# Знакомство с GPCPU на базе технологии Nvidia Cuda

**Задание к лабораторной работе**

1. Установить и настроить среды Cuda на рабочей машине.

2. Модифицировать код программы "Cuda: Hello, world!" для вывода ядром в файл фразы "I am from N block, M thread (global index: K)", где параметры N, M, K имеют соответствующие значения.

3. Создать и скомпилировать проект, который выводит параметры среды Cuda, объяснить значения соответствующих параметров.

4. Ознакомиться и описать примеры приведены в GPU Computing SDK.

2. Модифицированный код программы "Cuda: Hello, world!" для вывода ядром в файл фразы "I am from N block, M thread (global index: K)", где параметры N, M, K имеют соответствующие значения, где в украинском алфавите N – первая буква фамилии, М – первая буква имени и K – первая буква отчества. Но у некоторых вариантов возможна такая ситуация когда например перемножение количество блоков и количество потоков будет меньше чем глобальный индекс. Количество блоков задается в гриде с помощью функции:

dim3 dimGrid(4, 4, 1); //Размерность грида

, где параметры обозначают

uint3 blockIdx.x = 4

uint3 blockIdx.y = 4

uint3 blockIdx.z = 1

Следовательно количество блоков равно 4×4×1=16

Количество потоков задается в блоке с помощью функции:

dim3 dimBlock(1,1,1); //Размерность блока

, где параметры обозначают

uint3 threadIdx.x = 1

uint3 threadIdx.y = 1

uint3 threadIdx.z = 1

Следовательно количество потоков равно 1×1×1=1

Функция, для вывода ядром:

kernel<<<dimGrid, dimBlock>>>();

Индекс блока:

int BlockIndex = blockIdx.x + blockIdx.y\*gridDim.x + blockIdx.z\*gridDim.y\*gridDim.x;

Индекс потока:

Int ThreadIndex= threadIdx.x+ threadIdx.y\*blockDim.x + threadIdx.z\*blockDim.y\*blockDim.x;

Глобальный индекс:

int GlobalThreadIndex = BlockIndex\*blockDim.x\*blockDim.y\*blockDim.z + ThreadIndex;

Функция:

cudaDeviceSynchronize();

Блокирует, пока устройство не выполнит все предыдущее запрошенные задачи, возвращает ошибку, если одна из предыдущих задач не выполнена.

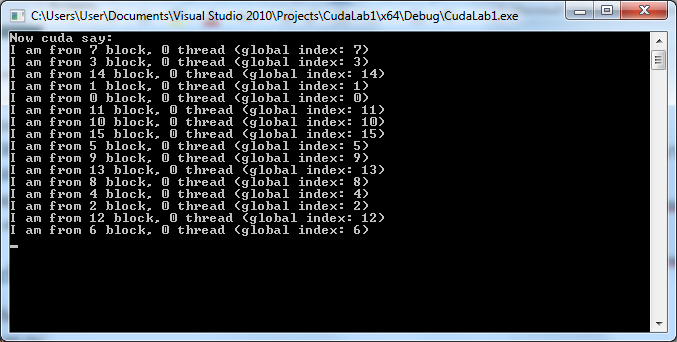


Рисунок 1 – Результат работы программы

Посмотреть характеристики карты можно вызвав функцию cudaGetDeviceProperties(), которая заполнит структуру типа cudaDeviceProp.

deviceProp.name - это строка ASCII, идентифицирующая устройство;

deviceProp.totalGlobalMem /(1024\*1024) - общий объем глобальной памяти, доступной на устройстве в Мегабайтах;

deviceProp.sharedMemPerBlock/1024 - это максимальный объем разделяемой памяти, доступный блоку потока в Килобайтах;

deviceProp.regsPerBlock - максимальное количество 32-разрядных регистров, доступных блоку потоков;

deviceProp.warpSize - размер основы в потоках;

deviceProp.memPitch - максимальный шаг в байтах, разрешенный функциями копирования памяти, которые связаны с областями памяти, выделенными cudaMallocPitch ();

deviceProp.maxThreadsPerBlock - максимальное количество потоков на блок;

deviceProp.maxThreadsDim[3] - содержит максимальный размер каждого измерения блока;

deviceProp.maxGridSize[3] - содержит максимальный размер каждого измерения сетки;

deviceProp.clockRate -  это тактовая частота в килогерце;

deviceProp.totalConstMem/1024 - общий объем постоянной памяти, доступной на устройстве в Килобайтах;

deviceProp.major, deviceProp.minor - это основные и младшие номера версий, определяющие вычислительную способность устройства;

deviceProp.textureAlignment - это требование выравнивания; текстурные базовые адреса, которые выровнены по тексту. Байт определения не требует смещения, применяемого к выборкам текстур;

deviceProp.deviceOverlap - 1, если устройство может одновременно копировать память между хостом и устройством во время выполнения ядра, или 0, если нет. Устаревший, используйте вместо deviceProp.asyncEngineCount.  
deviceProp.multiProcessorCount - это количество многопроцессоров на устройстве;

deviceProp.kernelExecTimeoutEnabled - 1, если для ядра, выполняемого на устройстве, существует ограничение на время выполнения, или 0, если нет.

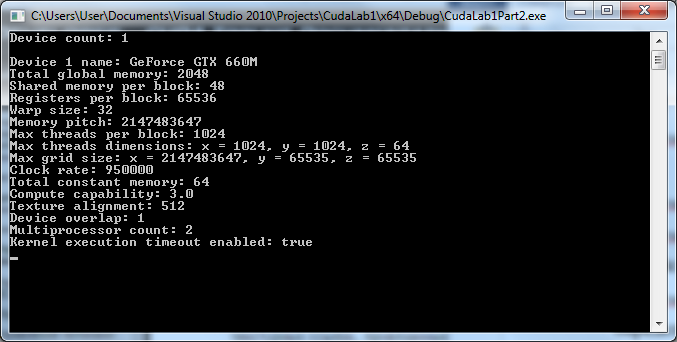


Рисунок 2 – Результат работы программы часть 2.

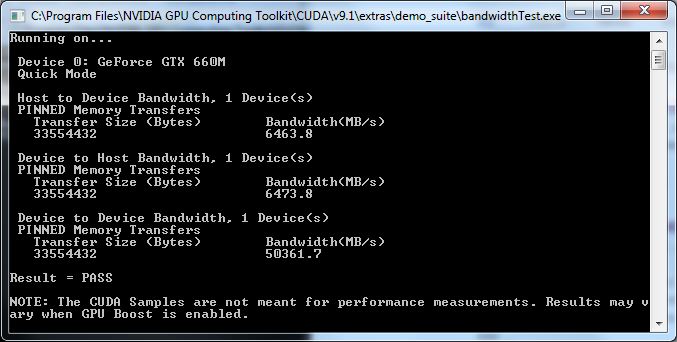


Рисунок 3 – Результат работы программы bandwidthTest.exe.

