвере `nv4_disp.dll` версии *335.28*.

## 15.14. Некоторые ссылки по теме занятия

`http://support.amd.com/en-us/kb-articles/Pages/OpenCL2-Driver.aspx`
`http://www2.ati.com/drivers/amd-14.41rc1-win8.1-64bit-opencl2-sep19.exe`
`http://www2.ati.com/drivers/linux-amd-14.41rc1-opencl2-sep19.zip`
`http://developer.amd.com/tools-and-sdks/opencl-zone/`
Информация о драйвере *OpenCL 2.0* от ***AMD***; ссылки на некоторые недавние версии; раздел для разработчиков *OpenCL*, использующих *GPU* и *CPU* фирмы ***AMD***: инструментарий, библиотеки и другие ресурсы.

`https://software.intel.com/en-us/articles/opencl-drivers`
`https://software.intel.com/en-us/intel-opencl`
Разнообразные *GPU/CPU OpenCL*-драйверы от ***Intel***, а также *SDK* для *OpenCL*-приложений.

`http://www.nvidia.com/drivers`
`https://developer.nvidia.com/opencl`
Первая ссылка — на страницу загрузки драйверов для видеокарт ***NVidia*** (поддержка *OpenCL* включена в драйверы). Страница по второй ссылке почти полностью посвящена примерам кода, включённым в ***NVIDIA OpenCL SDK***.

`https://01.org/beignet`
`https://www.freedesktop.org/wiki/Software/Beignet/`
Открытая реализация *OpenCL*, именуемая ***Beignet***, целевыми устройствами для которой являются *GPU*, интегрированные в процессоры *Intel Core* младших поколений, — поскольку на системах *Linux* они оказались не поддержанными реализацией *OpenCL* фирмы ***Intel***.

`http://pocl.sourceforge.net`
`http://portablecl.org`
`https://github.com/pocl/pocl`
`http://sourceforge.net/projects/pocl/files/`
Открытая реализация *OpenCL*, называемая ***POCL*** (**Portable Computing Language**), целевыми устройствами для которой являются *CPU* различных архитектур (**x86_64, ARM v7, PowerPC** и другие): информация о проекте, текущая версия исходного кода, предыдущие версии пакета.

`https://streamcomputing.eu/blog/2013-07-08/installing-and-using-pocl/`
Статья ***«Installing and using Portable Computing Language (PoCL)»*** об установке реализации *OpenCL* с открытым кодом (***POCL***) на *Ubuntu 64*.

`http://raytracey.blogspot.com/2016/11/opencl-path-tracing-tutorial-1-firing.html`
Вводная статья блога об использовании *OpenCL* для отрисовки реальных сцен (т.н. ***ray tracing***): ***«OpenCL path tracing tutorial 1: Firing up OpenCL»*** (автор — ***Sam Lapere***).

`http://raytracey.blogspot.com/2016/11/opencl-path-tracing-tutorial-2-path.html`
Другая статья (***«OpenCL path tracing tutorial 2: path tracing spheres»***) из этого же блога.

## 15.15. Задание для самостоятельной работы

Используя высказанные выше замечания (или игнорируя их), попробуйте по своему вкусу подкорректировать программы из первой и второй частей сопроводительных материалов к статье ***«OpenCL path tracing tutorial 2: path tracing spheres»*** и добиться их работоспособности на имеющихся в распоряжении реализациях *OpenCL* (в том числе — и в аудитории 5-42). Задокументируйте необходимые изменения для каждой использованной реализации *OpenCL* и каждого применённого компилятора.

# 16. *OpenCL 2.0* — новые возможности

Выпуск версии спецификации *OpenCL 2.0* был в определённом смысле знаковым, т.к. в ней члены группы *Khronos* попытались воплотить то, что считали наиболее важным для *OpenCL* на ближайшие годы: поддержку разделяемой виртуальной памяти (***Shared Virtual Memory***), динамического параллелизма (***Dynamic Parallelism***), обобщённого адресного пространства (***Generic Address Space***); улучшенную поддержку изображений (*Images*), включая изображения ***sRGB*** и запись в трёхмерные изображения; реализацию некоторого подмножества атомарных операций и операций синхронизации из стандарта языка *C11* (***C11 Atomics***); введение новых объектов памяти (***Pipes***), организованных как *FIFO*-очереди — для обмена информацией между ядрами; обнаружение и подгрузку реализаций *OpenCL* в рамках *Android* (***Android Installable Client Driver Extension***). Здесь мы остановимся чуть более подробно только на одном из этих нововведений, наиболее радикально влияющем на скорость работы определённого сорта программ (тех, которые могут быть сформулированы рекурсивно); нормальное изложение всего упомянутого выше потребовало бы, вероятно, ещё одного пособия...

