Лабораторная работа №1

Основы работы с технологией CUDA. Гибридное программирование. Работа с глобальной памятью

Цель: изучить модель программирования в CUDA, иерархию памяти в CUDA и основные особенности работы с глобальной памятью.

При подготовке к лабораторной работе рекомендуется изучить материалы, предоставленные в списке литературы, а также прочие материалы по тематике лабораторной работы, представленные в открытых источниках.

Далее следует краткий конспект теоретического материала для лабораторной работы, задания и требования к лабораторной работе, а также контрольные вопросы для самопроверки.

1 Основные понятия

CUDA (Compute Unified Device Architecture) –

- технология (библиотеки и расширенный Си), предназначенная для разработки приложений для массивно-параллельных вычислительных устройств, заметно облегчает написание GPGPU (General Purposed Graphical Processing Unit)-приложений;
- программно-аппаратная архитектура.

Гибридное программирование - это написание программы для гетерогенной аппаратной вычислительной структуры, например, для системы, состоящей из центрального процессора CPU и графического ускорителя GPU.

2 Модель программирования в CUDA

GPU (Graphical Processing Unit, device) – это вычислительное устройство, которое:

- состоит из массива потоковых мультипроцессоров (streaming multiprocessor, SM);
- является сопроцессором к центральному процессору CPU (host);
- имеет собственную память (DRAM);
- выполняет одновременно большое количество нитей.

Программный код состоит из последовательных и параллельных частей, выполняющихся на CPU и GPU соответственно. Программа, использующая GPU, состоит из следующих частей:

- программного кода для GPU, описывающего необходимые вычисления и работу с памятью;
- программного кода для CPU**,** в котором осуществляется управление памятью GPU (выделение/освобождение), обмен данными между GPU/CPU, запуск кода для GPU, обработка результатов и прочий последовательный код.

Параллельная часть кода выполняется на большом количестве нитей (thread) (см. рисунок 1), которые группируются в блоки (blocks) фиксированного размера, блоки объединяются в сеть блоков (grid). Каждая нить и блок имеют свой идентификатор. Нить использует идентификаторы для определения, с каким элементом работать.

Ядро (kernel) – это функция, которая работает на GPU и которая может быть вызвана только с CPU. Ядро выполняется на сетке из блоков.

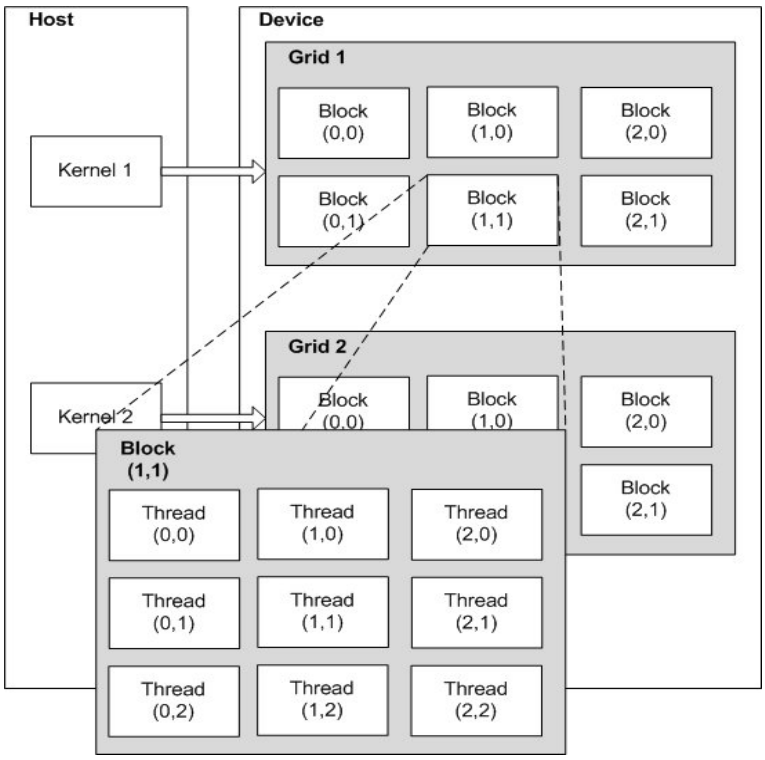

Рисунок 1 – Иерархия нитей в CUDA

Каждый блок целиком выполняется на одном SM. Нити одного блока могут взаимодействовать между собой через shared-память и через барьерную синхронизацию (__syncthreads()). Нити разным блоков не могут взаимодействовать между собой.

