Министерство науки и образования РФ
РГРТУ

Методические указания
по лабораторной работе №1
«Знакомство с OPENCL»

Рязань, 2017г.

Цель работы:

Получение практических навыков программирования на OpenCL.

Теоретическая часть

Гетерогенные вычислительные системы — электронные системы, использующие различные типы вычислительных блоков. Вычислительными блоками такой системы могут быть процессор общего назначения (GPP), процессор специального назначения (например, цифровой сигнальный процессор (DSP) или графический процессор (GPU)), сопроцессор, логика ускорения (специализированная интегральная схема (ASIC) или программируемая пользователем вентильная матрица (FPGA)). В общем, гетерогенная вычислительная платформа содержит процессоры с разными наборами команд (ISA). Спрос на повышение гетерогенности в вычислительных системах, частично связан с необходимостью в высокопроизводительных, высокореакционных системах, которые взаимодействуют с другим окружением (аудио/видео системы, системы управления, сетевые приложения и т.д.). Основным методом получения дополнительной производительности вычислительных систем является введение дополнительных специализированных ресурсов, в результате чего вычислительная система становится гетерогенной. Это позволяет разработчику использовать несколько типов вычислительных элементов, каждый из которых способен выполнять задачи, которые лучше всего для него подходят. Добавление дополнительных, независимых вычислительных ресурсов неизбежно приводит к тому, что большинство гетерогенных систем рассматриваются как параллельные вычислительные системы или многоядерные системы. Ещё один термин, который иногда используется для этого типа вычислений «гибридные вычисления». Hybrid-core computing — форма гетерогенных вычислений, в которой асимметричные вычислительные устройства сосуществуют в одном процессоре.

Современные GPU как потенциальный элемент гетерогенных вычислительных систем благодаря специализированной конвейерной архитектуре они намного эффективнее в обработке графической информации, чем центральный процессор (CPU).

GPU может использоваться как в составе дискретной видеокарты, так и в интегрированных решениях (встроенных в северный мост либо в гибридный процессор). Таким образом, возможно совмещение на одной плате нескольких GPU, имеющих доступ к одному и тому же адресному пространству оперативной памяти.

Оперативная память, используемая GPU, может быть разделяемой с CPU или отдельной. Как правило, на дискретных видеоадаптерах присутствует собственная оперативная память, используемая только GPU, тогда как встроенные GPU используют ту же оперативную память, что и CPU.

Отличительными особенностями GPU по сравнению с CPU являются:

- архитектура, максимально нацеленная на увеличение скорости расчёта текстур и сложных графических объектов;
- ограниченный набор команд.

Высокая вычислительная мощность GPU объясняется особенностями архитектуры. Если современные CPU содержат несколько ядер (на большинстве современных систем от 2 до 16, по состоянию на 2013 г.), графический процессор изначально создавался как многоядерная структура, в которой количество ядер может достигать сотен. Разница в архитектуре обусловливает и разницу в принципах работы. Если архитектура CPU предполагает последовательную обработку информации, то GPU исторически предназначался для обработки компьютерной графики.

Каждая из этих двух архитектур имеет свои достоинства. CPU лучше работает с последовательными задачами. При обработке больших объемов информации алгоритмами, допускающими эффективное с временной точки зрения распараллеливание вычислений, очевидное преимущество имеет GPU.

GPGPU (англ. general-purpose graphics processing units — «GPU общего назначения») — техника использования графического процессора видеоадаптера, предполагающая выполнение на GPU расчетов в приложениях для общих вычислений, которые обычно проводит центральный процессор. Это стало возможным благодаря добавлению программируемых шейдерных блоков и более высокой арифметической точности растровых конвейеров, что позволяет разработчикам ПО использовать потоковые процессоры для неграфических данных.

Принципиальная схема выполнения общих вычислений на GPU приведена на рисунке 1.

