**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский Авиационный Институт»**

**(Национальный Исследовательский Университет)**

**Институт: №8 «Информационные технологии   
и прикладная математика»   
Кафедра: 806 «Вычислительная математика   
и программирование»**

Курсовая работа  
по курсу «Численные методы»

Группа: М8О-407Б-21

Студент: И. Д. Павлов

Преподаватель: Ю.В. Сластушенский

Оценка:

Дата: 21.12.2024

Москва, 2024

**ОГЛАВЛЕНИЕ**

[**1** **Тема** 3](#_Toc167364649)

[**2** **Задание** 3](#_Toc167364650)

[**3** **Теория** 4](#_Toc167364651)

[**4** **Листинг кода** 8](#_Toc167364652)

[**5** **Выводы** 1](#_Toc167364653)4

[**6** **Список используемой литературы** 1](#_Toc167364654)7

# **Тема**

Распараллеливание вычислительных алгоритмов решения систем линейных алгебраических уравнений.

# **Задание**

* Изучить методы распараллеливания СЛАУ на CUDA.
* Найти любое решение системы уравнений, где A – матрица размерности , b - вектор-столбец свободных коэффициентов длины , x – вектор неизвестных длины .
* Разработать программу для решения СЛАУ на CUDA. Входные данные: на первой строке заданы числа n и m – размеры матрицы. В следующих n строках записано по m вещественных чисел – элементов матрицы. Выходные данные: m значений, являющихся элементами вектора неизвестных x.
* Провести сравнительный анализ времени работы данного алгоритма на разных конфигурациях CUDA и сравнить с результатами на процессоре.

# **Теория**

Для решения СЛАУ есть следующие методы:

* Метод Гаусса
* Метод прогонки
* Методы простых итераций и Зейделя

Методы прогонки, итераций и Зейделя плохо поддаются распараллеливанию на GPU, так как каждая итерация зависит от уже обновлённых значений текущего шага. В этих методах обновления переменных происходят последовательно, и каждое следующее значение рассчитывается с использованием уже обновлённого предыдущего.

Поэтому далее рассматривается метод Гаусса. Алгоритм состоит из двух основных этапов:

1. Прямой ход (приведение к верхнетреугольному виду). Путём элементарных преобразований строк матрицы (перестановка строк, умножение строки на число, сложение строк) матрица приводится к верхнетреугольному виду. Для каждой строки используется ведущий элемент (pivot), который выбирается как максимальный по модулю элемент в текущем столбце для уменьшения ошибок, связанных с численной нестабильностью.
2. Обратный ход (нахождение решений). Решение системы находится путём подстановки значений переменных, начиная с последнего уравнения.

Сам алгоритм традиционно считается последовательным со сложностью . Однако, он выдает абсолютно точные значения в отличие от методов итераций и Зейделя и он универсален в отличии от метода прогонки. С помощью CUDA можно распараллелить прямой обход, а обратный обход, подсчет ранга и остальные операции уже имеют сложность . Таким образом окончательная сложность алгоритма становится , что гораздо лучше кубической, что будет показательно видно в 5 главе данного отчета.

Этапы распараллеливания прямого хода метода Гаусса:

1. Выбор главного элемента. Необходимо выбрать ведущий элемент — наибольший по модулю элемент в текущем столбце. Каждый поток обрабатывает подмножество элементов столбца. С помощью атомарной операции (atomic\_max) ищется максимальный элемент по модулю. В результате мы получаем максимальный элемент в столбце.
2. Перестановка строк. После выбора ведущего элемента может потребоваться переставить строки, чтобы главный элемент оказался на диагонали. Столбцы матрицы обрабатываются параллельно потоками. Каждый поток меняет местами элементы двух строк в своём столбце. Индексация потоков и блоков позволяет распределить работу по всем элементам строк.
3. Обновление строк матрицы. Каждая строка и каждый элемент строки обрабатываются параллельно. Для каждой строки рассчитывается коэффициент для исключения переменной. Обновления элементов матрицы выполняются независимо для каждой строки. Для распределения вычислений используется двумерная сетка потоков, где индексы потоков вычисляются на основе номеров блоков и потоков в каждом измерении. Фактическое обнуление элементов не производится для предотвращения неправильного ответа при параллельной обработке, но результирующая матрица сохраняет правильную структуру, необходимую для решения СЛАУ.