## 16.1. Динамический параллелизм

Динамический параллелизм (***dynamic parallelism***; иногда встречается термин ***nested parallelism***) заключается в том, что ядра на устройстве могут инициировать запуск новых ядер на этом же устройстве, причём безо всякого обращения к хосту (т.е., затратные переносы данных между памятью устройства и памятью хоста уже не нужны).

## 16.2. Пример программы — «Ковёр Серпинского»

Данный пример присутствует в материалах фирмы *Intel*, иллюстрирующих возможности *OpenCL 2.0* (в файле, сопровождающем статью ***«Sierpiński Carpet in OpenCL 2.0»*** по адресу `https://software.intel.com/en-us/articles/sierpinski-carpet-in-opencl-20`).

В оригинале исходные файлы примера располагаются в каталоге `Sierpinski`, который содержит главный файл `sierpinski.cpp` и подкаталог `clu` с двумя вспомогательными файлами — `clu.h` и `clu_runtime.cpp` (ещё один файл `CMakeLists.txt` не принимаем во внимание, т.к. он нам пока не понадобится).

Некоторые особенности программы достойны отдельного упоминания. Ядро в программе содержит синтаксическую новинку, пришедшую в *OpenCL C 2.0* из компилятора *Clang*: так называемый *блок* (***block***). По сути это новый тип данных — исполняемый кодовый блок, объявляемый с помощью символа `"^"`. С помощью этого же символа помечается также и определение «функции», соответствующей этому кодовому блоку. Так, в упомянутой выше статье приводится небольшой пример, поясняющий это понятие; мы его упростим:

```
int (^myBlock)(int) = ^(int num) { return num * num; }
```

Здесь объявлена переменная `myBlock`, тип которой — блок, это указывается символом `"^"` в объявлении. Этот блок возвращает в результате исполнения целое число (`int`), в качестве единственного параметра он тоже принимает `int`. Объявленная переменная тут же проинициализирована указанным здесь определением блока (второй символ `"^"`). В определении блока мы видим конкретное значение передаваемого ему параметра (это целое число `num`) и далее в фигурных скобках показано, что с ним происходит: оно возвращается возведённым в квадрат. Такое определение блока выглядит очень похоже на определение функции, только у неё нет никакого имени. А так как этой «функцией» инициализировано значение переменной, понятно, что вызов «функции» осуществляется при помощи этой переменной — вероятно, примерно так, как указано в объявлении.

Нечто похожее, только немного более простое, можно увидеть в определении функции ядра `sierpinski()` из рассматриваемого примера (несущественные детали ядра далее для простоты опущены):

```
__kernel void sierpinski(__global char* src, int width, int ox, int oy)
{
  . . .
  queue_t q = get_default_queue();
  . . .
  const size_t grid[2] = { ..., ...};
  enqueue_kernel(q,0,ndrange_2D(grid),^{sierpinski(src,width,x+ox,y+oy);});
  . . .
}
```

Ядро начинается вполне традиционно, оно имеет название `sierpinski` и 4 параметра. В его определении можно заметить только несколько не совсем обычных строк, именно они оставлены для пояснения. Первая — это создание очереди на основе некой «очереди по умолчанию». Вторая — запуск ядра на исполнение из него же как блока — с использованием созданной очереди на некоторой сформированной «сетке». Для запуска используется функция `enqueue_kernel()`, доступная только в ядрах, начиная с версии *2.0*, а её последним аргументом является блок с вызовом этого же ядра, но с несколько другими значениями параметров из текущей области действия.

> **Внимание!** Ядра, в которых используется версия *OpenCL C 2.0*, должны компилироваться с параметром командной строки `-cl-std=CL2.0` (см., например, файл `sierpinski.cpp`). Если же при компиляции программы ядра опция `-cl-std` не указана, то будет использоваться максимально возможная версия *OpenCL C 1.x*.

