Нижегородский государственный университет им. Н.И. Лобачевского

Факультет Вычислительной математики и кибернетики

Программирование на OpenCL

Бастраков С.И.

ВМК ННГУ

sergey.bastrakov@gmail.com

Молодежная школа «Высокопроизводительные вычисления для гибридных вычислительных систем», ННГУ, 2011

Содержание

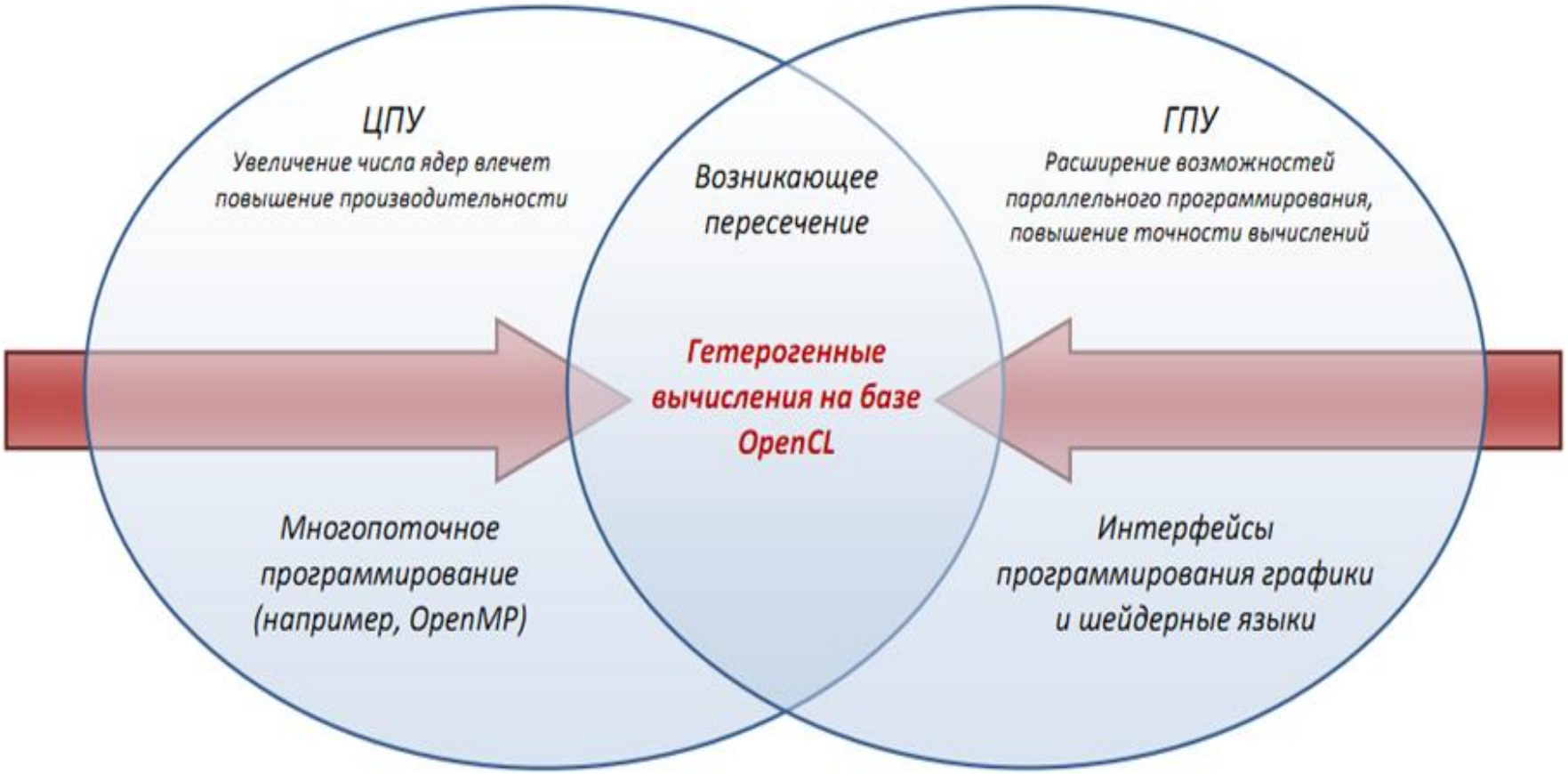
- Стандарт гетерогенных вычислений OpenCL
- Пример приложения с использованием OpenCL
- Обзор реализаций OpenCL