3 Расширения языка Си

Программы для CUDA (соответствующие файлы имеют расширение .cu) пишутся на «расширенном» Си и компилируются при помощи nvcc компилятора.

Вводимые в CUDA расширения языка Си состоят из:

- спецификаторов функций, показывающих, где будет выполняться функция и откуда она может быть вызвана;
- спецификаторов переменных, задающих тип памяти, используемый для данных переменных;
- директивы для запуска ядра из кода;
- встроенные переменные, содержащие информацию о текущей нити;
- дополнительные типы данных.

Спецификаторы функций, вводимые в CUDA, представлены в таблице 1. Спецификатор __global__ соответствует ядру и функция может возвращать только void. Спецификаторы __host__ и __device___ могут использоваться одновременно. Спецификаторы __global__ и __host__ не могут быть использованы одновременно.

Спецификаторы переменных, вводимые в CUDA, представлены в таблице 2.

Таблица 1 – Спецификаторы функций в CUDA

Спецификатор	**Выполняется на**	**Может вызываться из**
__device__	device	device
__global__	device	host
__host__	host	host

Таблица 2 – Спецификаторы переменных в CUDA

Спецификатор	**Находится**	**Доступна**	**Вид доступа**
__device__	device	device	R
__constant__	device	device / host	R / W
__shared__	device	block	R / W

Ограничения на функции, выполняемые на GPU:

- нельзя брать адрес функции (за исключением __global__);
- не поддерживается рекурсия;
- не поддерживаются static-переменные внутри функции;
- не поддерживается переменное число входных аргументов.

Ограничения на спецификаторы переменных:

- нельзя применять к полям структуры (struct или union);
- не могут быть extern;
- запись в __constant__ может выполнять только CPU через специальные функции;
- __shared__ - переменные не могут инициализироваться при объявлении.

Новые типы данных:

- одно-, двух-, трех-, четырехмерные вектора из базовых типов (u)char, (u)int, (u)short, (u)long, longlong, float, double (float3, int3 и другие);
- dim3 – uint3 с конструктором, позволяющим задавать не все компоненты (не заданные инициализируются единицей).

Для векторов не определены покомпонентные операции. Для типа данных double и long возможны только вектора размера 1 и 2. Пример работы с новыми типами данных представлен на рисунке 2.

```
int2   a = make_int2 ( 1, 7 );
float4 b = make_float4 ( a.x, a.y, 1.0f, 7 );
float2 x = make_float2 ( b.z, b.w );
dim3   grid   = dim3 ( 10 ); //эквивалентно grid(10, 1, 1)
dim3   blocks = dim3 ( 16, 16 );
```

Рисунок 2 – Пример программного кода для работы с новыми типами данных в CUDA

В CUDA поддерживаются следующие встроенные переменные, содержащие информацию о текущей нити (рисунок 3):

- dim3 gridDim; // размер сетки
- uint3 blockIdx; // индекс текущего блока в сетке
- dim3 blockDim; // размер блока
- uint3 threadIdx; // индекс текущей нити в блоке
- int warpSize; // размер warp’a

Встроенные переменные доступны в функции ядра.