Здесь важно содержимое пары строк: второй, которая отображает имя модели устройства, и предпоследней, которая сообщает, что все требуемые тесты пройдены.

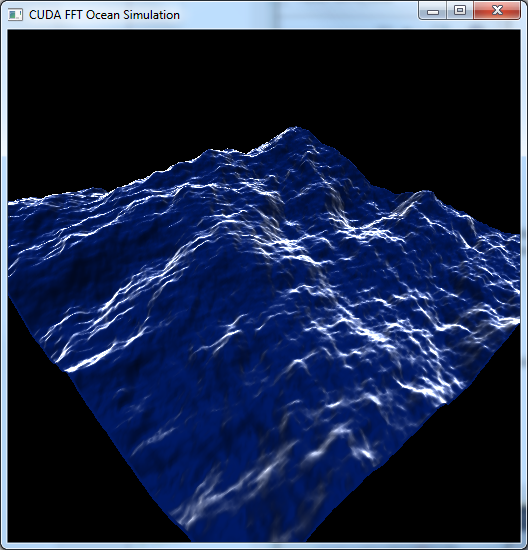


Рисунок 4 – Результат работы программы oceanFFT.exe.

Простые методы, демонстрирующие

* Основные подходы к вычислению GPU
* Рекомендации по наиболее важным функциям
* Эффективная работа с пользовательскими типами данных
* Быстрое интегрирование ускорения графического процессора в приложения C и C ++

Примеры использования, охватывающие такие темы, как:

* Добавление поддержки библиотек с ускорением GPU в приложение
* Использование таких функций, как память нулевого копирования, асинхронные передачи данных, унифицированная виртуальная адресация, одноранговая связь, параллельные ядра и многое другое
* Совместное использование данных между графическими API-интерфейсами CUDA и Direct3D / OpenGL (совместимость)

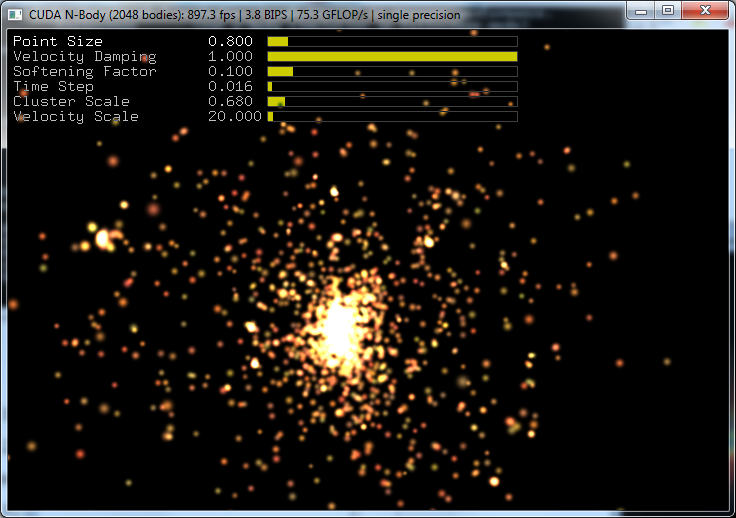


Рисунок 5 – Результат работы программы nbody.exe.

Данные-параллельные алгоритмы и примитивы для операций линейной алгебры:

* Матрица транспонирована
* Матричное умножение
* Матричное умножение с несколькими правыми частями
* Параллельная префиксная сумма больших массивов

Измерение и оптимизация производительности

* Тесты пропускной способности
* Профилирование приложений с использованием таймеров

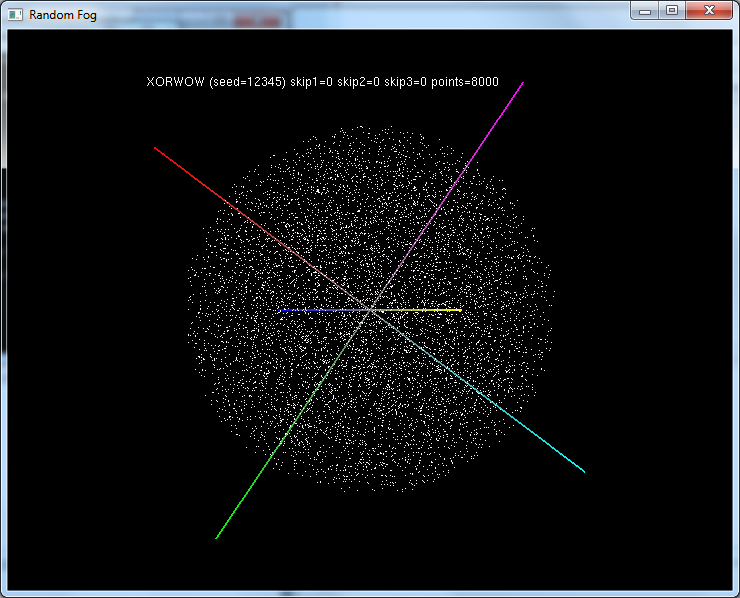


Рисунок 6 – Результат работы программы randomFog.exe.

Расширенные примеры приложений

* Использование CUDA с MPI и OpenMP
* Динамика вычислительной текучей среды (CFD)
* Гравитационное моделирование n-тела
* Блэк-Скоулз и биномиальная цена опциона
* 3D-конечная разность во времени (FDTD)
* Видеокодирование / декодирование
* Свертка изображений

# Знакомство с ключевыми моментами работы с GPGPU на базе технологии Nvidia Cuda

**Задание к лабораторной работе**

1. Реализовать программу, которая описана в пункте 4 теоретической части

2. Написать аналогичную программу, со следующими отличиями:

a. Предусмотреть возможности получения векторов для расчета из файла или их случайной генерации

b. Вариант получения векторов и вариант вывода результата расчета должен задаваться аргументом командной строки

c. Реализовать поддержку векторов большего размера, чем размер блока грида

3. Модифицировать написанную во втором пункте программу для сравнения временных характеристик в зависимости от размера векторов. Имеются в виду показатели времени выполнения суммирования на CPU, чистого времени суммирования на GPU и времени с передачей данных в GPU. Сделать выводы с полученными результатами.

4. Реализовать программу, осуществляющую составления двух матриц в которой выставляются требования аналогичны указанным во втором пункте. А также выполнить оценку аналогичную описанной в третьем пункте.

