一、实验环境配置

1．实验环境

操作系统：Win10 1607 64位

显卡配置：NVIDIA GTX960 4G显存

Cuda版本：NVIDIA CUDA 9.0

IDE：Visual Studio2013

开发语言：C语言。

2．环境配置过程

2.1安装CUDA：

（1）下载和安装:

CUDA是由显卡厂商NVIDIA推出的通用并行计算架构，该架构使GPU能够解决复杂的计算问题，安装文件从NVIDIA官网下载。

下载CUDA9.0安装包后按照提示安装即可，本次安装选择精简安装模式，安装路径可选默认路径或自定义安装路径。

（2）设置环境变量：

安装CUDA后查看系统的环境变量，可以看到系统中自动添加了CUDA\_PATH和CUDA\_PATH\_V9\_0两个环境变量，接下来，还要在系统中添加以下几个环境变量.

CUDA\_SDK\_PATH = C:\ProgramData\NVIDIA Corporation\CUDA Samples\v9.0

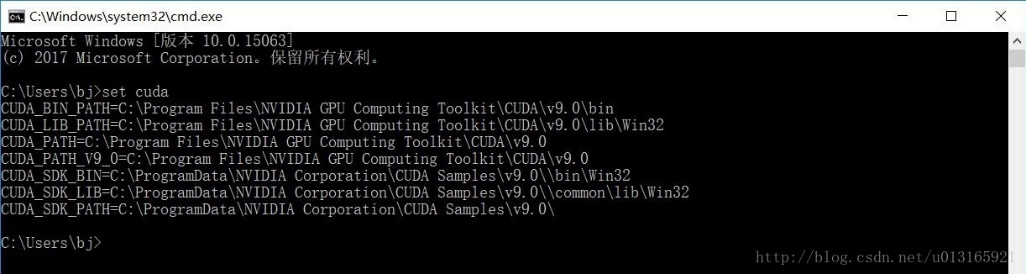
CUDA\_LIB\_PATH = %CUDA\_PATH%\lib\x64

CUDA\_BIN\_PATH = %CUDA\_PATH%\bin

CUDA\_SDK\_BIN\_PATH = %CUDA\_SDK\_PATH%\bin\win64

CUDA\_SDK\_LIB\_PATH = %CUDA\_SDK\_PATH%\common\lib\x64

添加完成后，在cmd中查看环境变量：

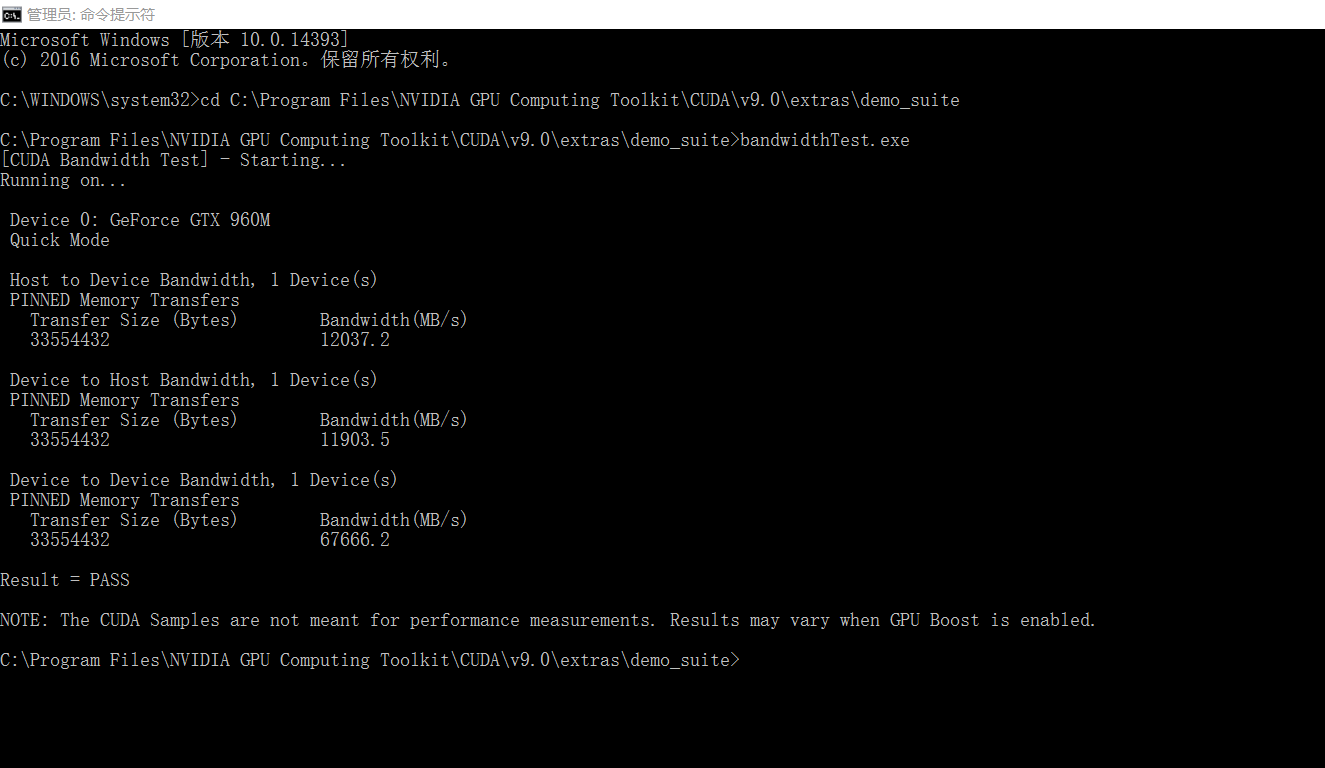