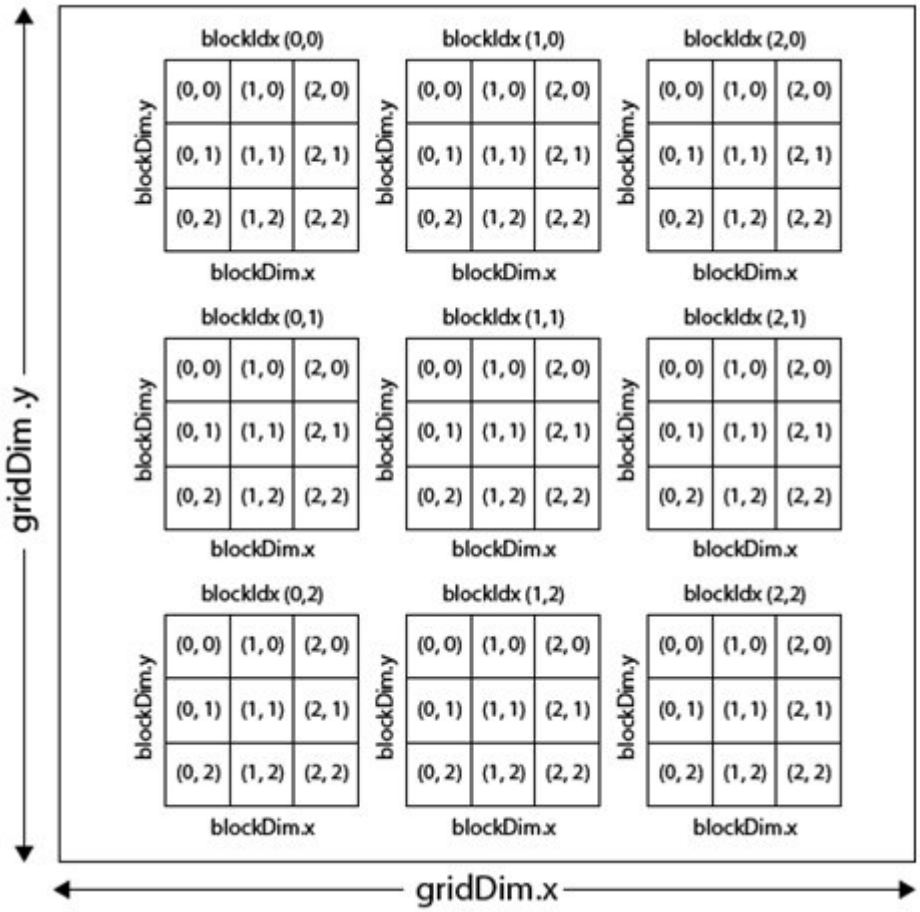

Рисунок 3 – Иерархия нитей в CUDA

Общий вид команды для запуска ядра:

kernel<<<bl, th, ns, st>>> (data);

где

- dim3 bl – число блоков в сетке;
- dim3 th – число нитей в сетке;
- size_t ns – количество дополнительной shared-памяти, выделяемое блоку;
- cudaStream_t st – поток, в котором нужно запустить ядро.

Программный код для запуска ядра с общим количеством потоков равным nx представлен на рисунке 4.

```
float * data;
dim3 threads(256, 1, 1);
dim3 blocks(nx / 256, 1);
KernelName<<<blocks, threads>>> ( data );
```

Рисунок 4 – Пример программного кода для запуска ядра с общим количеством потоков равным nx

4 Программный стек CUDA. CUDA host API

CUDA предоставляет в распоряжение программиста ряд функций (CUDA host API), которые могут быть использованы только из CPU и отвечают за:

- управление GPU;
- работу с контекстом;
- работу с памятью;
- работу с модулями;
- управление выполнением кода;
- работу с текстурами;
- взаимодействие OpenGL и Direct3D.