2. а)

Пример кода получения вектора для расчёта из файла

Запись в файл

ofstream f("C:\\Users\\User\\Documents\\Visual Studio 2010\\Projects\\CudaLab1\\TestAddVector\\file0.txt");

// Записываем значения в файл

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

{

f << i << endl;

vec1[i] = vec2[i] = i;

}

Чтение из файла

ifstream f("C:\\Users\\User\\Documents\\Visual Studio 2010\\Projects\\CudaLab1\\TestAddVector\\file0.txt");

// Чтение значений из файла

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

{

f >> vec1[i]; // обрахунок з файлу

vec2[i] = vec1[i]; // обрахунок з файлу

}

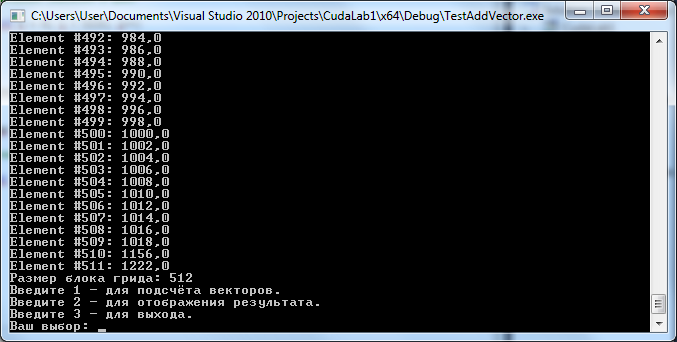


Рис. 1 – Чтение вектора из файла

Пример кода получения вектора при случайной генерации

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

{

vec1[i] = rand(); // випадкова генерація

vec2[i] = vec1[i]; // випадкова генерація

}

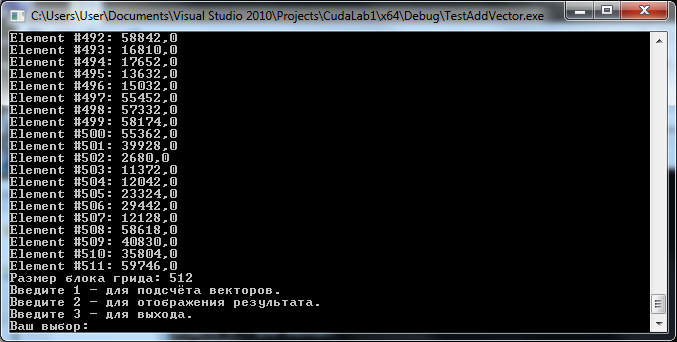


Рис. 2 – Случайная генерация вектора

б) Вариант получения векторов и вариант вывода результата расчета задаётся аргументом командной строки

int main()

{

int Size;

setlocale(LC\_ALL, "Rus");

int n;

cout << "Введите 1 - для подсчёта векторов." << endl;

cout << "Введите 2 - для отображения результата." << endl;

cout << "Введите 3 - для выхода." << endl;

cout << "Ваш выбор: ";

cin >> n;

if(n == 1)

{

Size = computingVectors();

computingVectors();

cout << "Подсчёт окончен." << endl;

}

if(n == 2)

showVectors(Size);

if(n == 3)

return 0;

do

{

if(n!= 1)

{

cout << "Введите 1 - для подсчёта векторов." << endl;

cout << "Введите 2 - для отображения результата." << endl;

cout << "Введите 3 - для выхода." << endl;

cout << "Ваш выбор: ";

}

else if ( n == 1)

{

cout << "Введите 1 - для подсчёта векторов." << endl;

cout << "Введите 2 - для отображения результата." << endl;

cout << "Введите 3 - для выхода." << endl;

cout << "Ваш выбор: ";

}

cin >> n;

if(n == 1)

{

computingVectors();

cout << "Подсчёт окончен." << endl;

}

if(n == 2)

showVectors(Size);

} while(n == 1 || n == 2);

if(n == 3)

delete[] vec1, vec2, vec3;

\_getch();

return 0;

}

Где функции,

void computingVectors() – вычисление вектора

void showVectors(int Size) – вывод результата расчёта, где Size – размер блока.

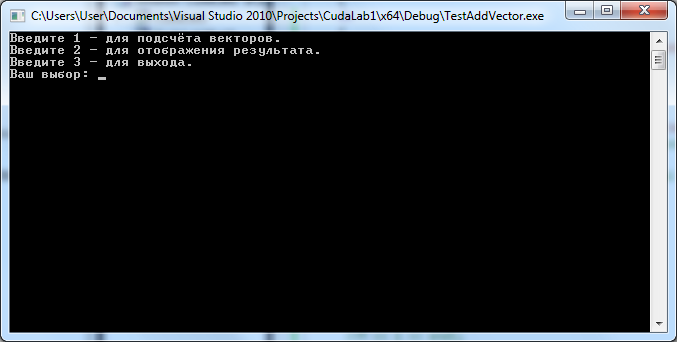


Рис. 3 – Вариант получения векторов и вариант вывода результата расчета задаётся аргументом командной строки

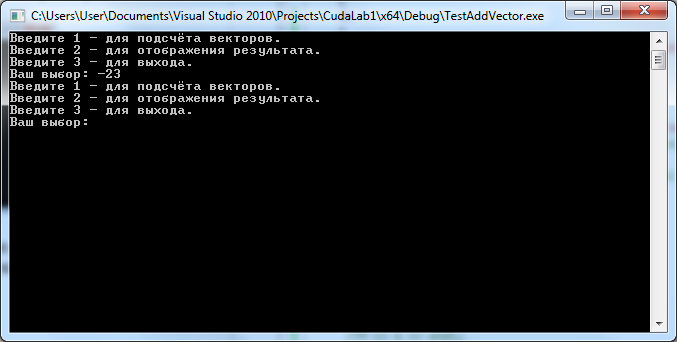


Рис. 4 – Вариант получения векторов и вариант вывода результата расчета с ошибочным значением

с) Поддержка векторов большего размера, чем размер блока грида

if(f == false)

{

do

{

cout << "Введите размер блока грида: (0 - 1024): " ;

cin >> Size;

} while(Size < 0 || Size > 1024);

f = true;

}

Использования значения в ядре

addVector<<<Size, SIZE>>>(devVec1, devVec2, devVec3);

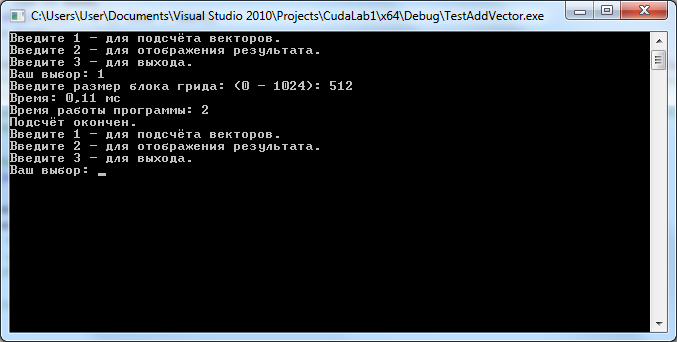


Рис. 5 – Поддержка векторов большего размера, чем размер блока грида

3) Для сравнении значений на CPU буду использовать встроенную функцию с++ clock( );

Пример работы в программе

// Покажчики на пам'ять відеокарти

float \*devVec1, \*devVec2, \*devVec3;