Данные 3 этапа проходят каждый в своем ядре CUDA. Сами ядра вызываются min(n, m) раз в цикле.

Ядро reduce\_max\_col предназначено для параллельного поиска максимального по модулю элемента в столбце матрицы и его индекса с использованием CUDA. Каждый поток внутри блока обрабатывает часть элементов столбца, находя максимальное значение среди своих данных. Для синхронизации и избежания конфликтов при обновлении общего максимума используется атомарная операция atomic\_max, реализованная на уровне устройства. Эта операция гарантирует корректное обновление максимального значения в разделяемой памяти (sh\_val) между потоками. После завершения обработки каждым потоком, главный поток (с threadIdx.x == 0) записывает итоговое максимальное значение и его индекс в глобальные переменные \*value и \*idx.

Ядро swap\_rows предназначено для параллельного обмена двух строк в матрице на GPU. Каждый поток обрабатывает отдельный столбец матрицы, вычисляя индексы элементов из заданных строк row1 и row2, а затем обменивает их значения. Перемещение потоков по столбцам осуществляется с шагом, равным количеству всех потоков (offset), чтобы гарантировать охват всех элементов строки при большом количестве столбцов.

Ядро gauss выполняет параллельное обновление элементов матрицы на этапе прямого хода метода Гаусса. Каждый поток GPU отвечает за обработку отдельных элементов ниже и правее текущего ведущего элемента (pivot) на i-й итерации. Координаты обрабатываемых элементов определяются через индексы потоков (idx, idy) и их смещение (offsetx, offsety). Для каждого элемента вычисляется корректирующий множитель с использованием ведущего элемента строки, после чего элемент обновляется по формуле исключения Гаусса.

Обратный ход Гаусса:

Для дальнейшего решения из матрицы извлекается квадратная подсистема размером rank×(rank+1) для переменных, которые могут быть определены однозначно. Из исходной матрицы отбираются строки и столбцы, соответствующие базисным переменным. Строки, не влияющие на базисные переменные, исключаются. Столбцы, соответствующие зависимым переменным, отбираются в новую матрицу. Правая часть системы (вектор свободных членов) также переносится в последний столбец новой подсистемы.

На этапе обратного хода каждая переменная вычисляется начиная с последней строки. Система приводится к диагональному виду, а затем д ля каждой строки вычисляется поправка для элементов выше по столбцу. Коэффициенты, отвечающие за переменные в верхних строках, обнуляются.

Затем переменные рассчитываются напрямую:

* Диагональные элементы используются для нахождения значений переменных.
* Если ведущий элемент слишком мал (по модулю меньше eps), переменная считается равной нулю.
* Для избыточных переменных выводится нулевое значение.

# **Листинг кода**

#include <iostream>

#include <vector>

#include <math.h>

#include <iomanip>

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

const double eps = 1e-7;

#define CSC(call)                                                   \

do {                                                                \

    cudaError\_t res = call;                                         \

    if (res != cudaSuccess) {                                       \

        fprintf(stderr, "ERROR in %s:%d. Message: %s\n",            \

                \_\_FILE\_\_, \_\_LINE\_\_, cudaGetErrorString(res));       \

        exit(0);                                                    \

    }                                                               \

} while(0)

\_\_device\_\_ void atomic\_max(double\* const address, const double value)

{

    unsigned long long int\* const \_address = (unsigned long long int\*)address;

    unsigned long long int prev = \*\_address, next;

    do {

        next = prev;

        if (\_\_longlong\_as\_double(next) >= value) {

            break;

        }

        prev = atomicCAS(\_address, next, \_\_double\_as\_longlong(value));

    } while (next != prev);

}

\_\_global\_\_ void reduce\_max\_col(double \*matrix, int sz, double \*value, int \*idx) {

    \_\_shared\_\_ double sh\_val;

    \_\_shared\_\_ int sh\_idx;

    if (threadIdx.x == 0) {

        sh\_val = 0.0;

        sh\_idx = 0;

    }

    \_\_syncthreads();

    double max\_val = 0.0;

    int max\_idx = 0;

    for (int i = threadIdx.x; i < sz; i += blockDim.x) {

        if (fabs(max\_val) < fabs(matrix[i])) {

            max\_val = matrix[i];

            max\_idx = i;

        }

    }

    atomic\_max(&sh\_val, max\_val);

    \_\_syncthreads();

    if (sh\_val == max\_val) {

        sh\_idx = max\_idx;