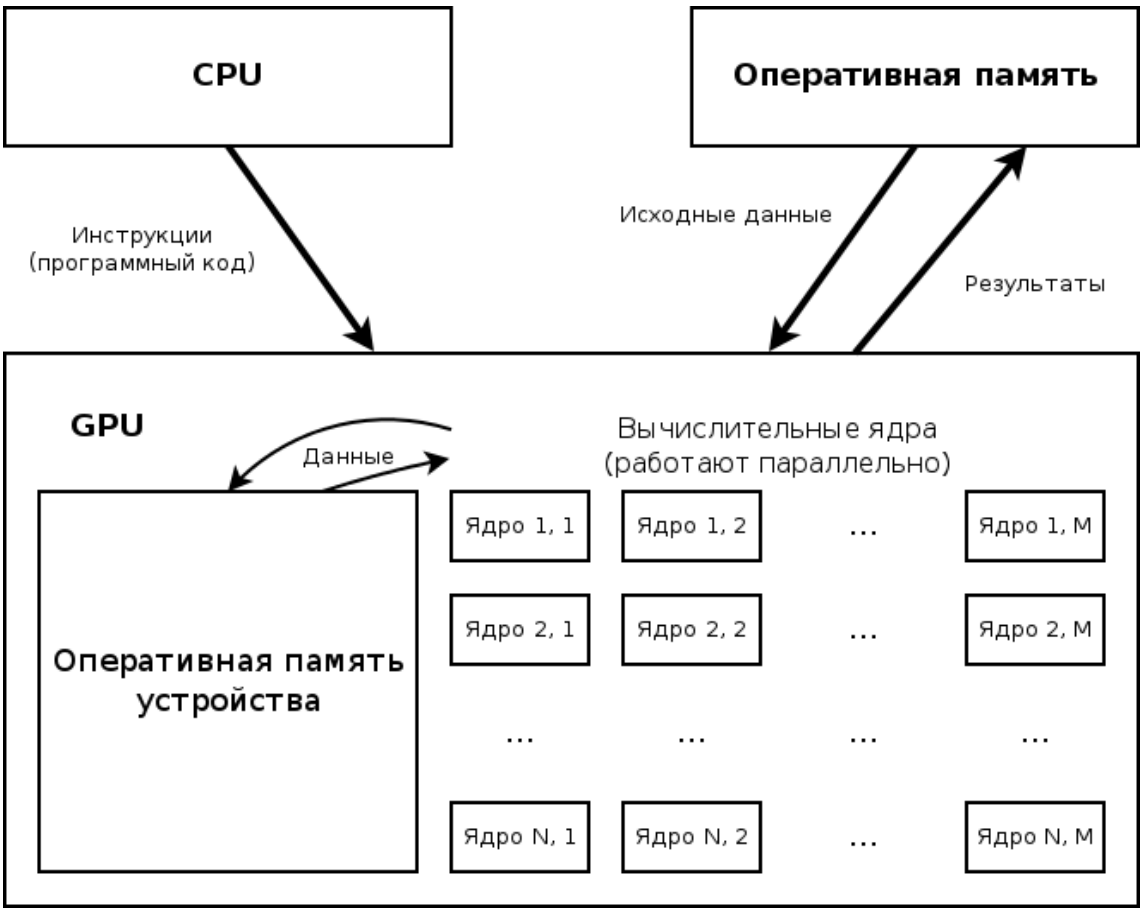

Рисунок 1 — Принципиальная схема выполнения общих вычислений на GPU

Существуют следующие программные средства программирования GPGPU:

- OpenCL - является открытым стандартом программирования задач на языках программирования C и C++, связанных с параллельными вычислениями на различных графических и центральных процессорах; стандарт поддерживается некоммерческим консорциумом Khronos Group;
- CUDA — технология GPGPU, позволяющая программистам реализовывать на языке программирования Си (а также C++/C#) алгоритмы, выполнимые на GPU, производимые компанией NVIDIA (серия GPU GeForce восьмого поколения и старше);
- AMD FireStream — технология GPGPU, позволяющая программистам реализовывать алгоритмы, выполнимые на графических процессорах ускорителей ATI;
- DirectCompute — вычислительный шейдер;
- OpenACC - стандарт, описывающий набор директив для написания гетерогенных программ, задействующих как центральный, так и графический процессор; используется для распараллеливания программ на языках C, C++ и Fortran; OpenACC был создан группой компаний, в которую вошли CAPS, Cray, NVIDIA и PGI;
- C++ AMP.

