**Лабораторная работа №1**

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

**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. Запустить программу«Hello world» на всех мультипроцессорах в GPU. Измерить время выполнения программы.

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

\_\_global\_\_ void cuda\_hello() {

printf("Hello World from GPU!\n");

}

int main() {

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);

printf("maxBlocksPerMultiProcessor : %d\n", devProp.maxBlocksPerMultiProcessor);

printf("maxThreadsPerMultiProcessor : %d\n", devProp.maxThreadsPerMultiProcessor);

printf("Multiprocessor count : %d\n", devProp.multiProcessorCount);

}

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

cuda\_hello<<<1, 1>>>();

//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);

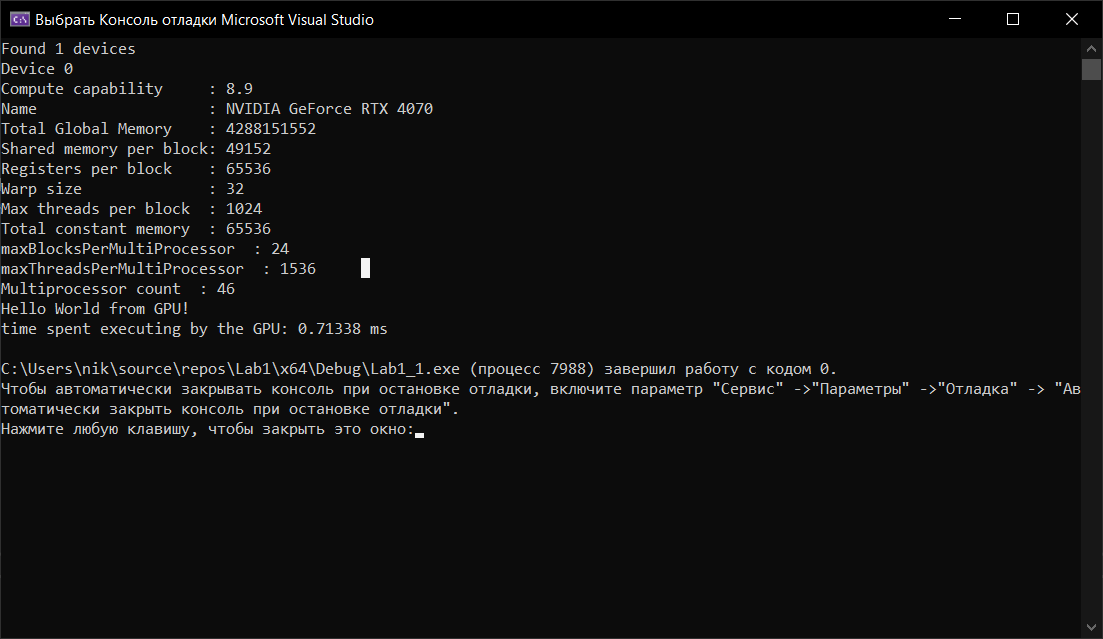
// уничтожаем созданные события

cudaEventDestroy(start);

cudaEventDestroy(stop);

return 0;

}



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

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

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#define VAR 0;

void checkError(cudaError\_t cudaStatus, char\* msg, int\* dev\_a, int\* dev\_b, int\* dev\_c);

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

void printMatrix(int\* matrix, int size);

void printArray(int\* a, int size);

// вариант 1

#if VAR == 1

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

{

int i = blockIdx.x \* blockDim.x + threadIdx.x;

c[i] = a[i] \* b[i];

}

#else

\_\_global\_\_ void multiplyKernel(int\* c, int\* a, int\* b, unsigned int size)

{

int index = blockIdx.y \* blockDim.x \* blockDim.y \* gridDim.x + blockIdx.x \* blockDim.y + threadIdx.y \* blockDim.x \* gridDim.x + threadIdx.x;

int i = blockIdx.x \* blockDim.x + threadIdx.x;

int j = blockIdx.y \* blockDim.y + threadIdx.y;

index -= (blockDim.x \* gridDim.x - size) \* j;

if (i >= size || j >= size)

return;

for (size\_t k = 0; k < size; k++)

{

c[index] += a[j \* size + k] \* b[k \* size + i];

