Министерство науки и образования РФ

РГРТУ

Методические указания по лабораторной работе №1

«Знакомство с OPENCL»

Рязань, 2022г.

# 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](http://dic.academic.ru/dic.nsf/ruwiki/11856) (GPP), процессор специального назначения (например, [цифровой сигнальный процессор](http://dic.academic.ru/dic.nsf/ruwiki/70891) (DSP) или [графический процессор](http://dic.academic.ru/dic.nsf/ruwiki/69730) (GPU)), сопроцессор, логика ускорения (специализированная интегральная схема (ASIC) или [программируемая](http://dic.academic.ru/dic.nsf/ruwiki/1268229) [пользователем вентильная матрица](http://dic.academic.ru/dic.nsf/ruwiki/1268229) (FPGA)).

В общем, [гетерогенная](http://dic.academic.ru/dic.nsf/ruwiki/444719) вычислительная платформа содержит процессоры с разными [наборами команд](http://dic.academic.ru/dic.nsf/ruwiki/1734270) (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](http://dic.academic.ru/dic.nsf/ruwiki/88479) [вычислительные системы](http://dic.academic.ru/dic.nsf/ruwiki/88479) 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.

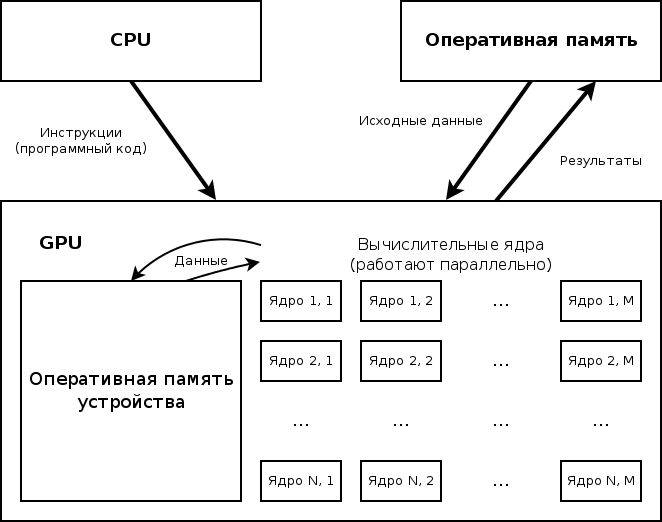
Принципиальная схема выполнения общих вычислений на GPU приведена на рисунке 1.

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

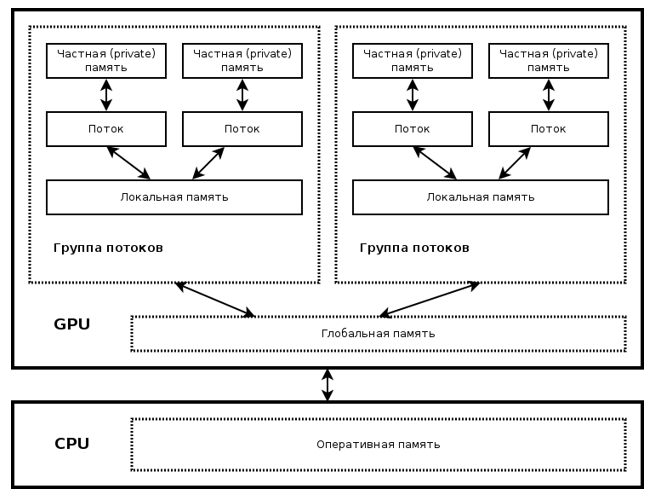
. 

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 соответствующего SDK или ручной перенос некоторых заголовочных файлов в папку с проектом.

# 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 (рисунок 3). Если путь указан как переменная окружения, то допускается следующее содержимое данной строки, например, «$(AMDAPPSDKROOT)/include».