// Виділяємо пам'ять для векторів на відеокарті

cudaMalloc((void\*\*)&devVec1, sizeof(float) \* SIZE);

cudaMalloc((void\*\*)&devVec2, sizeof(float) \* SIZE);

cudaMalloc((void\*\*)&devVec3, sizeof(float) \* SIZE);

// Копіюємо дані в пам'ять відеокарти

cudaMemcpy(devVec1, vec1, sizeof(float) \* SIZE, cudaMemcpyHostToDevice);

cudaMemcpy(devVec2, vec2, sizeof(float) \* SIZE, cudaMemcpyHostToDevice);

unsigned int start\_time = clock();

addVector<<<Size, SIZE>>>(devVec1, devVec2, devVec3);

unsigned int end\_time = clock();

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

if( b == false)

{

cout << "Время работы программы: " << search\_time << endl;

b = true;

}

Для сравнении значений на GPU буду использовать встроенную функцию cuda cudaEventRecord (eventName, 0), cudaEventElapsedTime(&gpuTime, start, stop);

eventName start, stop;

cudaEvent\_t start , stop;

cudaEventCreate (&start);

cudaEventCreate (&stop);

float gpuTime;

// Покажчики на пам'ять відеокарти

float \*devVec1, \*devVec2, \*devVec3;

// Виділяємо пам'ять для векторів на відеокарті

cudaMalloc((void\*\*)&devVec1, sizeof(float) \* SIZE);

cudaMalloc((void\*\*)&devVec2, sizeof(float) \* SIZE);

cudaMalloc((void\*\*)&devVec3, sizeof(float) \* SIZE);

// Копіюємо дані в пам'ять відеокарти

cudaMemcpy(devVec1, vec1, sizeof(float) \* SIZE, cudaMemcpyHostToDevice);

cudaMemcpy(devVec2, vec2, sizeof(float) \* SIZE, cudaMemcpyHostToDevice);

cudaEventRecord (start, 0 );

addVector<<<Size, SIZE>>>(devVec1, devVec2, devVec3);

cudaEventRecord (stop , 0 );

// Обробник події

cudaEvent\_t syncEvent;

// Створюємо подію

cudaEventCreate(&syncEvent);

// Записуємо подію

cudaEventRecord(syncEvent, 0);

// Синхронізуємо подію

cudaEventSynchronize(syncEvent);

cudaEventSynchronize(stop);

// Тільки тепер отримуємо результат розрахунку

cudaMemcpy(vec3, devVec3, sizeof(float) \* SIZE, cudaMemcpyDeviceToHost);

cudaEventElapsedTime(&gpuTime, start, stop);

if( d == false)

{

printf("Время: %.2f мс\n", gpuTime);

d = true;

}

Вывод выполнение вычислений на GPU в 10 раз быстрее чем на CPU

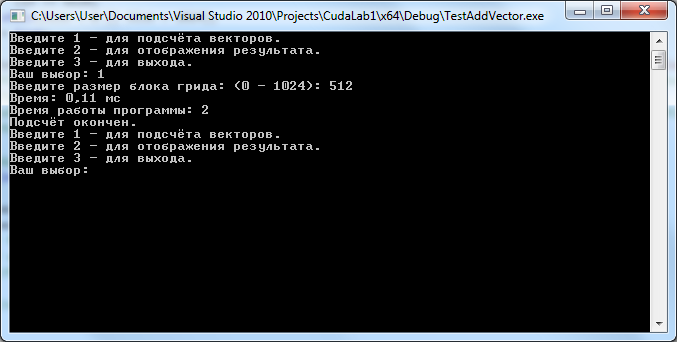


Рис. 5 – Время работы программы на CPU и GPU

4) Реализовать программу, осуществляющую сложение двух матриц

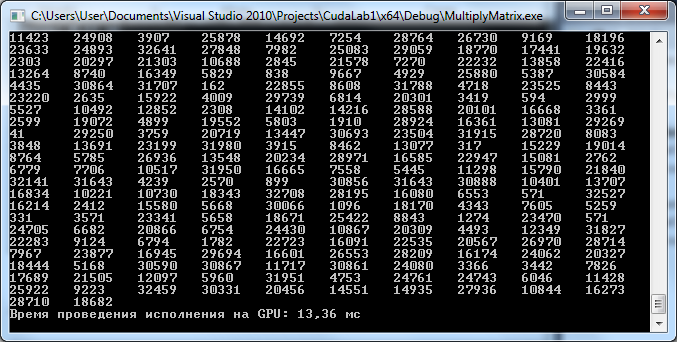


Рис. 6 – Время работы программы сложение двух матриц

# Знакомство с работой с памятью в GPGPU на базе технологии Nvidia Cuda

**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

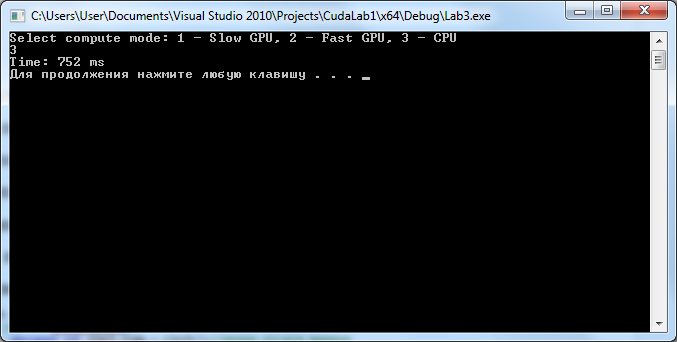


Рис. 1 - Время выполнения транспонирования матрицы на CPU

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

}

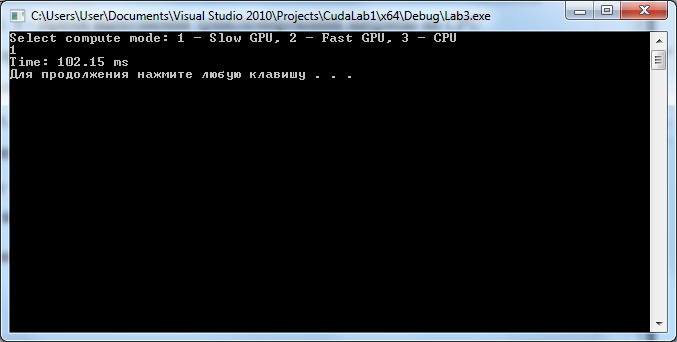


Рис. 2 - Время выполнения транспонирования матрицы на GPU без использования общей памяти