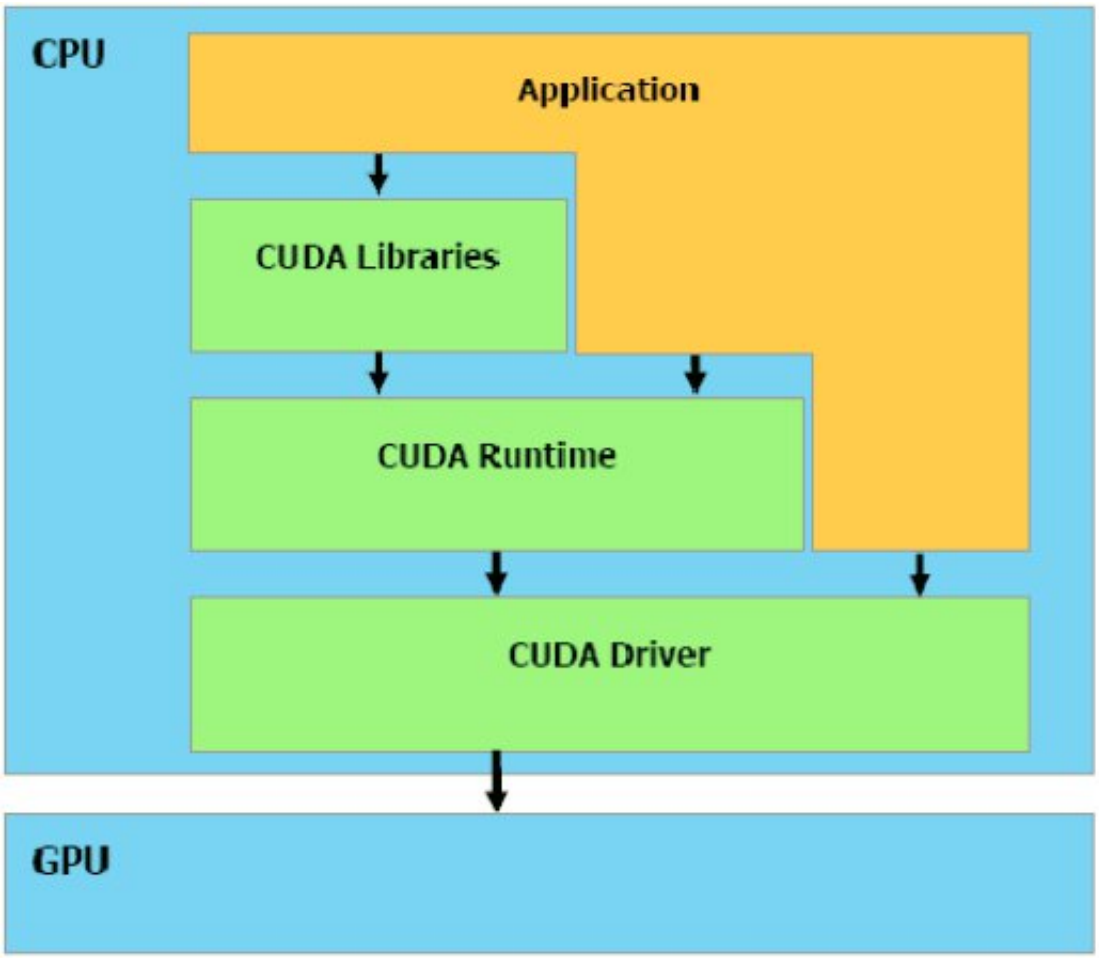

Рисунок 5 – Программный стек CUDA

CUDA host API выступает в двух формах: низкоуровневый CUDA driver API; высокоуровневый CUDA runtime API. На рисунке 5 приведены различные уровни программно-аппаратного стека CUDA. Взаимодействие с GPU происходит только через драйвер устройства. Программы могут использовать GPU посредством:

- обращения к стандартным функциям библиотек (BLAS, FFTW) – простота использования, не всегда эффективно;
- использования CUDA runtime API;
- использованием CUDA driver API.

5 CUDA «Hello world»

Программа «Hello world» с использованием CUDA runtime API представлена на рисунке 6. Ключевые функции:

- cudaMalloc ((void**)&dev, N * sizeof (float)) – выделить память на GPU под N элементов типа данных float;

- cudaMemcpy (a, dev, N * sizeof (float), cudaMemcpyDeviceToHost) – скопировать результаты из памяти GPU (DRAM) в память CPU (N элементов);
- cudaFree (dev) – освободить память GPU;
- kernel<<<dim3((N/512),1), dim3(512,1)>>> (dev) – запустить N нитей блоками по 512 нитей, выполняемая на нити функция – kernel, массив данных – dev.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include <math.h>
#define          N        (1024*1024)

__global__ void kernel ( float * data )
{
  int   idx = blockIdx.x * blockDim.x + threadIdx.x;
  float x   = 2.0f * 3.1415926f * (float) idx / (float) N;
  data [idx] = sinf ( sqrtf ( x ) );
}

int main(int argc, char *argv[])
{
         float *  a = (float*)malloc(N * sizeof(float));
   float * dev = nullptr;
          // выделить память на GPU
   cudaMalloc ( (void**)&dev, N * sizeof ( float ) );
   // конфигурация запуска N нитей
   kernel<<<dim3((N/512),1), dim3(512,1)>>> ( dev );
    // скопировать результаты в память CPU
   cudaMemcpy ( a, dev, N * sizeof ( float ), cudaMemcpyDeviceToHost );
   // освободить выделенную память
   cudaFree   ( dev   );
   free(a);
   for (int idx = 0; idx < N; idx++)
     printf("a[%d] = %.5f\n", idx, a[idx]);