}

}

#endif

int main(int argc, char\* argv[])

{

// вариант 1

#if VAR == 1

//printf("Variant 1\n");

printf("Enter array size: ");

int arraySize;

scanf("%d", &arraySize);

int\* a = (int\*)malloc(arraySize \* sizeof(int));

int\* b = (int\*)malloc(arraySize \* sizeof(int));

int\* c = (int\*)malloc(arraySize \* sizeof(int));

for (size\_t i = 0; i < arraySize; i++)

{

a[i] = rand() % 100 + 1;

b[i] = rand() % 100 + 1;

c[i] = 0;

}

#else

// вариант 0

//printf("Variant 0\n");

int arraySize;

arraySize = atoi(argv[1]);

int\* a = (int\*)malloc(arraySize \* arraySize \* sizeof(int));

int\* b = (int\*)malloc(arraySize \* arraySize \* sizeof(int));

int\* c = (int\*)malloc(arraySize \* arraySize \* sizeof(int));

for (size\_t i = 0; i < arraySize; i++)

{

for (size\_t j = 0; j < arraySize; j++)

{

a[i \* arraySize + j] = rand() % 100 + 1;

b[i \* arraySize + j] = rand() % 100 + 1;

c[i \* arraySize + j] = 0;

}

}

#endif

// Add vectors in parallel.

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

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "addWithCuda failed!");

return 1;

}

// cudaDeviceReset must be called before exiting in order for profiling and

// tracing tools such as Nsight and Visual Profiler to show complete traces.

cudaStatus = cudaDeviceReset();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaDeviceReset failed!");

return 1;

}

#if VAR == 1

printf("A: ");

printArray(a, arraySize);

printf("\nB: ");

printArray(b, arraySize);

printf("\nC: ");

int sum = 0;

for (int i = 0; i < arraySize; i++) {

sum += c[i];

}

printf("%d", sum);

#else

/\*printf("A:\n");

printMatrix(a, arraySize);

printf("B:\n");

printMatrix(b, arraySize);

printf("C:\n");

printMatrix(c, arraySize);\*/

for (int i = 0; i < arraySize; i++)

for (int j = 0; j < arraySize; j++)

{

int test = 0;

for (int k = 0; k < arraySize; k++)

test += a[i \* arraySize + k] \* b[k \* arraySize + j];

if (test != c[i \* arraySize + j]) {

printf("wrong answer");

}

}

#endif

return 0;

}

// Helper function for using CUDA to add vectors in parallel.

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

{

cudaDeviceProp devProp;

cudaGetDeviceProperties(&devProp, 0);

int\* dev\_a = 0;

int\* dev\_b = 0;

int\* dev\_c = 0;

cudaError\_t cudaStatus;

#if VAR == 1

int dev\_size = size;

#else

int dev\_size = size \* size;

#endif

// Choose which GPU to run on, change this on a multi-GPU system.

cudaStatus = cudaSetDevice(0);

checkError(cudaStatus, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?", dev\_a, dev\_b, dev\_c);

// Allocate GPU buffers for three vectors (two input, one output) .

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

checkError(cudaStatus, "cudaMemcpy failed!", dev\_a, dev\_b, dev\_c);

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

checkError(cudaStatus, "cudaMemcpy failed!", dev\_a, dev\_b, dev\_c);

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

checkError(cudaStatus, "cudaMemcpy failed!", dev\_a, dev\_b, dev\_c);

// Copy input vectors from host memory to GPU buffers.

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

checkError(cudaStatus, "cudaMemcpy failed!", dev\_a, dev\_b, dev\_c);

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

checkError(cudaStatus, "cudaMemcpy failed!", dev\_a, dev\_b, dev\_c);

cudaStatus = cudaMemcpy(dev\_c, c, dev\_size \* sizeof(int), cudaMemcpyHostToDevice);

checkError(cudaStatus, "cudaMemcpy failed!", dev\_a, dev\_b, dev\_c);

// Launch a kernel on the GPU with one thread for each element.