Поскольку для компиляции примера понадобятся заголовочные файлы *OpenCL* версии *2.0* (конкретно, два файла: `cl.h` и `cl_platform.h`), имеет смысл скопировать их локально в каталог `CL`, созданный там же, где и находится каталог `Sierpinski`. Понадобится изменить строку инициализации параметров в самом начале функции `main()` (там предполагается исполнение кода на реализации *OpenCL* от *Intel*, поэтому нужно использовать вместо этой строки что-то от применяемой конкретной реализации), а также, возможно (мне на *Ubuntu 16.04* x86_64, по крайней мере, пришлось), добавить определение типа `errno_t` (`typedef int errno_t;`) и подключить файл `string.h` — для прототипа функции `memset()`. После этого пример откомпилируется с помощью `g++`, но может и не заработать (у меня точно не заработал, поскольку в реализации от *AMD* указанный в программе размер очереди — $16 * 1024 * 1024$ — не мог быть выделен).

Командная строка для компилятора `g++` (при указанном выше расположении файлов):

```
g++ -o SC -std=c++0x clu/clu_runtime.cpp sierpinski.cpp -I.. -I clu -lOpenCL
```

Здесь `SC` — это имя исполняемого файла (сокращение от *Sierpinski Carpet*), файлы исходного кода `clu/clu_runtime.cpp` и `sierpinski.cpp` компилируются с «продвинутой» опцией (как минимум `-std=c++0x`, можно также `-std=c++11`), заголовочные файлы для компиляции берутся из расположенного «снаружи» каталога `CL` и вложенного каталога `clu`, поиск библиотеки *OpenCL* производится в обычных местах, ибо пути не указаны.

> Внимательно просматривая сформированный графический файл `sierpinski.ppm`, удалось заметить, что часть его точек, которые должны были быть чёрными, оказались не заполненными. Это, вне всякого сомнения, связано с размером используемой очереди: при увеличении этого размера число незаполненных точек уменьшается, правда, далеко не пропорционально указанному размеру. Отсюда можно сделать вывод (малоутешительный!), что при подобном запуске ядер у пользователя уже нет возможности контролировать, в каком количестве теперь будут запущены ядра — всё определяется размером очереди...

## 16.3. Некоторые ссылки по теме занятия

`https://www.khronos.org/news/press/khronos-releases-opencl-2.0`
Пресс-релиз о выпуске спецификации *OpenCL 2.0* с перечислением дополнений и обновлений.

`https://streamcomputing.eu/blog/2015-02-10/overview-of-opencl-2-0-support-samples...`
Статья ***«Overview of OpenCL 2.0 hardware support, samples, blogs and drivers»*** аналитика сайта `streamhpc.com` с обзором поддержки *OpenCL 2.0* изделиями фирм ***AMD***, ***Intel***, ***NVidia***.

`https://software.intel.com/ru-ru/articles/sierpinski-carpet-in-opencl-20`
Статья ***«Sierpiński Carpet in OpenCL* 2.0»*** с примером исходного кода, иллюстрирующая динамический параллелизм в *OpenCL 2.0* (*dynamic parallelism* или *nested parallelism*).

`https://software.intel.com/ru-ru/articles/gpu-quicksort-in-opencl-20-using...`
Статья-руководство ***«GPU-Quicksort в OpenCL 2.0: вложенные параллельные вычисления и сканирование групп обработки»***, показывающая применение функции `enqueue_kernel`, а также наборов функций `work_group_scan_exclusive_add` и `work_group_scan_inclusive_add`.

`https://software.intel.com/ru-ru/articles/using-opencl-20-read-write-images`
`https://software.intel.com/sites/default/files/managed/db/33/Read-Write_Images.pdf`
`https://software.intel.com/sites/default/files/managed/12/d4/ReadWriteImageSDK.zip`
Материалы по использованию изображений для чтения и записи в *OpenCL 2.0*: статья и код.

`https://software.intel.com/sites/default/files/managed/9d/6d/TutorialSVMBasic.pdf`
`https://software.intel.com/en-us/articles/opencl-20-shared-virtual-memory-overview`
Статья ***«OpenCL™ 2.0 Shared Virtual Memory Overview»*** посвящена новой особенности *OpenCL 2.0*: так называемой разделяемой виртуальной памяти (*Shared Virtual Memory*, *SVM*), позволяющей иметь разделяемые между хостом и устройством структуры данных с большим числом указателей (связные списки, деревья и т.п.).

`https://software.intel.com/en-us/articles/opencl-20-non-uniform-work-groups`
Статья ***«OpenCL™ 2.0 Non-Uniform Work-Groups»*** демонстрирует, что теперь всё рабочее пространство (*NDRange*) необязательно должно состоять из рабочих групп одинакового размера.

`https://software.intel.com/en-us/articles/using-opencl-20-work-group-functions`
Статья ***«Using OpenCL™ 2.0 Work-group Functions»*** вводит новые встроенные функции, работающие со скалярными типами данных на уровне рабочей группы.

