1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —

**Институт кибербезопасности и защиты информации**

**ЛАБОРАТОРНАЯ РАБОТА № 4**

# «Параллельные вычисления на GPu»

1. по дисциплине «языки программирования»
2. Выполнили
3. Студенты гр. 4831001/10001 Гусаров В.В., Бушмелев А.С.
4. <*подпись*>
5. Проверил
6. преподаватель Малышев Е.В.

<*подпись*>

Санкт-Петербург

1. 2022
2. **ЦЕЛЬ РАБОТЫ**
3. Реализовать параллельные вычисления на GPU для взятия определителя матрицы

**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. Правильность работы алгоритма проверена с помощью онлайн – калькулятора (Рисунок 1). Алгоритм заключается в приведении матрицы к треугольному виду и последующее умножение чисел на диагонали этой матрицы.

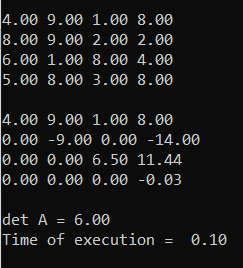
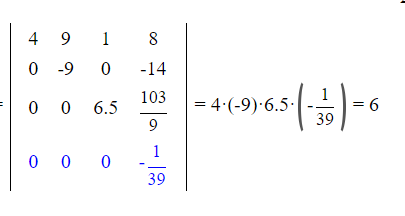


Рисунок 1 – Результаты работы онлайн – калькулятора и разработанной программы на CUDA.

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

**Изображение выглядит как стол

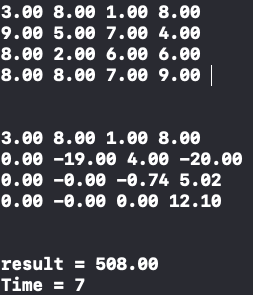
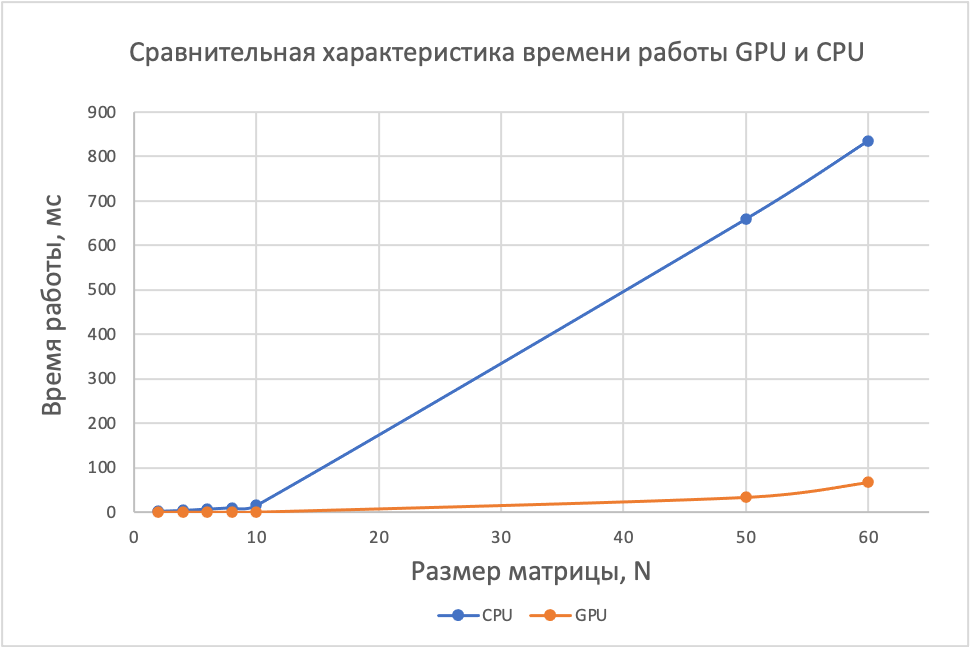
Автоматически созданное описание**

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 СИ

#include <time.h>

#include <stdio.h>

#include <stdlib.h>

void determinant(double\*\* matrix, int N)

{

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

{

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

{

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

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

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

}

}

}

void filling\_the\_matrix(double\*\* matrix,int N)

{

srand(time(NULL));

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

{

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

{

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

}

}

}

void matrix\_output(double\*\* matrix, int N)

{

printf("\n");

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

{

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

{

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

if (j==N-1) printf("\n");

}

}

printf("\n");

}

int main()

{

int N;

printf("size = ");

scanf("%i", &N);

double result = 1.0;

int time\_end;

double\*\* input\_matrix = NULL;

input\_matrix = (double\*\*)malloc(N \* sizeof(double\*));

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

{

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

}

filling\_the\_matrix(input\_matrix,N);

matrix\_output(input\_matrix,N);

int time\_start = clock();

determinant(input\_matrix, N);

time\_end=clock();

matrix\_output(input\_matrix,N);

for (int i = 0; i < N; i++) result \*= input\_matrix[i][i];

printf("\nresult = %.2f \n", result);

printf("Time = %i\n", time\_end - time\_start);

}

**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().***