ТехнологияOpenCL

OpenCL (от англ. Open Computing Language — открытый язык вычислений) — фреймворк для написания компьютерных программ, связанных с параллельными вычислениями на различных графических и центральных процессорах.

В фреймворк OpenCL входят язык программирования, который базируется на стандарте C99, и интерфейс программирования приложений (англ. API). OpenCL обеспечивает параллелизм на уровне инструкций и на уровне данных и является реализацией техники GPGPU.

Цель OpenCL состоит в том, чтобы дополнить OpenGL и OpenAL, которые являются открытыми отраслевыми стандартами для трёхмерной компьютерной графики и звука, пользуясь возможностями GPU.OpenCL разрабатывается и поддерживается некоммерческим консорциумом KhronosGroup, в который входят много крупных компаний, включая Apple, AMD, Intel, NVIDIA, ARM, Sony Computer Entertainment и другие.

Ключевыми отличиями используемого языка от C99 являются:

- отсутствие поддержки указателей на функции, рекурсии, битовых полей, массивов переменной длины, стандартных заголовочных файлов;
- отсутствие поддержки динамического выделения памяти: функций malloc(), calloc(), realloc(), free();
- расширения языка для параллелизма: векторные типы, синхронизация, функции для work - items / work - groups;

- квалификаторы типов памяти: __global, __local, __constant, __private;
- иной набор встроенных функций.

Перечисленные отличия (особенно — отсутствие поддержки рекурсии и отсутствие поддержки динамического выделения памяти) делают процесс разработки программ более трудоемким, однако разработанная программа, потенциально, содержит меньшее количество логических ошибок.

Все вычислительные потоки, выполняющиеся на GPU в любой момент времени, разделяются OpenCL по рабочим группам (англ. work — group). Вычислительный поток (англ. work - item), таким образом, является членом одной и только одной рабочей группы.

OpenCL реализует сложную модель памяти, принципиальная схема которой приведена на рисунке 2.