Стандарт гетерогенных вычислений OpenCL

Стандарт OpenCL

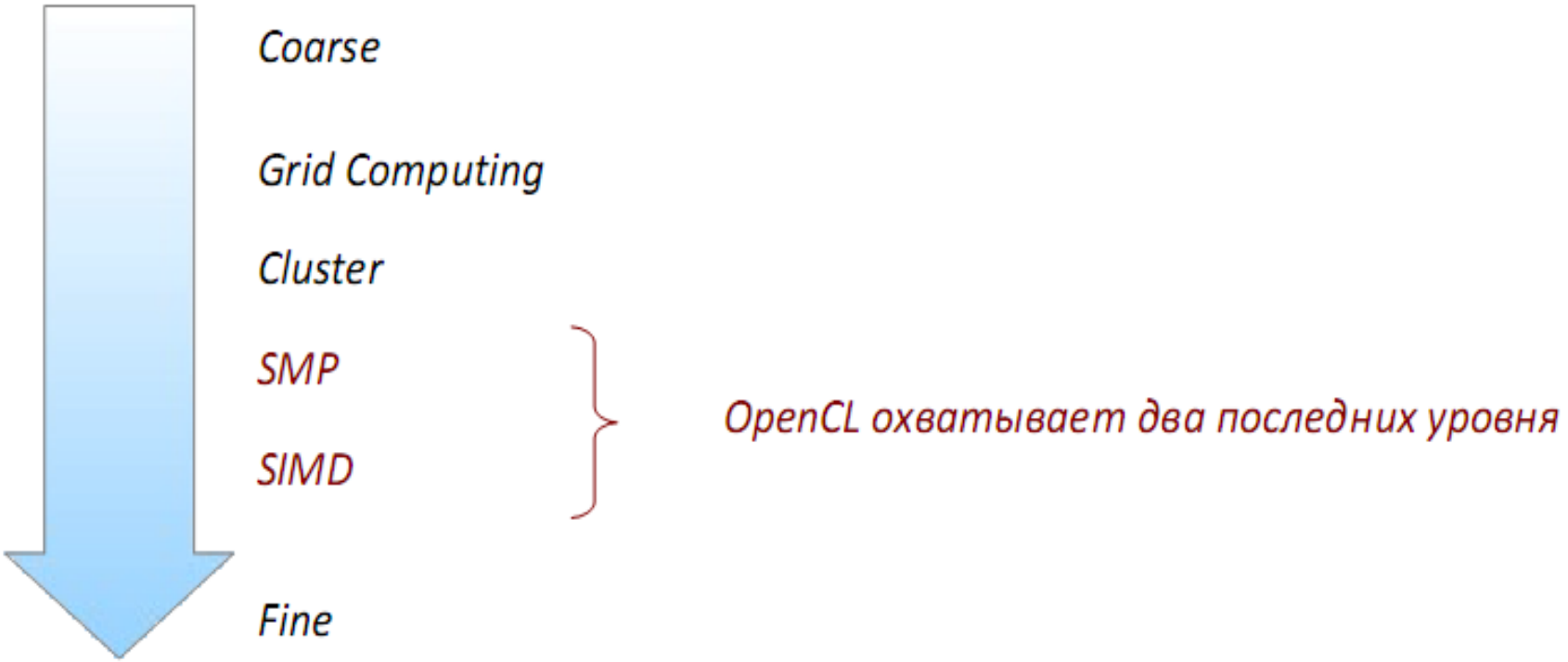
- **OpenCL – Open Computing Language**, открытый стандарт для гетерогенных вычислений, разрабатываемый Khronos Group совместно с представителями производителей устройств и ПО.
- Первая версия стандарта – ноябрь 2008 года.
- Поддерживается Apple, NVIDIA, AMD/ATI, Intel, …
- Поддержка широкого класса вычислительных устройств за счет введения обобщенных моделей (модели платформы, памяти, исполнения, …).

Область применения OpenCL



Источник: Д.К. Боголепов, В.Е. Турлапов «Вычисления общего назначения на графических процессорах»

Охват областей параллелизма



Источник: Д.К. Боголепов, В.Е. Турлапов «Вычисления общего назначения на графических процессорах»

Основные особенности стандарта

- Исходный код приложения легко портируется на другие платформы.
- Поддержка широкого класса устройств достигается за счет введения **обобщенных моделей** данных систем:
  - **модель платформы** (*platform model*);
  - **модель памяти** (*memory model*);
  - **модель исполнения** (*execution model*);
  - **модель программирования** (*programming model*).
- Все модели являются абстрактными (не привязанными к конкретным устройствам), реализация предоставляется производителем.

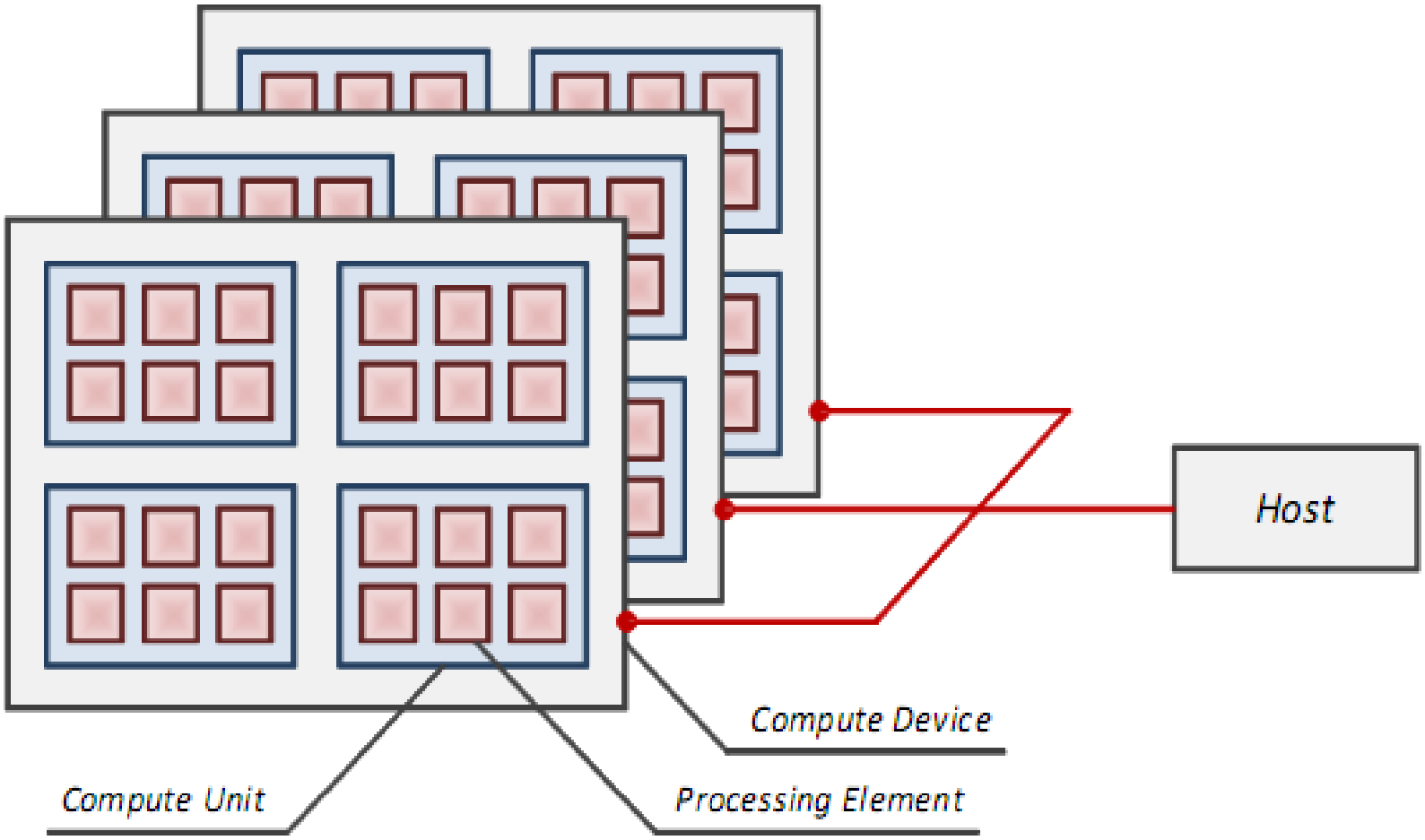
Инструментарий OpenCL

- **Platform Layer API**:
  - уровень аппаратной абстракции над различными вычислительными устройствам;
  - запрос, выбор и инициализация устройств;
  - создание контекстов и очередей команд.
- **Runtime API**:
  - исполнение вычислительных ядер;
  - планирование, вычисления и ресурсы памяти.
- Язык **OpenCL C**:
  - потоковые расширения языка C для написания ядер.

