Донецкий Национальный Технический Университет

Лабораторная работа № 3

«Знакомство с работой с памятью в GPGPU

на базе технологии Nvidia Cuda»

Выполнил:

ст. группы ИПОИм -17

Лысенко А. С.

Проверила:

доцент каф. ПМИ

Назарова И. А.

Покровск 2017

**Задание к лабораторной работе**

1. Реализовать программу, которая описана в пункте 3 теоретической части. И выполнить оценки скорости выполнения вычислений при различных данных.

2. Реализовать параметры программы, которые выполняют умножения матрицы на вектор и транспонирования матриц.

1. Оценки скорости выполнения вычислений данных на CPU

Для подсчёта времени, использую функцию clock() которая подключается с помощью библиотеки <ctime> и возвращает время в мс.

if (mode == CPU) {

// int start = GetTickCount();

unsigned int start\_time = clock();//начало отсчета времени

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

transposeMatrixCPU(inputMatrix, outputMatrix, width, height);

}

unsigned int end\_time = clock();//конец отсчета времени

unsigned int search\_time = end\_time - start\_time; // подсчёт результата

cout << "Time: " << search\_time << " ms" << endl; //вывод результата

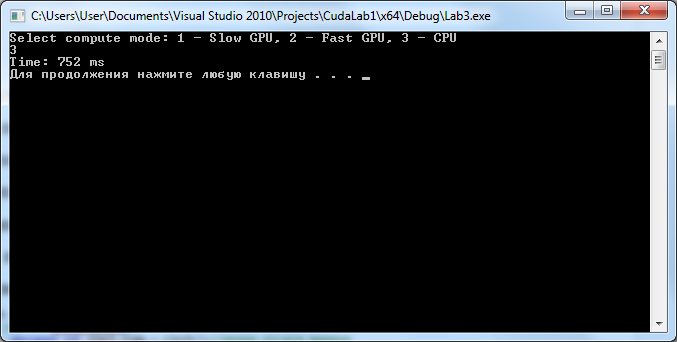


Рис. 1 - Время выполнения транспонирования матрицы на CPU

Оценка скорости выполнения вычислений данных на GPU без использования общей памяти вычисляю с помощью cuda runtime api Event Management. Где для начало, создаются event's, затем синхронизируются и вычисляется время по функции cudaEventElapsedTime(&gpuTime, start, stop);

, где &gpuTime – ссылка на переменную в которую записываем время

start, stop – начальный и конечный event's

float gpuTime;

cudaEvent\_t start , stop;

// Створюємо події для синхронізації і виміру часу роботи GPU

cudaEventCreate (&start);

cudaEventCreate (&stop);

// Використовується функція без суспільної пам'яті

if (mode == GPU\_SLOW) {

//Отмечаем старт расчетов на GPU

cudaEventRecord(start, 0);

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

transposeMatrixGlobal<<<gridSize, blockSize>>>(devInputMatrix, devOutputMatrix, width, height);

}

cudaEventRecord (stop, 0 );

cudaEventSynchronize(stop); /конец отсчета времени

cudaEventElapsedTime(&gpuTime, start, stop); // подсчёт результата

printf("Time: %.2f ms\n", gpuTime); //вывод результата

}

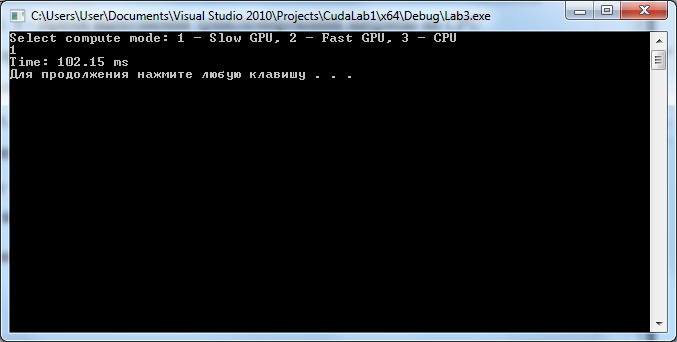


Рис. 2 - Время выполнения транспонирования матрицы на GPU без использования общей памяти

// Використовується функція з суспільною пам'яттю

else if (mode == GPU\_FAST) {

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

//Отмечаем старт расчетов на GPU

cudaEventRecord(start, 0);

transposeMatrixShared<<<gridSize, blockSize>>>(devInputMatrix, devOutputMatrix, width, height);