`https://software.intel.com/en-us/articles/the-generic-address-space-in-opencl-20`
Статья ***«The Generic Address Space in OpenCL™ 2.0»*** даёт примеры работы с обобщённым адресным пространством. Теперь программисты могут ограничиваться одним вариантом кода, работающим в разных именованных адресных пространствах (`local`, `global`, `private`).

`https://software.intel.com/en-us/articles/using-opencl-20-srgb-image-format`
Статья ***«Using OpenCL™ 2.0 sRGB Image Format»*** — о работе с *sRGB*-изображениями. *sRGB* — это цветовое пространство, появившееся во времена распространения электронно-лучевых мониторов, поэтому изображения *sRGB* могут быть отображены на таких мониторах без какой-либо обработки (гамма-коррекции).

`https://software.intel.com/en-us/articles/using-opencl-20-atomics`
Статья ***«Using OpenCL™ 2.0 Atomics»*** обсуждает тонкости атомарных операций в *OpenCL 2.0*: теперь нужно использовать специальные типы данных и отказаться от привычных операций, так как они заменены соответствующими встроенными функциями.

# 17. *OpenCL* «под капотом» или «обёртки» *OpenCL*

Надо признать, что прямое использование вызовов функций *OpenCL API* — достаточно трудоёмкий способ написания программ: число необходимых для работы фунций весьма велико (и продолжает увеличиваться от версии к версии), дополнительно надо также использовать функции управления компиляцией/построением программ из текста ядер. А если учесть, что вызов каждой функции *OpenCL* следует сопровождать обязательной обработкой возможных ошибок, то станет понятно, что в таких условиях о программе удобочитаемой и легко воспринимаемой можно забыть — если не использовать что-то уже готовое и отлаженное (или гораздо более выразительное программное средство).

Далее обсуждается не просто оформление кода в виде объектов языка *C++* — как это было сделано в заголовочных файлах консорциума `cl.hpp`, `cl2.hpp`, — мы знакомимся с библиотеками, использующими *OpenCL* как основу взаимодействия с параллельными устройствами, причём нейтральную по отношению к производителям.

## 17.1. Библиотека *Boost.Compute*

Автор этой библиотеки ***Kyle Lutz*** позиционирует её как вариант библиотеки шаблонов для параллельных устройств (***«STL for Parallel Devices»***): *GPU*, многоядерных *CPU* и акселераторов. Не очень давно (начиная с версии *Boost 1.61*) ***Boost.Compute*** стала, наконец, частью библиотеки ***Boost*** (`https://github.com/boostorg/compute`).

*Boost.Compute* является чисто заголовочной библиотекой, поэтому не нужно никакого поиска/включения дополнительных статических библиотек на этапе сборки программы; все возможности библиотеки могут быть включены в программу директивой:

```
#include <boost/compute.hpp>
```

или можно ограничиться основным заголовком «обёртки» (минимизируя зависимость от остальных возможностей библиотеки *Boost*):

```
#include <boost/compute/core.hpp>
```

Все классы и функции *Boost.Compute* определены в рамках своего пространства имён `boost::compute`, а потому для извлечения их в глобальную область видимости можно добавить директиву (небезопасно, т.к. имена здесь и в *STL* частично совпадают!):

```
using namespace boost::compute;
```

Существуют некоторые дополнительные конфигурационные макросы библиотеки, скажем, если определён `BOOST_COMPUTE_DEBUG_KERNEL_COMPILATION`, тогда исходный код ядра и лог его построения попадают в стандартный вывод. `BOOST_COMPUTE_USE_OFFLINE_CACHE` разрешает кэширование откомпилированных ядер на диске (программа при этом должна линковаться с библиотеками *Boost* — `Boost.Filesystem` и `Boost.System`).

Типичный пример программы (сортировка вектора вещественных величин `float`) дан в сопровождающем пояснительном тексте на сайте (чуть изменён, комментарии убраны):

```
#include <vector>
#include <algorithm>
#include <boost/compute.hpp>

namespace compute = boost::compute;

int main()
{
    compute::device gpu = compute::system::default_device();
```

```
    compute::context ctx(gpu);
    compute::command_queue queue(ctx, gpu);

    std::vector<float> hV(1000000);
    std::generate(hV.begin(), hV.end(), rand);
    compute::vector<float> dV(1000000, ctx);

    compute::copy(hV.begin(), hV.end(), dV.begin(), queue);
    compute::sort(dV.begin(), dV.end(), queue);
    compute::copy(dV.begin(), dV.end(), hV.begin(), queue);

    return 0;
}
```

Здесь выбирается устройство по умолчанию, для него создаются контекст и очередь, на хосте генерируется вектор случайных величин, а на устройстве создаётся вектор такого же размера — для сортировки этих значений. Затем содержимое хост-вектора копируется на устройство, там сортируется и результат возвращается на хост.

