Лекция 5. Введение в технологию OpenCL

Параллельное программирование

17 декабря 2010 г.

Назначение OpenCL

Определение

OpenCL (*Open Computing Language*) — открытый стандарт параллельного программирования для гетерогенных платформ, включающих центральные, графические процессоры и другие дискретные вычислительные устройства.

Компоненты

- библиотечные функции (API) — для управления параллельными вычислениями на устройствах со стороны центрального процессора;
- язык программирования — для реализации вычислений;
- система времени выполнения для поддержки разработки.

Краткая история OpenCL

Год	Событие
18 ноября 2008 г.	OpenCL 1.0
14 июня 2010 г.	OpenCL 1.1

Таблица 1: Основные этапы развития OpenCL

Модель платформы OpenCL

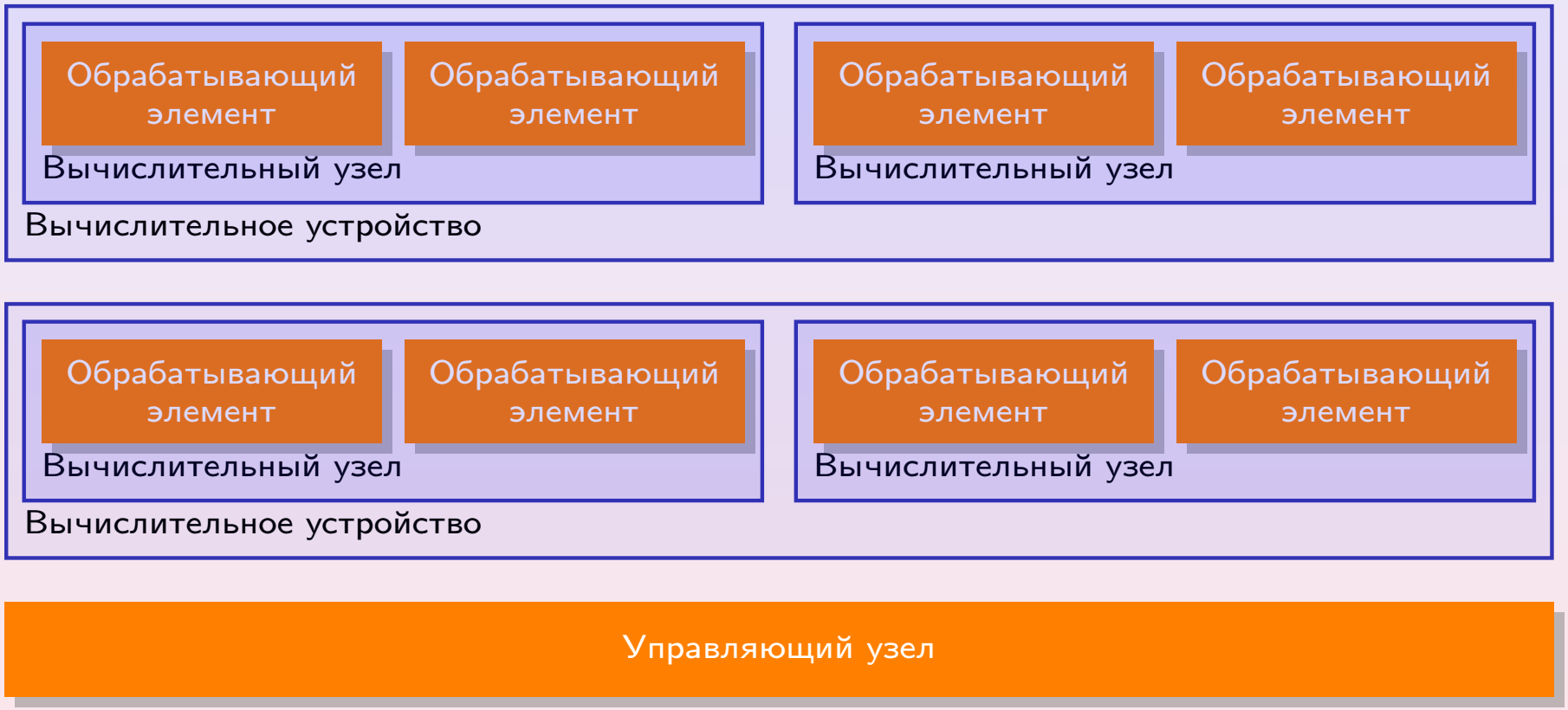

Рис. 1: Модель платформы OpenCL

Иерархия памяти OpenCL

Рис. 2: Архитектура памяти OpenCL

Программная модель

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, манипуляции с памятью и т. д.)

