1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт компьютерных механики и математики
5. **Высшая школа кибербезопасности и защиты информации**

**ЛАБОРАТОРНАЯ РАБОТА № 4**

# «Параллельные вычисления на GPu»

1. по дисциплине «языки программирования»
2. Выполнили
3. Студенты гр. 3631001/80001 Смирнягина В.Н., Якимова Н.П
4. <*подпись*>
5. Проверил
6. преподаватель Малышев Е.В.

<*подпись*>

Санкт-Петербург

1. 2019
2. **ЦЕЛЬ РАБОТЫ**
3. Реализовать параллельные вычисления на GPU для решения задачи перемножения матриц.

**ЗАДАЧИ РАБОТЫ**

1. Изучить основы работы с CUDA
2. Разбить исходную задачу на блоки, вычисление которых будет происходить в многопоточном режиме.
3. Разработать код программы, реализующий вычисление исходной задачи на GPU.

**ХОД РАБОТЫ**

# ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

**CUDA** – это программно-аппаратная архитектура параллельных вычислений, позволяющая существенно увеличить вычислительную продуктивность благодаря использованию графических процессоров NVIDIA.

**БАЗОВЫЕ ПОНЯТИЯ И ТЕРМИНЫ CUDA:**

* **Устройство (device)** — **GPU**. Выполняет роль «подчиненного» — делает только то, что ему говорит CPU;
* **Хост (host) —** **CPU**. Выполняет управляющую роль — запускает задачи на устройстве, выделяет память на устройстве, перемещает память на/с устройства;
* **Ядро (kernel)** — задача, запускаемая хостом на устройстве;

Для организации параллельных вычислений GPU запустит большое количество копий одного и того же ядра в разных потоках. Чем больше потоков, тем эффективнее.

Код для ядер отличается от обычного последовательного кода в таких моментах:

1) Внутри ядер имеется возможность узнать «идентификатор» или, проще говоря, позицию потока, который сейчас выполняется. Используя эту позицию, можно добиться того, что одно и то же ядро будет работать с разными данными в зависимости от потока, в котором оно запущено. Такая организация параллельных вычислений называется [SIMD](http://en.wikipedia.org/wiki/SIMD) (Single Instruction Multiple Data) — когда несколько процессоров выполняют одновременно одну и ту же операцию, но на разных данных.

2) В некоторых случаях в коде ядра необходимо использовать различные способы синхронизации.

В учебных целях для ознакомления с CUDA был реализован был алгоритм нахождения определителя матрицы. Его реализация подразумевает разделение на вычисляемые блоки – детерминант для каждого элемента высчитывается параллельно с остальными детерминантами. Правильность работы алгоритма проверена с помощью онлайн – калькулятора (Рисунок 1).

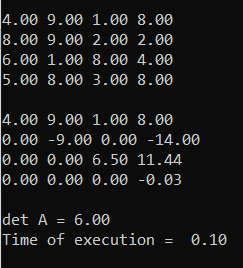
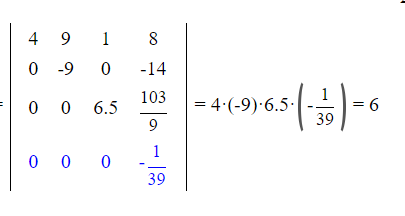


Рисунок 1 – Результаты работы онлайн – калькулятора и разработанной программы на CUDA.

Также, для сравнительной характеристики был реализован тот же алгоритм на языке С, работа которого также проверена с помощью онлайн – калькулятора (Рисунок 2).

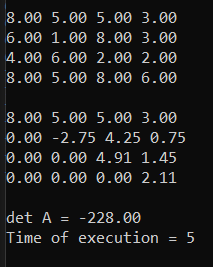
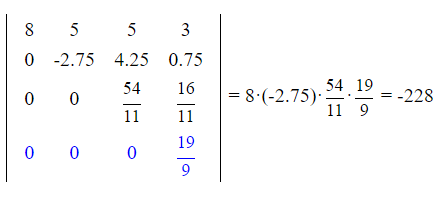
****

Рисунок 2 – Результаты работы онлайн – калькулятора и разработанной программы на C.

**СРАВНИТЕЛЬНАЯ ХАРАКТЕРИСТИКА ДВУХ РЕАЛИЗАЦИЙ**

|  |  |  |  |
| --- | --- | --- | --- |
| Размер матрицы, N | Время высчитывания определителя на CPU, мс | Время высчитывания определителя на GPU, мс | Соотношение времени выполнения на CPU и GPU |
| 2 | 2 | 0,06 | 33,3 |
| 4 | 4 | 0,1 | 40,0 |
| 6 | 7 | 0,12 | 58,3 |
| 8 | 10 | 0,12 | 83,3 |
| 10 | 15 | 0,16 | 93,8 |
| 50 | 659 | 33,73 | 19,5 |
| 60 | 835 | 67,35 | 12,4 |

Таблица 1 – Результаты измерений времени вычисления на CPU и GPU.