}

cudaEventRecord (stop, 0 );

cudaEventSynchronize(stop); //конец отсчета времени

cudaEventElapsedTime(&gpuTime, start, stop); // подсчёт результата

printf("Time: %.2f ms\n", gpuTime); //вывод результата

}

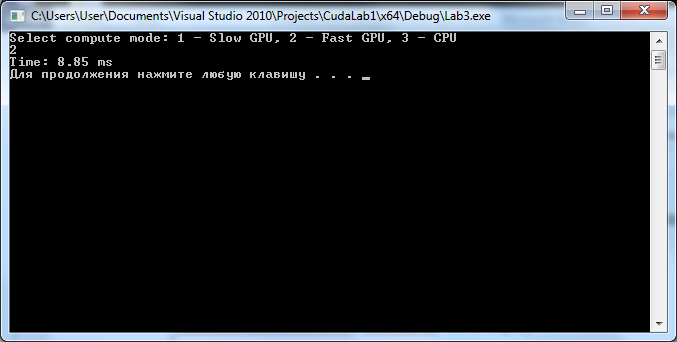


Рис. 3 - Время выполнения транспонирования матрицы на GPU с использования общей памяти

2. Параметры программы умножения матрицы на вектор.

Реализация в ядре

\_\_global\_\_ void MatrVectMul(float \*d\_c, float \*d\_a, float \*d\_b, int Size)

{

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

int k;

d\_c[i]=0;

for (k=0;k<Size;k++)

d\_c[i]+=d\_a[i\*Size+k]\*d\_b[k];

}

Оценка скорости выполнения вычислений данных на GPU без использования общей памяти вычисляю с помощью cuda runtime api Event Management.

// установка количества блоков

dim3 grid((Size+255)/256, 1, 1);

// установка количества потоков в блоке

dim3 threads(256, 1, 1);

// вызов функции

cudaEventRecord (start, 0 );

MatrVectMul<<< grid, threads >>> (d\_c, d\_a, d\_b,Size);

cudaEventRecord (stop, 0 );

// копирование из видеопамяти в оперативную память

cudaMemcpy(h\_c, d\_c, sizeof(float)\*Size,cudaMemcpyDeviceToHost) ;

cudaEventElapsedTime(&gpuTime, start, stop);

printf("Время: %.2f мс\n", gpuTime);

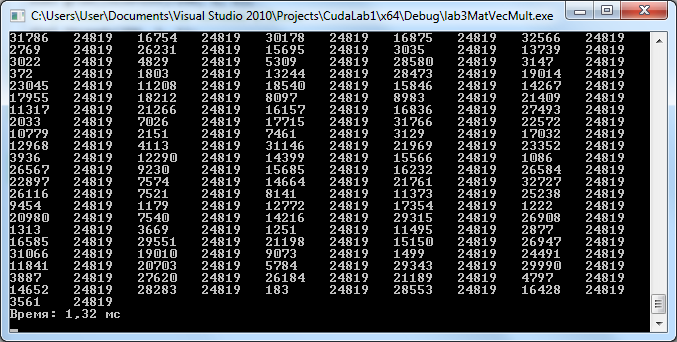


Рис. 4 - Время выполнения умножения матрицы на вектор на GPU без использования общей памяти

Инициализация массивов происходи с помощью встроенной функции rand( );

for (int i=0;i<Size;i++) // инициализация массивов a и b

{

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

{

h\_a[i\*Size+k]=rand();

}

h\_b[i]=rand();

}

Перемножение выполняется в ядре функции \_\_global\_\_ void MatrVectMul(float \*d\_c, float \*d\_a, float \*d\_b, int Size)

, где float \*d\_c – указатель на результативный массив

float \*d\_a - указатель на массив данных

float \*d\_b - указатель на вектор данных

int Size – размер вектора данных

\_\_global\_\_ void MatrVectMul(float \*d\_c, float \*d\_a, float \*d\_b, int Size)

{

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

int k;

d\_c[i]=0;

for (k=0;k<Size;k++)

d\_c[i]+=d\_a[i\*Size+k]\*d\_b[k];

}

Программа транспонирования матрицы рассмотрена в 1 пункте, сейчас детальнее познакомимся с алгоритмом её работы.

