# Лабораторная работа № 5
# Разработка программ с поддержкой технологии CUDA с использованием компилятора командной строки

**Цель работы:** познакомиться с особенностями разработки программ с использованием технологии CUDA.

Для разработки программ с использованием технологии CUDA необходима установка инструментария из CUDA SDK, CUDA Driver и CUDA Toolkit, в состав которого входит компилятор командной строки `nvcc`. Инструментарий свободно доступен по адресу https://developer.nvidia.com/cuda-downloads/. Программа с поддержкой технологии CUDA разбивается на `.cu`-файлы, компилируемые с использованием `nvcc`, и обычные файлы программы (в данной работе `.cpp`), компилируемые и линкуемые обычным компилятором (в данной работе `cl.exe`, входящим в состав Microsoft Visual Studio).

В состав `.cu`-файлов входят специфичные для технологии CUDA элементы: функции CUDA-ядер (англ. kernel), выполняемые потоковыми мультипроцессорами (англ. Streaming Multiprocessor, SM) видеокарты, и функции, осуществляющие вызов CUDA-ядер. Пример CUDA-ядра, осуществляющего поэлементное умножение векторов:

```
__global__ void VecMulKernel(float *a, float *b, float *c)
{
    // Определение индекса потока
    int i = threadIdx.x + blockIdx.x*blockDim.x;

    // Обработка соответствующей порции данных
    c[i] = a[i] * b[i];
}
```

Пример вызова CUDA-ядра:

```
VecMulKernel<<<blocks, threads>>>(a, b, c);
```

При компиляции `.cu`-файлов может потребоваться подключение ряда заголовочных файлов:

```
#include <cuda.h>
#include <cuda_runtime.h>
//...
```

располагающихся в папке

```
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0\include\
```

(для Windows XP x86).

`.cpp`-файлы содержат весь остальной код (например, функцию `main()`), включающий вызовы функций из `.cu`-файлов.

Компиляция простейшего проекта с использованием компилятора `nvcc` производится из командной строки следующим образом:

```
nvcc file1.cu file2.cu … fileN.cu file1.cpp file2.cpp … fileN.cpp -o program_name.exe
```

причем для корректной работы пути к файлам `nvcc.exe` и `cl.exe` должны содержаться в переменной окружения `PATH` (при необходимости их нужно добавить в список вручную). Если компиляция завершена успешно, то в результате будет сформирован исполняемый файл с указанным именем (`program_name.exe` в данном примере).

Общая стратегия обработки информации на видеокарте сводится к трем действиям:

1. Передача исходных данных из оперативной памяти в память видеокарты.
2. Запуск ядра.
3. Передача результирующих данных из памяти видеокарты и оперативную память.

Управление динамической памятью на видеокарте происходит с использованием функций `cudaMalloc()` и `cudaFree()`, функциональность которых аналогична соответствующим функциям стандартной библиотеки для работы с динамической памятью.

Для копирования данных между оперативной памятью и памятью видеокарты применяется функция `cudaMemcpy(pDst, pSrc, SizeInBytes, FromTo)`, в параметрах которой передаются адреса областей памяти источника и приемника, размер копируемой области данных в байтах и константа, определяющая направление копирования:

```
cudaMemcpyHostToHost     // RAM (Host) := RAM (Host)
cudaMemcpyHostToDevice   // GPU (Device) := RAM (Host)
cudaMemcpyDeviceToHost   // RAM (Host) := GPU (Device)
cudaMemcpyDeviceToDevice // GPU (Device) := GPU (Device)
```

При работе с динамической памятью необходимо помнить о том, что указатели на область памяти видеокарты не действительны в основной программе, и, наоборот, указатели на область оперативной памяти не действительны в коде CUDA-ядра.

При вызове функции-ядра в параметрах `blocks` и `threads` указывается конфигурация запуска ядра (число потоков в блоке и число блоков в сетке). Например:

```
// Конфигурация запуска ядра
dim3 threads = dim3(512, 1);         // 512 потоков в блоке
dim3 blocks = dim3(n/threads.x, 1); // n/512 блоков в сетке

// Вызов ядра
MulKernel<<<blocks, threads>>>(a, b, c);
```

В функции-ядре при помощи обращения к предопределенным переменным `threadIdx` и `blockIdx` можно определить «координаты» потока в блоке и блоке в сетке.