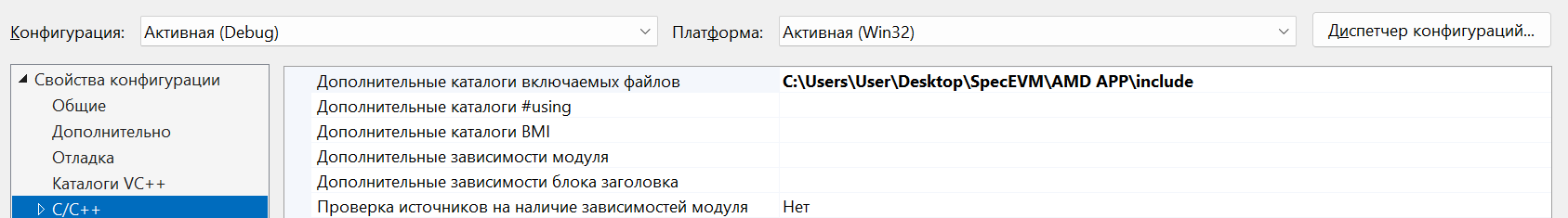


Рисунок 3 – Дополнительные каталоги включаемых файлов

По аналогии в Configuration Properties → Linker в строке Additional Library Directories добавить полный путь к \lib\x86 или $(AMDAPPSDKROOT)\lib\x86 (рисунок 4).

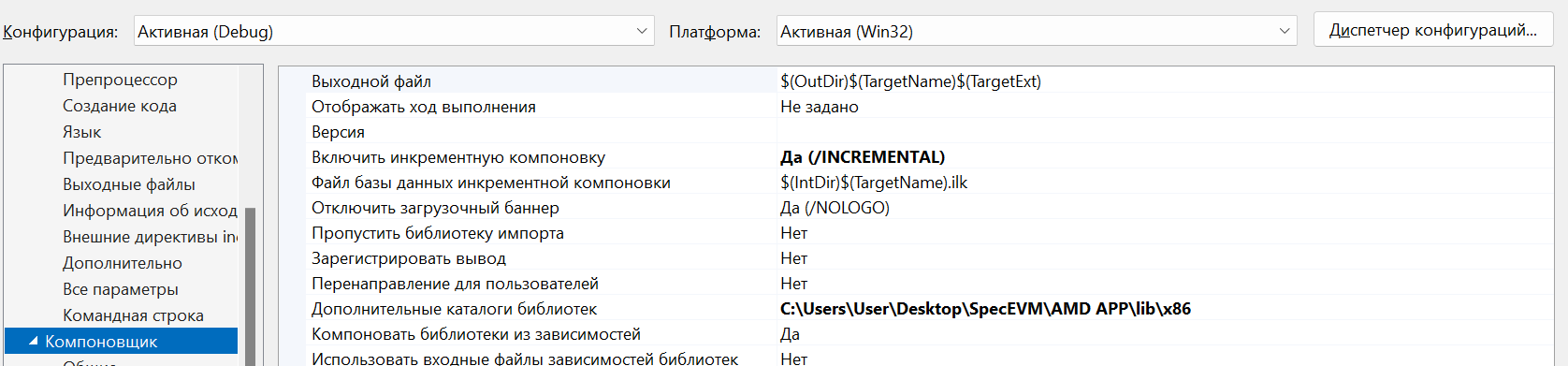


Рисунок 4 – Дополнительные каталоги библиотек

В пункте Configuration Properties → Linker в строке Additional Dependencies указать OpenCL.lib. Настройка проекта на этом считается законченной (рисунок 5).

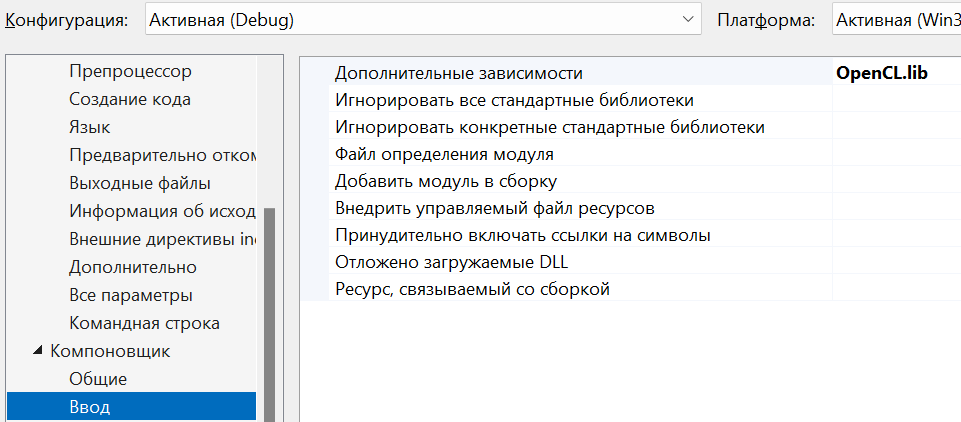


Рисунок 5 – Дополнительные зависимости.