（3）监测cuda是否安装成功

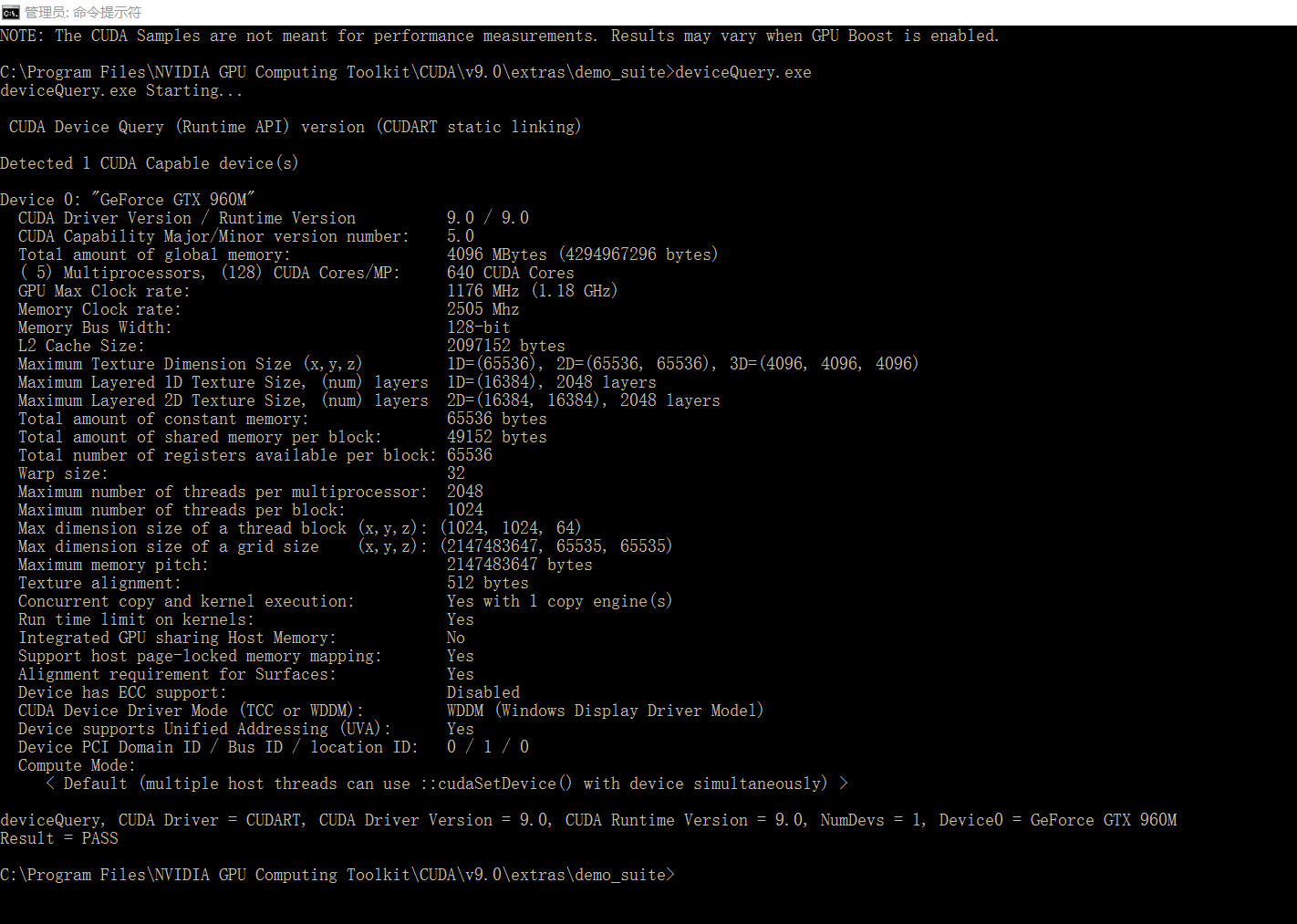
启动cmd DOS命令窗口 ，进入CUDA的安装目录下

cd C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.0\extras\demo\_suite

执行bandwidthTest.exe

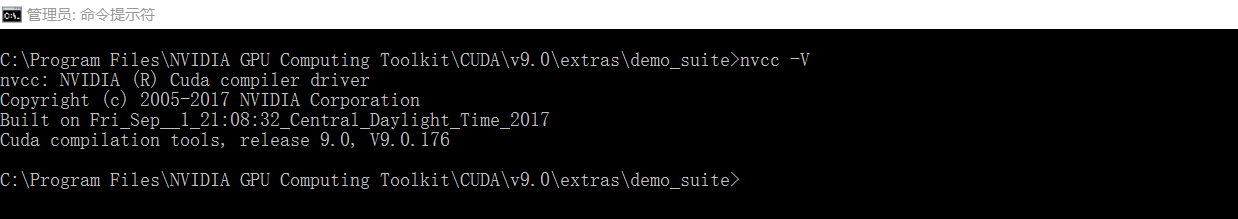


再执行deviceQuery.exe



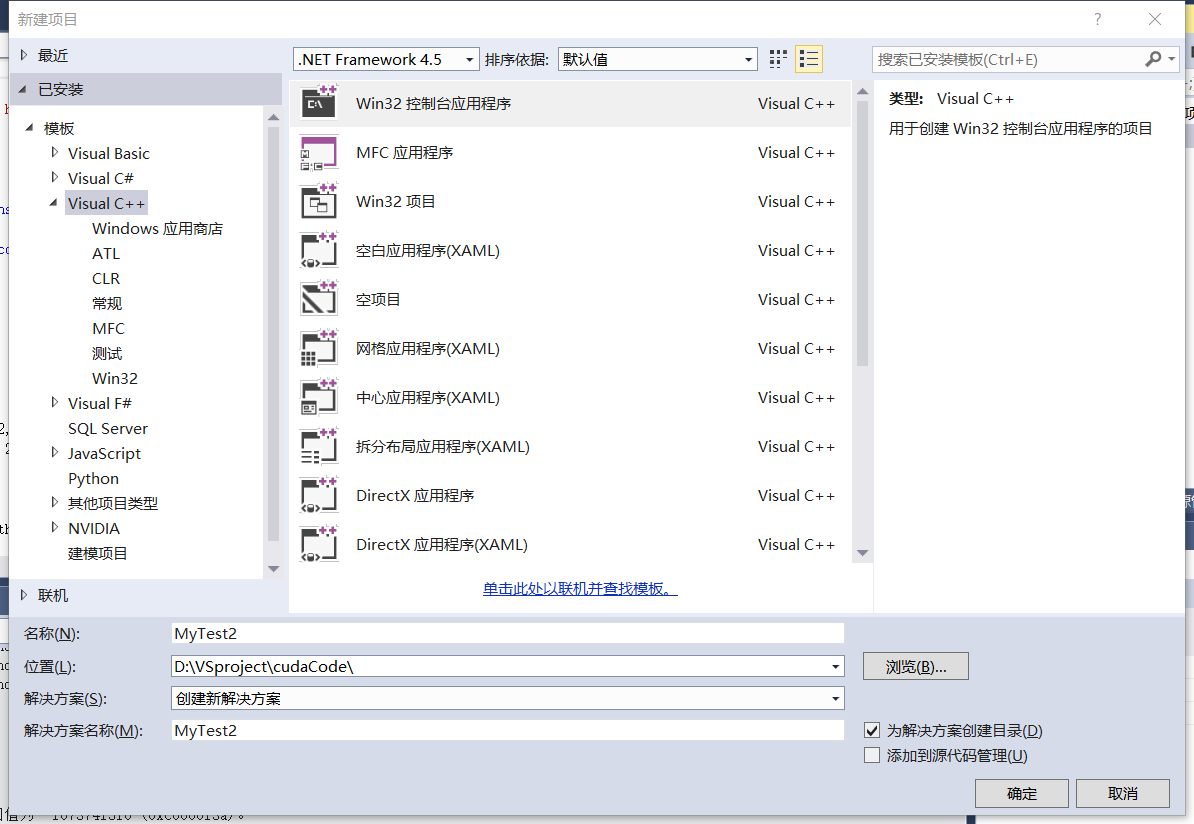
Result=PASS，说明CUDA安装成功。

打开命令窗口，输入：nvcc –V，可以查看CUDA安装信息



2.2.配置VS2013

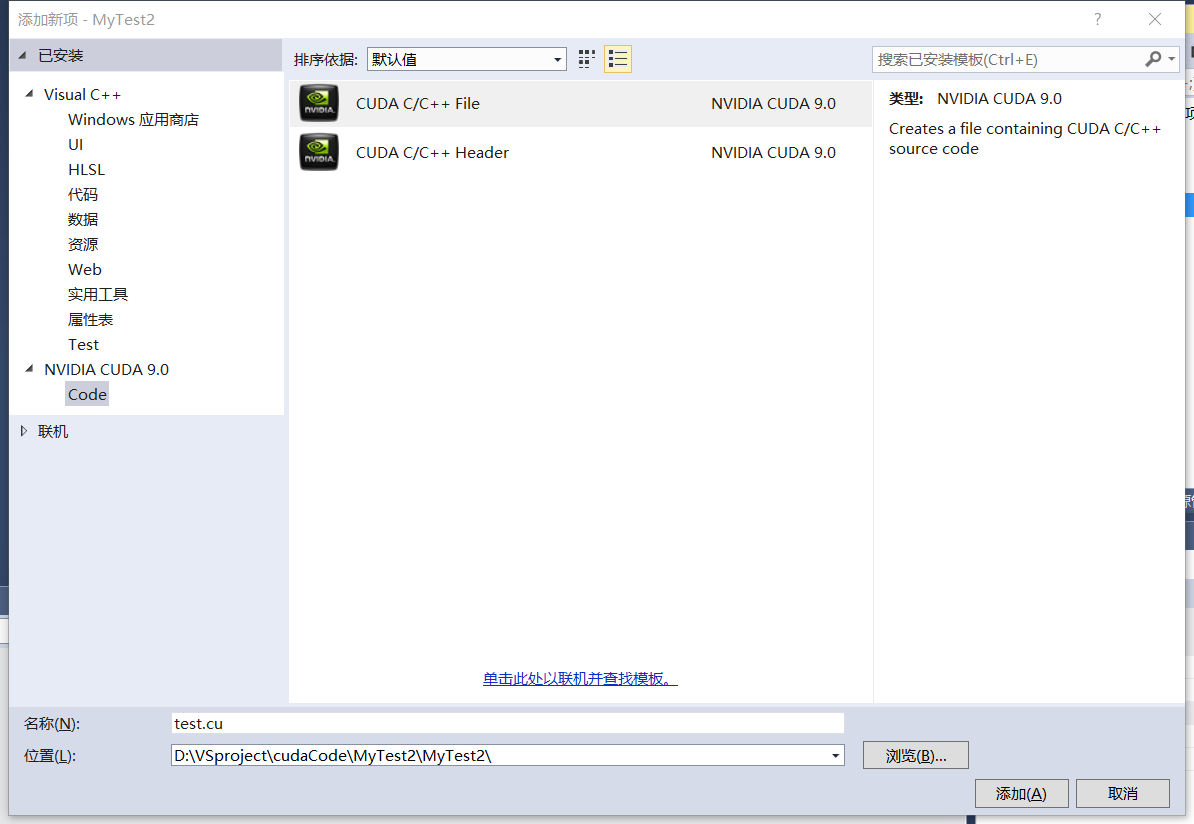
（1）启动VS2013 ，新建一个win32的控制台工程。





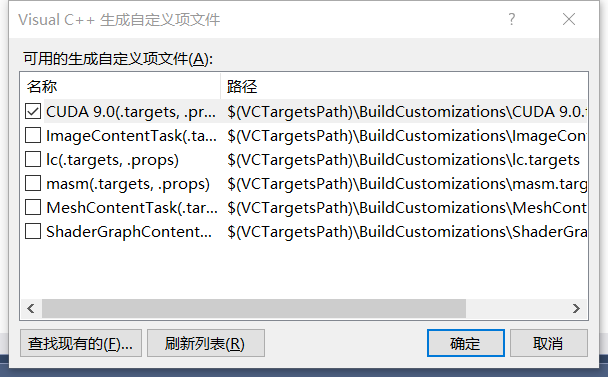
（2）新建CUDA文件

右键源文件文件夹->新建项->选择cuda c/c++- Header>新建一个以.cu结尾的文件

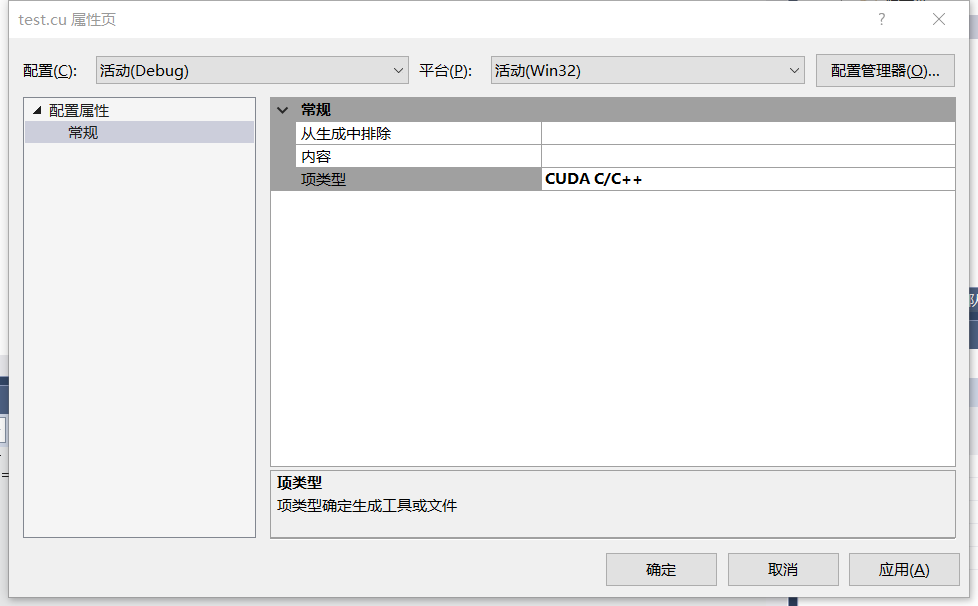


（3）生成CUDA

右键项目名-》生成依赖项-》生成自定义-》选择cuda生成

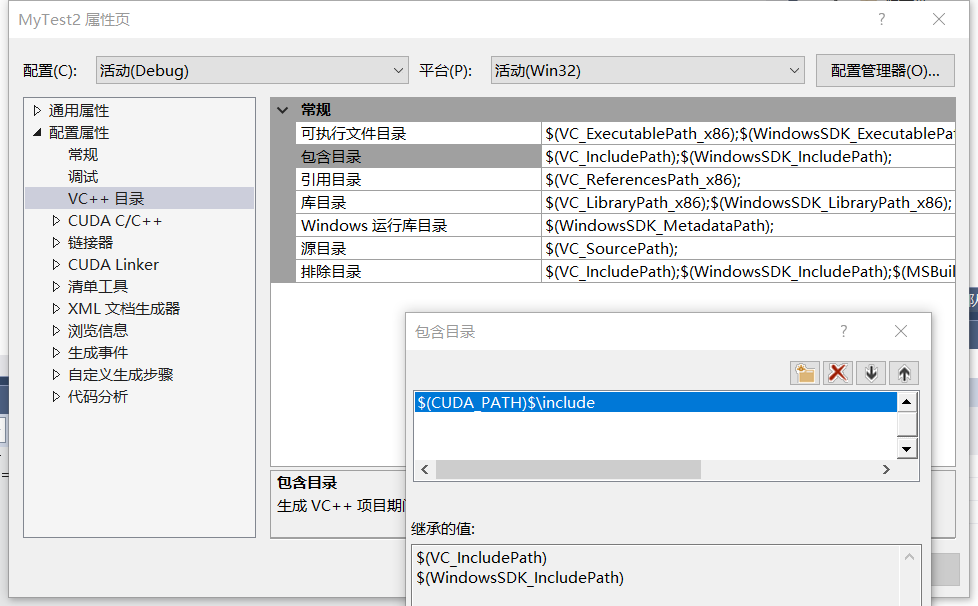


（4）右键kernal.cu-》属性-》选择cuda c/c++编译器