Программная модель

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, манипуляции с памятью и т. д.)

Программная модель

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, манипуляции с памятью и т. д.)

Программная модель

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, манипуляции с памятью и т. д.)

Объекты

Определения

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

Определения

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

Определения

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

Определения

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 (окончание)

Определения

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 вызовов API.

Объект памяти (*memory object*) — ссылается на область глобальной памяти.

Объекты (окончание)

Определения

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 вызовов API.

Объект памяти (*memory object*) — ссылается на область глобальной памяти.

Объекты (окончание)

Определения

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 вызовов API.

Объект памяти (*memory object*) — ссылается на область глобальной памяти.

Устройства

Определения

Рабочий элемент (*work-item*) — набор параллельно исполняемых ядер на устройстве, вызванных при помощи команды.

Рабочая группа (*work-group*) — набор взаимодействующих рабочих элементов, исполняющихся на одном устройстве. Исполняют одно и то же ядро, разделяют локальную память и барьеры рабочей группы.

Обрабатывающий элемент (*processing element*) — виртуальный скалярный процессор. Рабочий элемент может выполняться на одном или нескольких обрабатывающих элементах.

Вычислительный узел (*compute unit*) — исполняет одну рабочую группу. Устройство может состоять из одного или нескольких вычислительных элементов.

Устройства

Определения

Рабочий элемент (*work-item*) — набор параллельно исполняемых ядер на устройстве, вызванных при помощи команды.

Рабочая группа (*work-group*) — набор взаимодействующих рабочих элементов, исполняющихся на одном устройстве. Исполняют одно и то же ядро, разделяют локальную память и барьеры рабочей группы.

Обрабатывающий элемент (*processing element*) — виртуальный скалярный процессор. Рабочий элемент может выполняться на одном или нескольких обрабатывающих элементах.

Вычислительный узел (*compute unit*) — исполняет одну рабочую группу. Устройство может состоять из одного или нескольких вычислительных элементов.

Устройства

Определения

Рабочий элемент (*work-item*) — набор параллельно исполняемых ядер на устройстве, вызванных при помощи команды.

Рабочая группа (*work-group*) — набор взаимодействующих рабочих элементов, исполняющихся на одном устройстве. Исполняют одно и то же ядро, разделяют локальную память и барьеры рабочей группы.

Обрабатывающий элемент (*processing element*) — виртуальный скалярный процессор. Рабочий элемент может выполняться на одном или нескольких обрабатывающих элементах.

Вычислительный узел (*compute unit*) — исполняет одну рабочую группу. Устройство может состоять из одного или нескольких вычислительных элементов.

Устройства

Определения

Рабочий элемент (*work-item*) — набор параллельно исполняемых ядер на устройстве, вызванных при помощи команды.

Рабочая группа (*work-group*) — набор взаимодействующих рабочих элементов, исполняющихся на одном устройстве. Исполняют одно и то же ядро, разделяют локальную память и барьеры рабочей группы.

Обрабатывающий элемент (*processing element*) — виртуальный скалярный процессор. Рабочий элемент может выполняться на одном или нескольких обрабатывающих элементах.

Вычислительный узел (*compute unit*) — исполняет одну рабочую группу. Устройство может состоять из одного или нескольких вычислительных элементов.

Контекст

Определения

Контекст (*context*) — среда, в которой выполняются ядра, а также область определения синхронизации и управления памятью. Включает:

- набор устройств;
- память, доступную устройствам;
- свойства памяти;
- одну или несколько очередей команд.

Объект программы (*program object*) — включает:

- ссылку на связанный контекст;
- исходный текст или двоичное представление;
- последний удачно собранный код, список устройств, для которых он собран, настройки и журнал сборки;
- набор текущих связанных ядер.

Контекст

Определения

Контекст (*context*) — среда, в которой выполняются ядра, а также область определения синхронизации и управления памятью. Включает:

- набор устройств;
- память, доступную устройствам;
- свойства памяти;
- одну или несколько очередей команд.

Объект программы (*program object*) — включает:

- ссылку на связанный контекст;
- исходный текст или двоичное представление;
- последний удачно собранный код, список устройств, для которых он собран, настройки и журнал сборки;
- набор текущих связанных ядер.