# 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 int main() или main, в зависимости от вида приложения, укажем вызов процедуры выбора и инициализации. Для этого в коде запишем строку: Init;

Выше процедуры int main () 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

const char\* output = "Ошибка";

cout << output << endl;

}

Обратите внимание, что в переменную **numPlatforms** заносим значение 0, а в **platform** – значение **NULL**, 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.

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[0], CL\_DEVICE\_OPENCL\_C\_VERSION,

sizeof(openclVersion), openclVersion,

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.

cl\_context context = clCreateContext(NULL, 1, devices, NULL, NULL, NULL);

Создаем очередь команд

cl\_command\_queue commandQueue = clCreateCommandQueue(context, devices[0], 0, NULL);

Загружаем текст программы для ядра

const char\* filename = "HelloWorld\_Kernel.cl";

string sourceStr;

status = convertToString(filename, sourceStr);

const char\* source = sourceStr.c\_str();

size\_t sourceSize[] = { strlen(source) };

cl\_program program = clCreateProgramWithSource(context, 1, &source, sourceSize, NULL);

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

Перед Init() вставим код функции convertToString который выглядит следующим образом:

int convertToString(const char \*filename, std::string& s)

{

size\_t size;

char\* str;

std::fstream f(filename, (std::fstream::in | std::fstream::binary));

if (f.is\_open())

{

size\_t fileSize;

f.seekg(0, std::fstream::end);

size = fileSize = (size\_t)f.tellg();

f.seekg(0, std::fstream::beg);

str = new char[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, описанной 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\_t strlength = 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\_mem inputBuffer = clCreateBuffer(context, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR, (strlength + 1) \* sizeof(char), (void\*)input, NULL);

cl\_mem outputBuffer = 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\_t global\_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';

Вывод содержимого буфера можно осуществить с помощью следующей строки кода

cout << output << endl;

Следующим шагом является очистка результата.

status = clReleaseKernel(kernel);

status = clReleaseProgram(program);

status = clReleaseMemObject(inputBuffer);

status = clReleaseMemObject(outputBuffer);

status = clReleaseCommandQueue(commandQueue);

status = clReleaseContext(context);

if (output != NULL)

{

free(output);

output = NULL;

}

if (devices != NULL)

{

free(devices);

devices = NULL;

}

# Задания

1. Обработать буфер входных данных по формуле O=I+K, где I - символ (типа char) во входном буфере, О - 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 c применением CPU (меняется константа с CL\_DEVICE\_TYPE\_GPU на CL\_DEVICE\_TYPE\_СPU).
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.).

**Контрольные вопросы**

1. Что такое гетерогенные вычислительные системы

2. Как вы понимаете Hybrid-core computing

3. Какие программные средства GPGPU вы знаете

4. OpenCl понятие, цели.

5. К каким областям памяти имеет доступ вычислительный поток

Министерство науки и образования РФ

РГРТУ

Методические указания по лабораторной работе №2

«Освоение основ синтаксиса языка»

Рязань, 2022г.

**Цель работы:** Изучение языка OpenCL на примере обработки 2D изображений.

**Теоретическая часть**

На предыдущей лабораторной работе. Мы научились обращаться к устройству OpenCL, опрашивать его параметры, передавать ему данные и даже выполнят простую обработку значений в буфере данных.

Теперь целью работы будет освоение основ синтаксиса языка на примере работы с двухмерным массивом данных, представленном как полноцветное изображение.

Для выполнения лабораторной работы у Вас уже будет готовая хост программа, которая будет открывать изображение и выполнять обработку кодом, который необходимо будет сформировать в файле «prog.cl» в соответствии с заданием, этот файл находится в папке «Host». Программа представляет собой исполняемый файл, главное окно которого представлено на рисунке 1.