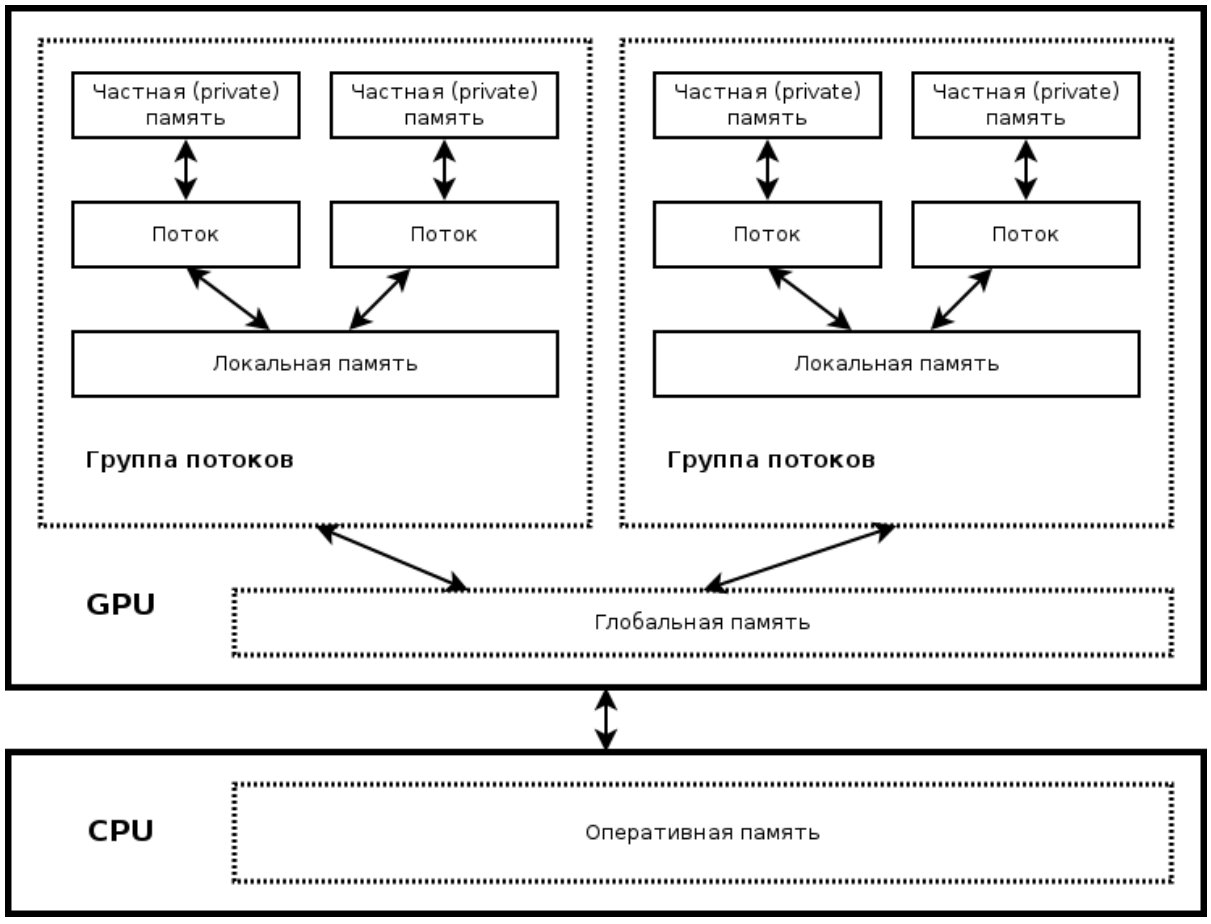

Рисунок 2 — Модель памяти, реализуемая OpenCL

Каждый вычислительный поток, работающий на GPU, имеет доступ к следующим областям памяти:

- глобальная память (англ. global / constant memory) — общая для всех потоков, выполняющихся на GPU;
- локальная память (англ. Local memory) — память, общая для всех потоков, выполняющихся в составе одной рабочей группы (англ. workgroup);
- собственная память потока (англ. private memory) — память, доступная только вычислительному потоку.

Таким образом, поток имеет доступ к трем областям памяти, при этом:

- глобальная память является наибольшей по размеру (сотни мегабайт — несколько гигабайт), однако самой медленной из всех;
- собственная память является наиболее быстрой, однако ее объем составляет несколько десятков килобайт;
- локальная память рабочей группы по быстроте доступу занимает промежуточное место между глобальной и собственной памятью, ее объем составляет несколько десятков килобайт;
- локальная память не доступна потокам из других рабочих групп;
- динамическое выделение памяти запрещено.

Перечисленные особенности работы с памятью оказывают существенное влияние на программирование решений задач — как правило, существует проблема выбора между использованием глобальной памяти всегда и промежуточным кэшированием данных из глобальной памяти в локальную и собственную.

Узкое место при использовании вычислений на графических процессорах составляет скорость передачи данных по шине с хоста во внутреннюю память устройства с GPU. По оценкам это время составляет приблизительно 90% общего времени обработки данных на GPU.

Практическая часть

В лабораторной работе разработка приложения будет основываться на пакете AMD OpenCL SDK. Для разработки приложений, использующих технологию OpenCL для устройств NVIDIA или Intel требуется наличие соответствующегоSDK или ручной перенос некоторых заголовочных файлов в папку с проектом.

1) Первичная настройка проекта.

Запустите Visual Studio и выполните пункт меню File → New Project → Visual c++ → CLR → CLR console application. В свойствах проекта «Property Pages» в Configuration Properties → C/C++, в строке Additional Include Directories необходимо указать полный путь до каталога с содержимым CL\cl.h. Если путь указан как переменная окружения, то допускается следующее содержимое данной строки, например «$(AMDAPPSDKROOT)/include». При этом переменная окружения должна быть объявлена соответствующим образом, например как на рисунке 1.