Рассмотрим функцию транспонирования матрицы без использования общей памяти

// Функція транспонування матриці без використання глобальної пам'яті

// \* inputMatrix - покажчик на вихідну матрицю

// \* outputMatrix - покажчик на матрицю результат

// \* width - ширина вихідної матриці (вона ж висота матриці-результату)

// \* height - висота вихідної матриці (вона ж ширина матриці-результату)

\_\_global\_\_ void transposeMatrixGlobal(float\* inputMatrix, float\* outputMatrix, int width, int height)

{

// Розрахунок індексів матриці

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

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

if ((xIndex < width) && (yIndex < height)) {

// Лінійний індекс елемента рядки вихідної матриці

int inputIdx = xIndex + width \* yIndex;

// Лінійний індекс елемента стовпця матриці-результату

int outputIdx = yIndex + height \* xIndex;

// Встановлення елементу

outputMatrix[outputIdx] = inputMatrix[inputIdx];

}

}

Функция транспонирования матрицы с использования общей памяти

// Функція транспонування матриці з використанням колективної пам'яті

// \* inputMatrix - покажчик на вихідну матрицю

// \* outputMatrix - покажчик на матрицю результат

// \* width - ширина вихідної матриці (вона ж висота матриці-результату)

// \* height - висота вихідної матриці (вона ж ширина матриці-результату)

\_\_global\_\_ void transposeMatrixShared(float\* inputMatrix, float\* outputMatrix, int width, int height)

{

//int BLOCK\_DIM = width;

\_\_shared\_\_ float temp[BLOCK\_DIM][BLOCK\_DIM];

// Розрахунок індексів матриці

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

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

if ((xIndex < width) && (yIndex < height)) {

// Линейный индекс элемента строки исходной матрицы

int idx = yIndex \* width + xIndex;

//Копируем элементы исходной матрицы

temp[threadIdx.y][threadIdx.x] = inputMatrix[idx];

}

//Синхронизируем все нити в блоке

\_\_syncthreads();

xIndex = blockIdx.y \* blockDim.y + threadIdx.x;

yIndex = blockIdx.x \* blockDim.x + threadIdx.y;

if ((xIndex < height) && (yIndex < width)) {

// Линейный индекс элемента строки исходной матрицы

int idx = yIndex \* height + xIndex;

//Копируем элементы исходной матрицы

outputMatrix[idx] = temp[threadIdx.x][threadIdx.y];

}

}

Функция транспонирования матрицы которая выполняется на CPU

// Функція транспонування матриці, яка виконується на CPU

\_\_host\_\_ void transposeMatrixCPU(float \*inputMatrix, float \*outputMatrix, int width, int height) {

for (int y = 0; y < height; y++) {

for (int x = 0; x < width; x++) {

outputMatrix[x \* height + y] = inputMatrix[y \* width + x];

}

}

}

Листинг программы

Код на С++

Транспонирование матрицы

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <conio.h>

#include <device\_functions.h>

#include <iostream>

#include <Windows.h>

#include <ctime>

#define BLOCK\_DIM 20

#define GPU\_SLOW 1

#define GPU\_FAST 2

#define CPU 3

// Кількість навантажувальних циклів

#define ITERATIONS 20

using namespace std;

// Визначення змінної середовища

#define CUDA\_DEBUG

// Виведення діагностичної інформації

#ifdef CUDA\_DEBUG

#define CUDA\_CHECK\_ERROR(err) \

if (err != cudaSuccess) \

{ \

printf("Cuda error: %s\n", cudaGetErrorString(err)); \

printf("Error in file: %s, line: %i\n", \_\_FILE\_\_, \_\_LINE\_\_); \

}

#else

#define CUDA\_CHECK\_ERROR(err)

#endif

// Функція транспонування матриці без використання глобальної пам'яті

// \* inputMatrix - покажчик на вихідну матрицю

// \* outputMatrix - покажчик на матрицю результат

// \* width - ширина вихідної матриці (вона ж висота матриці-результату)

// \* height - висота вихідної матриці (вона ж ширина матриці-результату)

\_\_global\_\_ void transposeMatrixGlobal(float\* inputMatrix, float\* outputMatrix, int width, int height)

{

// Розрахунок індексів матриці

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

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

if ((xIndex < width) && (yIndex < height)) {

// Лінійний індекс елемента рядки вихідної матриці

int inputIdx = xIndex + width \* yIndex;

// Лінійний індекс елемента стовпця матриці-результату

int outputIdx = yIndex + height \* xIndex;

// Встановлення елементу

outputMatrix[outputIdx] = inputMatrix[inputIdx];

}

}

// Функція транспонування матриці з використанням колективної пам'яті

// \* inputMatrix - покажчик на вихідну матрицю

// \* outputMatrix - покажчик на матрицю результат

// \* width - ширина вихідної матриці (вона ж висота матриці-результату)

// \* height - висота вихідної матриці (вона ж ширина матриці-результату)