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);

#if VAR == 1

dim3 threads;

if (dev\_size < devProp.maxThreadsPerBlock) {

threads = dim3(dev\_size, 1, 1);

}

else {

threads = dim3(devProp.maxThreadsPerBlock, 1, 1);

}

dim3 blocks(ceil(dev\_size / threads.x), 1, 1);

multiplyKernel << <blocks, threads >> > (dev\_c, dev\_a, dev\_b);

#else

double d;

if (size < sqrt(devProp.maxThreadsPerBlock)) {

d = size;

}

else {

d = sqrt(devProp.maxThreadsPerBlock);

}

d = 16;

dim3 threads(d, d, 1);

dim3 blocks(ceil(size / d), ceil(size / d), 1);

multiplyKernel<<<blocks, threads>>>(dev\_c, dev\_a, dev\_b, size);

#endif

// Check for any errors launching the kernel

cudaStatus = cudaGetLastError();

checkError(cudaStatus, "multiplyKernel launch failed: %s\n", dev\_a, dev\_b, dev\_c);

// cudaDeviceSynchronize waits for the kernel to finish, and returns

// any errors encountered during the launch.

cudaStatus = cudaDeviceSynchronize();

checkError(cudaStatus, "cudaDeviceSynchronize returned error code %d after launching multiplyKernel!\n", dev\_a, dev\_b, dev\_c);

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);

// Copy output vector from GPU buffer to host memory.

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

checkError(cudaStatus, "cudaMemcpy failed!", dev\_a, dev\_b, dev\_c);

return cudaStatus;

}

void checkError(cudaError\_t cudaStatus, char\* msg, int\* dev\_a, int\* dev\_b, int\* dev\_c)

{

if (cudaStatus == cudaSuccess)

return;

printf(msg);

cudaFree(dev\_c);

cudaFree(dev\_a);

cudaFree(dev\_b);

}

void printArray(int\* a, int size)

{

for (int i = 0; i < size; i++)

printf("%d ", a[i]);

}

void printMatrix(int\* matrix, int size)

{

for (int i = 0; i < size; i++)

{

for (int j = 0; j < size; j++)

printf("%d ", matrix[i \* size + j]);

printf("\n");

}

printf("\n");

}

Для проверки зависимости времени выполнения перемножения матриц на GPU от размерности матриц был написан batch-скрипт:

set /a size=1

:do

echo %size% >> text.txt

Lab1\_2.exe %size% >> text.txt

set /a size += 100

:while

if %size% geq 2000 (goto next) else (goto do)

:next

Результат выполнения скрипта:

1

time spent executing by the GPU: 0.03200 ms

101

time spent executing by the GPU: 0.12934 ms

201

time spent executing by the GPU: 0.23350 ms

301

time spent executing by the GPU: 0.65462 ms

401

time spent executing by the GPU: 1.24925 ms

501

time spent executing by the GPU: 2.32477 ms

601

time spent executing by the GPU: 3.74614 ms

701

time spent executing by the GPU: 5.75920 ms

801

time spent executing by the GPU: 8.76138 ms

901

time spent executing by the GPU: 12.04682 ms

1001

time spent executing by the GPU: 17.13197 ms

1101

time spent executing by the GPU: 21.62048 ms

1201

time spent executing by the GPU: 28.03622 ms

1301

time spent executing by the GPU: 35.24906 ms

1401

time spent executing by the GPU: 46.92128 ms

1501

time spent executing by the GPU: 53.41181 ms

1601

time spent executing by the GPU: 65.28070 ms

1701

time spent executing by the GPU: 77.70701 ms

1801

time spent executing by the GPU: 97.48870 ms

1901

time spent executing by the GPU: 107.96179 ms

Matlab:

hold on, grid on

title('Лаб №1.2 Перемножение матриц:')

xlabel('Размерность'), ylabel('Время выполнения, мс')

x\_cuda\_size = [1, 101, 201, 301, 401, 501, 601, 701, 801, 901, 1001, 1101, ...

1201, 1301, 1401, 1501, 1601, 1701, 1801, 1901];

y\_cuda\_time = [0.03200, 0.12934, 0.23350, 0.65462, 1.24925, 2.32477, ...

3.74614, 5.75920, 8.76138, 12.04682, 17.13197, 21.62048, 28.03622, ...

35.24906, 46.92128, 53.41181, 65.28070, 77.70701, 97.48870, 107.96179];

plot(x\_cuda\_size, y\_cuda\_time)