// 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

}

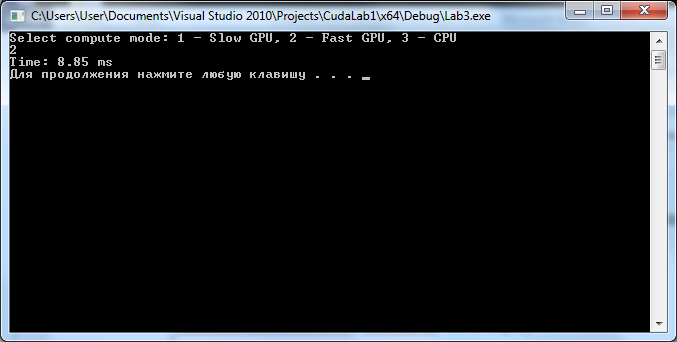


Рис. 3 - Время выполнения транспонирования матрицы на GPU с использования общей памяти

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);

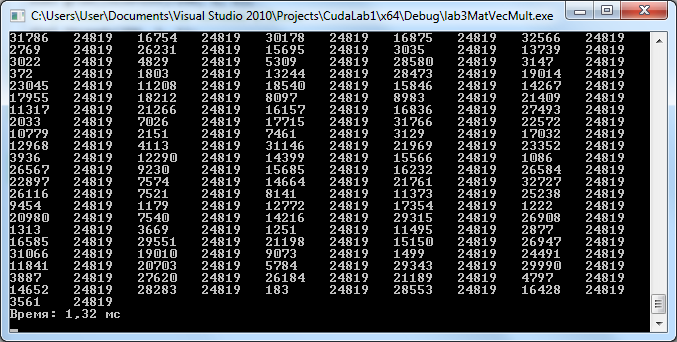


Рис. 4 - Время выполнения умножения матрицы на вектор на GPU без использования общей памяти

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];

}

}

}

# Знакомство с оптимизацией памяти в GPGPU на базе технологии Nvidia Cuda

**Задание к лабораторной работе**

1. Практически проверить изложенные в теоретической работе сведения и наглядно продемонстрировать полученные результаты

2. Модифицировать созданные в рамках 3 лабораторной работы с точки зрения рассматриваемой теоретической информации. Сравнить результаты работы соответствующих программ

1. Практическая проверка изложенных в теоретической работе сведений и наглядно продемонстрированы полученные результаты.

Выравнивание размеров используемых типов.

Обращение к глобальной памяти происходит через чтение/запись 32/64/128-битовых слов. Важно, что адрес, по которому происходит доступ, должен быть выровнен по размеру слова (кратен размеру слова в байтах).



Рис 1. - Пример выровненного и невыровненного 4-х байтного слова.

Все функции выделяющие глобальную память выделяют её выровненной по 256 байтам. Пусть есть массив из следующих структур в глобальной памяти:

struct vector {

float x;

float y;

float z;

};

Каждый элемент массива — 12 байт. Адрес первого элемента выровнен по 16 байтам, но адрес второго элемента — нет, и его чтение потребует двух обращений.

Выравнивание:

struct \_\_align\_\_(16) vector3 {

float x;

float y;

float z;

};

Теперь все элементы массива будут хранится по адресам кратным 16 байтам.

Для проверки оптимизации при использовании структур с выравниванием и без, использую функции ядра в которых произвожу вычисления с заданными структурами. Замеры времени будут проводиться с помощью cuda event's в cuda runtime API.

Примеры кода функции ядра:

\_\_global\_\_ void testFunc()

{

vector3 d;

d.x = 257;

d.y = 663;

d.z = 771;

int idx = d.x \* d.y + d.z;

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

{

int g =+ (d.x \* d.x \* d.z) / d.x;

idx += g;

}

};

\_\_global\_\_ void testFunc1()

{

vector b;

b.x = 257;

b.y = 663;

b.z = 771;

int idx = b.x \* b.y + b.z;

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

{

int c = (b.x \* b.x \* b.z) / b.x;

idx += c;

}

};

В самих функциях выполняются обычные операции при вычислении индекса потока с операциями со структурами данных определенных типов.

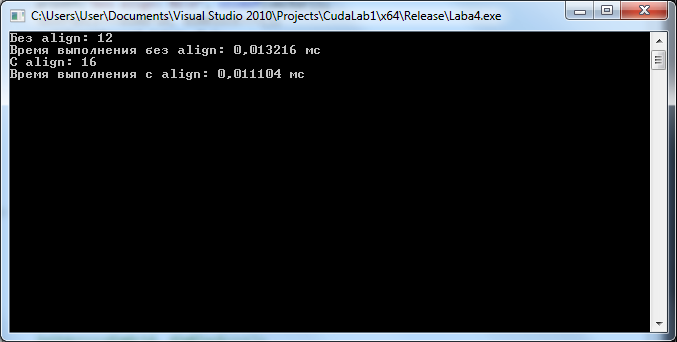


Рис. 2 – Пример выполнения программы с выравниванием размеров используемых типов

Продемонстрирую ещё один пример с выравниванием размеров используемых типов.

Пример кода функции ядра:

\_\_device\_\_ float3 data[512];

\_\_global\_\_ void initData()

{

int idx = threadIdx.x;

data[idx] = make\_float3(idx, idx, idx);

};

\_\_device\_\_ float4 dataFast[512];

\_\_global\_\_ void initDataFast()

{

int idx = threadIdx.x;

dataFast[idx] = make\_float4(idx, idx, idx, 0);

};

cudaEvent\_t start , stop;

cudaEventCreate (&start);

cudaEventCreate (&stop);

dim3 dimGrid(256,2,1); // кол-во блоков

dim3 dimBlock(16,16,1); // кол-во потоков

cudaEventRecord (start, 0 );

initData<<<dimGrid, dimBlock>>>();

cudaEventRecord (stop, 0 );

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

printf("Время выполнения int3: %.4f мс\n", gpuTime);

cudaEventRecord (start, 0 );

initDataFast<<<dimGrid, dimBlock>>>();

cudaEventRecord (stop, 0 );

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

printf("Время выполнения int4: %.4f мс\n", gpuTime);

В данном примере сравнивается время работы выравненного и невыравненного вектора в глобальной памяти.

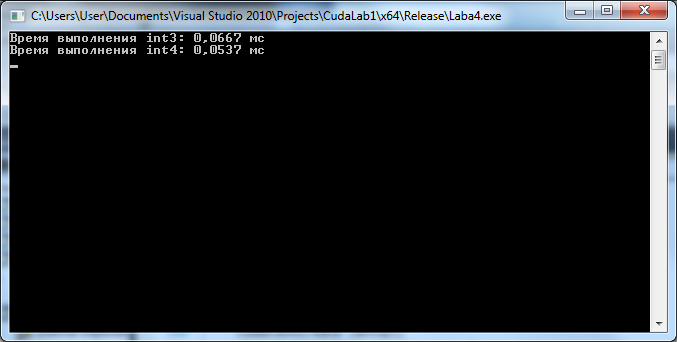


Рис. 3 – Пример работы выравненного и невыравненного вектора в глобальной памяти.

Использование объединенных запросов.

GPU имеет возможность объединять несколько запросов к глобальной памяти в один (coalescing). Все обращения MP к памяти происходят независимо для каждой половины warp’a. Максимальное объединение — все запросы одного полу-warp’a удается объединить в один большой запрос на чтение из глобальной памяти.