Индексное пространство рабочих элементов

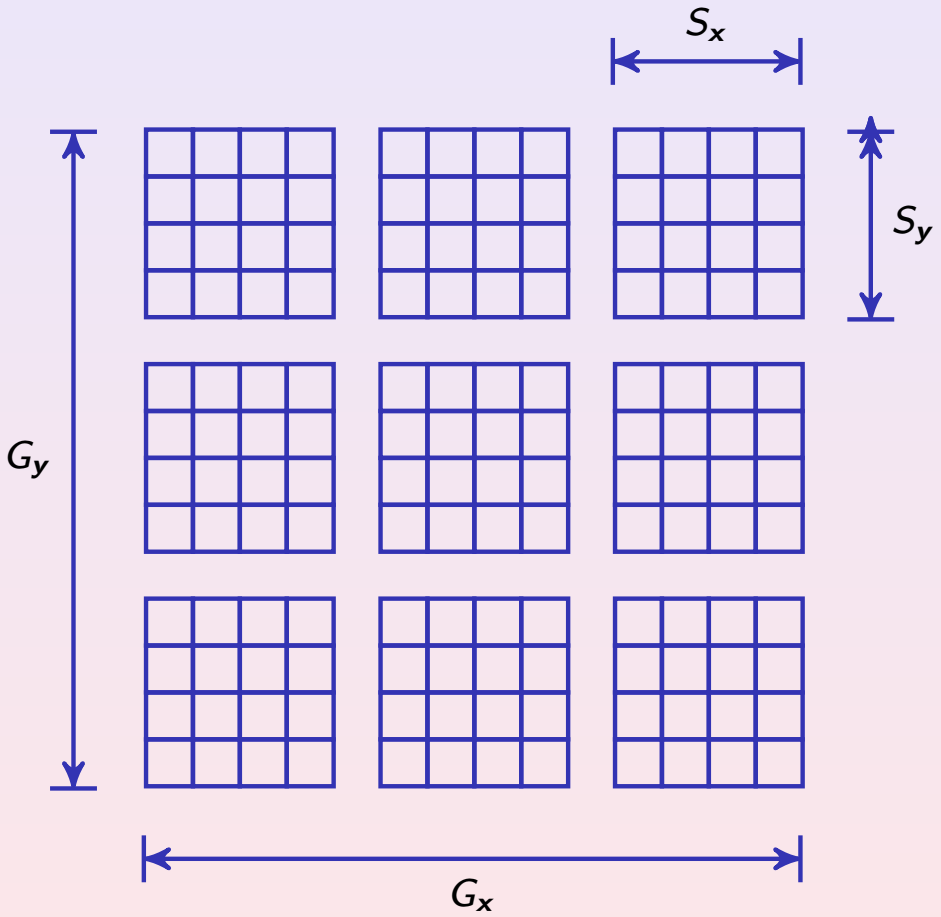

Рис. 3: Индексное пространство

Обозначения

(G_x, G_y): глобальные размеры;

(S_x, S_y): локальные размеры рабочей группы;

(F_x, F_y): глобальные смещения рабочей группы;

(g_x, g_y): глобальный идентификатор;

(s_x, s_y): локальный идентификатор;

Индексное пространство рабочих элементов (окончание)

Соотношения

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

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

$$(w_x, w_y) =$$

Индексное пространство рабочих элементов (окончание)

Соотношения

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

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

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

Информационные зависимости

Допустимые виды информационных зависимостей

MIMD — допускаются входные зависимости (возможно, нужна рассылка данных).

SIMD — допускаются зависимости от вхождений, выполняемых раньше ко вхождениям, выполняемым позже:

- прямые зависимости;
- антизависимости;
- выходные зависимости.

Информационные зависимости

Допустимые виды информационных зависимостей

MIMD — допускаются входные зависимости (возможно, нужна рассылка данных).

SIMD — допускаются зависимости от вхождений, выполняемых раньше ко вхождениям, выполняемым позже:

- прямые зависимости;
- антизависимости;
- выходные зависимости.

Возможности разных видов памяти

	Управляющий узел	Ядро
Глобальная	динамическое чтение/запись	— чтение/запись
Константная	динамическое чтение/запись	статическое чтение/—
Локальная	динамическое —/—	статическое чтение/запись
Закрытая	— —/—	статическое чтение/запись

Таблица 2: Вид размещения, доступ на чтение/запись для памяти

Система времени выполнения

Состав системы времени выполнения (Framework)

Слой платформы (*Platform layer*) — позволяет управляющему узлу:

- обнаруживать устройства OpenCL;
- определять их возможности;
- создавать контексты.

Среда выполнения (*Runtime*) — позволяет управляющему узлу управлять созданными контекстами.