**Задание.** Создать .cu-файл, поместив в него ссылки на заголовочные файлы `cuda.h` и `cuda_runtime.h`, код ядра для поэлементного умножения векторов (приведен выше) и следующий код с вызовом ядра:

```
// a, b - указатели на исходные массивы
// c - указатель на результирующий массив
// n - размер массивов (число элементов)
void vec_mul_cuda(float *a, float *b, float *c, int n)
{
    int SizeInBytes = n * sizeof(float);

    // Указатели на массивы в видеопамяти
    float *a_gpu = NULL;
    float *b_gpu = NULL;
    float *c_gpu = NULL;

    // Выделение памяти под массивы на GPU
    cudaMalloc( (void **)&a_gpu, SizeInBytes );
```

```
    cudaMalloc( (void **)&b_gpu, SizeInBytes );
    cudaMalloc( (void **)&c_gpu, SizeInBytes );

    // Копирование исходных данных из CPU на GPU
    cudaMemcpy(a_gpu, a, SizeInBytes, cudaMemcpyHostToDevice); // a_gpu = a
    cudaMemcpy(b_gpu, b, SizeInBytes, cudaMemcpyHostToDevice); // b_gpu = b

    // Задание конфигурации запуска ядра
    dim3 threads = dim3(512, 1);        // 512 потоков в блоке
    dim3 blocks = dim3(n/threads.x, 1); // n/512 блоков в сетке

    // Запуск ядра (покомпонентное умножение векторов c = a * b)
    VecMulKernel<<<blocks, threads>>>(a_gpu, b_gpu, c_gpu);

    // Копирование результата из GPU в CPU
    cudaMemcpy(c, c_gpu, SizeInBytes, cudaMemcpyDeviceToHost); // c = c_gpu

    // Освобождение памяти GPU
    cudaFree(a_gpu);
    cudaFree(b_gpu);
    cudaFree(c_gpu);
}
```

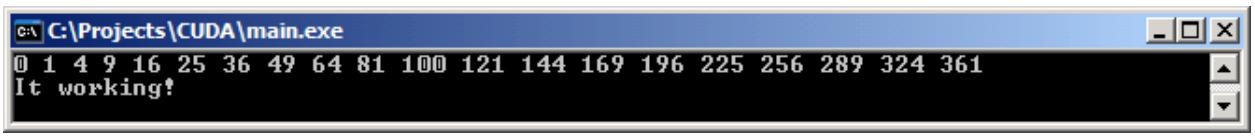
Создать `.cpp`-файл, поместив в него код для вызова разработанных подпрограмм:

```
#include <iostream>
#include <cuda.h>
#include <cuda_runtime.h>

using namespace std;

// Описание внешней функции, располагающейся в .cu-файле
void vec_mul_cuda(float *a, float *b, float *c, int n);

// Размер вектора (должен быть кратен 512)
const int N = 1024;

// Вектора
float a[N], b[N], c[N];

void main()
{
    // Заполнение векторов исходными данными
    for (int i=0; i<N; i++)
    {
        a[i] = i;
        b[i] = i;
        c[i] = 0;
    }

    // Покомпонентное умножение векторов на GPU
    vec_mul_cuda(a, b, c, 1024);

    // Вывод первых 20 значений результата
    for (int i=0; i<20; i++)
      cout << c[i] << " ";

    getchar();
    return;
}
```

Откомпилировать проект с использованием компилятора командной строки:

```
nvcc имя_cu_файла имя_cpp_файла -o main.exe
```

Убедиться в работоспособности разработанной программы:



Модифицировать программу следующим образом:

- разработать CUDA-ядро для поэлементного сложения векторов;
- сформировать вектора из случайных исходных данных;
- найти сумму векторов с использованием разработанного CUDA-ядра и с использованием подпрограммы на CPU, сравнить полученные результаты, сделать вывод о корректности расчетов на GPU.

**Содержание отчета.**

1. Титульный лист
2. Цель работы
3. Задание
4. Листинг программы
5. Скриншот с результатами работы программы
6. Выводы