（5）添加包含目录：

$(CUDA\_PATH)$\include

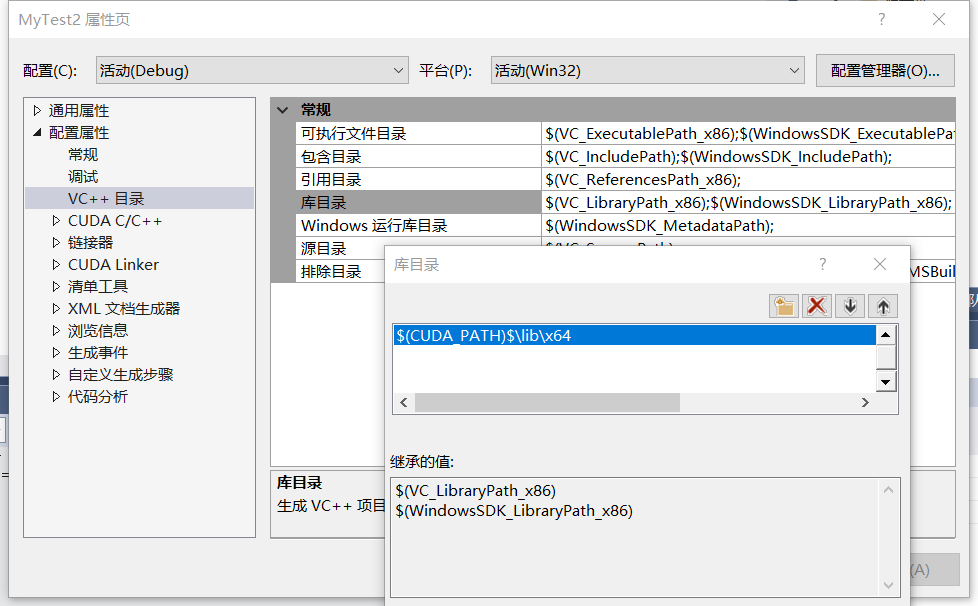


（6）库目录配置

1.VC++目录–>库目录

2.添加库目录：

$(CUDA\_PATH)$\lib\x64



3.依赖项

配置属性–>连接器–>输入–>附加依赖项

添加库文件：

cublas.lib

cuda.lib

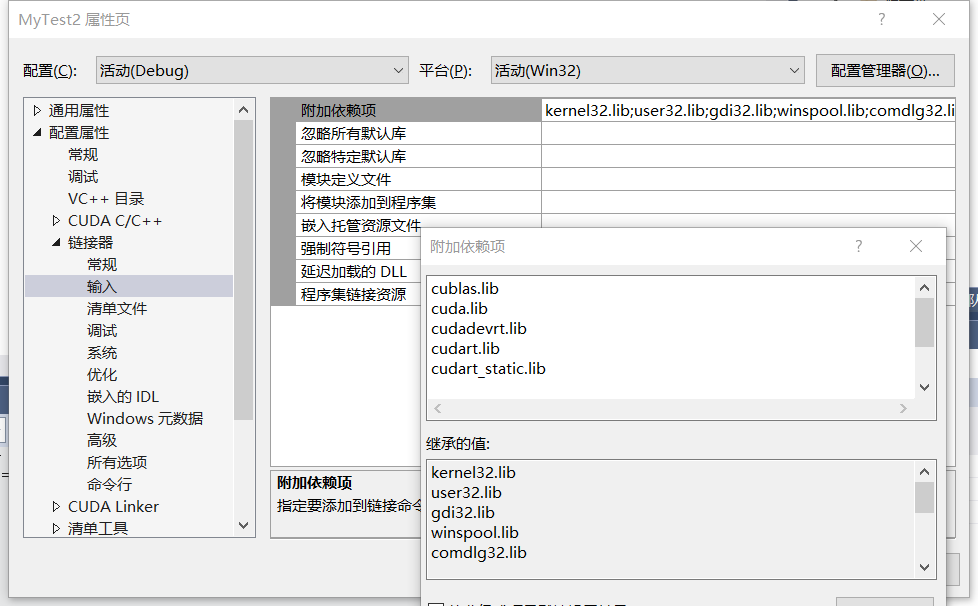
cudadevrt.lib

cudart.lib

cudart\_static.lib

nvcuvid.lib

OpenCL.lib



3 . 遇到的问题及解决方案

（1）下载cuda的版本要和cudnn相匹配，并且要下载与电脑适配的显卡驱动

（2）配置vs环境时添加nvcuvenc.lib库文件，编译时，报找不到该文件的错误。去掉后，程序也能运行

二、编程实验

1向量相加

使用向量相加测试cuda程序的运行情况

1.1程序设计清单

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <stdlib.h>

cudaError\_t addWithCuda(int \*c, const int \*a, const int \*b, unsigned int size);

\_\_global\_\_ void addKernel(int \*c, const int \*a, const int \*b)

{