   return 0;
}
```

Рисунок 6 – Программа «Hello world» с использованием CUDA runtime API

6 Получение информации об имеющихся GPU и их возможностях

Перед началом работы с GPU очень важно получить информацию об их возможностях. Это можно сделать с помощью функции cudaGetDeviceProperties, которая возвращает структуру cudaDeviceProp. Программный код получения информации об имеющихся GPU представлен на рисунке 8.

```
struct cudaDeviceProp {
        char name[256];             // название устройства
        size_t totalGlobalMem;  // полный объем глобальной памяти в байтах
        size_t sharedMemPerBlock; //объем разделяемой памяти в блока в байтах
        int regsPerBlock;               // количество 32-битных регистров в блоке
        int warpSize;                    // размер warpa
        size_t memPitch;               // максимальный Pitch в байтах
        int maxThreadsPerBlock;   // максимальное число активных нитей в блоке
```

```
        int maxThreadsDim[3];     // максимальный размер блока по каждому измерению
        int maxGridSize[3];  // максимальный размер сетки по каждому измерению
        size_t totalConstMem; // объем константной памяти
        int major;                  // Compute Capability, старший номер
        int minor;                  // Compute Capability, младший номер
        int clockRate;             // частота в килогерцах
        size_t textureAlignment; // выравнивание памяти для текстур
        int deviceOverlap; // можно ли осуществлять копирование || вычислениям
        int multiProcessorCount; // количество мультипроцессоров в GPU
        int kernelExecTimeoutEnabled; // 1, если есть ограничения на время выполнения
ядра
        int integrated;   //1, если GPU встроено в материнскую плату
        int canMapHostMemory; //1, если можно отображать память CPU в память CUDA
        int computeMode; // режим GPU
        int concurrentKernels; // 1, если устройство поддерживает выполнение нескольких
ядер в одном контексте;
        int ECCEnabled;
        int pciBusID; // идентификатор PCI шины
        int pciDeviceID; //  идентификатор PCI устройства
        int tccDriver; //1, при использовании TCC драйвера
    }
```

Рисунок 7 – Описание структуры cudaDeviceProp

```
int                 deviceCount;
cudaDeviceProp      devProp;

cudaGetDeviceCount ( &deviceCount );
printf              ( "Found %d devices\n", deviceCount );

for ( int device = 0; device < deviceCount; device++ )
{
            cudaGetDeviceProperties ( &devProp, device );
            printf ( "Device %d\n", device );
            printf ( "Compute capability     : %d.%d\n", devProp.major, devProp.minor );
            printf ( "Name                   : %s\n", devProp.name );
            printf ( "Total Global Memory    : %u\n", devProp.totalGlobalMem );
            printf ( "Shared memory per block: %d\n", devProp.sharedMemPerBlock );
            printf ( "Registers per block    : %d\n", devProp.regsPerBlock );
            printf ( "Warp size              : %d\n", devProp.warpSize );
            printf ( "Max threads per block  : %d\n", devProp.maxThreadsPerBlock );
            printf ( "Total constant memory  : %d\n", devProp.totalConstMem );
}
```

Рисунок 8 – Программный код получения информации об имеющихся GPU

7 Получение времени выполнения ядра на GPU

CUDA runtime API предоставляет возможность замера времени, затраченного GPU на выполнение различных операций с помощью событий CUDA. Событие – это объект типа cudaEvent_t, используемый для обозначения «точки» среди вызовов CUDA. При помощи функций для работы с событиями можно создавать и уничтожать события, привязывать к определенным местам в коде, узнавать, наступило ли данное событие, а также получить интервал времени в миллисекундах между наступлениями двух

событий. На рисунке 9 приводится пример кода, замеряющий время выполнения ядра на GPU.

