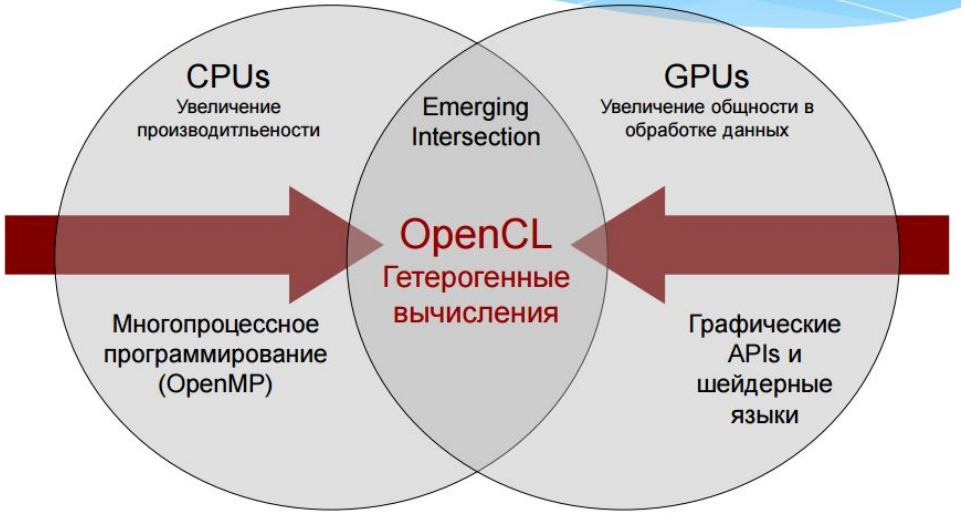


Параллельное и распределенное программирование.

Технология программирования гетерогенных систем OpenCL.

Лекция 2

План лекции

- OpenCL архитектура
- Простейшая программа

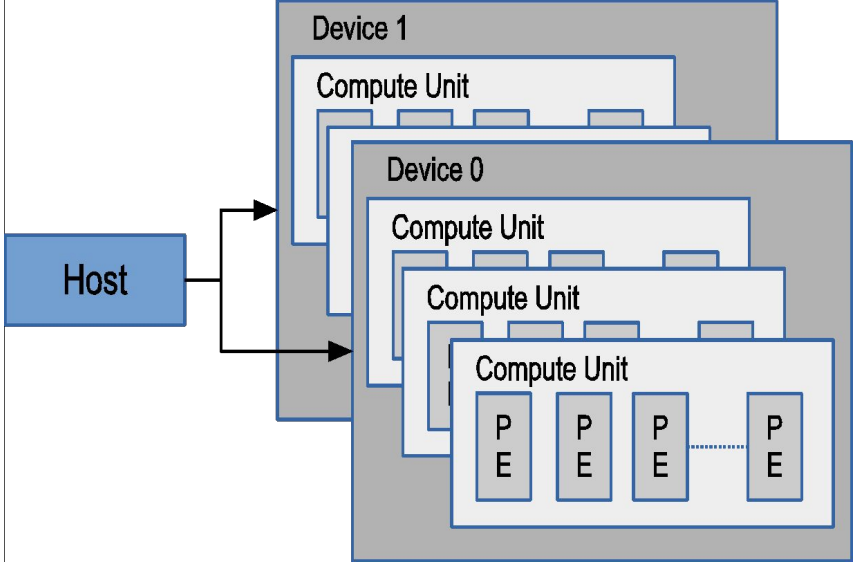
OpenCL архитектура

- OpenCL позволяет проводить параллельные вычисления на гетерогенных устройствах
  - Процессоры, графические процессоры, ПЛИС и т. д.
- Предоставляет переносимый код. OpenCL определяется в четырех моделях:
  - модель платформы;
  - модель исполнения;
  - модель памяти;
  - модель программирования.

Модель платформы OpenCL

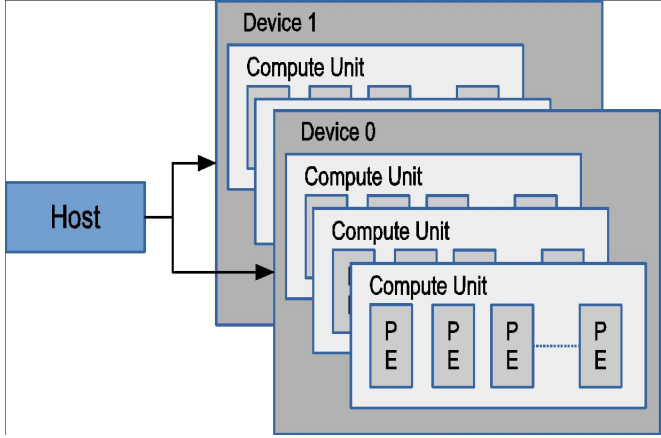
- Модель платформы описывает вычислительные ресурсы, используемые OpenCL и их взаимосвязь между собой
- Каждая реализация OpenCL (т. е. библиотека OpenCL) может создавать платформы, состоящие из ресурсов в системе, с которыми она способна взаимодействовать
  - Например, платформа AMD может состоять из процессоров X86 и графических процессоров Radeon
- OpenCL использует модель «Installable Client Driver»
  - Цель состоит в том, чтобы позволить платформам от разных поставщиков сосуществовать
  - Приложения могут выбирать платформу во время выполнения

Модель платформы OpenCL



- Модель платформы определяет хост (Host), подключенный к одному или нескольким вычислительным устройствам
- Устройство разделено на один или несколько вычислительных блоков
- Вычислительные единицы (Compute Unit) делятся на один или несколько элементов обработки
  - Каждый обрабатывающий элемент поддерживает собственный счетчик программ