int i = threadIdx.x;

c[i] = a[i] + b[i];

}

int main()

{

const int arraySize = 5;

const int a[arraySize] = { 1, 2, 3, 4, 5 };

const int b[arraySize] = { 10, 20, 30, 40, 50 };

int c[arraySize] = { 0 };

// Add vectors in parallel.

cudaError\_t cudaStatus = addWithCuda(c, a, b, arraySize);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "addWithCuda failed!");

return 1;

}

printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}\n",

c[0], c[1], c[2], c[3], c[4]);

cudaStatus = cudaDeviceReset();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaDeviceReset failed!");

return 1;

}

system("pause");

return 0;

}

// 使用CUDA来并行计算向量的辅助函数.

cudaError\_t addWithCuda(int \*c, const int \*a, const int \*b, unsigned int size)

{

int \*dev\_a = 0;

int \*dev\_b = 0;

int \*dev\_c = 0;

cudaError\_t cudaStatus;

//选择要运行的GPU

cudaStatus = cudaSetDevice(0);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");

goto Error;

}

//为三个向量分配GPU缓冲区（两个输入，一个输出） .

cudaStatus = cudaMalloc((void\*\*)&dev\_c, size \* sizeof(int));

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

cudaStatus = cudaMalloc((void\*\*)&dev\_a, size \* sizeof(int));

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

cudaStatus = cudaMalloc((void\*\*)&dev\_b, size \* sizeof(int));

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

//将输入向量从主机内存复制到GPU缓冲区.

cudaStatus = cudaMemcpy(dev\_a, a, size \* sizeof(int), cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

cudaStatus = cudaMemcpy(dev\_b, b, size \* sizeof(int), cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

// 在GPU上启动一个内核，每个元素都有一个线程.

addKernel<<<1, size>>>(dev\_c, dev\_a, dev\_b);

//检查启动内核的错误

cudaStatus = cudaGetLastError();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaStatus));

goto Error;

}

// cudadevicesyn同步等待内核完成，并返回在启动过程中遇到的错误.

cudaStatus = cudaDeviceSynchronize();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addKernel!\n", cudaStatus);

goto Error;

}

//从GPU缓冲区复制输出矢量到主机内存.

cudaStatus = cudaMemcpy(c, dev\_c, size \* sizeof(int), cudaMemcpyDeviceToHost);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

Error:

cudaFree(dev\_c);