```
cudaEvent_t start, stop;                    //описываем переменные типа  cudaEvent_t
float       gpuTime = 0.0f;
// создаем события начала и окончания выполнения ядра
cudaEventCreate(&start);
cudaEventCreate(&stop);
//привязываем событие start  к данному месту
cudaEventRecord(start, 0);
// вызвать ядро
kernel<<<blocks, threads>>> ( dev_a, dev_b, N, dev_c);
//привязываем событие stop  к данному месту
cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);
// запрашиваем время между событиями
cudaEventElapsedTime(&gpuTime,  start, stop);
printf("time spent executing by the GPU: %.5f ms\n", gpuTime);
// уничтожаем созданные события
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Рисунок 9 – Пример программного кода получения времени выполнения ядра

Использование CUDA в MS Visual Studio

Для использования возможностей технологии CUDA в MS Visual Studio необходимо:

вариант 1:

- создать новый проект;
- добавить к нему .cu файл;
- задать правила компиляции .cu файлов (CUDA SDK – Cuda.rules)
- подключить библиотеки: в поле Project Properties -> Linker -> General -> Additional Library установить $(CUDA_LIB_PATH);
- в поле Linker->Input->Additional Dependencies добавить cudart.lib.

вариант 2: воспользоваться проектом CUDA VS Wizard (см. рисунок 10), добавив библиотеку cudart.lib (рисунок 11).