\_\_global\_\_ void transposeMatrixShared(float\* inputMatrix, float\* outputMatrix, int width, int height)

{

//int BLOCK\_DIM = width;

\_\_shared\_\_ float temp[BLOCK\_DIM][BLOCK\_DIM];

// Розрахунок індексів матриці

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

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

if ((xIndex < width) && (yIndex < height)) {

// Линейный индекс элемента строки исходной матрицы

int idx = yIndex \* width + xIndex;

//Копируем элементы исходной матрицы

temp[threadIdx.y][threadIdx.x] = inputMatrix[idx];

}

//Синхронизируем все нити в блоке

\_\_syncthreads();

xIndex = blockIdx.y \* blockDim.y + threadIdx.x;

yIndex = blockIdx.x \* blockDim.x + threadIdx.y;

if ((xIndex < height) && (yIndex < width)) {

// Линейный индекс элемента строки исходной матрицы

int idx = yIndex \* height + xIndex;

//Копируем элементы исходной матрицы

outputMatrix[idx] = temp[threadIdx.x][threadIdx.y];

}

}

// Функція транспонування матриці, яка виконується на CPU

\_\_host\_\_ void transposeMatrixCPU(float \*inputMatrix, float \*outputMatrix, int width, int height) {

for (int y = 0; y < height; y++) {

for (int x = 0; x < width; x++) {

outputMatrix[x \* height + y] = inputMatrix[y \* width + x];

}

}

}

// Функція виведення матриці на екран

\_\_host\_\_ void printMatrixToFile(char\* fileName, float\* matrix, int width, int height) {

FILE \*file = fopen(fileName, "wt");

for (int y = 0; y < height; y++) {

for (int x = 0; x < width; x++) {

fprintf(file, "%.0f\t", matrix[y \* width + x]);

}

fprintf(file, "\n");

}

fclose(file);

}

int main()

{

float gpuTime;

cudaEvent\_t start , stop;

cudaEventCreate (&start);

cudaEventCreate (&stop);

int width = 2048, height = 1536;

// Розмір масиву для збереження матриці

int matrixSize = width \* height;

// Кількість байтів що займає матриця

int byteSize = matrixSize \* sizeof(float);

//Выделяем память под матрицы на хосте

float\* inputMatrix = new float[matrixSize];

float\* outputMatrix = new float[matrixSize];

//Заполняем исходную матрицу данными

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

inputMatrix[i] = i;

// Вибираємо спосіб розрахунку транспонованою матриці

printf("Select compute mode: 1 - Slow GPU, 2 - Fast GPU, 3 - CPU\n");

int mode;

scanf("%i", &mode);

// Записуємо вихідну матрицю в файл

printMatrixToFile("before.txt", inputMatrix, width, height);

// Якщо используеться тільки CPU

if (mode == CPU) {

// int start = GetTickCount();

unsigned int start\_time = clock();

//cudaEventRecord (start, 0 );

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

transposeMatrixCPU(inputMatrix, outputMatrix, width, height);

}

// Виводимо час виконання функції на CPU (в мілліекундах)

// cudaEventRecord (stop, 0 );

// cudaEventSynchronize(stop);

// cudaEventElapsedTime(&gpuTime, start, stop);

//printf("Time: %.2f ms\n", gpuTime);

unsigned int end\_time = clock();

unsigned int search\_time = end\_time - start\_time;

cout << "Time: " << search\_time << " ms" << endl;

// printf("CPU compute time: %i\n", GetTickCount() - start);

}

// У разі розрахунку на GPU

else {

float \*devInputMatrix, \*devOutputMatrix;

// Виділяємо глобальну пам'ять для зберігання даних на пристрої

CUDA\_CHECK\_ERROR(cudaMalloc((void\*\*)&devInputMatrix, byteSize));

CUDA\_CHECK\_ERROR(cudaMalloc((void\*\*)&devOutputMatrix, byteSize));

// Копіюємо вихідну матрицю з хоста на девайс

CUDA\_CHECK\_ERROR(cudaMemcpy(devInputMatrix, inputMatrix, byteSize, cudaMemcpyHostToDevice));

// Конфігурація запуску ядра

dim3 gridSize = dim3(width / BLOCK\_DIM, height / BLOCK\_DIM, 1);

dim3 blockSize = dim3(BLOCK\_DIM, BLOCK\_DIM, 1);

//cudaEvent\_t start, stop;

// Створюємо події для синхронізації і виміру часу роботи GPU

//CUDA\_CHECK\_ERROR(cudaEventCreate(&start));