Модель платформы

- Платформа представляется в виде **хост-системы** (*host*), связанной с одним или несколькими **устройствами** (*device*).
  - Центральный процессор может являться одновременно и хост-системой и устройством.
- Устройство состоит из одного или более **вычислительных модулей** (*compute units*), которые могут включать в себя несколько **обрабатывающих элементов** (*processing elements*).
- Непосредственно вычисления производятся в обрабатывающих элементах устройства.

Хост и устройства



Источник: The OpenCL Specification v. 1.1

Выбор платформы

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

- Функция для получения всех доступных платформ:
  - *num_entries* – максимальное количество, которое может быть возвращено;
  - *platforms* – память для записи платформ, если NULL, платформы не записываются;
  - *num_platforms* – память для записи количества платформ.
- Типичная схема работы:
  - первый вызов для определения количества платформ;
  - выделение памяти для объектов платформ;
  - второй вызов для получения объектов платформ.

Выбор платформы

```
cl_int  clGetPlatformInfo (cl_platform_id platform,
                           cl_platform_info param_name,
                           size_t param_value_size,
                           void *param_value,
                           size_t *param_value_size_ret)
```

- Функция для получения характеристик платформы:
  - *platform* – платформа (ее ID);
  - *param_name* – имя запрашиваемой характеристики;
  - *param_value* – указатель на память для записи результата;
  - *param_value_size* – количество памяти, выделенной под *param_value*;
  - *param_value_size_ret* – записанное количество байт.

Выбор платформы

- Возможные значения *param_name*:
  - CL_PLATFORM_PROFILE – поддерживаемый профиль OpenCL (полный или частичный);
  - CL_PLATFORM_VERSION – версия платформы;
  - CL_PLATFORM_NAME – имя платформы;
  - CL_PLATFORM_VENDOR – название производителя;
  - CL_PLATFORM_EXTENSIONS – поддерживаемые расширения стандарта.
- На основе этой информации можно, к примеру, выбрать платформу нужного производителя, если установлено несколько реализаций OpenCL.

Выбор устройства

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

- Функция для получения всех устройств указанного типа (*device_type*) в данной платформе (*platform*).
- Типичная схема работы с 2 вызовами (подобно работе с *clGetPlatformIDs*), *num_entries* задает максимальное количество устройств, которые могут быть записаны в *devices*.

Выбор устройства

- Возможные значения *device_type*:
  - CL_DEVICE_TYPE_CPU – центральный процессор (возможно, многоядерный);
  - CL_DEVICE_TYPE_GPU – графический процессор, поддерживающий работу с графическими API;
  - CL_DEVICE_TYPE_ACCELERATOR – периферийный ускоритель (например, IBM Cell);
  - CL_DEVICE_TYPE_DEFAULT – тип процессора по умолчанию (свойство системы);
  - CL_DEVICE_TYPE_ALL – все доступные OpenCL-совместимые устройства.

Выбор устройства

```
cl_int      clGetDeviceInfo (cl_device_id device,
                             cl_device_info param_name,
                             size_t param_value_size,
                             void *param_value,
                             size_t *param_value_size_ret)
```

- Функция для получения характеристик устройства, смысл параметров аналогичен параметрам функции *clGetPlatformInfo*.
- Позволяет получить широкий перечень характеристик, от типа устройства и названия производителя до размеров памяти всех типов, поддерживаемой арифметики и др.

Контекст

- **Контекст** (*context*) служит для управления объектами и ресурсами OpenCL.
- Все ресурсы OpenCL привязаны к контексту.
- С контекстом ассоциированы следующие данные:
  - устройства;
  - объекты программ;
  - ядра;
  - объекты памяти;
  - очереди команд.

Создание контекста

```
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)
```

- Функция для создания контекста с указанными устройствами.
- *pfn_notify* – callback-функция, вызываемая при возникновении ошибок при дальнейшей работе с контекстом.
- Есть также функция *clCreateContextFromType* для создания контекста, ассоциированного с устройствами определенного типа.

Очередь команд

- **Очередь команд** (*command queue*) является механизмом запроса действия на устройстве со стороны хоста.
- В качестве действия на устройстве могут выступать операции с памятью, запуск ядер, синхронизация.
- Для каждого устройства требуется своя очередь команд.
- Команды внутри очереди могут выполняться синхронно и асинхронно; в порядке установки или нет.

Создание очереди команд

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

- Функция для создания очереди команд, служащей для взаимодействия между заданными контекстом и устройством.

Объекты памяти

- Все операции работы с памятью на устройстве осуществляются с использованием **объектов памяти**.
- Прямая работа с памятью устройства со стороны хоста невозможна (даже если устройство является центральным процессором).
- Для представления одномерных массивов данных используются **буферы** (*buffer objects*). Данные представлены в непрерывном участке памяти, есть прямой доступ со стороны устройства как к массивам.
- Для представления 2- и 3-мерных массивов данных используются **изображения** (*image objects*). Для доступа со стороны устройства используются специальные объекты – **сэмплеры** (*sampler objects*).

Создание буфера

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

- Функция для создания буфера (объект типа *cl_mem*) указанного размера *size* байт в указанном контексте.
- Флаги определяют вариант доступа к буферу со стороны устройства, нужно ли копировать в буфер данные из *host_ptr* и некоторые другие свойства.

Создание буфера

- *flags* является битовым полем со следующими значениями:
  - CL_MEM_READ_WRITE – доступ на чтение и запись;
  - CL_MEM_WRITE_ONLY – доступ только на запись;
  - CL_MEM_READ_ONLY – доступ только на чтение;
  - CL_MEM_USE_HOST_PTR – использовать для хранения объекта буфера (на стороне хоста) в указанной памяти;
  - CL_MEM_ALLOC_HOST_PTR – выделить для хранения буфера новую память;
  - CL_MEM_COPY_HOST_PTR – скопировать в созданный буфер *size* байт из *host_ptr*.