Вообще же библиотека обеспечивает подобно *STL* общие алгоритмы (типа `transform()`, `accumulate()`, `sort()`) и контейнеры (`vector<T>`, `flat_set<T>`). Имеются в ней также и расширения: параллельные вычислительные алгоритмы (такие, как `exclusive-scan()`, `scatter()`, `reduce()`) и некоторые «воображаемые» итераторы (`transform_iterator<>`, `permutation_iterator<>`, `zip_iterator<>`).

## 17.2. Библиотека *ViennaCL*

***ViennaCL*** (`http://viennacl.sourceforge.net`) — это бесплатная открытая библиотека линейной алгебры для вычислений на многоядерных архитектурах (имеются в виду *GPU* или *CPU* с несколькими ядрами), написанная на *C++* и поддерживаюшая *CUDA*, *OpenCL* и *OpenMP* (в качестве т.н. *backend*) и допускающая их переключение при исполнении.

В библиотеку входят функции первого, второго и третьего уровня библиотеки *BLAS*, быстрые умножения разреженных матриц на матрицы и разреженных матриц на векторы (поддержка разреженных матриц пока является ограниченной), итерационные методы решения систем линейных уравнений, алгоритмы *БПФ*. Имеется также *Python*-интерфейс к библиотеке *ViennaCL* — «обёртка», именуемая ***PyViennaCL***.

Поскольку *ViennaCL* является чисто заголовочной библиотекой, то для её установки достаточно скопировать каталог `viennacl/` либо в каталог проекта, либо в то место, которое указано как путь поиска заголовочных файлов (в *Unix*-системах — каталоги `/usr/include/` или `/usr/local/include/`). Предполагается, что *OpenCL* заголовочные файлы ранее уже были установлены; если это не так, то тогда надо скопировать их в подкаталоге `CL/` в то же самое место. В случае системы *Windows* и работы в среде разработки пути поиска заголовочных файлов добавляются в свойства проекта.

По умолчанию библиотека *ViennaCL* (начиная с версии *1.4.0*) использует *CPU backend* и поэтому для использования *OpenCL* необходимо определить константу препроцессора `VIENNACL_WITH_OPENCL` в начале исходного файла либо указать её в командной строке при компиляции (`-DVIENNACL_WITH_OPENCL`). Разумеется, должны быть доступны также библиотека (или *ICD*-загрузчик) и драйверы реализации *OpenCL*.

## 17.3. Библиотека *VexCL*

Это шаблонная библиотека векторных выражений для использования с *OpenCL*/*CUDA*; её автор — наш соотечественник ***Денис Демидов*** (`https://github.com/ddemidov/vexcl`). ***VexCL*** позволяет использовать обычную интуитивную нотацию для векторных операций,

операции редукции, произведений разреженных матриц на векторы и т.п. Допускает определяемые пользователем ядра, а также взаимодействие в другими библиотеками. В *VexCL* даже имеется шаблонный класс `vex::svm_vector<T>` для реализации введённой в *OpenCL 2.0* разделяемой виртуальной памяти (*Shared Virtual Memory*, *SVM*).

> Следует иметь в виду, что библиотека *VexCL* использует особенности *C++11*, поэтому компиляция при её применении возможна не любым компилятором!

Для использования библиотеки подключается один файл-заголовок — `vexcl/vexcl.hpp`. Выбрать *OpenCL* в качестве *backend* можно, определяя величину `VEXCL_BACKEND_OPENCL` с помощью препроцессора. Но какой бы *backend* ни был определён, на этапе линковки понадобятся (кроме, разумеется `libOpenCL.so` или `OpenCL.dll`) ещё и `Boost.System` и `Boost.Filesystem`, а также, возможно, `Boost.Thread` и `Boost.Date_Time`.

> Если необходимо вывести исходный код сформированного библиотекой *OpenCL*-ядра, то перед запуском программы, использующей библиотеку *VexCL*, можно установить переменную окружения `VEXCL_SHOW_KERNELS=1`.