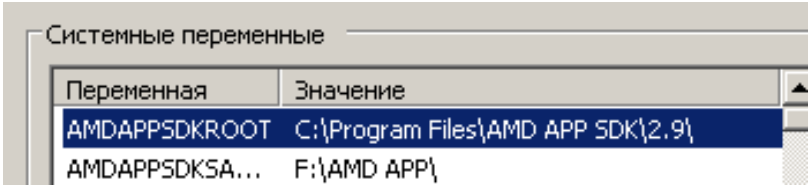

Рисунок 1 – Фрагмент окна «свойства система –> дополнительные параметры системы -> переменные среды»

По аналогии в Configuration Properties → Linker встроке Additional Library Directories добавить полный путь к \lib\x86 или $(AMDAPPSDKROOT)\lib\x86.

В пункте Configuration Properties → Linker встроке Additional Dependencies указать OpenCL.lib. Настройка проекта на этом считается законченной.

2) Создания шаблона для выбора и инициализации устройства.

Первым делом необходимо подключить заголовочный файл cl.h. В нашем случае в коде необходимо указать строку

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

Добавим также следующие модули:

```
#include<string.h>
#include<stdio.h>
#include<stdlib.h>
#include<iostream>
#include<string>
#include<fstream>
```

Они пригодятся нам в дальнейшем. Не забываем указать пространство имен:

```
Using namespace std;
```

В процедуре _tmain или main, в зависимости от вида приложения, укажем вызов процедуры выбора и инициализации. Для этого в коде запишем строку: Init;

Выше процедуры _tmain объявим саму процедуру инициализации

```
void Init()
{
}
```

На первом шаге получим список доступных платформ, для этого в процедуре Init запишем следующий код:

```
cl_uint numPlatforms;
cl_platform_id platform = NULL;
cl_int status = clGetPlatformIDs(0, NULL, &numPlatforms);
```

Выполнение функций OpenCL по необходимости осуществляется проверкой значения переменной status на значение 0.

```
#define CL_SUCCESS 0
if (status != CL_SUCCESS)
{
    //Обрабатываем ошибку
}
```

Обратите внимание, что в переменную numPlatforms заносим значение 0, а в platform - значениеNULL,это делается с целью определения фактического количества платформ установленных на хосте. Далее выбираем первую платформу (в компьютере может быть установлено несколько различных вычислительных платформ, в нашем случае на лабораторном компьютере имеется только одна платформа). Следующие строки кода позволяют выбрать платформу с максимально доступным номером (в данной работе №1).

```
if(numPlatforms > 0)
{
    cl_platform_id* platforms = (cl_platform_id* )malloc(numPlatforms* sizeof(cl_platform_id));
    status = clGetPlatformIDs(numPlatforms, platforms, NULL);
```

```
            platform = platforms[0];
            free(platforms);
        }
```

В результате получим идентификатор платформы в переменной platforms.

На втором шаге произведем опрос платформы и выберем первое GPU устройство. Для этого используем идентификатор выбранной платформы и параметр CL_DEVICE_TYPE_GPU, определяющий выбор устройств среди класса GPU-устройств. Доступны к выбору также все устройства с CPU и ускорители, для их выбора параметр соответственно имеет вид:

```
CL_DEVICE_TYPE_CPU  (для устройств CPU),
CL_DEVICE_TYPE_ACCELERATOR (для ускорителей),
CL_DEVICE_TYPE_CUSTOM (другие типы устройств),
CL_DEVICE_TYPE_ALL (все доступные устройства).
```

Следующий код демонстрирует выбор устройств с GPU.

```
        cl_uint numDevices = 0;
        cl_device_id *devices;
        status  =  clGetDeviceIDs(platform,  CL_DEVICE_TYPE_GPU,  0,
NULL,&numDevices);
    devices        =        (cl_device_id*)malloc(numDevices        *
sizeof(cl_device_id));
          status   =   clGetDeviceIDs(platform,   CL_DEVICE_TYPE_GPU,
numDevices, devices, NULL);
```

Функция вызывается последовательно два раза. При первом вызове в третий параметр функции передают значение 0, а в четвертый – NULL, функция вернет количество устройств на платформе в переменную numDevices. При повторном вызове третьим параметром передается полученное значение количества устройств,а четвертым указатель на массив, в который будет внесен перечень идентификаторов устройств на платформе. Далее останется выбрать из массива необходимый идентификатор устройства для дальнейшей работы с устройством, например первый в массиве.