    }

    \_\_syncthreads();

    if (threadIdx.x == 0) {

        \*value = sh\_val;

        \*idx = sh\_idx;

    }

}

\_\_global\_\_ void swap\_rows(double \*matrix, int n, int m, int row1, int row2) {

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

    int offset = blockDim.x \* gridDim.x;

    while (j < m) {

        int idx1 = j \* n + row1;

        int idx2 = j \* n + row2;

        double tmp = matrix[idx1];

        matrix[idx1] = matrix[idx2];

        matrix[idx2] = tmp;

        j += offset;

    }

}

\_\_global\_\_ void gauss(double \*matrix, int n, int m, int i) {

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

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

    int offsetx = blockDim.x \* gridDim.x;

    int offsety = blockDim.y \* gridDim.y;

    int h = m - i - 1;

    int w = n - i - 1;

    for (int y = idy; y < h; y += offsety) {

        for (int x = idx; x < w; x += offsetx) {

            int k = i + 1 + y;

            int r = i + 1 + x;

            double coef = matrix[k \* n + i];

            double num  = matrix[i \* n + r];

            double div  = matrix[i \* n + i];

            matrix[k \* n + r] -= coef \* num / div;

        }

    }

}

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

{

    int n = 0, m = 0;

    scanf("%d %d", &n, &m);

    double \*matrix = new double[n \* (m + 1)];

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < m; j++) {

            std::cin >> matrix[j \* n + i];

        }

    }

    for (int i = 0; i < n; i++) {

        std::cin >> matrix[m \* n + i];

    }

    double \*dev\_matrix;

    CSC(cudaMalloc(&dev\_matrix, sizeof(double) \* n \* (m + 1)));

    CSC(cudaMemcpy(dev\_matrix, matrix, sizeof(double) \* n \* (m + 1), cudaMemcpyHostToDevice));

    double \*max\_elem;

    int    \*max\_ptr;

    CSC(cudaMalloc(&max\_elem, sizeof(double)));

    CSC(cudaMalloc(&max\_ptr,  sizeof(int)));

    std::vector<bool> rank\_cols(m, false);

    int skip = 0;

    int iterCount = std::min(n, m);

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

        int begin\_col\_idx = (i + skip) \* n + (i - skip);

        int end\_col\_idx   = (i + 1 + skip) \* n;

        int col\_size      = end\_col\_idx - begin\_col\_idx;

        reduce\_max\_col<<<512, 512>>>(dev\_matrix + begin\_col\_idx, col\_size, max\_elem, max\_ptr);

        double h\_max\_elem;

        int    h\_max\_ptr;

        CSC(cudaMemcpy(&h\_max\_elem, max\_elem, sizeof(double), cudaMemcpyDeviceToHost));

        CSC(cudaMemcpy(&h\_max\_ptr, max\_ptr,   sizeof(int),    cudaMemcpyDeviceToHost));

        int shift  = begin\_col\_idx + h\_max\_ptr;

        int center = (i + skip) \* n + (i - skip);

        int row1 = shift  % n;

        int row2 = center % n;

        if (row1 != row2) {

            swap\_rows<<<1024, 1024>>>(dev\_matrix, n, m + 1, row1, row2);

            CSC(cudaGetLastError());

            CSC(cudaDeviceSynchronize());

        }

        double main\_elem;

        CSC(cudaMemcpy(&main\_elem, dev\_matrix + center, sizeof(double), cudaMemcpyDeviceToHost));

        if (fabs(main\_elem) < eps) {

            skip++;

            continue;

        } else {

            rank\_cols[i + skip] = true;

        }

        gauss<<<dim3(64, 64), dim3(16, 16)>>>(dev\_matrix, n, m + 1, i);

        CSC(cudaGetLastError());

        CSC(cudaDeviceSynchronize());

    }

    CSC(cudaMemcpy(matrix, dev\_matrix, sizeof(double) \* n \* (m + 1), cudaMemcpyDeviceToHost));

    int rank = 0;

    for (int j = 0; j < m; j++) {

        if (rank\_cols[j]) {

            rank++;

        }

    }

    if (rank == m) {

        std::cout << "System has one solution:\n";

    } else {

        std::cout << "System has infinity solutions. One of them:\n";

    }

    double \*square\_matrix = new double[rank \* (rank + 1)];

    int ri = 0;

    for (int row = 0; row < n; row++) {

        if (row >= m) {

            break;