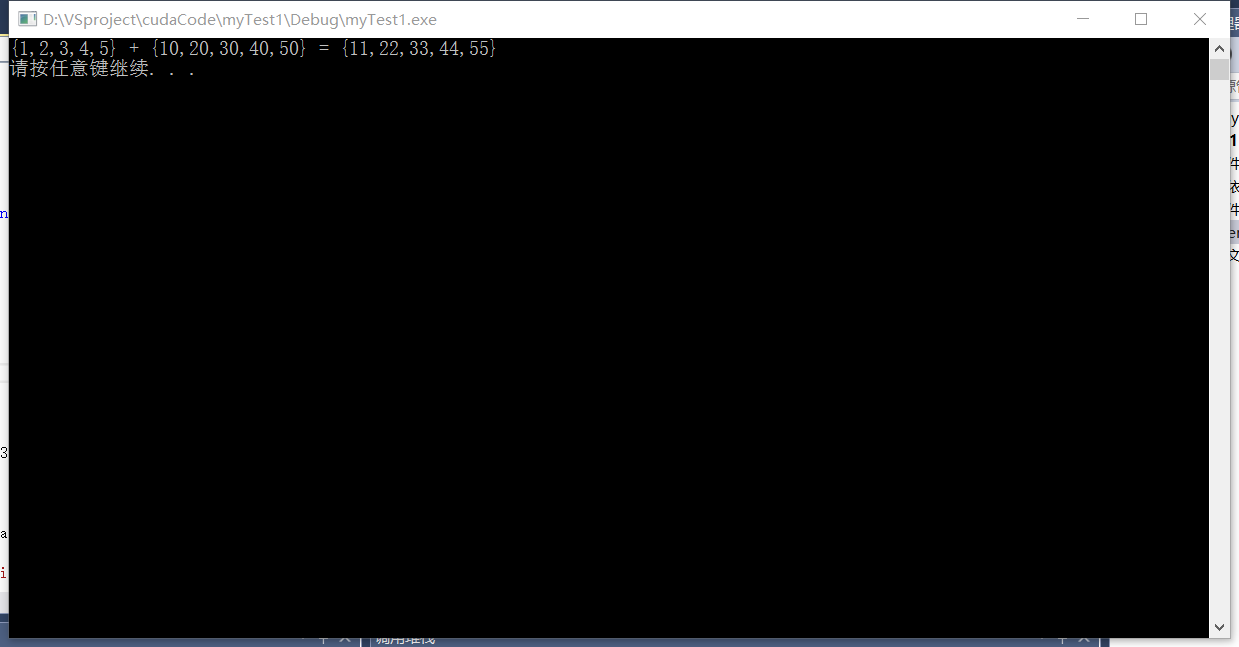
cudaFree(dev\_a);

cudaFree(dev\_b);

return cudaStatus;

}

1.2程序运行结果：



说明配置可用，测试完成

2基于CUDA的矩阵相乘算法

2.1程序设计清单

#include "cuda\_runtime.h"

#include "cublas\_v2.h"

#include <time.h>

#include <iostream>

using namespace std;

// 定义测试矩阵的维度

int const M = 5;

int const N = 10;

int main()

{

// 定义状态变量

cublasStatus\_t status;

// 在 内存 中为将要计算的矩阵开辟空间

float \*h\_A = (float\*)malloc (N\*M\*sizeof(float));

float \*h\_B = (float\*)malloc (N\*M\*sizeof(float));

// 在 内存 中为将要存放运算结果的矩阵开辟空间

float \*h\_C = (float\*)malloc (M\*M\*sizeof(float));

// 为待运算矩阵的元素赋予 0-10 范围内的随机数

for (int i=0; i<N\*M; i++) {

h\_A[i] = (float)(rand()%10+1);

h\_B[i] = (float)(rand()%10+1);

}

// 打印待测试的矩阵

cout << "矩阵 A :" << endl;

for (int i=0; i<N\*M; i++){

cout << h\_A[i] << " ";

if ((i+1)%N == 0) cout << endl;

}

cout << endl;

cout << "矩阵 B :" << endl;

for (int i=0; i<N\*M; i++){

cout << h\_B[i] << " ";

if ((i+1)%M == 0) cout << endl;

}

cout << endl;

/\*

\*\* GPU 计算矩阵相乘

\*/

// 创建并初始化 CUBLAS 库对象

cublasHandle\_t handle;

status = cublasCreate(&handle);

if (status != CUBLAS\_STATUS\_SUCCESS)

{

if (status == CUBLAS\_STATUS\_NOT\_INITIALIZED) {

cout << "CUBLAS 对象实例化出错" << endl;

}

getchar ();

return EXIT\_FAILURE;

}

float \*d\_A, \*d\_B, \*d\_C;

// 在 显存 中为将要计算的矩阵开辟空间

cudaMalloc (

(void\*\*)&d\_A, // 指向开辟的空间的指针

N\*M \* sizeof(float) //　需要开辟空间的字节数

);

cudaMalloc (

(void\*\*)&d\_B,

N\*M \* sizeof(float)

);

// 在 显存 中为将要存放运算结果的矩阵开辟空间

cudaMalloc (

(void\*\*)&d\_C,

M\*M \* sizeof(float)

);

// 将矩阵数据传递进 显存 中已经开辟好了的空间

cublasSetVector (

N\*M, // 要存入显存的元素个数

sizeof(float), // 每个元素大小

h\_A, // 主机端起始地址

1, // 连续元素之间的存储间隔

d\_A, // GPU 端起始地址

1 // 连续元素之间的存储间隔

);

cublasSetVector (

N\*M,

sizeof(float),

h\_B,

1,

d\_B,

1

);

// 同步函数

cudaThreadSynchronize();

// 传递进矩阵相乘函数中的参数，具体含义请参考函数手册。

float a=1; float b=0;