Обмен данными между хостом и устройством

- Для обмена данными служат функции:

  clEnqueue{Read|Write}{Buffer|Image}

- Под записью (write) понимается копирование данных с хоста на устройства, под чтением (read) – с устройства на хост.
- Возможна также установка прямого соответствия между участками памяти на хосте и устройстве при помощи **clEnqueueMap{Buffer|Image}**

Обмен данными между хостом и устройством

```
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)

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

Объекты программы и ядер

- **Ядром** называется функция, являющаяся частью программы и параллельно исполняющаяся на устройстве. Ядро является аналогом потоковой функции.
- Часть, выполняющаяся на устройстве, состоит из набора ядер, объявленных с квалификатором **__kernel**.
- Компилирование ядер может осуществляться во время исполнения программы с помощью функций API.
- **Объект программы** (*program object*) служит для представления следующих данных:
  - исходные и/или скомпилированные тексты ядер;
  - данные о компиляции.
- Работа с ядрами со стороны осуществляется при помощи **объектов ядер**.

Создание объекта программы

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

- Функция для создания объекта программы из исходного кода ядер (компилирование при этом не производится).

Компилирование программы

```
cl_int      clBuildProgram (cl_program program,
                            cl_uint num_devices,
                            const cl_device_id *device_list,
                            const char *options,
                            void (CL_CALLBACK *pfn_notify)(cl_program program,
                                                           void *user_data),
                            void *user_data)
```

- Функция для компилирования и сборки ядер в составе программы для указанных устройств. Опции сборки (макросы, опции компилятора) указываются через options.
- В случае ошибок компиляции возвращаемый результат отличен от CL_SUCCESS, подробная информация может быть получена при помощи функции **clGetProgramBuildInfo()**

Создание объектов ядер

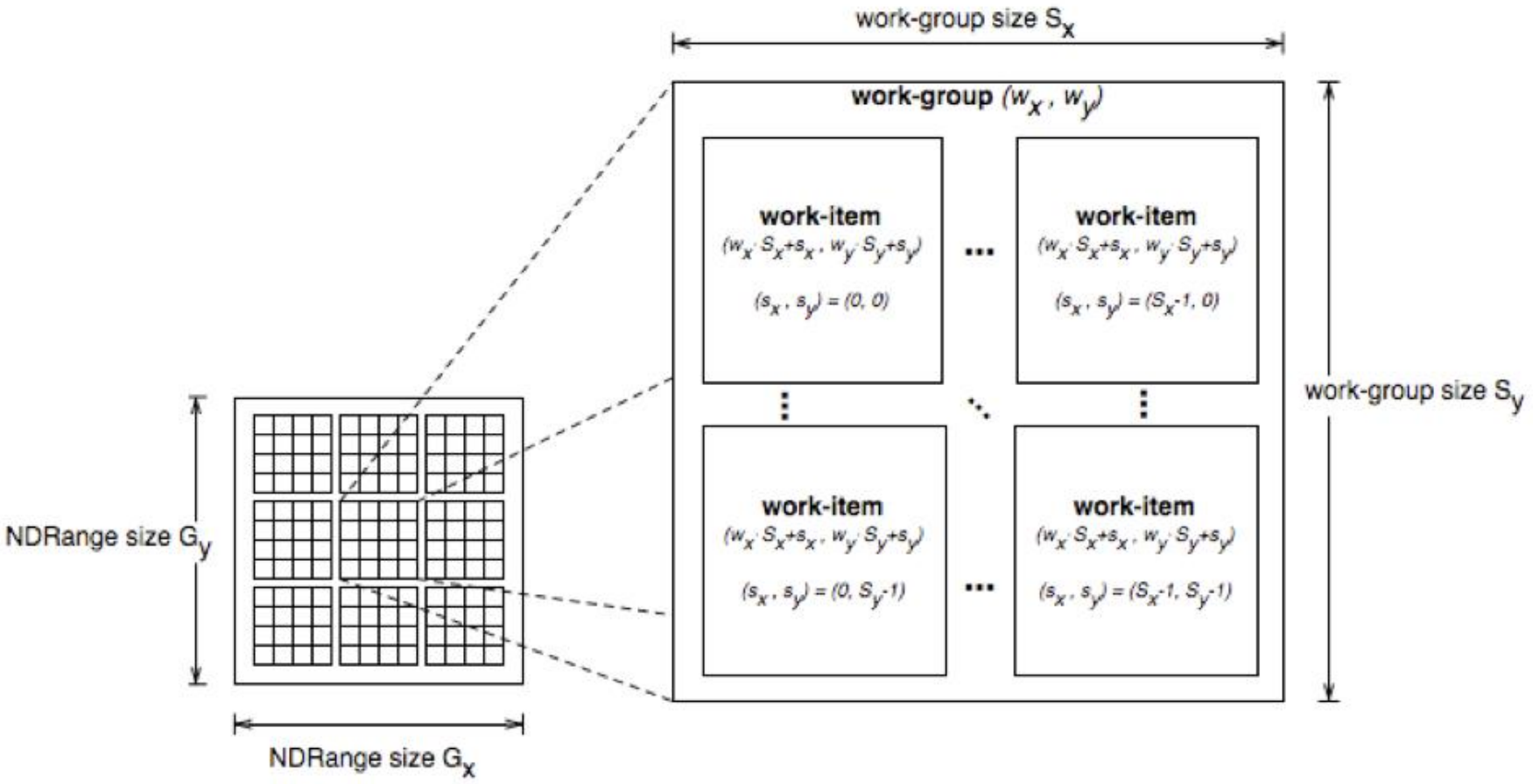
```
cl_kernel    clCreateKernel (cl_program program,
                             const char *kernel_name,
                             cl_int *errcode_ret)
```

- Функция для создания объекта ядра по имени функции-ядра в исходном коде.

Модель исполнения

- Каждый экземпляр ядра называется **элементом работы** (*work-item*). При исполнении ядра элементы работы могут выполняться параллельно.
- Элементы работы объединены в **группы работ** (*work-group*), независимые друг от друга.
- Иерархия элементов работы и групп работ определяется **пространством индексов** (*index space*).
- Для распределения работы каждая группа работ имеет индекс, каждый элемент работы имеет уникальный глобальный и локальный (внутри группы работы) индексы.
- Индексы могут быть 1-, 2- и 3-мерные.
- Пример: ядро вычисляет матричное произведение, каждый элемент работы вычисляет один элемент результирующей матрицы.

Пространство индексов



Написание ядер

- Ядро является функцией со спецификатором **__kernel**, возвращающей void.
- Доступ к индексам элемента работы внутри ядра осуществляется при помощи функций:

  get_global_id(dim)

  get_global_size(dim)

  get_group_id(dim)

  get_num_groups(dim)

  get_local_id(dim)

  get_local_size(dim),

  где dim – номер размерности (0, 1 или 2 в текущих реализациях OpenCL).

Пример ядра

- Сложение двух векторов. Пространство индексов одномерно, каждый элемент работы вычисляет один элемент результирующего вектора:

```
__kernel void vecAdd (__global int * a,
      __global int * b, __global int * c)
{
      int idx = get_global_id(0);
      c[idx] = a[idx] + b[idx];
}
```

Запуск ядра

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

- Функция для установки значений аргументов ядра при его вызове.
- Необходимо вызвать ее для каждого аргумента ядра.
- Для передачи одномерных массивов необходимо передать соответствующий буфер.

Источник: The OpenCL Specification v. 1.1

Запуск ядра

```
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)
```

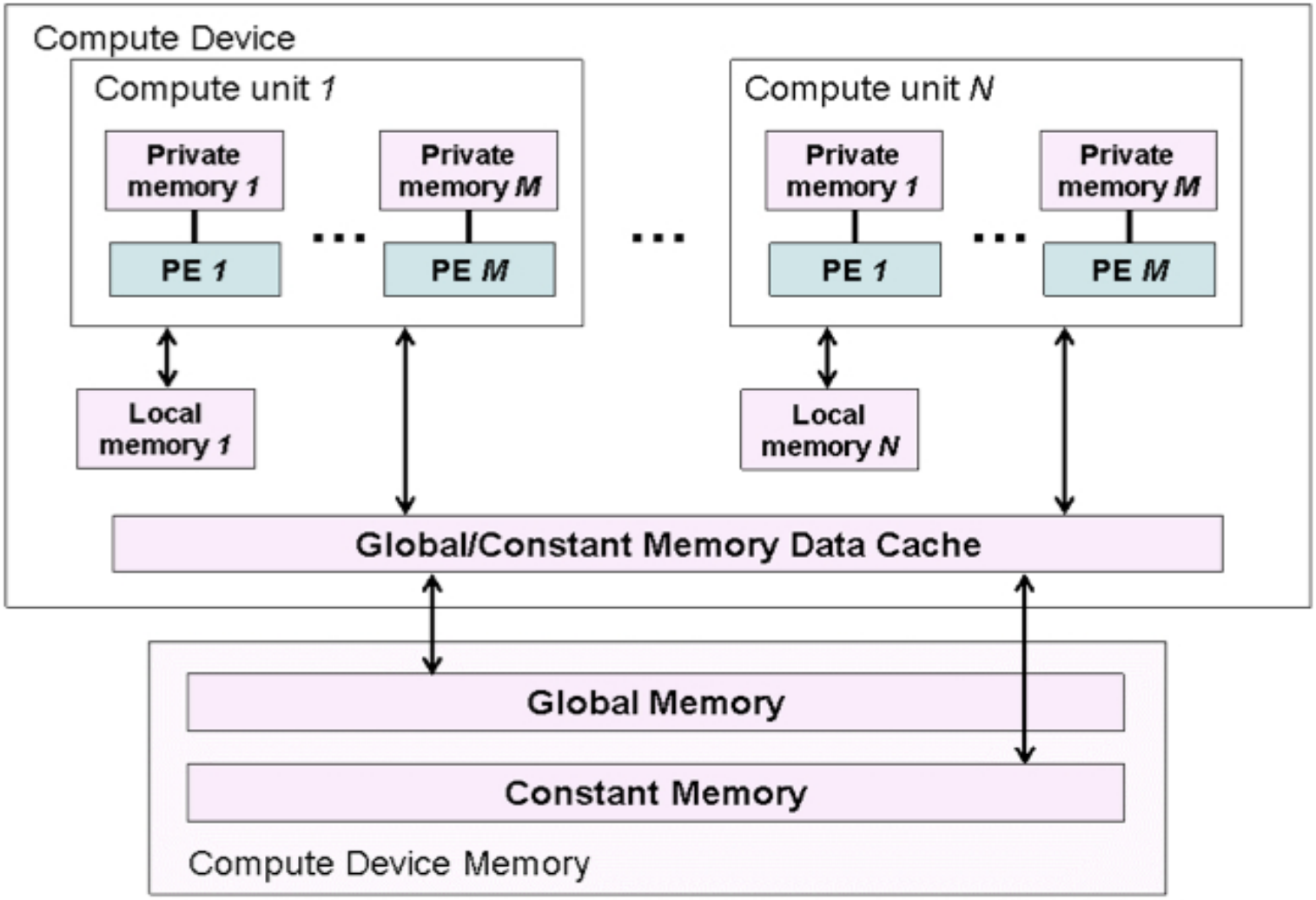
- Функция для постановки запуска ядра в очередь команд, указываются параметры пространства индексов:
  - *work_dim* – размерность пространства индексов;
  - *global_work_offset* – начальные глобальные индексы;
  - *global_work_size* – общее количество элементов работы;
  - *local_work_size* – количество элементов работы в группе работ.

Источник: The OpenCL Specification v. 1.1

Модель памяти

- Типы памяти на устройстве:
  - **глобальная** (*global*), доступ из всех элементов работы;
  - **константная** (*constant*), доступ из всех элементов работы только на чтение;
  - **локальная** (*local*), доступ из элементов работы в одной группе работ (эксклюзивна для группы работ);
  - **частная** (*private*), эксклюзивна для каждого элемента работы.
- Гарантируется область видимости, но не конкретная реализация и размещение различных областей памяти.

Модель памяти



Источник: The OpenCL Specification v. 1.1

Квалификаторы памяти

- **__global** или **global** – данные в глобальной памяти.
- **__constant** или **constant** – данные в константной памяти.
- **__local** или **local** – данные в локальной памяти.
- **__private** или **private** – данные в частной памяти.
- Для изображений (image) используются квалификаторы режима доступа **__read_only**/**__write_only**.
- Явное указание квалификаторов памяти обязательно для указателей в ядре.

Синхронизация в ядре

void **barrier** (cl_mem_fence_flags *flags*)

- Функция для барьерной синхронизации элементов работы внутри одной группы работы.
- *flags* определяют операции упорядочивания обращений к памяти, выполняемые при синхронизации, возможные значения:
  - CLK_LOCAL_MEM_FENCE;
  - CLK_GLOBAL_MEM_FENCE.
- Нет явной возможности для барьерной синхронизации элементов работы в разных группах работ в ходе работы ядра.
- Есть атомарные функции для локальной и глобальной памяти.

Источник: The OpenCL Specification v. 1.1

Синхронизация в очереди команд

- Гибкий механизм синхронизации и асинхронного выполнения команд в одной очереди команд:
  - барьерная синхронизация;
  - синхронизация на основе событий.
- Позволяет эффективно задействовать устройства за счет перекрытия вычислений и обменов данными.