Все нити обращаются к 32(Computing Platform от 1.2) /32/64-битовым словам, давая в результате один 32(CP от 1.2) /64/128-байтный блок;

Получившийся блок выровнен по своему размеру (адрес кратен 32(CP от 1.2) /64/128);

Все 16 слов, к которым обращаются нити лежат в пределах одного блока;

Нити обращаются к словам последовательно: k–ая нить обращается к k–му слову; допускается, что отдельные нити пропустят обращение к соответствующим словам.

Если нити полу-warp’a не удовлетворяют любому условию, то каждое обращение к памяти происходит как отдельная транзакция!

Ещё один пример оптимизации это использование структур массивов а не массивов структур. Позволит уменьшить количество запросов в глобальную память за счет объединений.

Пример кода функции ядра:

\_\_device\_\_ vec3 data1[SIZE]; // инициализация массива структур

\_\_global\_\_ void initDataStruct()

{

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

data1[idx].x = idx;

data1[idx].y = idx \* 2;

data1[idx].z = idx \* 3;

}

\_\_device\_\_ float x[SIZE]; //инициализация структур массивов

\_\_device\_\_ float y[SIZE]; //инициализация структур массивов

\_\_device\_\_ float z[SIZE]; //инициализация структур массивов

\_\_global\_\_ void initArr()

{

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

x[idx] = idx;

y[idx] = idx \* 2;

z[idx] = idx \* 3;

};

cudaEventRecord (start, 0 );

initDataStruct<<<dimGrid, dimBlock>>>();

cudaEventRecord (stop, 0 );

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

printf("Время выполнения массива структур: %.4f мс\n", gpuTime);

gpuTime = 0;

cudaEventRecord (start, 0 );

initArr<<<dimGrid, dimBlock>>>();

cudaEventRecord (stop, 0 );

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

printf("Время выполнения структур массивов: %.4f мс\n", gpuTime);\*/

cudaDeviceSynchronize();

Комментариями обозначены места объявления данных

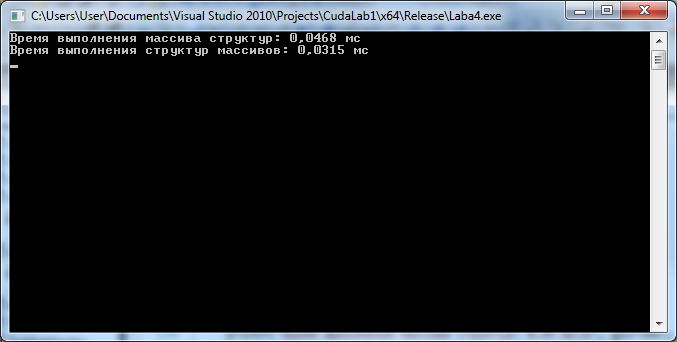


Рис 4. – Пример работы программы с использованием объединенных запросов.

2. Модифицировать созданные в рамках 3 лабораторной работы с точки зрения рассматриваемой теоретической информации. Сравнить результаты работы соответствующих программ.

В лабораторной работе 3 происходит оптимизация с помощью использования структуры массивов.

Пример кода с объявлением и инициализацией:

// 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];

}

}

Ещё хочу рассказать о топ что оптимизацию можно осуществить на этапе компиляции программы для этого нужно выбирать конфигурацию Release а не Debug при использовании Release компилятор перестаёт выполнять некоторые проверки на итерации и выключает из списка ненужные пространства имён что приводит к оптимизации проекта. Демонстрации примера приведена ниже:

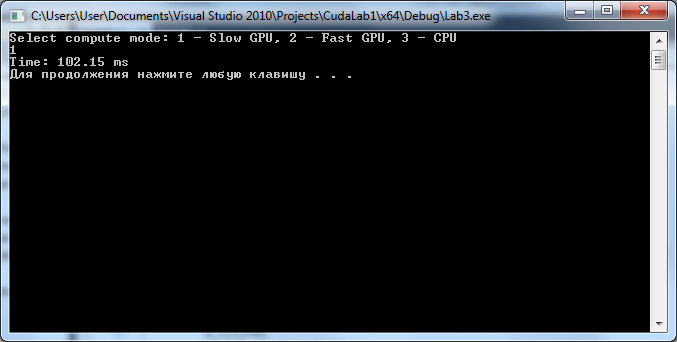


Рис. 5 – Пример выполнения конфигурации Debug

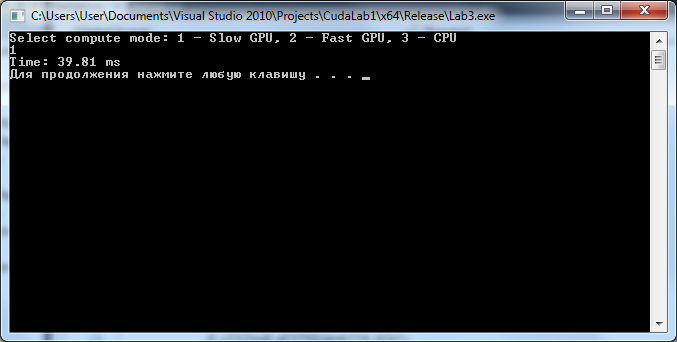


Рис. 5 – Пример выполнения конфигурации Release

Как видим увеличение скорости выполнения 2,57 раза.

# Листинг программы

# Код на С

# Параметры ядра

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

#include <cuda.h>

#include <iostream>

#include <fstream>

#include <string>

using namespace std;

\_\_global\_\_ void kernel()

{

int BlockIndex = blockIdx.x + blockIdx.y\*gridDim.x + blockIdx.z\*gridDim.y\*gridDim.x;

int ThreadIndex = threadIdx.x + threadIdx.y\*blockDim.x + threadIdx.z\*blockDim.y\*blockDim.x;

int GlobalThreadIndex = BlockIndex\*blockDim.x\*blockDim.y\*blockDim.z + ThreadIndex;

printf("I am from %d block, %d thread (global index: %d)\n", BlockIndex, ThreadIndex, GlobalThreadIndex);

}

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

{

printf("Now cuda say:\n");

dim3 dimGrid(4,4,1); //16 - Л

dim3 dimBlock(1,1,1); //1 - А

kernel<<<dimGrid, dimBlock>>>();

cudaDeviceSynchronize();

\_getch();

return 0;

}

Характеристики видеокарты

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <conio.h>

#include <iostream>

using namespace std;

int main()

{

int deviceCount;

cudaDeviceProp deviceProp;

cudaGetDeviceCount(&deviceCount);

printf("Device count: %d\n\n", deviceCount);