// 矩阵相乘。该函数必然将数组解析成列优先数组

cublasSgemm (

handle, // blas 库对象

CUBLAS\_OP\_T, // 矩阵 A 属性参数

CUBLAS\_OP\_T, // 矩阵 B 属性参数

M, // A, C 的行数

M, // B, C 的列数

N, // A 的列数和 B 的行数

&a, // 运算式的 α 值

d\_A, // A 在显存中的地址

N, // lda

d\_B, // B 在显存中的地址

M, // ldb

&b, // 运算式的 β 值

d\_C, // C 在显存中的地址(结果矩阵)

M // ldc

);

// 同步函数

cudaThreadSynchronize();

// 从 显存 中取出运算结果至 内存中去

cublasGetVector (

M\*M, // 要取出元素的个数

sizeof(float), // 每个元素大小

d\_C, // GPU 端起始地址

1, // 连续元素之间的存储间隔

h\_C, // 主机端起始地址

1 // 连续元素之间的存储间隔

);

// 打印运算结果

cout << "计算结果的转置 ( (A\*B)的转置 )：" << endl;

for (int i=0;i<M\*M; i++){

cout << h\_C[i] << " ";

if ((i+1)%M == 0) cout << endl;

}

// 清理掉使用过的内存

free (h\_A);

free (h\_B);

free (h\_C);

cudaFree (d\_A);

cudaFree (d\_B);

cudaFree (d\_C);

// 释放 CUBLAS 库对象

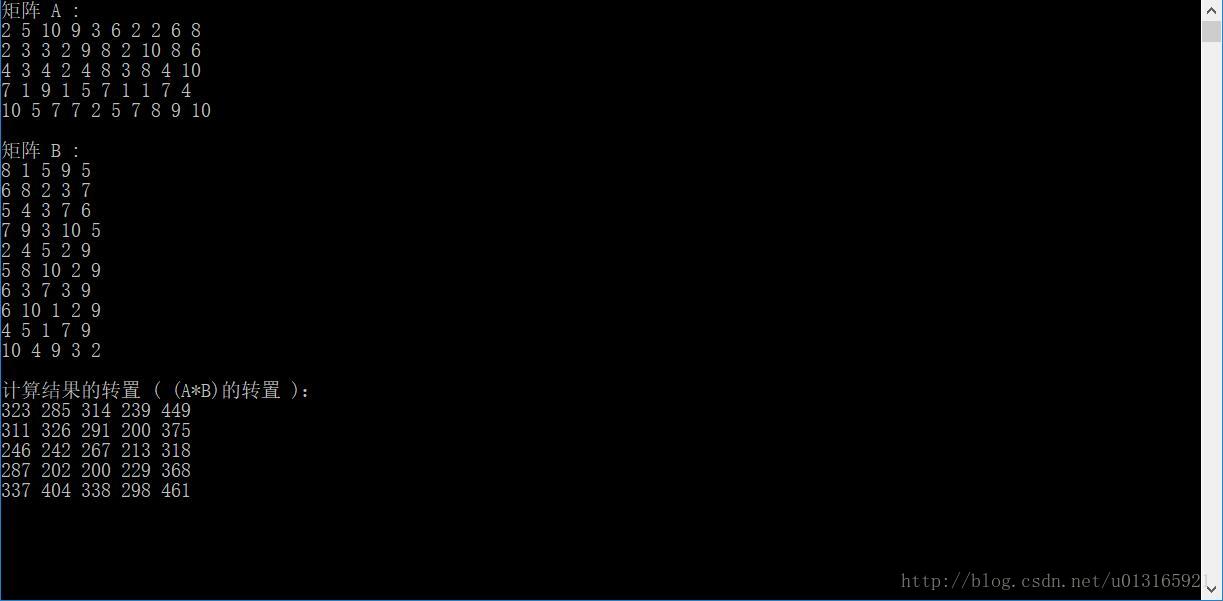
cublasDestroy (handle);

getchar();

return 0;

}

2.2程序运行结果：



3遇到的问题和思考总结

3.1遇到的问题

生成项目的时候报缺少“unistd.h”库的错误，网上查找发现：unistd.h是linux下的，windows不支持linux的系统调用。头文件unistd.h是Linux/Unix的系统调用，包含了许多UNIX系统服务函数原型，如open、read、write、\_exit、getpid等函数。在linux下能够编译通过的包含此头文件的程序，在VC下编译时出现了如下问题fatal error C1083: Cannot open include file: 'unistd.h': No such file or directory

　　写一个unistd.h文件即可，其内容如下：

　　#ifndef \_UNISTD\_H

　　#define \_UNISTD\_H

#include <io.h>

#include <process.h>

#endif /\* \_UNISTD\_H \*/

3.2思考总结

通过本实验，还提高了对Cuda的使用熟练度，更深层次的了解了Cuda的使用原理以及使用方法。同时更让我知道了自己的不足之处，例如：输入马虎出错、步骤遗漏等。这为我以后的项目开发都有一定的警醒作用，督促自己需要更加认真。同时也感谢这个过程中老师和同学的悉心帮助，这让我在本次实验的进行更加顺利，在此谢谢大家。