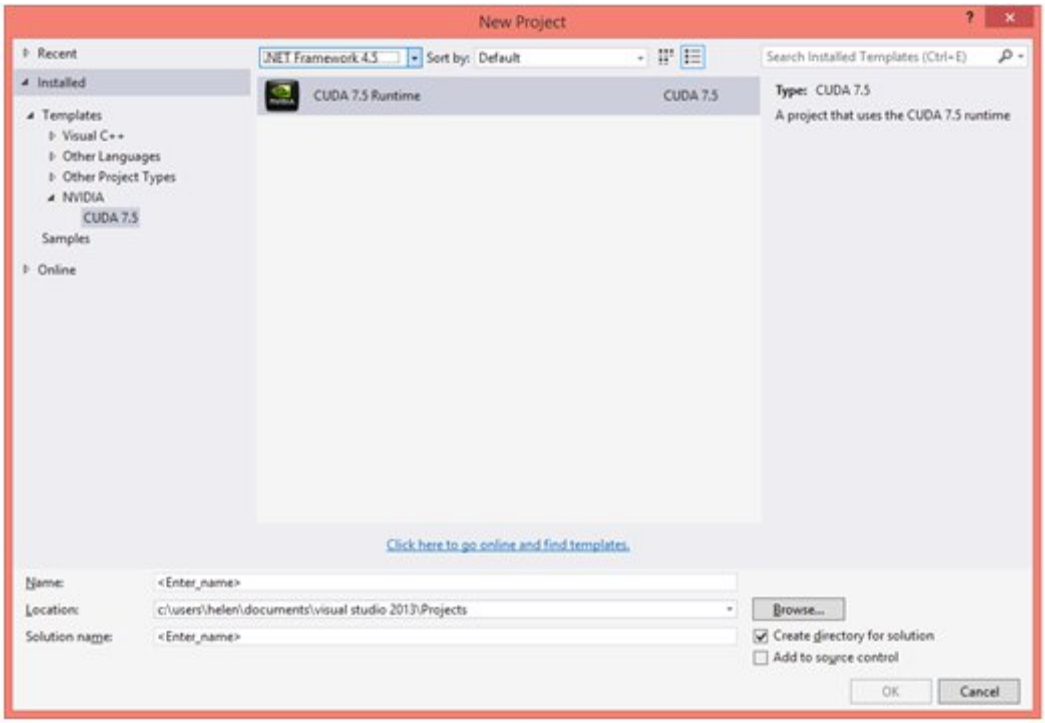

Рисунок 10 – Создание нового проекта CUDA с помощью VS Wizard

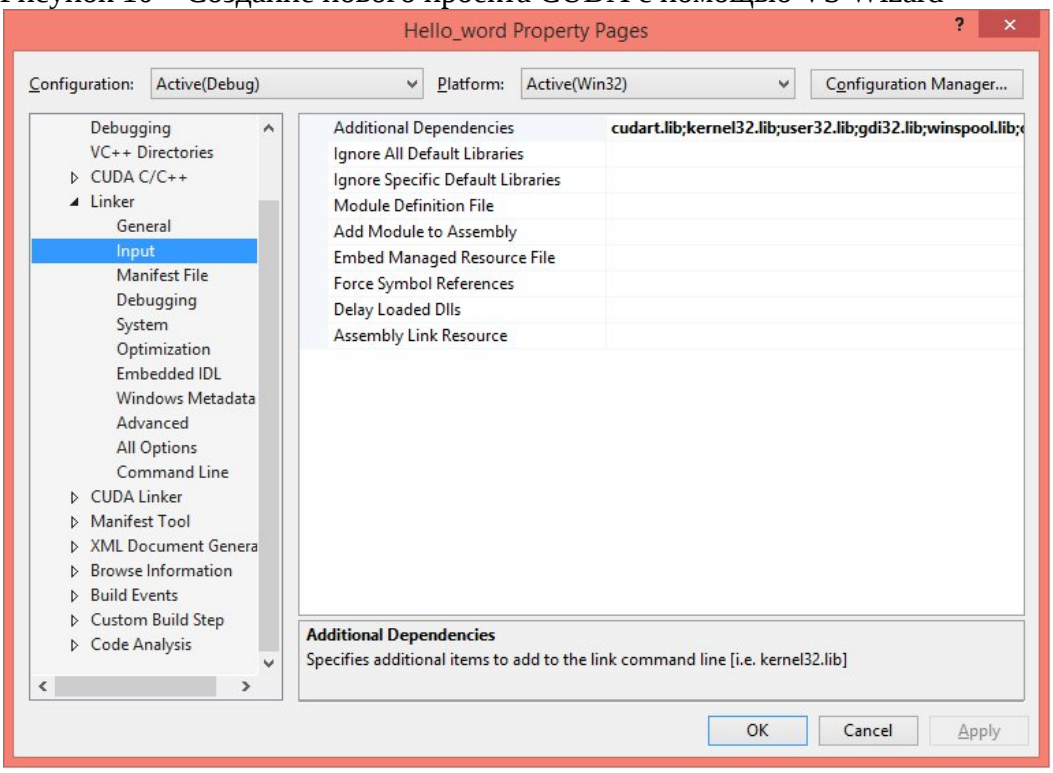

Рисунок 11 – Добавление библиотеки cudart.lib

<u>Лабораторные задания</u> (№ варианта = № компьютера%2)

Задание 1. В MS Visual Studio создать проект CUDA VS Wizard. Ознакомиться и запустить программу «Hello world». Получить информацию об устройстве. Измерить время выполнения программы. **Результаты занести в отчёт.** Запустить программу «Hello world» на всех мультипроцессорах в GPU. Измерить время выполнения программы. **Результаты занести в отчёт.**

Задание 2. Написать программу на Си с использованием CUDA runtime API в соответствии с вариантом задания. Измерить время работы программы для различных значений параметров. **Результаты занести в отчёт.** Написать программу для верификации результатов.

Вариант	**Задание**
0	Даны матрицы A и B из NxN натуральных (ненулевых) элементов (задаются случайно). Матрицы расположены в глобальной памяти. Написать программу, выполняющую перемножение двух матриц на GPU.
1	Даны два вектора A и B из N натуральных (ненулевых) элементов (задаются случайно). Вектора расположены в глобальной памяти. Написать программу, выполняющую перемножение двух векторов на GPU.

Контрольные вопросы

1. Что такое гибридное программирование?
2. Что такое CUDA?
3. Основные положения программной модели CUDA?
4. Из чего состоит программный стек CUDA?
5. Что такое ядро в CUDA?
6. Какие расширения языка Си вводятся в CUDA?
7. Какие встроенные переменные поддерживаются в CUDA и для чего они нужны?
8. Какие ограничения вводятся на функции, выполняемые на GPU?

Требования к сдаче работы

1. При домашней подготовке изучить теоретический материал по тематике лабораторной работы, представленный в списке литературы ниже, выполнить представленные примеры, занести в отчёт результаты выполнения.
2. Продемонстрировать выполнение лабораторных заданий.
3. Ответить на контрольные вопросы.
4. Показать преподавателю отчет.

Литература

1. http://www.nvidia.ru/object/cuda-parallel-computing-ru.html
2. А.В. Боресков, А.А. Харламов. Основы работы с технологией Cuda. – М: ДМК Пресс, 2010. – 232 с.