// CUDA\_CHECK\_ERROR(cudaEventCreate(&stop));

//Отмечаем старт расчетов на GPU

// Використовується функція без суспільної пам'яті

if (mode == GPU\_SLOW) {

cudaEventRecord(start, 0);

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

transposeMatrixGlobal<<<gridSize, blockSize>>>(devInputMatrix, devOutputMatrix, width, height);

}

cudaEventRecord (stop, 0 );

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

printf("Time: %.2f ms\n", gpuTime);

}

// Використовується функція з суспільною пам'яттю

else if (mode == GPU\_FAST) {

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

cudaEventRecord(start, 0);

transposeMatrixShared<<<gridSize, blockSize>>>(devInputMatrix, devOutputMatrix, width, height);

}

cudaEventRecord (stop, 0 );

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

printf("Time: %.2f ms\n", gpuTime);

}

// Відзначаємо закінчення розрахунку

cudaEventRecord(stop, 0);

// Синхронізуються з моментом закінчення розрахунків

cudaEventSynchronize(stop);

// Розраховуємо час роботи GPU

float time = 0;

cudaEventElapsedTime(&time, start, stop);

// Виводимо час розрахунку в консоль

// printf("GPU compute time: %.0f\n", time);

// Копіюємо результат з девайса на хост

CUDA\_CHECK\_ERROR(cudaMemcpy(outputMatrix, devOutputMatrix, byteSize, cudaMemcpyDeviceToHost));

// Чистимо ресурси на відеокарті

CUDA\_CHECK\_ERROR(cudaFree(devInputMatrix));

CUDA\_CHECK\_ERROR(cudaFree(devOutputMatrix));

CUDA\_CHECK\_ERROR(cudaEventDestroy(start));

CUDA\_CHECK\_ERROR(cudaEventDestroy(stop));

}

// Записуємо матрицю-результат в файл

printMatrixToFile("after.txt", outputMatrix, height, width);

// Чистимо пам'ять на хості

delete[] inputMatrix, outputMatrix;

system("pause");

return 0;

}

Умножение матрицы на вектор

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <conio.h>

#include <iostream>

#include <clocale>

using namespace std;

\_\_global\_\_ void MatrVectMul(float \*d\_c, float \*d\_a, float \*d\_b, int Size)

{

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

int k;

d\_c[i]=0;

for (k=0;k<Size;k++)

d\_c[i]+=d\_a[i\*Size+k]\*d\_b[k];

}

int main()

{

setlocale(LC\_ALL, "Rus");

float gpuTime;

cudaEvent\_t start , stop;

cudaEventCreate (&start);

cudaEventCreate (&stop);

int Size = 256;

// обычные массивы в оперативной памяти

float \*h\_a,\*h\_b,\*h\_c;

h\_a = new float[Size\*Size];

h\_b = new float[Size];

h\_c = new float[Size];

for (int i=0;i<Size;i++) // инициализация массивов a и b

{

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

{

h\_a[i\*Size+k]=rand();

}

h\_b[i]=rand();

}

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

{

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

{

cout << h\_a[i\*Size+k] << "\t";

cout << h\_b[i] << "\t";

}

cout << endl;

}

// указатели на массивы в видеопамяти

float \*d\_a,\*d\_b,\*d\_c;

// выделение видеопамяти

cudaMalloc((void \*\*)&d\_a, sizeof(float)\*Size\*Size);

cudaMalloc((void \*\*)&d\_b, sizeof(float)\*Size);

cudaMalloc((void \*\*)&d\_c, sizeof(float)\*Size);

// копирование из оперативной памяти в видеопамять

cudaMemcpy(d\_a, h\_a, sizeof(float)\*Size\*Size, cudaMemcpyHostToDevice) ;

cudaMemcpy(d\_b, h\_b, sizeof(float)\*Size, cudaMemcpyHostToDevice) ;

// установка количества блоков

dim3 grid((Size+255)/256, 1, 1);

// установка количества потоков в блоке

dim3 threads(256, 1, 1);

// вызов функции

cudaEventRecord (start, 0 );

MatrVectMul<<< grid, threads >>> (d\_c, d\_a, d\_b,Size);

cudaEventRecord (stop, 0 );

// копирование из видеопамяти в оперативную память

cudaMemcpy(h\_c, d\_c, sizeof(float)\*Size,cudaMemcpyDeviceToHost) ;

cudaEventElapsedTime(&gpuTime, start, stop);

printf("Время: %.2f мс\n", gpuTime);

// освобождение памяти

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

\_getch();

}