Компилятор (*Compiler*) — создаёт исполняемые программы с ядрами.

Система времени выполнения

Состав системы времени выполнения (Framework)

Слой платформы (*Platform layer*) — позволяет управляющему узлу:

- обнаруживать устройства OpenCL;
- определять их возможности;
- создавать контексты.

Среда выполнения (*Runtime*) — позволяет управляющему узлу управлять созданными контекстами.

Компилятор (*Compiler*) — создаёт исполняемые программы с ядрами.

Система времени выполнения

Состав системы времени выполнения (Framework)

Слой платформы (*Platform layer*) — позволяет управляющему узлу:

- обнаруживать устройства OpenCL;
- определять их возможности;
- создавать контексты.

Среда выполнения (*Runtime*) — позволяет управляющему узлу управлять созданными контекстами.

Компилятор (*Compiler*) — создаёт исполняемые программы с ядрами.

Обнаружение платформ

Функция

```
cl_int clGetPlatformIDs(
  cl_uint nNumEntries, cl_platform_id *pPlatforms,
  cl_uint *pnNumPlatforms);
```

Параметры

`nNumEntries`	—	размер массива, на который указывает `pPlatforms` (0, если `pPlatforms == NULL`);
`pPlatforms`	—	массив для возврата информации об устройствах (`NULL` $\Rightarrow$ не возвращать);
`pnNumPlatforms`	—	возвращаемое количество устройств OpenCL, (`NULL` $\Rightarrow$ не возвращать);

Обнаружение устройств на платформе

Функция

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

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 молчанию в системе
CL_DEVICE_TYPE_ALL	все доступные устройства OpenCL

Таблица 3: Категории устройств OpenCL

Создание контекста

Функция

```
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);
```

Создание очереди команд

Функция

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

nProperties	Описание
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE	исполнение не по порядку;
CL_QUEUE_PROFILING_ENABLE	включение профилирования.

Таблица 4: Свойства очередей команд

Создание буфера

Функция

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

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.

Таблица 5: Свойства буферов памяти

Чтение/запись в буфер

Функция

```
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);
```

Функция

```
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);
```

Чтение 2-х и 3-х мерного буфера

Функция

```
cl_int clEnqueueReadBufferRect(
  cl_command_queue command_queue, cl_mem buffer, cl_bool bBlockingRead,
  const size_t auBufferOrigin[3],
  const size_t auHostOrigin[3],
  const size_t auRegion[3],
  size_t uBufferRowPitch,
  size_t uBufferSlicePitch,
  size_t uHostRowPitch,
  size_t uHostSlicePitch,
  void *pvData,
  cl_uint uNumEventsInWaitList, const cl_event *pnEventWaitList,
  cl_event *pEvent);
```

Чтение 2-х и 3-х мерного буфера (окончание)

Вычисление смещения с начала буфера

```
auBufferOrigin[2] * uBufferSlicePitch +
auBufferOrigin[1] * uBufferRowPitch +
auBufferOrigin[0]
```

Копирование буфера

Функция

```
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);
```

Отображение буфера в память управляющего узла

Функция

```
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);
```

nMapFlags	Описание
CL_MAP_READ	чтение;
CL_MAP_WRITE	запись.

Таблица 6: Флаг отображения

Завершение отображения буфера

Функция

```
cl_int clEnqueueUnmapMemObject(
  cl_command_queue command_queue,
  cl_mem memobj,
  void *pvMappedPtr,
  cl_uint uNumEventsInWaitList,
  const cl_event *pEventWaitList,
  cl_event *pEvent);
```

Создание объекта программы

Функция

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

Сборка программы

Функция

```
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);
```

Создание ядра

Функция

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

Задание аргументов ядра

Функция

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

Исполнение ядра

Функция

```
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);
```

Исполнение ядра на одном рабочем элменте

Функция

```
cl_int clEnqueueTask(
  cl_command_queue command_queue,
  cl_kernel kernel,
  cl_uint uNumEventsInWaitList,
  const cl_event *pEventWaitList,
  cl_event *pEvent);
```

Заполнение вектора

Пример

```
#include <CL/cl.h>

#include <iostream>

const int g_cuNumItems = 128;

const char *g_pcszSource =
  "__kernel void memset(__global uint *puDst)     \n"
  "{                                              \n"
  "  puDst[get_global_id(0)] = get_global_id(0);  \n"
  "}                                              \n";
```

Заполнение вектора (продолжение)

Пример (продолжение)

```
int main()
{
  //
  //   1. Получение платформ
  //
  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);
```