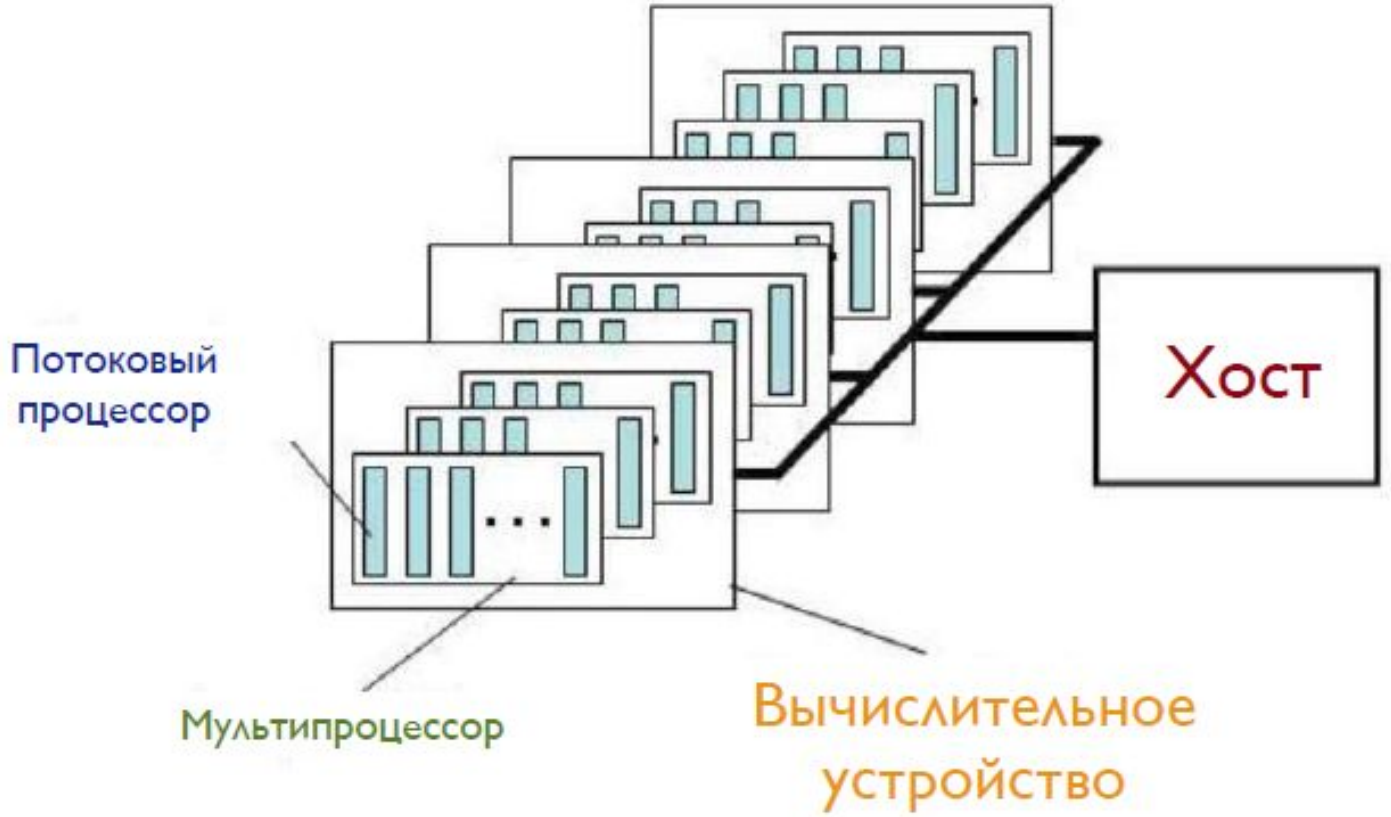
Host/Devices



- Хост - это любой процессор, на котором работает библиотека OpenCL
  - Процессоры x86 в целом
- Устройства - это процессоры, с которыми библиотека может разговаривать
  - Процессоры, графические процессоры, ПЛИС и другие ускорители
- Для AMD
  - Все ЦП объединены в одно устройство (каждое ядро является вычислительным блоком и обрабатывающим элементом)
  - Каждый графический процессор представляет собой отдельное устройство

Модель платформы OpenCL

Модель платформы. Аппаратная система



Обнаружение платформ

получить платформу:

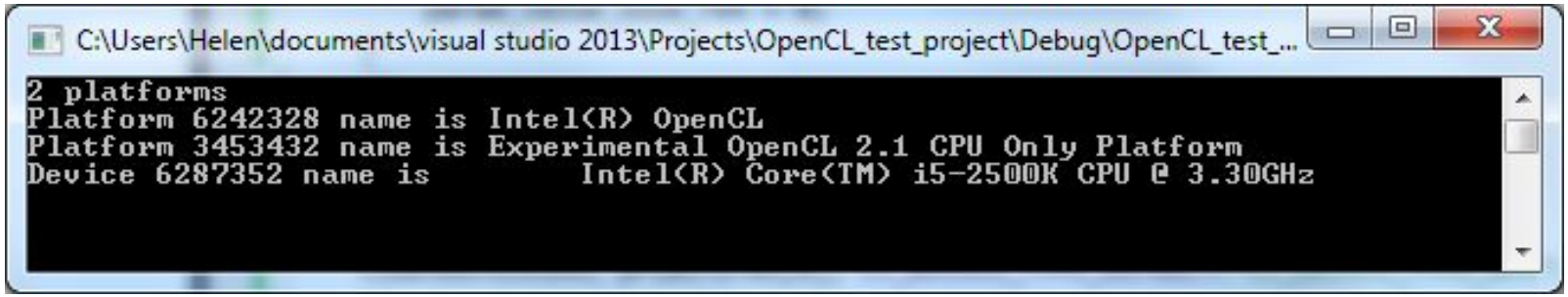
```
cl_int clGetPlatformIDs(
            // размер массива, на который указывает pPlatforms
            cl_uint nNumEntries,
            // массив возврата информации об устройствах
            cl_platform_id *pPlatforms,
            // возвращаемое количество устройств OpenCL
            cl_uint *pnNumPlatforms);
```

- Платформа выбирается с помощью двойного вызова API
  - Первый вызов используется для получения количества платформ, доступных для реализации
  - Затем пространство памяти выделяется для объектов платформы
  - Второй вызов используется для извлечения объектов платформы

Обнаружение платформ

Информация о платформе:

```
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)
```



Обнаружение устройств на платформе

- Получить устройства

```
cl_int clGetDeviceIDs(
cl_platform_id platformID,
cl_device_type nDeviceType, cl_uint nNumEntries,
cl_device_id *pDevices, cl_uint *pnNumDevices);
```

Таблица - Категории устройств в OpenCL

nDeviceType	Описание
CL_DEVICE_TYPE_CPU	центральный процессор
CL_DEVICE_TYPE_GPU	видеокарта
CL_DEVICE_TYPE_ACCELERATOR	специализированный ускоритель
CL_DEVICE_TYPE_DEFAULT	устройство по умолчанию в системе
CL_DEVICE_TYPE_ALL	все доступные устройства OpenCL

- Получить информацию

```
clGetDeviceInfo()
```

Контекст

- Контекст - это среда для управления объектами и ресурсами OpenCL
- Для управления программами OpenCL следующее связано с контекстом
  - Устройства: вычислительные устройства
  - Объекты программы: источник программы, который реализует ядра
  - Ядра: функции, выполняемые на устройствах OpenCL
  - Объекты памяти: данные, которыми управляет устройство
  - Командные очереди: механизмы взаимодействия с устройствами
    - Команды включают: передачу данных, выполнение ядра и синхронизацию
- Когда вы создаете контекст, вы предоставляете список устройств для связи с ним
  - Для остальных ресурсов OpenCL вы свяжете их с контекстом по мере их создания

Программная модель

- Хост-программа
- Программа для устройства
  - Набор ядер

Аппаратура	Программа
Потоковый процессор (обрабатывающий элемент)	Поток
Мультипроцессор (вычислительный узел)	Блок потоков
Вычислительное устройство	Ядро
Хост	Хост-программа

Программная модель. Основные определения

Ядро (kernel) – функция, исполняемая устройством. Имеет в описании спецификацию __kernel.