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

cudaGetDeviceProperties(&deviceProp, i);

printf("Device %d name: %s\n", i + 1, deviceProp.name);

printf("Total global memory: %lu\n", deviceProp.totalGlobalMem/(1024\*1024));

printf("Shared memory per block: %lu\n", deviceProp.sharedMemPerBlock/1024);

printf("Registers per block: %d\n", deviceProp.regsPerBlock);

printf("Warp size: %d\n", deviceProp.warpSize);

printf("Memory pitch: %lu\n", deviceProp.memPitch);

printf("Max threads per block: %d\n", deviceProp.maxThreadsPerBlock);

printf("Max threads dimensions: x = %d, y = %d, z = %d\n", deviceProp.maxThreadsDim[0], deviceProp.maxThreadsDim[1], deviceProp.maxThreadsDim[2]);

printf("Max grid size: x = %d, y = %d, z = %d\n", deviceProp.maxGridSize[0], deviceProp.maxGridSize[1], deviceProp.maxGridSize[2]);

printf("Clock rate: %d\n", deviceProp.clockRate);

printf("Total constant memory: %lu\n", deviceProp.totalConstMem/1024);

printf("Compute capability: %d.%d\n", deviceProp.major, deviceProp.minor);

printf("Texture alignment: %lu\n", deviceProp.textureAlignment);

printf("Device overlap: %d\n", deviceProp.deviceOverlap);

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

printf("Kernel execution timeout enabled: %s\n", deviceProp.kernelExecTimeoutEnabled ? "true" : "false");

}

\_getch();

return 0;

}

# Листинг программы на С++

# Сложение векторов

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <conio.h>

#include <iostream>

#include <fstream>

#include <locale>

#include <ctime>

#define SIZE 512

using namespace std;

// Функція складання двох векторів

\_\_global\_\_ void addVector(float\* left, float\* right, float\* result)

{

// Отримати id поточної нитки

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

if(idx >= 1024) return;

// Розраховуємо результат.

result[idx] = left[idx] + right[idx];

}

// Виділяємо пам'ять під вектора

float \*vec1 = new float[SIZE], \*vec2 = new float[SIZE], \*vec3 = new float[SIZE];

bool f = false;

bool b = false;

bool d = false;

int Size;

float gpuTime;

int computingVectors()

{

cudaEvent\_t start , stop;

cudaEventCreate (&start);

cudaEventCreate (&stop);

if(f == false)

{

do

{

cout << "Введите размер блока грида: (0 - 1024): " ;

cin >> Size;

} while(Size < 0 || Size > 1024);

f = true;

}

ifstream f("C:\\Users\\User\\Documents\\Visual Studio 2010\\Projects\\CudaLab1\\TestAddVector\\file0.txt");

//ofstream f("C:\\Users\\User\\Documents\\Visual Studio 2010\\Projects\\CudaLab1\\TestAddVector\\file0.txt");

//fclose(f);

// Ініціалізіруем значення векторів

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

{

vec1[i] = rand(); // випадкова генерація

vec2[i] = vec1[i]; // випадкова генерація

// f >> vec1[i]; // обрахунок з файлу

// vec2[i] = vec1[i]; // обрахунок з файлу

//f << i << endl;

//vec1[i] = vec2[i] = i;

}

// Покажчики на пам'ять відеокарти

float \*devVec1, \*devVec2, \*devVec3;

// Виділяємо пам'ять для векторів на відеокарті

cudaMalloc((void\*\*)&devVec1, sizeof(float) \* SIZE);

cudaMalloc((void\*\*)&devVec2, sizeof(float) \* SIZE);

cudaMalloc((void\*\*)&devVec3, sizeof(float) \* SIZE);

// Копіюємо дані в пам'ять відеокарти

cudaMemcpy(devVec1, vec1, sizeof(float) \* SIZE, cudaMemcpyHostToDevice);

cudaMemcpy(devVec2, vec2, sizeof(float) \* SIZE, cudaMemcpyHostToDevice);

unsigned int start\_time = clock();

cudaEventRecord (start, 0 );

addVector<<<Size, SIZE>>>(devVec1, devVec2, devVec3);

cudaEventRecord (stop , 0 );

// Обробник події

cudaEvent\_t syncEvent;

// Створюємо подію

cudaEventCreate(&syncEvent);

// Записуємо подію

cudaEventRecord(syncEvent, 0);

// Синхронізуємо подію

cudaEventSynchronize(syncEvent);

cudaEventSynchronize(stop);

// Тільки тепер отримуємо результат розрахунку

cudaMemcpy(vec3, devVec3, sizeof(float) \* SIZE, cudaMemcpyDeviceToHost);

cudaEventElapsedTime(&gpuTime, start, stop);

if( d == false)

{

printf("Время: %.2f мс\n", gpuTime);

d = true;

}

unsigned int end\_time = clock();

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

if( b == false)

{

cout << "Время работы программы: " << search\_time << endl;

b = true;

}

// Вивільняються ресурсів

// Видалення події

cudaEventDestroy(syncEvent);

// Вивільнення пам'яті на відеокарті

cudaFree(devVec1);

cudaFree(devVec2);

cudaFree(devVec3);

// Вивільнення памьяті основної програми

return Size;

}

void showVectors(int Size)

{

// Результати розрахунку

for (int i = 0; i < SIZE; i++) printf("Element #%i: %.1f\n", i, vec3[i]);

cout << "Размер блока грида: " << Size << endl;

}

int main()

{

int Size;

setlocale(LC\_ALL, "Rus");

int n;

cout << "Введите 1 - для подсчёта векторов." << endl;

cout << "Введите 2 - для отображения результата." << endl;

cout << "Введите 3 - для выхода." << endl;

cout << "Ваш выбор: ";

cin >> n;

if(n == 1)

{

Size = computingVectors();

computingVectors();

cout << "Подсчёт окончен." << endl;

//cout << "Введите 2 - для отображения результата." << endl;

}

if(n == 2)

showVectors(Size);

if(n == 3)

return 0;

do

{

if(n!= 1)

{

cout << "Введите 1 - для подсчёта векторов." << endl;

cout << "Введите 2 - для отображения результата." << endl;

cout << "Введите 3 - для выхода." << endl;

cout << "Ваш выбор: ";

}

else if ( n == 1)

{

cout << "Введите 1 - для подсчёта векторов." << endl;

cout << "Введите 2 - для отображения результата." << endl;

cout << "Введите 3 - для выхода." << endl;

cout << "Ваш выбор: ";

}

cin >> n;

if(n == 1)

{

computingVectors();

cout << "Подсчёт окончен." << endl;

}

if(n == 2)

showVectors(Size);

} while(n == 1 || n == 2);

if(n == 3)

delete[] vec1, vec2, vec3;

\_getch();

return 0;

}

# Листинг программы на С++

# Сложение матриц

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

#include <iostream>

#include <iomanip>

#include <clocale>

using namespace std;

#define BLOCK\_SIZE 16 // submatrix size

#define N 256 // matrix size is N\*N

\_\_global\_\_ void matMult ( float \* a, float \* b, int n, float \* c )