## 17.4. Библиотека *ArrayFire*

***ArrayFire*** — вычислительная библиотека общего назначения, нацеленная на широкий круг параллельных архитектур (многоядерные *CPU*, *GPU*, акселераторы) и включает в себя сотни функций: векторные алгоритмы, функции обработки изображений/сигналов, компьютерное зрение, функции визуализации и др. В настоящее время поддерживаются две «обёртки» (для языков *Python* и *Rust*), а в перспективе ожидаются ещё семь...

Загрузка дистрибутива этой библиотеки с сайта (`https://arrayfire.com/download`) требует регистрации на нём. Можно и самостоятельно откомпилировать библиотеку, пользуясь её открытым кодом на *GitHub* (`https://github.com/arrayfire/arrayfire`). В составе откомпилированной библиотеки имеются модули для работы с *CUDA*, *OpenCL* и *CPU* (поддерживаются процессоры x86 и ARM), называются они, соответственно, `libafcuda`, `libafopencl` и `libafcpu`. Если устройства *CUDA* или *OpenCL* недоступны, библиотека будет задействовать центральный процессор.

Для пользования библиотекой в программе надо просто включить в неё заголовочный файл `arrayfire.h` (это унифицированный для *C/C++* заголовок), после чего уже можно вызывать *C*-функции или создавать *C++*-объекты (определены в рамках пространства имён `af`, а потому требуют применения префикса `af::` или извлечения в глобальную область видимости).

В качестве простого примера в документации к библиотеке приводится фрагмент кода для оценки величины числа $\pi$ методом Монте-Карло; он использует располагающиеся на устройствах *OpenCL* (или *CUDA*) массивы `array` (содержат 40 миллионов чисел):

```
array x = randu(20e6), y = randu(20e6);
array dist = sqrt(x * x + y * y);

float num_inside = sum<float>(dist < 1);
float pi = 4.0 * num_inside / 20e6;
af_print(pi);
```

Объект `array` в библиотеке ***ArrayFire*** — типичный контейнер и может содержать один из легко опознаваемых типов: `b8`, `f32`, `c32`, `s32`, `u32`, `f64`, `c64`, `s64`, `s16` или `u16`. Если тип хранимых величин явно не указан, контейнер создаётся для вещественных величин одинарной точности (`f32`).

## 17.5. «Привязки» к *OpenCL* из других языков

Всё то, что обсуждалось выше, можно рассматривать как привязку к *OpenCL* из языка *C++*, причём дополняющую усилия консорциума (речь идёт о `cl.hpp`, `cl2.hpp` и *SYCL*).

Привязки к другим языкам можно обнаружить в объёмном списке, приводимом в статье по адресу `https://streamcomputing.eu/knowledge/for-developers/opencl-wrappers/`.

> Разница между «обёртками» и «привязками» достаточно условна, пока не даются чёткие определения того и другого. Мы здесь говорим об «обёртке», если можно не думать (хотя бы в большинстве случаев) о написании кода ядер, т.к. они формируются самой «обёрткой» (как в случае *Boost.Compute*, *ViennaCL*, *VexCL* и *ArrayFire*), — хотя и могут в принципе быть заданы пользователем библиотеки. В случае же «привязки» текст ядра обязательно указывается пользователем.

Мы остановимся здесь всего на двух примерах такой привязки: к языку *Python* (т.к. *Python* уже практически достиг уровня популярности *C/C++*), а также к сравнительно новому языку *Julia* (поскольку в нём видится потенциал языка для исследователей).

## 17.6. *PyOpenCL* — *Python*-«привязка» к *OpenCL*

Пример тестовой программы со страницы `https://documen.tician.de/pyopencl/` — был немного модифицирован в части поименования переменных, чтобы сблизить варианты.

```
import numpy as np
import pyopencl as cl

a = np.random.rand(50000).astype(np.float32)
b = np.random.rand(50000).astype(np.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)

prg = cl.Program(ctx, """
  __kernel void sum(__global const float *A,
                    __global const float *B,
                    __global float *R)
  {
    int gid = get_global_id(0);
    R[gid] = A[gid] + B[gid];
  }
""").build()

r_g = cl.Buffer(ctx, mf.WRITE_ONLY, a.nbytes)
prg.sum(queue, a.shape, None, a_g, b_g, r_g)

result = np.empty_like(a)
cl.enqueue_copy(queue, result, r_g)

# Check on CPU with Numpy:
print(result - (a + b))
print(np.linalg.norm(result - (a + b)))
```