Программа (program) – набор ядер, а также, возможно, вспомогательных функций, вызываемых ими, и константных данных.

Приложение (application) – комбинация программ, работающих на управляющем узле и вычислительных устройствах.

Команда (command) – операция OpenCL, предназначенная для исполнения (исполнение ядра на устройстве, манипуляция с памятью и т.д.)

Объекты (1/2)

Объект (object) – абстрактное представление ресурса, управляемого OpenCL API (объект ядра, памяти и т.д.).

Дескриптор (handle) – непрозрачный тип, ссылающийся на объект, выделяемый OpenCL. Любая операция с объектом выполняется через дескриптор.

Очередь команд (command-queue) – объект, содержащий команды для исполнения на устройстве.

Объект ядра (kernel object) – хранит отдельную функцию ядра программы вместе со значениями аргументов.

Объекты (2/2)

Объект события (event object) – хранит состояние команды. Предназначен для синхронизации.

Объект буфера (buffer object) – последовательный набор байт. Доступен из ядра через указатель и из управляющего узла при помощи вызова API.

Объект памяти (memory object) – ссылается на область глобальной памяти.

Контекст

```
// создание контекста
cl_context clCreateContext(
        const cl_context_properties *pProperties,
        cl_uint num_devices, const cl_device_id *pDevices,
        void (CL_CALLBACK *pfnNotify)(
                const char *pcszErrInfo,
                const void *pvPrivateInfo, size_t uSizePrivateInfo,
                void *pvUserData),
        void *pvUserData, cl_int *pnErrCodeRet);
```

- Функция создает контекст с учетом списка устройств
- Аргумент properties указывает, какую платформу использовать (если NULL будет использоваться по умолчанию, выбранным поставщиком)
- Функция также обеспечивает механизм обратного вызова для сообщения об ошибках пользователю

Очередь команд

```
// создание очереди команд
cl_command_queue clCreateCommandQueue(
        cl_context context, cl_device_id deviceID,
        cl_command_queue_properties nProperties,
        cl_int *pnErrCodeRet);
```

Таблица – Свойства очередей команд

nProperties	Описание
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE	исполнение не по порядку
CL_QUEUE_PROFILING_ENABLE	включение профилирования

- Очередь команд - это механизм, по которому хост запрашивает, чтобы действие выполнялось устройством (т. е. Хост посылает команды на устройство)
  - Команды включают запуск передачи памяти, начало выполнения ядра и т. д.
- Свойства очереди команд задают:
  - Разрешено ли выполнение команд вне очереди
  - Включено ли профилирование
  - Должна ли эта очередь находиться на устройстве

Очередь команд

- Поскольку очередь команд нацелена на одно устройство, для каждого устройства требуется отдельная очередь команд
- Некоторые команды в очереди могут быть указаны как синхронные или асинхронные
- Команды могут выполняться в порядке или по порядку
- Командные очереди связывают контекст с устройством

События

- События (Events) - это механизм OpenCL для определения зависимостей между командами
- Все вызовы OpenCL API для включения команды в очередь команд имеют возможность генерации события и выбор списка событий, которые должны выполняться до выполнения этой команды
- Список событий, определяющих зависимости, называется списком ожидания
- События также используются для профилирования
- С командной строкой в порядке (по умолчанию) каждая команда будет завершена до начала следующей команды, поэтому вручную не указывать зависимости не требуется

Синхронизация очередей

cl_int **clFinish** (cl_command_queue *command_queue*)

- Вызов clFinish блокирует хост-программу до тех пор, пока все команды не будут завершены
  - На практике этот вызов имеет более высокие накладные расходы, чем определение зависимостей с использованием событий, и их следует использовать экономно, когда требуется высокая производительность

Привязка на стороне устройства (Device)

- OpenCL 2.0 представил командные очереди на стороне устройства
  - Позволяет устройству вставлять команды самому себе
  - Например, Ядро может вставить другое выполнение ядра на одно и то же устройство
  - Родительские и дочерние ядра выполняются асинхронно
  - Родительское ядро не зарегистрировано как завершено до тех пор, пока все его дочерние ядра не будут завершены
- Командные очереди на стороне устройства контролируются с помощью событий
  - События могут использоваться для обеспечения зависимостей

Объекты памяти

- Объектами памяти являются дескрипторы данных, к которым может обращаться ядро
  - Типы объектов OpenCL - это буферы, изображения и каналы (pipe)
- Буферы
  - Смежные куски памяти сохраняются последовательно и могут быть доступны напрямую (массивы, указатели, структуры)
  - Возможность чтения / записи
- Изображения
  - Непрозрачные объекты (2D или 3D)
  - Доступ только через встроенные функции read_image () и write_image ()
  - Могут быть прочитаны, записаны или оба в ядре (новое в OpenCL 2.0)
- Pipe (новое в OpenCL 2.0)
  - Упорядоченная последовательность элементов данных, называемых пакетами
  - Доступ к ним возможен только через инструкции read_pipe () и write_pipe ()

Буфер данных

- Одномерный массив в памяти хоста или устройства
  - Копирование
    - `clEnqueue{Read,Write,Copy}Buffer()`
    - Блокирующее/неблокирующее
  - Отображение
    - `clEnqueue{Map,Unmap}Buffer()`

Создание буфера

```
// создание буфера
cl_mem clCreateBuffer(
    cl_context context, cl_mem_flags nFlags,
    size_t uSize, void *pvHostPtr, cl_int *pnErrCodeRet);
```

Таблица – Свойство буферов памяти

nFlags	Описание
CL_MEM_READ_WRITE	чтение/запись
CL_MEM_WRITE_ONLY	только запись
CL_MEM_USE_HOST_PTR	использовать pvHostPtr
CL_MEM_ALLOC_HOST_PTR	выделять память управляющего узла
CL_MEM_COPY_HOST_PTR	копировать память управляющего узла

Передача данных

- Хотя среда OpenCL отвечает за обеспечение доступности данных ядром, явные команды передачи памяти могут использоваться для повышения производительности
- OpenCL предоставляет команды для передачи данных на и с устройств
  - clEnqueue {Написать | Читать} {Buffer | Изображение}
  - Запись - копирует с хоста на устройство
  - Чтение - это копирование с устройства на хост
- Существуют также вызовы API OpenCL, чтобы напрямую отображать все или часть объекта памяти указателю на хоста

Буфер данных

```
// копирование буфера
cl_int clEnqueueCopyBuffer(
    cl_command_queue command_queue,
    cl_mem src_buffer,
    cl_mem dst_buffer,
    size_t uSrcOffset,
    size_t uDstOffset,
    size_t uBytes,
    cl_uint uNumEventsInWaitList,
    const cl_event *pEventWaitList,
    cl_event *pEvent);
```