{

int bx = blockIdx.x; // block index

int by = blockIdx.y;

int tx = threadIdx.x; // thread index

int ty = threadIdx.y;

float sum = 0.0f; // computed subelement

int ia = n \* BLOCK\_SIZE \* by + n \* ty; // a [i][0]

int ib = BLOCK\_SIZE \* bx + tx;

// Multiply the two matrices together;

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

{

sum += a [ia + k] + b [ib + k\*n];

}

/\*for(int k = 0; k < n; k++)

{

printf("%.2f\n", a[ia + k]);

printf("%.2f\n", b[ib + k\*n]);

}\*/

// Write the block sub-matrix to global memory;

// each thread writes one element

int ic = n \* BLOCK\_SIZE \* by + BLOCK\_SIZE \* bx;

c [ic + n \* ty + tx] = sum;

}

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

{

setlocale(LC\_ALL, "Rus");

int numBytes = N \* N \* sizeof ( float );

// allocate host memory

float \* a = new float [N\*N];

float \* b = new float [N\*N];

float \* c = new float [N\*N];

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

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

{

int k = N\*i + j;

a [k] = rand();

b [k] = rand();

}

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

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

{

int k = N\*i + j;

//cout << setw(3) << a[k] << b[k];

printf("%.0f\t", a[k]);

printf("%.0f\t", b[k]);

}

printf("\n");

//cout << endl;

// allocate device memory

float \* adev = NULL;

float \* bdev = NULL;

float \* cdev = NULL;

cudaMalloc ( (void\*\*)&adev, numBytes );

cudaMalloc ( (void\*\*)&bdev, numBytes );

cudaMalloc ( (void\*\*)&cdev, numBytes );

// set kernel launch configuration

dim3 threads ( BLOCK\_SIZE, BLOCK\_SIZE );

dim3 blocks ( N / threads.x, N / threads.y);

// create cuda event handles

cudaEvent\_t start, stop;

float gpuTime = 0.0f;

cudaEventCreate (&start);

cudaEventCreate (&stop);

// asynchronously issue work to the GPU (all to stream 0)

cudaEventRecord (start, 0);

cudaMemcpy (adev, a, numBytes, cudaMemcpyHostToDevice );

cudaMemcpy (bdev, b, numBytes, cudaMemcpyHostToDevice );

matMult<<<blocks, threads>>> (adev, bdev, N, cdev);

cudaMemcpy (c, cdev, numBytes, cudaMemcpyDeviceToHost );

cudaEventRecord (stop, 0);

cudaEventSynchronize (stop);

cudaEventElapsedTime (&gpuTime, start, stop);

// print the cpu and gpu times

printf("Время проведения исполнения на GPU: %.2f мс\n", gpuTime);

// release resources

cudaEventDestroy (start);

cudaEventDestroy (stop);

cudaFree (adev);

cudaFree (bdev);

cudaFree (cdev);

delete a;

delete b;

delete c;

\_getch();

return 0;

}

# 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();

}

# Листинг программы

# Оптимизация памяти с помощью запросов и типов

# Код на Си

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <stdio.h>

#include <conio.h>

#include <clocale>

#include <ctime>

#define SIZE 256

#define ITERATIONS 20000

struct \_\_align\_\_(16) vec3

{

float x;

float y;

float z;

};

\_\_device\_\_ vec3 data1[SIZE]; // инициализация массива струтур

\_\_global\_\_ void initDataStruct()

{

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

data1[idx].x = idx;

data1[idx].y = idx \* 2;

data1[idx].z = idx \* 3;

}

\_\_device\_\_ float x[SIZE]; //инициализация структур массивов

\_\_device\_\_ float y[SIZE]; //инициализация структур массивов

\_\_device\_\_ float z[SIZE]; //инициализация структур массивов

\_\_global\_\_ void initArr()

{

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

x[idx] = idx;

y[idx] = idx \* 2;

z[idx] = idx \* 3;

};

\_\_device\_\_ float3 data[512];

\_\_global\_\_ void initData()

{

int idx = threadIdx.x;

data[idx] = make\_float3(idx, idx, idx);

};

\_\_device\_\_ float4 dataFast[512];

\_\_global\_\_ void initDataFast()

{

int idx = threadIdx.x;

dataFast[idx] = make\_float4(idx, idx, idx, 0);

};

struct vector {

float x;

float y;

float z;

};

struct \_\_align\_\_(16) vector3 {

float x;

float y;

float z;

};

\_\_global\_\_ void testFunc()

{

vector3 d;

d.x = 257;

d.y = 663;

d.z = 771;

int idx = d.x \* d.y + d.z;

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

{

int g =+ (d.x \* d.x \* d.z) / d.x;

idx += g;

}

};

\_\_global\_\_ void testFunc1()

{

vector b;

b.x = 257;

b.y = 663;

b.z = 771;

int idx = b.x \* b.y + b.z;

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

{

int c = (b.x \* b.x \* b.z) / b.x;

idx += c;

}

};

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

{

setlocale(LC\_ALL, "Rus");

float gpuTime;

float number;

cudaEvent\_t start , stop;

cudaEventCreate (&start);

cudaEventCreate (&stop);

dim3 dimGrid(256,2,1); // кол-во блоков

dim3 dimBlock(16,16,1); // кол-во потоков

cudaEventRecord (start, 0 );

initData<<<dimGrid, dimBlock>>>();

cudaEventRecord (stop, 0 );

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

printf("Время выполнения int3: %.4f мс\n", gpuTime);

cudaEventRecord (start, 0 );

initDataFast<<<dimGrid, dimBlock>>>();

cudaEventRecord (stop, 0 );

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

printf("Время выполнения int4: %.4f мс\n", gpuTime);

cudaEventRecord (start, 0 );

printf("Без align: %i\n", sizeof(vector));

testFunc1<<<dimGrid, dimBlock>>>();

cudaEventRecord (stop, 0 );

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

printf("Время выполнения без align: %.6f мс\n", gpuTime);

cudaEventRecord (start, 0 );

printf("С align: %i\n", sizeof(vector3));

testFunc<<<dimGrid, dimBlock>>>();

cudaEventRecord (stop, 0 );

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

printf("Время выполнения c align: %.6f мс\n", gpuTime);

cudaEventRecord (start, 0 );

initDataStruct<<<dimGrid, dimBlock>>>();

cudaEventRecord (stop, 0 );

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

printf("Время выполнения массива структур: %.4f мс\n", gpuTime);

gpuTime = 0;

cudaEventRecord (start, 0 );

initArr<<<dimGrid, dimBlock>>>();

cudaEventRecord (stop, 0 );

cudaEventSynchronize(stop);

cudaEventElapsedTime(&gpuTime, start, stop);

printf("Время выполнения структур массивов: %.4f мс\n", gpuTime);

cudaDeviceSynchronize();

\_getch();

return 0;

}