## 17.7. `OpenCL.jl` — *Julia*-«привязка» к *OpenCL*

Пример тестовой программы со страницы `https://github.com/JuliaGPU/OpenCL.jl` — немного модифицирован в части поименования переменных, чтобы сблизить варианты.

```
using OpenCL

const cl = OpenCL

const sum_kernel = "
   __kernel void sum(__global const float *A,
                     __global const float *B,
                     __global float *C)
   {
      int gid = get_global_id(0);
      C[gid] = A[gid] + B[gid];
   }
"
a = rand(Float32, 50_000)
b = rand(Float32, 50_000)

device, ctx, queue = cl.create_compute_context()

a_g = cl.Buffer(Float32, ctx, (:r, :copy), hostbuf=a)
b_g = cl.Buffer(Float32, ctx, (:r, :copy), hostbuf=b)
c_g = cl.Buffer(Float32, ctx, :w, length(a))

prg = cl.Program(ctx, source=sum_kernel) |> cl.build!
k = cl.Kernel(prg, "sum")

queue(k, size(a), nothing, a_g, b_g, c_g)

result = cl.read(queue, c_g)

if isapprox(norm(result - (a+b)), zero(Float32))
    info("Success!")
else
    error("Norm should be 0.0f")
end
```

## 17.8. Некоторые ссылки по теме

`https://streamcomputing.eu/knowledge/for-developers/opencl-wrappers/`
Озаглавленный ***«OpenCL Wrappers»*** краткий обзор/список библиотек, использующих *OpenCL*, а также «привязок» к *OpenCL* из различных языков программирования (*Delphi*/*Pascal*, *Fortran*, *Go*, *Haskell*, *Java*, *JavaScript*, *Julia*, *Lisp*, *.NET*, *Perl*, *Python*, *Ruby*, *Rust*).

`http://www.boost.org/doc/libs/release/libs/compute/doc/html/`
Документация к библиотеке ***Boost.Compute*** (автор библиотеки — ***Kyle Lutz***).

`http://viennacl.sourceforge.net`
Открытая библиотека линейной алгебры ***ViennaCL***: документация, примеры, исходный код, сравнительные тесты, публикации. Помимо *OpenCL* она может использовать также *CUDA* и *OpenMP*.

`http://www.boost.org/doc/libs/.../tutorial/using_opencl_via_vexcl.html`
`http://nut-code-monkey.blogspot.ru/2016/04/opencl-cuda-gpu-vexcl.html`
Раздел документации библиотеки *Boost* (***«Using OpenCL via VexCL»***) и небольшая заметка по использованию библиотеки ***VexCL*** на русском языке (автор библиотеки — ***Денис Демидов***).

# 18. Открытые репозитории с *OpenCL*-кодом

Помимо сайта `https://www.khronos.org/opencl/` консорциума *Khronos*, где размещены все версии спецификации *OpenCL* и другие интересные документы, полезно знать, что имеется раздел на ***GitHub.com***, где можно найти и открытый исходный код, созданный консорциумом (`https://github.com/KhronosGroup`). Там довольно много проектов, но впрямую к обсуждаемым в данном курсе вопросам относятся, во-первых, заголовочные файлы *OpenCL* (`https://github.com/KhronosGroup/OpenCL-Headers`), во-вторых, версии *C++*-заголовочных файлов (`https://github.com/KhronosGroup/OpenCL-CLHPP`), в-третьих, предлагаемая консорциумом «каноническая» реализация т.н. загрузчика инсталлируемых драйверов, *ICD Loader* (`https://github.com/KhronosGroup/OpenCL-ICD-Loader`). Кроме того, стоит обратить внимание на всё, что связано со ***SPIR*** и ***SPIR-V*** — предлагаемым бинарным форматом хранения откомпилированных ядер. Эта инициатива позволяет сопровождать программу *OpenCL* не исходным кодом ядер (что представляется не слишком удобным для защиты интеллектуальной собственности), а их «нечитабельным» вариантом, который заодно быстрее может быть трансформирован в исполняемый вид для конкретного устройства. При этом надо понимать, что переносимость программ *OpenCL* не ухудшается, потому как бинарный формат по-прежнему является нейтральным по отношению к реализациям *OpenCL*, т.е., не ориентирован на какого-то конкретного изготовителя устройств.