Буфер данных

```
// чтение буфера
cl_int clEnqueueReadBuffer(
    cl_command_queue command_queue,
    cl_mem buffer,
    cl_bool bBlockingRead,
      size_t uOffset,
      size_t uBytes,
      void *pvData,
      cl_uint nNumEventsInWaitList,
      const cl_event *pEventWaitList,
      cl_event *pEvent);
```

```
// запись в буфер
cl_int clEnqueueWriteBuffer(
    cl_command_queue command_queue,
    cl_mem buffer,
    cl_bool bBlockingWrite,
    size_t uOffset,
    size_t uBytes,
    const void *pcvData,
    cl_uint nNumEventsInWaitList,
    const cl_event *pEventWaitList,
    cl_event *pEvent);
```

Параметр bBlockingWrite указывает, что ptr можно повторно использовать после завершения команды

Буфер данных

```
// отображение буфера в память управляющего узла
void *clEnqueueMapBuffer(
    cl_command_queue command_queue,
    cl_mem buffer,
    cl_bool bBlockingMap,
    cl_map_flags nMapFlags,
    size_t uOffset,
    size_t uBytes,
    cl_uint uNumEventsInWaitList,
    const cl_event *pEventWaitList,
    cl_event *pEvent,
    cl_int *pnErrCodeRet);
```

Таблица – Флаг отображения

nMapFlags	Описание
CL_MAP_READ	чтение
CL_MAP_WRITE	запись

Буфер данных

```
// завершение отображения буфера в память
cl_int clEnqueueUnmapMemObject(
      cl_command_queue command_queue,
      cl_mem memobj,
      void *pvMappedPtr,
      cl_uint uNumEventsInWaitList,
      const cl_event *pEventWaitList,
      cl_event *pEvent);
```

Программа

- Программный объект представляет собой набор ядер OpenCL, функции и данные, используемые ядрами (исходный код (текст) или предварительно скомпилированный двоичный файл)
- Создание объекта программы требует либо чтения исходного кода, либо прекомпилированного двоичного кода
- Чтобы скомпилировать программу необходимо:
  - Указать целевые устройства (программа компилируется для каждого устройства)
  - Передать флаги компилятора (необязательно)
  - Проверить ошибки компиляции (необязательно, вывод на экран)

Программа

- Исполняемый код устройства
  - >= 1 ядер
- Создание
  - clCreateProgramWith{Source,Binary}()
- Сборка
  - clBuildProgram()
  - clGetProgramBuildInfo()

Создание объекта программы

```
// создание объекта программы
cl_program clCreateProgramWithSource(
    cl_context context,
    cl_uint uCount,
    const char **ppcszStrings,
    const size_t *puLengths,
    cl_int *pnErrCodeRet);
```

- Эта функция создает программный объект из строк исходного кода
  - count указывает количество строк
  - Пользователь должен создать функцию для чтения в исходном коде строки
- Если строки не имеют NULL-конца, то длина используется для указания длины строк

Сборка программы

```
// сборка программы
cl_int clBuildProgram(
    cl_program program,
    cl_uint uNumDevices,
    const cl_device_id *pcDeviceIDList,
    const char *pcszOptions,
    void (CL_CALLBACK *pfnNotify)(
        cl_program program, void *pvUserData),
    void *pvUserData);
```

- Эта функция компилирует и связывает исполняемый файл из объекта программы для каждого устройства в контексте
  - Если указан список устройств, то только те устройства являются целевыми
- Дополнительная предобработка, оптимизация и другие параметры могут предоставляться

Ядро

- «Точка входа» в устройство
- Создание
  - clCreateKernel(),
  - clCreateKernelsInProgram()
- Параметры
  - clSetKernelArg()
- Запуск
  - clEnqueueNDRangeKernel() – на решётке

Kernels

- Ядро - это функция, объявленная в программе, которая выполняется на устройстве OpenCL
  - Объект ядра - это функция ядра вместе со своими связанными аргументами
- Объект ядра создается из скомпилированной программы
- Пользователь должен явно связывать аргументы (объекты памяти, примитивы и т. Д.) С объектом ядра
- Объекты ядра создаются из объекта программы, указывая имя функции ядра

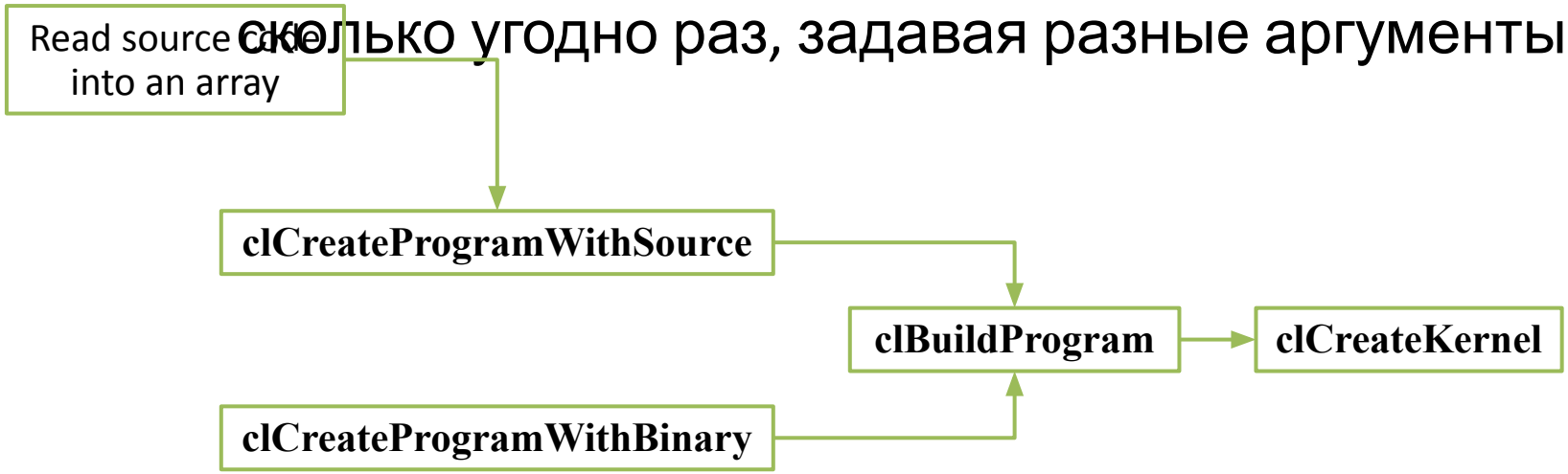
Создание ядра

```
// создание ядра
cl_kernel clCreateKernel(
    cl_program program,
    const char *pcszKernelName,
    cl_int *pnErrCodeRet);
```

- Создает ядро из данной программы
  - Созданное ядро задается строкой, которая соответствует имени функции внутри программы

Runtime Compilation of OpenCL kernels

- Существуют высокие накладные расходы для компиляции программ и создания ядер
  - Каждая операция должна выполняться только один раз (в начале программы)
    - Объекты ядра можно повторно использовать сколько угодно раз, задавая разные аргументы