Получение сведений об устройстве

Следующий шаг не обязателен, но позволяет получить информацию о параметрах устройства.

Все доступные информационные параметры доступны в таблице со станицы 38 в файле Opencl-1.2.pdf, который прилагается в каталоге как вспомогательное пособие для лабораторной работы.

Некоторые важные имена параметров приведены ниже:

CL_DEVICE_TYPE - тип устройства;

CL_DEVICE_MAX_COMPUTE_UNITS - количество вычислительных блоков;

CL_DEVICE_PLATFORM – имя платформы;

CL_DEVICE_NAME имя устройства;

и прочие.

Пример чтения одного параметра (номер версии OpenCL):

```
Char openclVersion[1024];
status = clGetDeviceInfo(devices,
                                    CL_DEVICE_OPENCL_C_VERSION,
sizeof(openclVersion),
openclVersion,  0);
```

Все параметры читаются однотипно с помощью функции clGetDeviceInfo , в ней:

- **Первый параметр** devices – получается в результате выполнения clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, numDevices, devices, NULL);
- **Второй параметр** - константное обозначение берется из колонки cl_device_info таблицы со станицы 38 в файле Opencl-1.2.pdf;
- **Третий параметр** - размер типа из второй колонки таблицы (например для типа char[] задается буфер 1024 элемента),
- **Четвертый параметр** - переменная куда будет возвращен результат.
- **Пятый параметр** 0.

Далее создаем контекст для выбранного устройства, передав в процедуру идентификатор выбранного устройства.

```
context = clCreateContext(NULL,1, devices,NULL,NULL,NULL);
```

Создаем очередь команд

```
commandQueue = clCreateCommandQueue(context, devices[0], 0,
NULL);
```

Загружаем текст программы для ядра

```
/*Step 5: Create program object */
constchar *filename = "HelloWorld_Kernel.cl";
stringsourceStr;
status = convertToString(filename, sourceStr);
constchar *source = sourceStr.c_str();
size_tsourceSize[] = {strlen(source)};
cl_program program = clCreateProgramWithSource(context, 1,
&source, sourceSize, NULL);
/*Step 6: Build program. */
status=clBuildProgram(program, 1,devices,NULL,NULL,NULL);
}
```

Код функции convertToString выглядит следующим образом

```
intconvertToString(constchar *filename, std::string& s)
{
     size_t size;
     char*  str;
     std::fstream      f(filename,      (std::fstream::in      |
std::fstream::binary));

     if(f.is_open())
     {
          size_tfileSize;
          f.seekg(0, std::fstream::end);
          size = fileSize = (size_t)f.tellg();
```

```
            f.seekg(0, std::fstream::beg);
            str = newchar[size+1];
            if(!str)
            {
                  f.close();
                  return 0;
            }

            f.read(str, fileSize);
            f.close();
            str[size] = '\0';
            s = str;
            delete[] str;
            return 0;
      }

      return -1;
}
```

Таким образом, из файла HelloWorld_Kernel.cl будет считан код на языке OpenCl (си подобный язык) и скомпилирован под конкретную архитектуру выбранного ранее устройства. Функция clCreateProgramWithSource представляет более универсальный способ создания программы для ядра, но имеет недостаток - любой может прочесть и изучить ваш код, работающий на GPU, также требуется время на компиляцию кода. Этого недостатка можно избежать, если хранить микропрограмму не в исходном коде, а в бинарном формате (заранее скомпилированную). Загрузить микропрограмму из бинарного файла можно функцией clCreateProgramWithBinary, при этом выполнение функции clBuildProgram, описануой ниже, уже не потребуется.

4) Приведем простейший пример программы для ядра GPU, которая добавляет единицу к каждому элементу массива входных данных (пояснения языка программирования на OpenCl будет дано позже.)