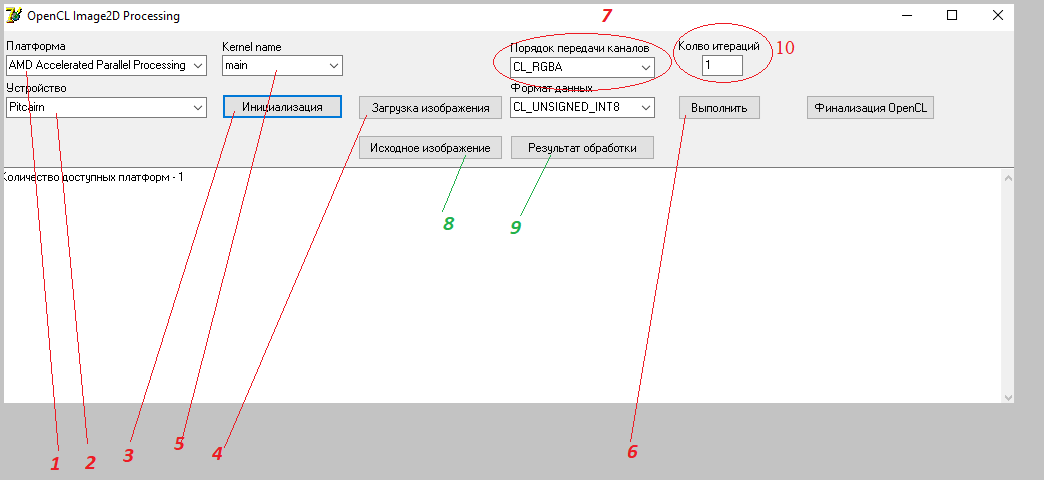


Рисунок 1 – Интерфейс хост программы

Хост программа открывает изображение в формате BMP (желательно цветностью 24 бит), трансформирует его в 32 битное и передает пиксели изображения в видеокарту как массив размером WхHх4,   
где W- ширина изображение в пикселях, H – высота изображения в пикселях, 4 – количество байт на пиксель (три цвета и альфа канал).

*Назначение кнопок интерфейса программы*:

1- указывается номер платформы, для некоторых комплектаций компьютеров может отличаться от 0;

2 - номер устройства, видеокарта не обязательна может иметь номер устройства 0;

3 - производит инициализацию устройства;

4 - загружает изображение BMP;

5 - имена функций;

6 – выполняет обработку на устройстве;

7 - выполняет обработку на устройстве;

8 - отображает загруженное изображение;

9 – результат обработки;

10 - заданное количество циклов повторной обработки изображения;

Сформированный буфер для видеокарты имеет следующие характеристики:

- тип данных Image2D – видеокарта считает буфер двухмерным изображением и дает возможность обращаться к его элементам по значению двух координат;

- формат цветового канала CL\_UNSIGNED\_INT8 (беззнаковое 8 битное целое на каждый канал цвета);

- порядок следования цветов CL\_BGRA.

Порядок CL\_BGRA означает, что при прочтении пикселя из массива в векторную переменную ее первая компонента будет содержать значение яркости синего цвета пикселя, вторая – зеленого, третья – красного и четвертая – значение альфа канала.

Стандартная заготовка микропрограммы для такой организации обработки данных выглядит следующим образом.

*\_\_constant sampler\_t sampler = CLK\_NORMALIZED\_COORDS\_FALSE |*

*CLK\_ADDRESS\_CLAMP | CLK\_FILTER\_NEAREST;*

*\_\_kernel void main(read\_only image2d\_t src\_image,*

*write\_only image2d\_t dst\_image)*

*{*

*uint x = get\_global\_id(0);*

*uint y = get\_global\_id(1);*

*uint width = get\_global\_size(0);*

*uint height = get\_global\_size(1);*

*}*

Где,

- *\_\_constant sampler\_t sampler –* объявляет объект типа *sampler\_t*, для возможности чтения и записи пикселей изображения,

- *CLK\_NORMALIZED\_COORDS\_FALSE –* константа указывает, что применяется не нормализованная система координат, т.е. значение координат находится в диапазоне [0...W-1, 0...H-1],

*- CLK\_ADDRESS\_CLAMP,* указывает возвращаемые значения при обращении за границы массива.