```
cl_int      clFinish (cl_command_queue command_queue)

cl_int      clFlush (cl_command_queue command_queue)

cl_int      clEnqueueMarker (cl_command_queue command_queue,
                             cl_event *event)

cl_int      clWaitForEvents (cl_uint num_events, const cl_event *event_list)
```

Источник: The OpenCL Specification v. 1.1

Освобождение ресурсов

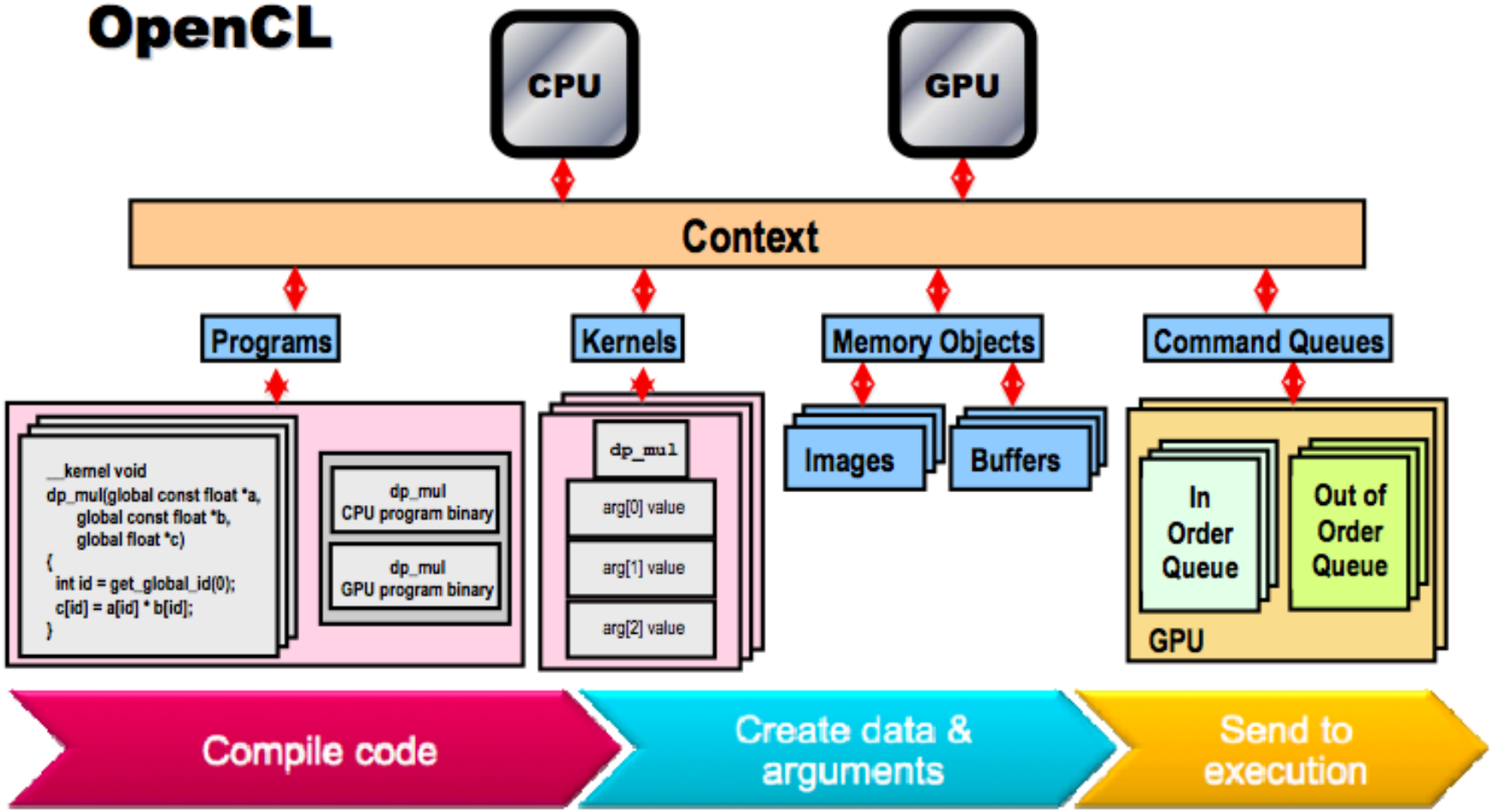
- Используется механизм подсчета ссылок на все ресурсы OpenCL (объекты памяти, ядра, программа, очередь команд, контекст).
- **clRetain…** увеличивает счетчик ссылок на 1 (вызывается автоматически при создании объектов), **clRelease…** уменьшает счетчик ссылок на 1 и освобождает ресурс при необходимости.
- Примеры:

```
cl_int    clReleaseMemObject (cl_mem memobj)
cl_int    clReleaseKernel (cl_kernel kernel)
cl_int    clReleaseProgram (cl_program program)
cl_int    clReleaseContext (cl_context context)
```

Контроль ошибок

- Все функции OpenCL API возвращают коды ошибок (в виде непосредственного результата либо через специальный аргумент-указатель на статус ошибки).
- Возвращаемое значение CL_SUCCESS, равное 0, соответствует успешному завершению функции.
- Возвращаемые отрицательные значения соответствуют ошибкам, определение соответствующих макросов в файле cl.h.

Общая схема работы



Модель программирования

- **Параллелизм по данным** (*data parallel*):
  - Соответствие между пространством индексов и размером задачи.
  - Каждый элемент работы выполняет фиксированное количество операций, масштабируется количество элементов работы и групп работ).
- **Параллелизм по задачам** (*task parallel*):
  - Разные ядра исполняются независимо на различных пространствах индексов.
  - Постановка в очередь нескольких задач.
- Синхронизация:
  - Между элементами работы в одной группе работ.
  - Между командами в одной очереди команд.

Пример приложения с использованием OpenCL

Постановка задачи

- В качестве учебного примера рассмотрим задачу поэлементного возведения в квадрат компонент вектора.
- На данном примере будут продемонстрированы все основные этапы разработки приложения с использованием OpenCL.

Этап 1 – разработка ядер

- Каждый элемент работы вычисляет квадрат одного из элементов массива.
- Для простоты сделаем ядро строковой константой.

```
01 | // Ядро для расчета квадрата каждого элемента входного массива
02 | const char * source =
03 | "__kernel void square (                                          \n"\
04 | "                       __global float * input,                  \n"\
05 | "                       __global float * output,                 \n"\
06 | "                       const unsigned int count                 \n"\
07 | "                     ) {                                        \n"\
08 | "    int i = get_global_id ( 0 );                                \n"\
09 | "                                                                \n"\
10 | "    if ( i < count )                                            \n"\
11 | "       output [i] = input [i] * input [i];                       \n"\
12 | "}"
```

Этап 2 – выбор платформы и устройств

- Получение информации о доступных платформах:

```
01 | cl_uint numPlatforms = 0;
02 |
03 | clGetPlatformIDs ( 0                  /* num_entries */,
04 |                    NULL               /* platforms */,
05 |                    &numPlatforms      /* num_platforms */ );
06 |
07 | cl_platform_id platform = NULL;
08 |
09 | if ( 0 < numPlatforms )
10 | {
11 |     cl_platform_id * platforms = new cl_platform_id [numPlatforms];
12 |
13 |     clGetPlatformIDs ( numPlatforms   /* num_entries */,
14 |                        platforms      /* platforms */,
15 |                        NULL           /* num_platforms */ );
16 |
17 |     platform = platforms [0]; delete [] platforms;
18 | }
```

Этап 2 – выбор платформы и устройств

- Создание контекста:

```
01 | // Создаем свойства контекста для задания конкретной платформы
02 | cl_context_properties properties [3] = {
03 |     CL_CONTEXT_PLATFORM,(  cl_context_properties ) platform, 0
04 | };
05 |
06 | // Создаем контекст с заданными свойствами для всех графических процессоров
07 | cl_context context = clCreateContextFromType (
08 |     ( NULL == platform ) ? NULL : properties    /* properties */,
09 |     CL_DEVICE_TYPE_GPU                          /* device_type */,
10 |     NULL                                        /* pfn_notify */,
11 |     NULL                                        /* user_data */,
12 |     NULL                                        /* errcode_ret */ );
13 |
14 | // Определяем размер массива (в байтах) для хранения списка устройств
15 | size_t size = 0;
16 |
17 | clGetContextInfo (
18 |     context                 /* context */,
19 |     CL_CONTEXT_DEVICES      /* param_name */,
20 |     0                       /* param_value_size */,
21 |     NULL                    /* param_value */,
22 |     &size                   /* param_value_size_ret */ );
```

Этап 2 – выбор платформы и устройств

- Выбор устройства:

```
01 | // Выбираем устройство для вычислений (в данном примере это первое устройство)
02 | cl_device_id device;
03 |
04 | if ( size > 0 )
05 | {
06 |     cl_device_id * devices = ( cl_device_id * ) alloca ( size );
07 |
08 |     clGetContextInfo (
09 |         context                 /* context */,
10 |         CL_CONTEXT_DEVICES      /* param_name */,
11 |         size                    /* param_value_size */,
12 |         devices                 /* param_value */,
13 |         NULL                    /* param_value_size_ret */ );
14 |
15 |     device = devices [0];
16 | }
```

- Замечание: возможен другой порядок – сначала запрашивается список доступных платформе устройств, затем для выбранного устройства создается контекст.

Этап 3 – создание очереди команд

- Создание очереди команд для заданного контекста и выбранного устройства:

```
01 | // Создаем очередь команд для заданного контекста и выбранного устройства
02 | cl_command_queue queue = clCreateCommandQueue (
03 |     context   /* context */,
04 |     device    /* device */,
05 |     0         /* properties */,
06 |     NULL      /* errcode_ret */ );
```

Этап 4 – объекты программы и ядер

- Создание объектов программы и ядра:

```
01 | // Создаем программный объект из исходного кода (определен выше)
02 | size_t srclen [] = { strlen ( source ) };
03 |
04 | cl_program program = clCreateProgramWithSource (
05 |     context       /* context */,
06 |     1             /* count */,
07 |     &source       /* strings */,
08 |     srclen        /* lengths */,
09 |     NULL          /* errcode_ret */ );
10 |
11 | // Создаем исполняемый файл программы для выбранного устройства (ГПУ)
12 | clBuildProgram ( program    /* program */,
13 |                  1          /* num_devices */,
14 |                  &device    /* device_list */,
15 |                  NULL       /* options */,
16 |                  NULL       /* pfn_notify */,
17 |                  NULL       /* user_data */ );
18 |
19 | // Создаем объект ядра для возведения массива в квадрат (ядро дано выше)
20 | cl_kernel kernel = clCreateKernel ( program        /* program */,
21 |                                     "square"       /* kernel_name */,
22 |                                     NULL           /* errcode_ret */ );
```

Этап 5 – объекты памяти

- Создание входного и выходного буферов:

```
01 | float data [SIZE];                    // Массив входных данных
02 | float results [SIZE];                 // Массив выходных данных
03 |
04 | for ( int i = 0; i < SIZE; i++ )
05 |     data [i] = rand ( );
06 |
07 | // Создаем объект памяти в виде буфера для передачи ядру входного массива
08 | cl_mem input = clCreateBuffer (
09 |     context                           /* context */,
10 |     CL_MEM_READ_ONLY                  /* flags */,
11 |     sizeof ( float ) * SIZE           /* size */,
12 |     NULL                              /* host_ptr */,
13 |     NULL                              /* errcode_ret */ );
```