> В конце лета 2015 года фирма *AMD* открыла исходный код своих *OpenCL* библиотек (***AMD Compute Libraries***, ***ACL***) для всеобщего доступа, разместив его на *GitHub* (местонахождение библиотек фирмы *AMD* — `https://github.com/clMathLibraries`). Там содержатся: ***clBLAS*** — функции линейной алгебры, ***clFFT*** — *БПФ*, ***clSPARSE*** — операции с разреженными матрицами, ***clRNG*** — генерация случайных чисел. Легко заметить, что названия этих библиотек почти совпадают с названиями библиотек конкурента (имеются в виду *cuBLAS*, *cuFTT*, *cuSPARSE* и *cuRAND* от *NVidia*).

Вообще же можно констатировать, что уже сформировались две встречные тенденции: часть изготовителей/разработчиков постепенно открывает свой исходный код *OpenCL* (как, скажем, *AMD*, *ArrayFire*), что можно, конечно, трактовать и как ослабление их позиций, а часть — говоря мягко — не торопится реализовывать предложения, в разработке которых они принимали самое непосредственное участие, рассчитывая, видимо, воспользоваться какими-то конкурентными преимуществами (как *NVidia*).

Прошло порядка четырёх лет после выпуска спецификации *OpenCL* версии *1.2* — и вот фирма *NVidia* (без особого шума) выпустила, наконец, драйверы для её поддержки. И это выглядит немного странно, поскольку вице-президент фирмы возглавляет рабочую группу по разработке *OpenCL* (*OpenCL Working Group*) в консорциуме *Khronos*.

Параллельно этому идёт постепенный процесс «избавления» *OpenCL* от исходного кода ядер: в спецификации *OpenCL 2.1* уже поддерживается *SPIR-V*.

Чем закончится такое «противоборство» и насколько оно продуктивно, покажет время.

## 18.1. Некоторые ссылки по теме

`https://www.khronos.org/opencl/`
Раздел сайта консорциума *Khronos*, посвящённый *OpenCL*.

`https://github.com/KhronosGroup`
Проекты *Khronos* на `github.com`: **OpenCL-Headers**, **OpenCL-ICD-Loader**, **OpenCL-CLHPP**.

`https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf`
Версия *2.1* спецификации *OpenCL*.

`https://github.com/clMathLibraries`
Исходные коды библиотек фирмы *AMD*: ***clBLAS***, ***clFFT***, ***clSparse***, ***clRNG***.

`https://en.wikipedia.org/wiki/Standard_Portable_Intermediate_Representation`
Статья из Википедии, посвящённая бинарному промежуточному представлению программ *OpenCL*, называемому ***SPIR***.

`https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf`
Версия 1.0 новой спецификации ***SPIR-V***.

`http://www.iwocl.org`
Форум разработчиков *OpenCL* (а также ежегодная конференция) **«International Workshop on OpenCL» (IWOCL)**.

`https://handsonopencl.github.io`
Обучающие материалы по *OpenCL*.

# Приложение. Работа с несколькими *GPU*

Если на компьютере имеется несколько одинаковых *GPU*, может возникнуть проблема их различения в *OpenCL*-программе. В заметке `https://anteru.net/blog/2014/08/01/2483/` предлагается такой подход для устройств *AMD* и *NVidia*.

### AMD

Используется расширение `cl_amd_device_topology`, работающее и на *Linux*, и под *Windows*, — для опроса *PCIe*-шины, уникальной для каждого *GPU*. Фрагмент кода (предполагается, что с помощью директивы `#include` включён файл `cl_ext.h`):

```
cl_device_topology_amd topology;

status = clGetDeviceInfo(devices[i], CL_DEVICE_TOPOLOGY_AMD,
                         sizeof(cl_device_topology_amd), &topology, NULL);

if (status == CL_SUCCESS) {
  if (topology.raw.type == CL_DEVICE_TOPOLOGY_TYPE_PCIE_AMD) {
    std::cout << "INFO: Topology: " << "PCI[ B#" << (int)topology.pcie.bus
              << ", D#" << (int)topology.pcie.device << ", F#"
              << (int)topology.pcie.function << " ]" << std::endl;
  }
} else {
  // Обработать ошибки
}
```

Упоминаемые здесь поля (`.bus`, `.device`, `.function`) структуры `topology` помогут образовать уникальный идентификатор для каждого *GPU* компьютера.

### NVidia

Здесь подход может быть аналогичным, ибо расширение `cl_nv_device_attribute_query` (как оказывается) поддерживает недокументированные значения для параметра функции `clGetDeviceInfo()`: `CL_DEVICE_PCI_BUS_ID_NV` (`0x4008`) и `CL_DEVICE_PCI_SLOT_ID_NV` (`0x4009`); с их помощью можно получить сходную информацию.

# Предметный указатель