Параметры основной процедуры main объявлены следующим образом:

- *read\_only image2d\_t src\_image –* входной буфер доступный только на чтение,

- *write\_only image2d\_t dst\_image –* выходной буфер доступный только на запись,

Необязательные, но очень полезные функции:

- *get\_global\_id* возвращает координаты текущего “кернеля” (применительно к этой программе, текущий пиксель в массиве),

- *get\_global\_size* возвращает размеры массива (применительно к этой программе, размеры изображения в пикселях).

**Синтаксис языка.**

Считать значение пикселя можно следующим образом:

*int2 coord; // переменная задает координаты чтения*

*coord = (int2)(x,y); // задаем конкретные координаты*

*uint4 i00 = read\_imageui(src\_image, sampler, coord); // читаем пиксель*

Обратите внимание суффикс **UI**на конце команды *read\_imageui* говорит о том, что пропитанные данные интерпретируются как беззнаковые целые числа.

Можно интерпретировать прочитанные пиксели и в другие форматы, в том числе и float, для этого применяется аналогичная команда, но с другим суффиксом.

Прочтенные данные помещаются в векторную переменную i00.

Записать результат по координатам можно так:

write\_imageui(dst\_image, coord, i00);

**Приведем пример как можно использовать векторные переменные.**

Пусть uint4 i00 содержит значение пикселя, тогда:

|  |  |
| --- | --- |
| i00=(uint4)(255)-i00; | Инверсия цветов |
| i00=(uint4)(255); | Белый цвет |
| i00=(uint4)(0); | Черный цвет |
| i00=(255,0,0,0); | Синий цвет |
| i00.x=i00.y | Зеленый копируется в синий |
| i00.xyzw=i00.yyyy | Во все цветовые каналы копируется значение зеленого |
| i00.xyzw=i00.y | Такое оформление недопустимо. ОШИБКА!!! |
| float4 Gx, Gy  float4 Gxy =hypot(Gx, Gy); | Производится вычисление гипотенузы по формуле: |

**Практическая часть**

Порядок работы с программой.

1. Запустить программу;



2. Указать номер платформы и устройства;

3. Нажать кнопку 3 (Инициализация);

4. Нажать кнопку 4 (Загрузить изображение) и выбрать изображение;

5. Указать количество итераций 1 и нажать кнопку 6 (Выполнить);

6. Если все вычисляется повторить операции с 1 по 4, количество итераций указать 20...30, выполнить обработку запомнить время выполнения.

7. Поменять номер устройства или платформы при необходимости, чтобы инициализировать OpenCl на процессоре;

8. Повторить предыдущие шаги на процессоре.

**Задания**

1. Обработать изображение фильтром Собеля.

матрица коэффициентов

**A**:

1;2;0;-2;-1;

2;3;0;-3; -2;

3;4;0;-4; -3;

2;3;0;-3; -2;

1;2;0;-2; -1;

**B:**

1;2;3;2;1

2;3;4;3;2

0;0;0;0;0

-2;-3;-4;-3;-2

-1;-2;-3;-2;-1

2. Инвертировать пиксели, отразить строки по вертикали (первая становится последней, вторая предпоследней и так далее).

3. Размытие Bluer 5x5

**Коэффициенты:**

0.003; 0.013; 0.022; 0.013; 0.003;

0.013; 0.059; 0.097; 0.059; 0.013;

0.022; 0.097; 0.159; 0.097; 0.022;

0.013; 0.059; 0.097; 0.059; 0.013;

0.003; 0.013; 0.022; 0.013; 0.003;

4. Мозаика из сжатого в два раза изображения, запись цветов согласно схеме.

|  |  |
| --- | --- |
| **B** | **R** |
| **G** | **BGR** |



5. Контрастирование по порогу для R - 20..120, для G - 40 ..140, для B - 100..200.

6. Сделать серым, отобразить строки слева направо.

7. Bluer3 (среднее значение среди 9 пикселей окно 3х3) + сетка черным цветом с шагом 50.

8. Сделать серым, инвертировать цвета, прямоугольник с координатами 0…50,0…50 заполнить белым цветом.

9. Bluer3 (среднее значение среди 9 пикселей окно 3х3) + оставить цвета согласно схеме.

|  |  |
| --- | --- |
| **B** | **R** |
| **G** | **BGR** |



10. Транспонировать прямоугольное окно в картинке с координатами 50,50 (ширина и высота 200х200).