Процедура, выполняемая ядром, называется helloworld. В одном файле может присутствовать несколько процедур, помимо этого можно загрузить и скомпилировать на GPU несколько файлов для одновременного выполнения. Названия функции, тип и количество входных параметров нам понадобятся далее, поэтому запомним их.

```
__kernel void helloworld(__global char* in, __global char* out)
{
      intnum = get_global_id(0);
      out[num] = in[num] + 1;
}
```

После того как вы записали

```
status=clBuildProgram(program, 1,devices,NULL,NULL,NULL);
```

инициализируем входной буфер и заполним его данными (обратите внимание, буфер создается в оперативной памяти Host-устройства):

```
      const char* input = "GdkknVnqkc";
```

Инициализируем выходной буфер, в который GPU вернет результат:

```
size_tstrlength = strlen(input);
char *output = (char*) malloc(strlength + 1);
```

Следующим шагом необходимо создать аналогичный буфер в памяти GPU и перенести в него данные с Host машины. Обратите внимание, что буфер с входными данными может быть объявлен только на чтение, а выходные данные только на запись. Буферы должны быть привязаны к созданному контексту и должен быть указан способ передачи данных (копировать с Host в память GPU или использовать буфер в памяти Host).

```
cl_meminputBuffer = clCreateBuffer(context,
CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR, (strlength + 1) *
sizeof(char),(void *) input, NULL);
cl_memoutputBuffer = clCreateBuffer(context,
CL_MEM_WRITE_ONLY , (strlength + 1) * sizeof(char), NULL, NULL);
```

Укажем имя исполняемой функции, создав тем самым объект ядра:

```
cl_kernel kernel = clCreateKernel(program,"helloworld", NULL);
```

Для корректного выполнения объекта яда необходимо передать аргументы, определенные в этой функции.

```
status = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void
*)&inputBuffer);
status = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void
*)&outputBuffer);
```

Далее необходимо определить размер рабочего массива (количество одновременно задействованных вычислительных блоков в устройстве). Запуск микропрограммы на выполнение на устройстве осуществляется функцией clEnqueueNDRangeKernel, в которую передают в качестве параметров: идентификатор созданной ранее командной очереди, указатель на kernel (можно интерпретировать его как исполняемую микропрограмму) и количество задействованных блоков (global_work_size).

```
size_tglobal_work_size[1] = {strlength};
status = clEnqueueNDRangeKernel(commandQueue, kernel, 1,
NULL, global_work_size, NULL, 0, NULL, NULL);
```

Результат выполнения необходимо прочитать из памяти GPU в память Host-машины:

```
status = clEnqueueReadBuffer(commandQueue, outputBuffer,
CL_TRUE, 0, strlength * sizeof(char), output, 0, NULL, NULL);
```

Добавим в буфер символ 0 так как строка должна заканчиваться символом 0.

```
output[strlength] = '\0';//Add the terminal character to the end of output.
```

Следующим шагом являетсяочистка результата.

```
/*Step 12: Clean the resources.*/
status = clReleaseKernel(kernel);//*Release kernel.
```

```
    status = clReleaseProgram(program);    //Release the program
object.
    status = clReleaseMemObject(inputBuffer);//Release mem object.
    status = clReleaseMemObject(outputBuffer);
    status = clReleaseCommandQueue(commandQueue);//Release Command
queue.
    status = clReleaseContext(context);//Release context.

    if (output != NULL)
    {
        free(output);
        output = NULL;
    }

    if (devices != NULL)
    {
        free(devices);
        devices = NULL;
    }
```

Вывод содержимого буфера можно осуществить с помощью следующей строки кода

```
    cout<<output<<endl;
```

Задания

1. Обработать буфер входных данных по формуле O=I+K, где I - символ (типа char) во входном буфере, O - символ (типа char) в выходном буфере, K - значение добавляемое к коду символа.

2. Набрать программу по пунктам методички, вывести результат конвертирования на экране с применением GPU (при наличии GPU с поддержкой OpenCL на лабораторном компьютере).

3. Выполнить программу с применением CPU (меняется константа с CL_DEVICE_TYPE_GPU на CL_DEVICE_TYPE_CPU).

4. Создать по аналогии с предыдущими пунктами создать два входных буфера *A* и *B* с типом значений - float (не менее 10 значений в буфере). Инициализировать их значениями. С помощью OpenCL вычислить парное произведение A[i]*B[i]. Вывести результат на мониторе.

5. Получить сведения об устройстве (Имя устройства, тип, количество вычислительных блоков и т.п.).