Reporting Compile Errors

- Если программа не скомпилирована в OpenCL требуется явно запрашивать вывод компилятора
  - Сбой компиляции определяется значением ошибки, возвращаемым командой clBuildProgram
  - Вызов clGetProgramBuildInfo с программным объектом и параметром CL_PROGRAM_BUILD_STATUS возвращает строку с выходом компилятора

Задание аргументов ядра

- Объекты памяти и отдельные значения данных могут быть заданы как аргументы ядра
- Аргументы ядра задаются повторными вызовами clSetKernelArgs

```
// задание аргумента ядра
cl_int clSetKernelArg(
        cl_kernel kernel,
        cl_uint uArgIndex,
        size_t uArgSize,
        const void *pcvArgValue);
```

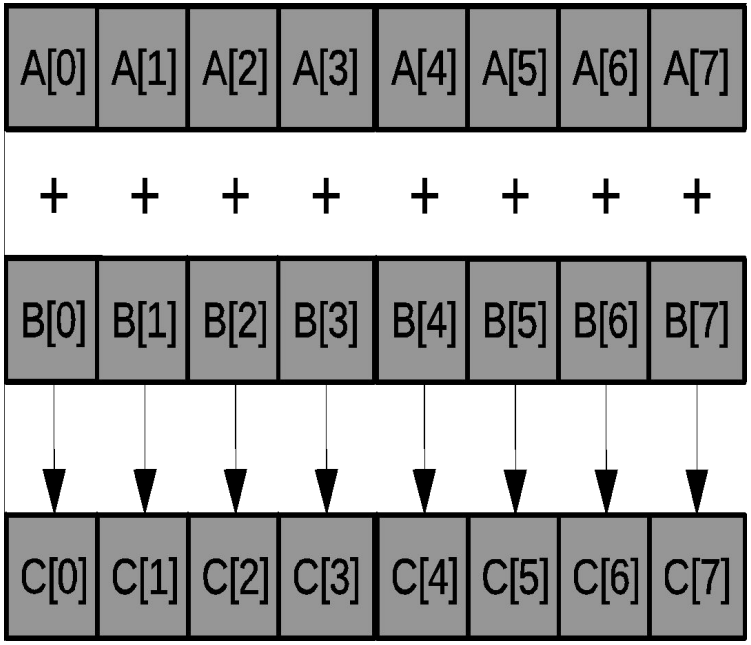
- Каждый вызов должен указывать
  - Индекс аргумента, размер и указатель на данные
- Примеры
  - clSetKernelArg (kernel, 0, sizeof (cl_mem), (void *) & d_iImage);
  - clSetKernelArg (kernel, 1, sizeof (int), (void *) & a);

Модель исполнения

- Массивно параллельные программы обычно пишутся так, что каждый поток вычисляет один элемент задачи
  - Для сложения векторов используются соответствующие элементы двух массивов, где каждый поток выполняет одно сложение

Модель исполнения

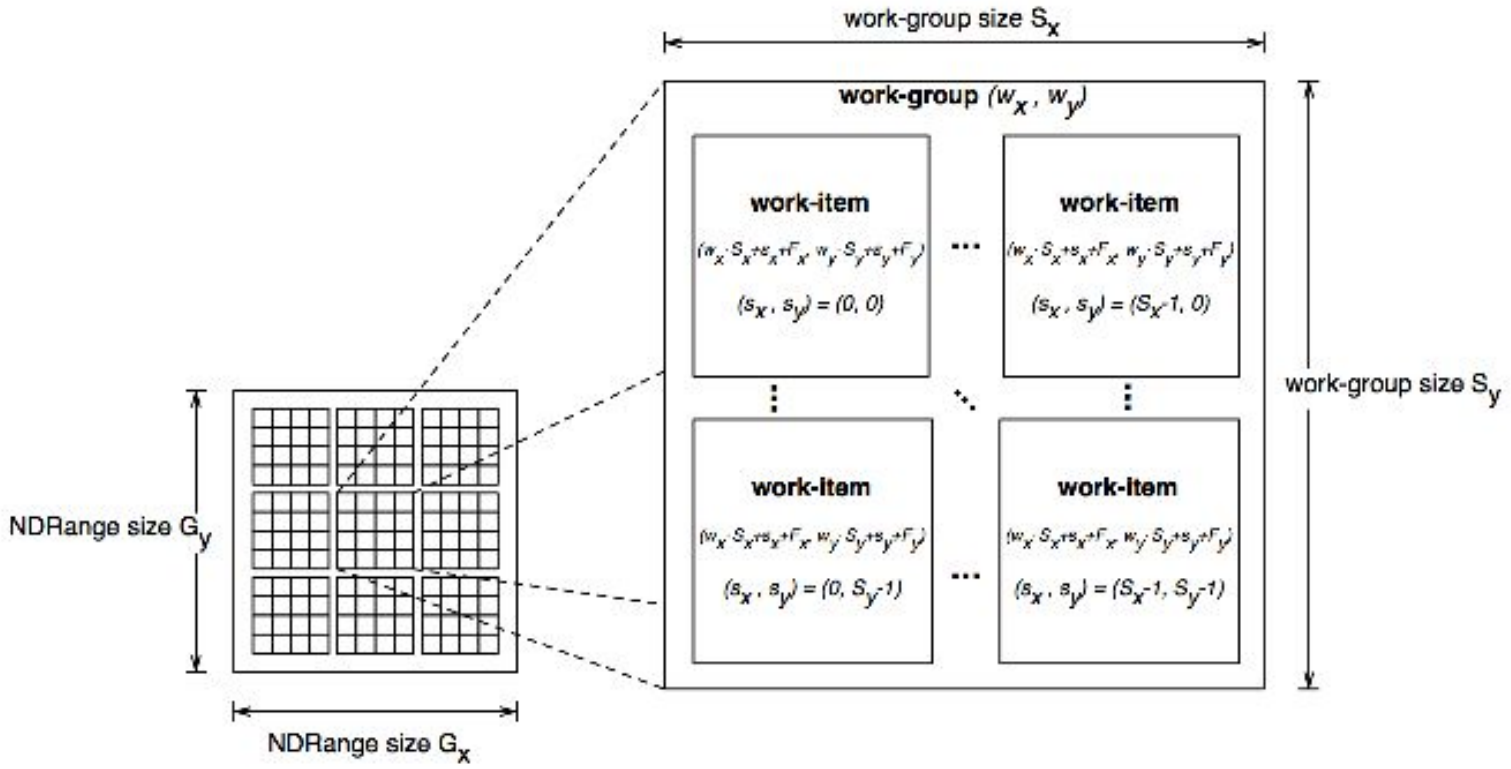
- Рассмотрим простое векторное сложение 8 элементов
  - Требуются 2 входных буфера (A, B) и 1 выходной буфер (C)
  - 1-мерная задача в этом случае
  - Каждый поток отвечает за добавление индексов, соответствующих его идентификатору



Модель исполнения

- Модель исполнения OpenCL предназначена для масштабирования
- Каждый экземпляр ядра называется рабочим элементом (хотя обычно используется «поток», **work-item**)
- **Рабочая группа (work-group)** – набор взаимодействующих рабочих элементов, исполняющихся на одном устройстве. Исполняют одно и тоже ядро, разделяют локальную память и барьеры рабочей группы.
- Пространство индексов определяет иерархию рабочих групп и рабочих элементов

Модель исполнения. Индексное пространство (1/3)



Gx , Gy – глобальные размеры;
Sx, Sy – локальные размеры рабочей группы;
Fx, Fy – глобальное смещение рабочей группы;

gx , gy – глобальный идентификатор;
sx, sy – локальный идентификатор.