**Контрольные вопросы**

1. В каком формате открывает изображение Хост программа

2. Что означает суффикс UI на конце некоторых команд

3. Расскажите, как инвертировать цвет пикселя.

4. Расскажите, как заполнить все цветовые каналы одним цветом.

Министерство науки и образования РФ

РГРТУ

Методические указания по лабораторной работе №3

«Несколько этапные неграфические вычисления на GPU»

Рязань, 2022г.

**Лабораторная работа №3**

Классическим вариантом применения вычислений на GPU считается следующий:

1. Инициализируется устройство с GPU;
2. Подготавливается один или несколько входных буферов в памяти хост машины с заполнением данными;
3. Данные буферов переносятся в память устройства с GPU;
4. В памяти устройства с GPU выделяется место под выходные данные;
5. В устройство передается микропрограмма функции и ее параметры;
6. После выполнения данные из памяти устройства с GPU копируются в память хост машины, затем цикл обработки можно повторять заново.

При конвейерной обработке данных, когда результат вычислений необходимо подать на вход следующего шага, такой подход дает проигрыш во времени. Существенные потери времени будут на операциях копирования данных на устройство и обратно. Если объем данных для обработки велик, то время копирования может свести на нет ускорение, полученное от применения GPU для вычислений.

Следует заметить, что при копировании данных из выходного буфера в память хоста, физически данные не удаляются из памяти устройства. Поэтому достаточно передавать указатель на область памяти результата как параметр указывающий на расположения входных данных для следующего шага выполнения обработки. Разумеется, область памяти при этом должна быть распределена на устройстве с атрибутами на чтение и запись. Т.е. в функции clCreateBuffer при создании выходного буфера на устройстве вместо параметра CL\_MEM\_WRITE\_ONLY следует указывать CL\_MEM\_READ\_WRITE. Если каждый поток будет писать только в одну ячейку (т.е. не будет случая когда два потока будут пытаться записать данные по одному адресу), то тогда в качестве выходного буфера параметром можно смело передать указатель на входной буфер. Например, так:

status = clSetKernelArg(kernel, 0, sizeof(cl\_mem), (void \*)&inputBuffer);

status = clSetKernelArg(kernel, 1, sizeof(cl\_mem), (void \*)&inputBuffer);

Здесь первым параметром передается указатель на входные данные, вторым на выходные, по факту это один и тот же адрес, значит при записи ядро будет модифицировать входные данные и ими еще раз можно будет воспользоваться на следующем шаге.

Еще как вариант можно на каждом шаге обработки менять указатели на буферы местами, например, так:

**На первом шаге**

status = clSetKernelArg(kernel, 0, sizeof(cl\_mem), (void \*)&inputBuffer);

status = clSetKernelArg(kernel, 1, sizeof(cl\_mem), (void \*)&outputBuffer);

**На втором**

status = clSetKernelArg(kernel, 0, sizeof(cl\_mem), (void \*)&outputBuffer);

status = clSetKernelArg(kernel, 1, sizeof(cl\_mem), (void \*)&inputBuffer);

**и т.д. далее циклически повторяя.**

Когда мы поняли механизм, который позволит не копировать данные «взад и вперед» по шине, с хоста на GPU и обратно перейдем к рассмотрению допустимых математических функций в микропрограмме.

Подробно синтаксис и перечень функций описан в документе «The OpenCL Specification» от Khronos group. Данная спецификация периодически дополняется.

При работе с векторными данными можно использовать суффиксы для выделения необходимых частей вектора, таких как .lo, .even, .hi, .odd.

*Пример:*

*float4 vf;*

*float2 low = vf.lo; // возврашается vf.xy*

*float2 high = vf.hi; // возврашается vf.zw*

*float2 even = vf.even; // возврашается vf.xz*

*float2 odd = vf.odd; // возврашается vf.yw*

Приведения к типу (TypeCasts), пример, как передать содержимое Float в Int:

*float f = 1.0f; int i = (int)f;*

Данное действие допустимо только для скалярных данных, для вектора подобные действия не разрешены, например:

*int4 i;*

*uint4 u = (uint4) i; // ОШИБКА!!!*

Для векторных данных каждая компонента должна быть обработана как скалярная

*float f = 1.0f;*

*float4 va = (float4)f;*

*// va будет содержать float4 vector с элементами (f, f, f, f).*

*Или так:*

*uchar u = 0xFF; float4 vb = (float4)u;*

Типы можно конвертировать один в другой с помощью команды convert\_destType(sourceType), примеры применения:

*uchar4 u;*

*int4 c = convert\_int4(u);*

*float f;*

*int i = convert\_int(f);*

Полная форма команды для скалярных еременных:

*destType* ***convert\_destType<\_sat><\_roundingMode>*** *(sourceType), для векторных (количество компонент n)*

*destTypen* ***convert\_destTypen<\_sat><\_roundingMode>*** *(sourceTypen)*

**\_sat -** учитывающий поведения при переполнении и насыщении значения, более подробно в OpenCl-1.2.pdf стр. 210.

**\_roundingMode -** задает способ округления

\_rte Округление до ближайшего четного

\_rtz Округление до 0

\_rtp Округление к положительной бесконечности

\_rtn Округление к отрицательной бесконечности по умолчанию установлено значение \_rtz

**Пример операций с векторами:**

*float4 v, u; float f;*

*v = u + f;*

*Это будет эквивалентно следующей записи*

***v.x = u.x + f;***

***v.y = u.y + f;***

***v.z = u.z + f;***

***v.w = u.w + f;***

Далее в описании аргументов и возвращаемых значений

для функций присутствуют обозначения generic type name.

Например, **gentype** – показывает, что параметры и возвращаемые значения функции могут быть следующих типов: float, float2, float3, float4, float8, float16, double, double2, double3, double4, double8 или double16.

**gentypef** – аналогично, показывает, что параметры и возвращаемое значение могут быть типами float, float2, float3, float4, float8, или float16.

**gentyped** – для double, double2, double3, double4, double8 или double16.

Математические функции (весь спектр доступных функций описан в спецификации со страницы 244 и далее). Мы приведем лишь часто используемые функции.

gentype **acos** (gentype) – Арккосинус,

gentype **acospi** (gentype *x*) – **acos** (*x*) / π,

gentype **asin** (gentype) – Арксинус,

gentype **asinpi** (gentype *x*) – **asin** (*x*) / π,

gentype **atan** (gentype *y\_over\_x*) – Арктангенс,

gentype **atan2** (gentype *y*, gentype *x*) – Арктангенс (*y* / *x)*,

gentype **atan2pi** (gentype *y*, gentype *x*) – **atan2** (*y*, *x*) / π,

gentype **cbrt** (gentype) – Кубический корень,

gentype **ceil** (gentype) – округление до целого в режиме округления с положительной бесконечностью,

gentype **cos** (gentype) – Косинус,

gentype **cospi** (gentype *x*) – **cos** (π *x*),

gentype **exp** (gentype *x*) – e в степени *x*,

gentype **exp2** (gentype) – e2,

gentype **expm1** (gentype *x*) – *ex*- 1.0,

gentype **fabs** (gentype) – Модуль для чисел с плавающей точкой,

gentype **floor** (gentype) – округление в режиме отрицательной бесконечности,

gentype **fmax** (gentype *x*, gentype *y*) – максимум,

gentype **fmin** (gentype *x*, gentype *y*) – минимум,

gentype **fmod** (gentype *x*, gentype *y*) – возврашает *x* – *y* \* **trunc** (*x*/*y*),

gentype **log** (gentype) – Натуральный логарифм,

gentype **log2** (gentype) – Логарифм по основанию 2,

gentype **log10** (gentype) – Десятичный логарифм,

gentype **log1p** (gentype *x*) – loge(1.0 + *x*),

gentype **mad** (gentype *a*, gentype *b*, gentype *c*) – a\*b+с,

gentype **round** (gentype *x*) – округление

gentype **sqrt** (gentype) – квадратный корень.

gentype **tan** (gentype) – тангенс.

gentype **trunc** (gentype) – отбросить целую часть

ugentype **abs** (gentype *x*) – возвращает | x |.