```
01 | // Создаем объект памяти в виде буфера для передачи ядру выходного массива
02 | cl_mem output = clCreateBuffer (
03 |     context                           /* context */,
04 |     CL_MEM_WRITE_ONLY                 /* flags */,
05 |     sizeof ( float ) * SIZE           /* size */,
06 |     NULL                              /* host_ptr */,
07 |     NULL                              /* errcode_ret */ );
```

Этап 5 – объекты памяти

- Копирование входного буфера в память устройства:

```
01 | // Помещаем в очередь команду записи входного массива в объект памяти
02 | clEnqueueWriteBuffer (
03 |     queue                          /* command_queue */,
04 |     input                          /* buffer */,
05 |     CL_TRUE                        /* blocking_write */,
06 |     0                              /* offset */,
07 |     sizeof ( float ) * SIZE        /* cb */,
08 |     data                           /* ptr */,
09 |     0                              /* num_events_in_wait_list */,
10 |     NULL                           /* event_wait_list */,
11 |     NULL                           /* event */ );
```

Этап 6 – запуск ядра

- Установка аргументов ядра:

```
01 | // Задаем аргументы ядра
02 | unsigned int count = SIZE;
03 |
04 | clSetKernelArg (
05 |     kernel                      /* kernel */,
06 |     0                           /* arg_index */,
07 |     sizeof ( cl_mem )           /* arg_size */,
08 |     &input                      /* arg_value */ );
09 |
10 | clSetKernelArg (
11 |     kernel                      /* kernel */,
12 |     1                           /* arg_index */,
13 |     sizeof ( cl_mem )           /* arg_size */,
14 |     &output                     /* arg_value */ );
15 |
16 | clSetKernelArg (
17 |     kernel                      /* kernel */,
18 |     2                           /* arg_index */,
19 |     sizeof ( unsigned int )     /* arg_size */,
20 |     &count                      /* arg_value */ );
```

Этап 6 – запуск ядра

- Определение глобального и локального размеров работы и запуск ядра:

```
01 | size_t group;   // Максимальный размер группы работ
02 |
03 | clGetKernelWorkGroupInfo (
04 |     kernel                          /* kernel */,
05 |     device                          /* device */,
06 |     CL_KERNEL_WORK_GROUP_SIZE       /* param_name */,
07 |     sizeof ( size_t )               /* param_value_size */,
08 |     &group                          /* param_value */,
09 |     NULL                            /* param_value_size_ret */ );
10 |
11 | // Выполнение ядра над всем множеством входных данных
12 | clEnqueueNDRangeKernel (
13 |     queue           /* command_queue */,
14 |     kernel          /* kernel */,
15 |     1               /* work_dim */,
16 |     NULL            /* global_work_offset */,
17 |     &count          /* global_work_size */,
18 |     &group          /* local_work_size */,
19 |     0               /* num_events_in_wait_list */,
20 |     NULL            /* event_wait_list */,
21 |     NULL            /* event */ );
22 |
23 | clFinish ( queue );   // Ожидаем завершения всех команд в очереди
```

Этап 7 – загрузка результатов вычислений

- Копирование результирующего буфера в память хоста:

```
01 | // Загрузка результатов вычислений с устройства
02 | clEnqueueReadBuffer (
03 |     queue                          /* command_queue */,
04 |     output                         /* buffer */,
05 |     CL_TRUE                        /* blocking_read */,
06 |     0                              /* offset */,
07 |     sizeof ( float ) * count       /* cb */,
08 |     results                        /* ptr */,
09 |     0                              /* num_events_in_wait_list */,
10 |     NULL                           /* event_wait_list */,
11 |     NULL                           /* event */ );
```

Этап 8 – освобождение ресурсов

- Освобождение использованных ресурсов:

```
01 | clReleaseMemObject ( input );
02 | clReleaseMemObject ( output );
03 | clReleaseProgram ( program );
04 | clReleaseKernel ( kernel );
05 | clReleaseCommandQueue ( queue );
06 | clReleaseContext ( context )
```

Обзор реализаций OpenCL

Использование OpenCL

- Основным достоинством OpenCL является переносимость между различными вычислительными платформами. На данный момент OpenCL является уникальным средством такого рода.
- Естественным требованием для этого является необходимость оперирования обобщенными терминами, что усложняет модель программирования и затрудняет оптимизацию для конкретных платформ.
- При этом стандарт хорошо проработан и содержит возможности для низкоуровневой оптимизации для конкретных устройств и достижения высокой эффективности. Техники оптимизации для разных платформ (например, CPU и GPU) существенно различны.

Intel OpenCL

- Реализация стандарта для многоядерных центральных процессоров. Вероятно, также будет поддерживать устройства архитектуры Intel MIC.
- Основана на Intel TBB.
- Использует оптимизирующий компилятор с возможностями автоматической векторизации кода.
- Содержит набор примеров (SDK) и отдельный компилятор с возможностью просмотра ассемблера и LLWM (промежуточного векторого языка).

NVIDIA OpenCL

- Реализация стандарта для графических процессоров NVIDIA.
- Использует архитектуру CUDA.
- OpenCL во многом похож на обобщенную версию CUDA C, тем не менее в настоящее время последний является значительно более популярным и динамично развивающимся средством разработки для GPU NVIDIA.
- NVIDIA предоставляет обобщенные средства разработки на CUDA C и OpenCL: CUDA Toolkit и GPU Computing SDK с примерами на CUDA C, OpenCL и DirectCompute.

AMD OpenCL

- Реализация стандарта для многоядерных центральных процессоров и графических процессоров ATI (а также APU).
- Является единственным развиваемым средством программирования для GPU AMD.
- Содержит набор примеров (SDK) и инструменты разработки (профилировщик).

Материалы

- OpenCL – официальный сайт: http://www.khronos.org/opencl/
- Intel OpenCL: http://software.intel.com/en-us/articles/intel-opencl-sdk/
- NVIDIA OpenCL: http://www.nvidia.ru/object/cuda_opencl_new_ru.html
- AMD OpenCL: http://www.amd.com/us/products/technologies/stream-technology/opencl/Pages/opencl.aspx