Модель исполнения. Индексное пространство (2/3)

$$(g_x, g_y) = (F_x + s_x + w_x S_x, F_y + s_y + w_y S_y)$$

$$(W_x, W_y) = \left(\frac{G_x}{S_x}, \frac{G_y}{S_y} \right)$$

$$(w_x, w_y) = \left(\frac{g_x - s_x - F_x}{S_x}, \frac{g_y - s_y - F_y}{S_y} \right)$$

Модель исполнения. Функции рабочих элементов (3/3)

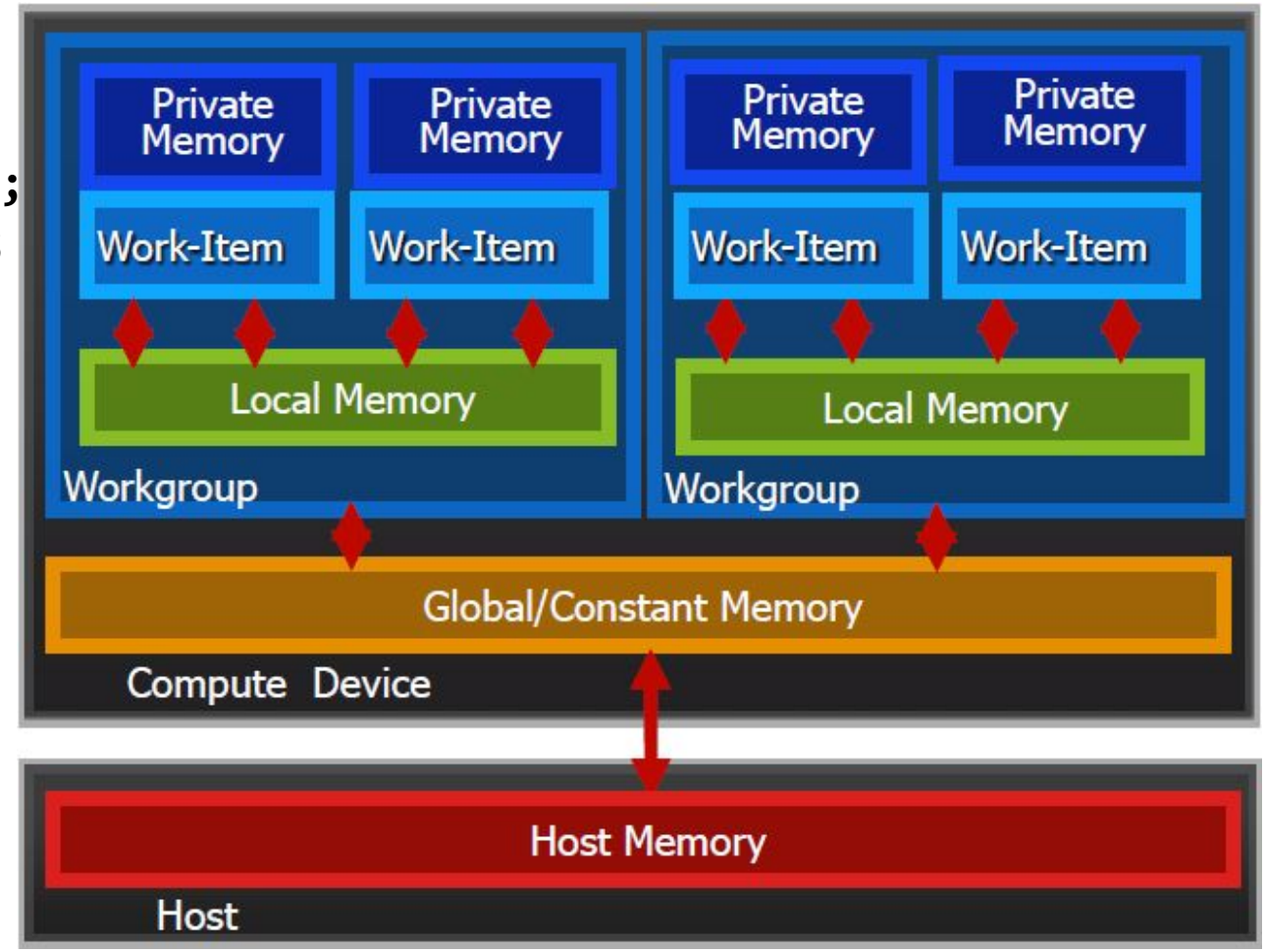
Таблица 11 – Функции рабочих элементов

Функция	Возвращаемое значение
uint get_work_dim();	Размерность пространства
size_t get_global_size(uint uDimIndex);	Глобальный размер
size_t get_global_id(uint uDimIndex);	Глобальный индекс
size_t get_local_size(uint uDimIndex);	Локальный размер
size_t get_local_id(uint uDimIndex);	Локальный индекс
size_t get_num_groups(uint uDimIndex);	Количество групп
size_t get_group_id(uint uDimIndex);	Индекс группы
size_t get_global_offset(uint uDimIndex);	Смещение группы

get_global_size(0) == get_local_size(0) * get_num_groups(0)

Модель платформы OpenCL. Иерархия памяти

- закрытая память;
- локальная память;
- константная память;
- глобальная память;
- хост-память.



Модель платформы OpenCL. Соответствие иерархий

Таблица – Квалификаторы адресного пространства

Память	Квалификатор	Доступ ядра	Доступ управляющего узла
Закрытая	__private	Чтение/запись	—
Локальная	__local	Чтение/запись	—
Глобальная	__global	Чтение/запись	Чтение/запись
Константная	__constant	Чтение	Чтение/запись
Хост-память	—	—	Чтение/запись

Таблица – Квалификаторы доступа

Квалификатор	Значение
__read_only	только для чтения
__write_only	только для записи
__read_write	для чтения/записи

Модель исполнения OpenCL. Квалификаторы

- либо **__xxx, либо xxx**
  - квалификатор **kernel**
    - Функция является ядром
  - квалификатор классов памяти
    - **private, local, constant, global**

Общее адресное пространство

- Одно общее адресное пространство добавляется после OpenCL 2.0
- Поддержка преобразования указателей в и из частных, локальных и глобальных адресных пространств