ugentype **abs\_diff** (gentype *x*, gentype *y*) – | x – y |

gentype **sign** (gentype *x*) – знак числа

gentype **degrees** (gentype *radians*) – (180 / π) \*

gentype **max** (gentype *x*, gentype *y*) – максимум

gentype **min** (gentype *x*, gentype *y*) – минимум

**Константы одинарной точности**

**M\_E\_F** – значение e **M\_LOG2E\_F – значение log2e**

**M\_LOG10E\_F** – значение log10e **M\_LN2\_F** – значение loge2

**M\_LN10\_F** – значение loge10

**M\_PI\_F** – значение π

**M\_PI\_2\_F** – значение π / 2

**M\_PI\_4\_F** – значение π / 4

**M\_1\_PI\_F** – значение 1 / π

**M\_2\_PI\_F** – значение 2 / π **M\_2\_SQRTPI\_F** – значение 2 / √π

**M\_SQRT2\_F** – значение √2 **M\_SQRT1\_2\_F** – значение 1 / √2

Аналогичные константы со значениями двойной точности (отличаются только отсутствием окончания \_F в имени).

**Задания**

1. Используйте программу, написанную на лабораторной работе №1 как основу для модернизации. Внесите в ней изменения обеспечивающие возможность не копировать буфер с устройства после вычислений а повторно использовать его для аналогичного вычисления. Чтобы в результате появилась надпись HelloWorld необходимо изменить строку

const char\* input = "GdkknVnqkc";

на

const char\* input = "FcjjmUmpjb";

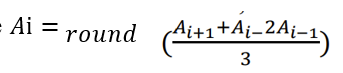
Выполните программу так чтобы kernel вызывался последовательно два раза. результат правильного выполнения – HelloWorld на экране монитора.

2. Измените размер буфера до 1000 элементов, заполните значениями и обработайте данные согласно варианта задания. Код программы для всех шагов разместите в одной процедуре выполняя, № шага передается в функцию как параметр.

Обязательный элемент отчета текст микропрограммы и фрагмент программы хоста с внесенными в нее изменениями.

**Варианты заданий:**

1.Заполните массив элементами типа Float (размер элемента 4 байта) значениями по возрастанию, на первом шаге рассчитайте 𝐴i =|tan(i)𝐴i+1 -sin(i)𝐴i−1| , на втором шаге 𝐴i = (𝐴i+1 + 2𝐴i−1)/3

2. Заполните массив элементами типа int (размер элемента 4 байта) значениями по возрастанию, на первом шаге рассчитайте 𝐴i =| 3𝐴i+1 𝐴i−1 |, на втором шаге 

3. Заполните массив элементами типа byte (размер элемента 1 байта) значениями по убыванию начиная с 1000, на первом шаге для нечетных номеров байтов рассчитайте сумму значений его соседних байт, на втором шаге для четных номеров байт вычислить значение 𝐴i = 5𝐴i−1

4. Заполните массив элементами типа int4 (размер элемента 4 байта) значениями по возрастанию, на первом шаге вычислите A.x=A.x+A.z, на втором A.y=trunc((A.w+A.x)/3).

5. Заполните массив элементами типа int4 (размер элемента 4 байта) значениями по возрастанию, на первом шаге вычислите A.x=A.x+A.z, на втором A.y=trunc((A.w+A.x)/3).

6. Заполните массив элементами типа float4 (размер элемента 4 байта) значениями по возрастанию, на первом шаге вычислите A.x=sin(A.x)+cos(A.z), на втором A.y=atan((A.w+A.x)/3).

7. Заполните массив элементами типа float4 (размер элемента 4 байта) значениями по возрастанию, на первом шаге вычислите A.x=sin(A.w)+cos(5\*A.z), на втором A.y=atan((|A.w+5A.x|)/3).

8. Заполните массив элементами типа integer (размер элемента 4 байта) значениями 1, на первом шаге вычислите𝐴i = 𝐴i+1 + i, на втором для каждых элементов с нечетным значением вычесть 1 для элементов с четным значением поменять знак.

9. Заполните массив элементами типа float 4 (размер элемента 4 байта) значениями 10, на первом шаге вычислите для четных номеров вычислить 𝐴i =(cos(𝐴i+1) sin(𝐴i+1)tan(𝐴i+1),0) на втором для каждых элементов с четным номером 𝐴i = 𝐴i−1/2.

10. Заполните массив элементами типа int (размер элемента 4 байта) значениями по возрастанию, на первом шаге рассчитайте 𝐴i = |3𝐴i+1 𝐴i−1 |, на втором шаге элементов кратных 10 вычислить 𝐴i = 5𝐴i−1