Заполнение вектора (продолжение)

Пример (продолжение)

```
//
//  2. Получение номера видеокарты
//
cl_device_id deviceID;
cl_uint uNumGPU;
clGetDeviceIDs(
  pPlatforms[0], CL_DEVICE_TYPE_GPU, 1, &deviceID, &uNumGPU);
//
//  3. Создание контекста
//
cl_context context = clCreateContext(
  NULL, 1, &deviceID, NULL, NULL, NULL);
```

Заполнение вектора (продолжение)

Пример (продолжение)

```
//
//  4. Создание очереди команд
//
cl_command_queue queue = clCreateCommandQueue(
  context, deviceID, 0, NULL);
//
//  5. Создание программы
//
cl_program program = clCreateProgramWithSource(
  context, 1, &g_pcszSource, NULL, NULL);
```

Заполнение вектора (продолжение)

Пример (продолжение)

```
//
//  6. Сборка программы
//
cl_int errcode = clBuildProgram(
  program, 1, &deviceID, NULL, NULL, NULL);
//
//  7. Получение ядра
//
cl_kernel kernel = clCreateKernel(program, "memset", NULL);
```

Заполнение вектора (продолжение)

Пример (продолжение)

```
//
//  8. Создание буфера
//
cl_mem buffer = clCreateBuffer(
  context, CL_MEM_WRITE_ONLY,
  g_cuNumItems * sizeof (cl_uint), NULL, NULL);
//
//  9. Установка буфера в качестве аргумента ядра
//
clSetKernelArg(kernel, 0, sizeof (buffer), (void *) &buffer);
```

Заполнение вектора (продолжение)

Пример (продолжение)

```
//
// 10. Запуск ядра
//
size_t uGlobalWorkSize = g_cuNumItems;
clEnqueueNDRangeKernel(
  queue, kernel, 1, NULL, &uGlobalWorkSize,
  NULL, 0, NULL, NULL);
clFinish(queue);
```

Заполнение вектора (продолжение)

Пример (продолжение)

```
//
// 11. 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);
```

Заполнение вектора (продолжение)

Пример (продолжение)

```
//
// 12. Использование результатов
//
for (int i = 0; i < g_cuNumItems; ++ i)
  std::cout << i << " – " << puData[i] << "; ";
std::cout << std::endl;
//
// 13. Завершение отображения буфера
//
clEnqueueUnmapMemObject(
  queue, buffer, puData, 0, NULL, NULL);
```

Заполнение вектора (окончание)

Пример (окончание)

```
  //
  // 14. Удаление объектов и освобождение памяти
  //   управляющего узла
  //
  clReleaseMemObject(buffer);
  clReleaseKernel(kernel);
  clReleaseProgram(program);
  clReleaseCommandQueue(queue);
  clReleaseContext(context);
  delete [] pPlatforms;
  //
}    // main()
```

Скалярные типы

Тип OpenCL	Тип C
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**

Таблица 7: Скалярные типы OpenCL

Векторные типы

Тип OpenCL	Тип C
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*

Таблица 8: Векторные типы OpenCL, значения n: 2, 3, 4, 8, 16

Квалификаторы

Квалификатор	Значение
__global	глобальная память;
__local	локальная память;
__constant	константная память;
__private	закрытая память.

Таблица 9: Квалификаторы адресного пространства

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

Таблица 10: Квалификаторы доступа

Объявление ядра

Пример (быстрое преобразование Фурье)

```
__kernel void clFFT_1DTwistSplit(
  __global float *pfInReal,
  __global float *pfInImag,
  unsigned int nStartRow,
  unsigned int nNumCols,
  unsigned int nN,
  unsigned int nNumRowsToProcess,
  int nDirection)
{
  // ...
}
```

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.

Таблица 11: Функции рабочих элементов

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];
}
```

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 (продолжение)

Пример (продолжение)

```
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);
```

Транспонирование матрицы (окончание)

Пример (окончание)

```
  //
  // Запись в глобальную память
  //
  uXIndex = get_group_id(1) * BLOCK_DIM + get_local_id(0);
  uYIndex = get_group_id(0) * BLOCK_DIM + get_local_id(1);
  if ((uXIndex < nHeight) && (uYIndex + nOffset < nWidth))
  {
    unsigned int uIndexOut = uYIndex * nHeight + uXIndex;
    pfOData[uIndexOut] =
      pfBlock[get_local_id(0) * (BLOCK_DIM + 1) + get_local_id(1)];
  }
}
```