Написание функции ядра

- Один экземпляр ядра выполняется для каждого рабочего элемента
- Ядра:
  - Необходимо начинать с ключевого слова __kernel
  - Должен иметь тип возврата void
  - Должен объявить адресное пространство каждого аргумента, являющегося объектом памяти
  - Использовать вызовы API (например, get_global_id ()), чтобы определить, какие данные будут работать над рабочим элементом

Написание функций ядра: идентификаторы адресного пространства

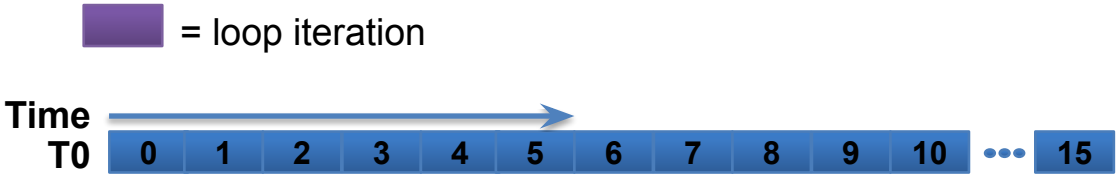
- Внутри ядра объекты памяти задаются с использованием классификаторов типов
  - `__global`: память, выделенная в глобальном адресном пространстве
  - `__constant`: специальный тип памяти только для чтения
  - `__local`: память, совместно используемая рабочей группой
  - `__private`: конфиденциально для каждой рабочей единицы
  - По умолчанию автоматические переменные помещаются в `private` пространство
- Аргументы ядра, являющиеся объектами памяти, должны быть глобальными, локальными или константными

Сложение векторов. Пример

Parallel Software – SPMD

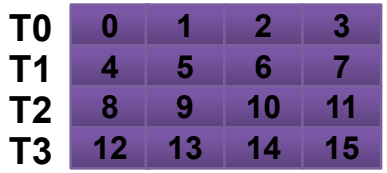
Single-threaded (CPU)

```
// there are N
elements
for(i = 0; i < N; i++)
  C[i] = A[i] + B[i]
```



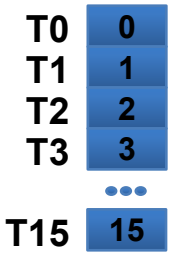
Multi-threaded (CPU)

```
// tid is the thread id
// P is the number of cores
for(i = tid*(N/P); i < (tid+1)*N/P;
i++)
  C[i] = A[i] + B[i]
```



Massively Multi-threaded (GPU)

```
// tid is the thread id
C[tid] = A[tid] +
B[tid]
```



Сложение векторов. Пример

```
__kernel void dp_add(
int nNumElements,
__global const float *pcfA,
__global const float *pcfB,
__global float *pfC)
{
int nID = get_global_id(0);
if (nID >= nNumElements)
return;
//
pfC[nID] = pcfA[nID] + pcfB[nID];
}
```

Последовательность. Сложение векторов

8.1 Создание трех буферов clCreateBuffer(два буфера для входных векторов CL_MEM_READ_ONLY, один – для выходного CL_MEM_WRITE_ONLY);

8.2 Инициализация входных векторов на CPU;

8.3 Запись в буфер входных векторов clEnqueueWriteBuffer;

8.4 Установка буферов в качестве аргумента ядра.

Написание функции ядра: выполнение ядра на устройстве

- Необходимо установить размеры индексного пространства и (необязательно) размеров рабочей группы
- Ядра выполняются асинхронно с хоста
  - clEnqueueNDRangeKernel просто добавляет ядро в очередь, но не гарантирует, что он начнет выполнение
- Структура потока, определяемая созданным индексным пространством
  - Каждый поток выполняет одно и то же ядро на разных данных

Executing Kernels

```
// исполнение ядра
cl_int clEnqueueNDRangeKernel(
    cl_command_queue command_queue,
    cl_kernel kernel,
    cl_uint uWorkDim,
    const size_t *pcuGlobalWorkOffset,
    const size_t *pcuGlobalWorkSize,
    const size_t *pcuLocalWorkSize,
    cl_uint uNumEventsInWaitList,
    const cl_event *pEventWaitList,
    cl_event *pEvent);
```

- Сообщает устройству, связанному с командной очередью, о начале выполнения указанного ядра
- Необходимо указать глобальное (индексное пространство) размер и указать локальные (рабочие группы) размеры
  - В предыдущих выпусках OpenCL глобальный размер должен был быть кратным локальному размеру. OpenCL 2.0 удалил это ограничение.

Освобождение ресурсов

- Объекты OpenCL должны быть освобождены после их использования
- Для большинства типов OpenCL существует команда clRelease {Resource}
  - Пример: clReleaseProgram, clReleaseMemObject

Простейшая программа

```
#include <CL/cl.h>
#include <iostream>
const int g_cuNumItems = 128;
const char *g_pcszSource =
"__kernel void memset(__global int * puDst) \n"
"{ \n"
" puDst[get_global_id(0)] = get_global_id(0); \n"
"} \n";
```

Простейшая программа (продолжение)

```
int main()
{
// 1. Получение платформы
cl_uint uNumPlatforms;
clGetPlatformIDs(0, NULL, &uNumPlatforms);
std::cout << uNumPlatforms << " platforms" << std::endl;
cl_platform_id *pPlatforms = new cl_platform_id[uNumPlatforms];
clGetPlatformIDs(uNumPlatforms, pPlatforms, &uNumPlatforms);

// 2. Получение информации о платформе
const size_t size = 128;
charparam_value[size] = {0};
size_t param_value_size_ret = 0;
for (int i = 0; i < uNumPlatforms; ++i)
{
cl_int res = clGetPlatformInfo(pPlatforms[i], CL_PLATFORM_NAME, size,
static_cast<void *>(param_value), &param_value_size_ret);
printf("Platform %i name is %s\n", pPlatforms[i], param_value);
param_value_size_ret = 0;
}
```

Простейшая программа (продолжение)

```
// 3. Получение номера CL устройства
cl_device_id deviceID;
cl_uint uNumGPU;
clGetDeviceIDs( pPlatforms[1], CL_DEVICE_TYPE_DEFAULT, 1, &deviceID,
&uNumGPU);

// 4. Получение информации о CL устройстве
param_value_size_ret = 0;
cl_int res1 = clGetDeviceInfo(deviceID, CL_DEVICE_NAME, size,
static_cast<void *>(param_value), &param_value_size_ret);
printf("Device %i name is %s\n", deviceID, param_value);

// 5. Создание контекста
cl_int errcode_ret;
cl_context context = clCreateContext(NULL, 1, &deviceID, NULL, NULL,
&errcode_ret);
```

Простейшая программа (продолжение)

```
// 6. Создание очереди команд
errcode_ret = 0;
cl_queue_properties qprop[] = {0 };
cl_command_queue queue = clCreateCommandQueueWithProperties(context,
deviceID, qprop, &errcode_ret);

// 7. Создание программы
errcode_ret = CL_SUCCESS;
size_t source_size = strlen(g_pcszSource);
cl_program program = clCreateProgramWithSource(context, 1,
&g_pcszSource, (const size_t *)&source_size, &errcode_ret);
```