# Разработка программного кода

РЕАЛИЗАЦИЯ НА CUDA

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <cuda.h>

#include <curand\_kernel.h>

#include <time.h>

#include <stdio.h>

#include <stdlib.h>

#include <random>

#include <conio.h>

//blockDim - размер блока

//blockIdx - индекс текущего блока

//threadIdx - индекс текущей нити в блоке

\_\_device\_\_ void elem(double\* ar, int m, int n, double k, int N)

{

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

if (g < N) ar[m \* N + g] -= k \* ar[n \* N + g];

}

\_\_global\_\_ void triangle(double\* arr, int N)

{

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

int j;

double kof;

for (j = 0; j < N \* N; j++)

{

// if (!arr[j \* N + j]) elem(arr, j, N - 1, N, N);

if (i >= j && i < N - 1)

{

kof = arr[(i + 1) \* N + j] / arr[j \* N + j];

elem(arr, i + 1, j, kof, N);

}

}

}

//Перемножаем элементы на главной диагонали уже треугольной матрицы, тем самым получаем определитель

double det(double\* arr, int N)

{

double d = 1.0;

for (int i = 0; i < N; i++) d \*= arr[i \* N + i];

return d;

}

\_\_host\_\_ int main()

{

int N;

printf("Input size of matrix N = ");

scanf\_s("%i",&N);

unsigned int timer;

int MatrSize = N \* N;//Размер матрицы

int byteSize = MatrSize \* sizeof(double);//Память, необходимая для массива на GPU

double\* inMatr = new double[MatrSize];//Выделяем память под массив

//Заполняем матрицу случайными числами и выводим на экран

srand(time(NULL));

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

{

inMatr[i] = 1 + rand() % 9;

}

printf("\n");

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

{

printf("%0.2f ", inMatr[i]);

if (((i + 1) % N == 0) && (i != 0)) printf("\n");

}

printf("\n");

double\* inMatr\_d;

cudaMalloc((void\*\*)&inMatr\_d, byteSize);//Выделяем память под массив на GPU

cudaMemcpy(inMatr\_d, inMatr, byteSize, cudaMemcpyHostToDevice);//Копируем значения матрицы на GPU

dim3 gridSize = dim3(N, N, 1);//Размерность сетки блоков (dim3), выделенную для расчетов

dim3 blockSize = dim3(1, 1, 1);//Размер блока (dim3), выделенного для расчетов

//Инициализируем переменные для замера времени работы

float recording;

cudaEvent\_t start, stop;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventRecord(start, 0);

float start2 = clock();

triangle <<< gridSize, blockSize >> > (inMatr\_d, N);//Запускаем матрицу для приведения к треугольному виду

cudaThreadSynchronize();//Синхронизируем потоки

float end = clock();

//Получаем время работы

cudaEventRecord(stop, 0);

cudaEventSynchronize(stop);

cudaEventElapsedTime(&recording, start, stop);

cudaMemcpy(inMatr, inMatr\_d, byteSize, cudaMemcpyDeviceToHost);//Копируем новую матрицу с GPU на CPU

//Выводим полученную матрицу

int string = 0;

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

{

if (string && i == string \*N)

{

int m = i;

for (int j = string \* N; j < string \*N + string; j++)

{

printf("0.00 ");

m++;

}

i = m;

}

printf("%.2f ", inMatr[i]);

if ((i + 1) % N == 0)

{

printf("\n");

string++;

}

}

printf("\ndet A = %.2f \n", det(inMatr, N));//Выводим определитель

if (recording > 0) printf("Time of execution = %.2f\n", recording);

else printf("Time of execution = %.2f\n", end - start2);

cudaFree(inMatr\_d);//Освобождаем память

getch();

return 0;

}

РЕАЛИЗАЦИЯ НА СИ

#include <time.h>

#include <stdio.h>

#include <stdlib.h>

#include <conio.h>

void det(double\*\* arr, int N)

{

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

{

for (int i = k + 1; i < N; i++)

{

double mu = arr[i][k] / arr[k][k];

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

arr[i][j] -= arr[k][j] \* mu;

}

}

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

{

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

{

printf("%.2f ", arr[i][j]);

if ((j + 1) % N == 0) printf("\n");

}

}

}

int main()

{

int N;

printf("Input size of matrix N = ");

scanf\_s("%i", &N);

int time\_start = clock();

double\*\* inMatr = NULL;

inMatr = (double\*\*)malloc(N \* sizeof(double\*));//Выделяем память под массив

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

{

inMatr[i] = (double\*)malloc(N \* sizeof(double));

}

//Заполняем матрицу случайными числами и выводим на экран

srand(time(NULL));

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

{

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

{

inMatr[i][j] = 1 + rand() % 9;

}

}

printf("\n");

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

{

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

{

printf("%0.2f ", inMatr[i][j]);

if (((j + 1) % N == 0) && (j != 0)) printf("\n");

}

}

printf("\n");

det(inMatr, N);

//Перемножаем элементы на главной диагонали уже треугольной матрицы, тем самым получаем определитель

double d = 1.00;

for (int i = 0; i < N; i++) d \*= inMatr[i][i];

printf("\ndet A = %.2f \n", d);

printf("Time of execution = %i", clock() - time\_start);

system("pause > 0");

}

**ВЫВОД**

У видеочипов работа простая и распараллеленная изначально. Видеочип принимает на входе группу полигонов, проводит все необходимые операции, и на выходе выдаёт пиксели. Обработка полигонов и пикселей независима, их можно обрабатывать параллельно, отдельно друг от друга. Поэтому из-за изначально параллельной организации работы в GPU используется большое количество исполнительных блоков, которые легко загрузить, в отличие от последовательного потока инструкций для CPU. Кроме того, современные GPU также могут исполнять больше одной инструкции за такт. **В процессе выполнения работы были изучены методы реализации параллельных вычислений на GPU. Синхронизация рабочих пространств GPU и CPU осуществляется при помощи копирования вычисленных данных из пространства GPU командой *CudaMemcpy().***