Простейшая программа (продолжение)

```
//
// 8. Сборка программы
//
cl_int errcode = clBuildProgram(
program, 1, &deviceID, NULL, NULL, NULL);
//
// 9. Получение ядра
//
cl_kernel kernel = clCreateKernel(program, "memset", NULL);
```

Простейшая программа (продолжение)

```
//
// 10. Создание буфера
//
cl_mem buffer = clCreateBuffer(
context, CL_MEM_WRITE_ONLY,
g_cuNumItems * sizeof (cl_uint), NULL, NULL);
//
// 11. Установка буфера в качестве аргумента ядра
//
clSetKernelArg(kernel, 0, sizeof (buffer), (void *) &buffer);
```

Простейшая программа (продолжение)

```
//
// 12. Запуск ядра
//
size_t uGlobalWorkSize = g_cuNumItems;
clEnqueueNDRangeKernel(
queue, kernel, 1, NULL, &uGlobalWorkSize,
NULL, 0, NULL, NULL);
clFinish(queue);
```

Простейшая программа (продолжение)

```
//
// 13. Отображение буфера в память управляющего узла
//
cl_uint *puData = (cl_uint *) clEnqueueMapBuffer(
queue, buffer, CL_TRUE, CL_MAP_READ, 0,
g_cuNumItems * sizeof (cl_uint), 0, NULL, NULL, NULL);
```

Простейшая программа (продолжение)

```
//
// 14. Использование результатов
//
for (int i = 0; i < g_cuNumItems; ++ i)
std::cout << i << " □ " << puData[i] << "; ";
std::cout << std::endl;
//
// 15. Завершение отображения буфера
//
clEnqueueUnmapMemObject(
queue, buffer, puData, 0, NULL, NULL);
```

Простейшая программа (продолжение)

```
//
// 16. Удаление объектов и освобождение памяти
// управляющего узла
//
clReleaseMemObject(buffer);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);
delete [] pPlatforms;
//
} // main()
```

Последовательность

1. Получение платформы
2. Получение номера CL устройства
3. Создание контекста
4. Создание очереди команд
5. Создание программы
6. Сборка программы
7. Получение ядра
8. **Передача параметров в функцию ядра** (создание буферов копирование данных, установка буферов в качестве аргумента ядра)
9. Запуск ядра
10. Отображение буфера(ов) с результатами в память управляющего узла
11. Использование результатов
12. Завершение отображения
13. Удаление объектов и освобождение памяти управляющего узла

Программная модель OpenCL

- Параллелизм на уровне данных
  - сопоставление между рабочими элементами и элементами в объекте памяти
  - рабочие группы могут быть определены явно или неявно
- Параллелизм на уровне задач
  - Ядро выполняется независимо от индексного пространства
  - выполнение нескольких задач, использование векторных типов устройств и т. д.

Скалярные типы OpenCL

Тип C	Тип OpenCL
bool	-
char	cl_char
unsigned char, uchar	cl_uchar
short	cl_short
unsigned short, ushort	cl_ushort
int	cl_int
unsigned int, uint	cl_uint
long	cl_long
unsigned long, ulong	cl_ulong
float	cl_float
half	cl_half
size_t, ptrdiff_t	-
intptr_t, uintptr_t	-
void	**void**

Векторные типы OpenCL

Тип C	Тип OpenCL
char*n*	cl_char*n*
uchar*n*	cl_uchar*n*
short*n*	cl_short*n*
ushort*n*	cl_ushort*n*
int*n*	cl_int*n*
uint*n*	cl_uint*n*
long*n*	cl_long*n*
ulong*n*	cl_ulong*n*
float*n*	cl_float*n*

Значение *n*: 2, 3, 4, 8, 16

Транспонирование матрицы. Пример

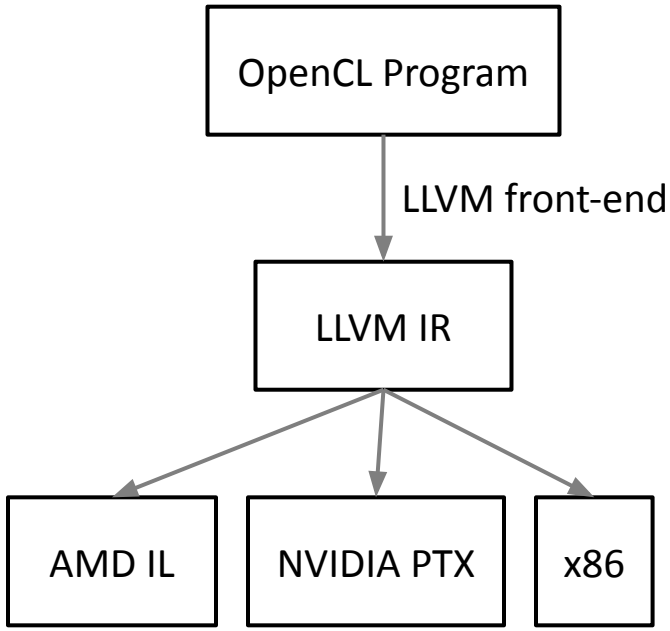
```
__kernel void transpose(
__global float *pfOData,
__global float *pfIData,
int nOffset, int nWidth, int nHeight,
__local float *pfBlock)
{
// Чтение из общей памяти
//
unsigned int uXIndex = get_global_id(0);
unsigned int uYIndex = get_global_id(1);
```

Транспонирование матрицы. Пример

```
if ((uXIndex + nOffset < nWidth) && (uYIndex < nHeight))
{
unsigned int uIndexIn = uYIndex * uWidth + uXIndex + nOffset;
pfBlock[get_local_id(1) * (BLOCK_DIM + 1) + get_local_id(0)] =
pfIData[uIndexIn];
}
//
barrier(CLK_LOCAL_MEM_FENCE);
```

Система компиляции OpenCL

- Ядро OpenCL обычно преобразуется в двоичное представление промежуточного языка
- Обычными промежуточными представлениями являются LLVM (виртуальная машина низкого уровня) IR или Khronos SPIR
- LLVM - это компилятор с открытым исходным кодом



More information at http://llvm.org

Ссылки

- http://www.khronos.org/opencl/
- http://developer.nvidia.com/object/opencl.html
-http://developer.amd.com/gpu/atistreamsdk/pages/default.aspx
- http://www.alphaworks.ibm.com/tech/opencl

Выводы

- OpenCL обеспечивает интерфейс для взаимодействия хостов с устройствами ускорителя
- Создается контекст, который содержит всю информацию и данные, необходимые для выполнения программы OpenCL
  - Создаются объекты памяти, которые можно перемещать и выключать
  - Командные очереди позволяют хосту запрашивать операции, выполняемые устройством
  - Программы и ядра содержат код, который необходимо выполнить устройствам