        }

        if (!rank\_cols[row]) {

            continue;

        }

        int rj = 0;

        for (int j = 0; j < m; j++) {

            if (rank\_cols[j]) {

                square\_matrix[rj \* rank + ri] = matrix[j \* n + row];

                rj++;

            }

        }

        square\_matrix[rank \* rank + ri] = matrix[m \* n + row];

        ri++;

        if (ri == rank) {

            break;

        }

    }

    for (int y = rank - 1; y >= 0; y--) {

        double diag = square\_matrix[y \* rank + y];

        if (fabs(diag) > eps) {

            for (int x = y - 1; x >= 0; x--) {

                double f = square\_matrix[y \* rank + x];

                square\_matrix[rank \* rank + x] -= square\_matrix[rank \* rank + y] \* f / diag;

                square\_matrix[y \* rank + x] = 0.0;

            }

        }

    }

    std::cout << std::scientific << std::setprecision(10);

    for (int i = 0; i < rank; i++) {

        double diag = square\_matrix[i \* rank + i];

        double rhs  = square\_matrix[rank \* rank + i];

        double val  = (fabs(diag) < eps ? 0.0 : (rhs / diag));

        std::cout << val << " ";

    }

    for (int i = rank; i < m; i++) {

        std::cout << 0.0 << " ";

    }

    std::cout << std::endl;

    CSC(cudaFree(dev\_matrix));

    CSC(cudaFree(max\_elem));

    CSC(cudaFree(max\_ptr));

    delete[] square\_matrix;

    delete[] matrix;

    return 0;

}

# **Выводы**

В таблице ниже представлено исследование зависимости времени работы алгоритма в мс от параметров сетки GPU ядер.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размер | 10 | 102 | 103 | 5\*103 |
| CPU | **0.002938** | **1.96897** | 1910.41 | - |
| <<<(16, 16), (16, 16)>>> | 1.523712 | 11.9897 | **559.59918** | 5042.01514 |
| <<<(32, 32), (16, 16)>>> | 1.60256 | 12.14976 | 560.24756 | **4334.9165** |
| <<<(32, 32), (32, 32)>>> | 1.6384 | 12.6382 | 566.29144 | 4612.94775 |
| <<<(64, 64), (32, 32)>>> | 1.6691 | 14.124 | 592.768 | 4563.656738 |

На малых размерах матриц CPU демонстрирует лучшие результаты, что объясняется накладными расходами на передачу данных между CPU и GPU, а также недостаточной загрузкой GPU. На более крупных размерах матриц GPU значительно превосходит CPU по времени выполнения, демонстрируя ускорение до нескольких раз. Конфигурация <<<(32, 32), (16, 16)>>> показывает наилучшее время выполнения для особо крупной матрицы, где CPU будет вести подсчет часами.

Увеличение размеров блоков не всегда приводит к улучшению производительности, что указывает на необходимость балансировки между числом потоков и доступной памятью.

Подводя итоги, реализация метода Гаусса на CUDA показала высокую эффективность при обработке больших матриц. Для малых размеров CPU остаётся конкурентоспособным из-за низких накладных расходов. Однако с увеличением размера задачи преимущества GPU становятся очевидными, и при правильной настройке параметров сетки и блоков можно добиться значительного ускорения вычислений.

Результаты работы программы:

=====================

TEST: t1.txt

IN:

3 4

1 3 2 0

1 10 3 1

4 19 9 1

0 -5 -5

OUT:

System has infinity solutions. One of them:

2.1428571429e+00 -7.1428571429e-01 0.0000000000e+00 0.0000000000e+00

=====================

TEST: t2.txt

IN:

2 3

1 2 3

4 5 6

5 14

OUT:

System has infinity solutions. One of them:

1.0000000000e+00 2.0000000000e+00 0.0000000000e+00

=====================

TEST: t3.txt

IN:

3 2

1 2

3 4

5 6

7 8 9

OUT:

System has one solution:

-6.0000000000e+00 6.5000000000e+00

=====================

TEST: t4.txt

IN:

2 2

1 2

3 4

5 6

OUT:

System has one solution:

-4.0000000000e+00 4.5000000000e+00

# **Список используемой литературы**

1. Раздел 1. Численные методы линейной алгебры. // Методические материалы. – URL: <https://mainfo.ru/mietodichieskiie-matierialy/> (дата обращения: 21.12.2024).
2. Боресков А.В., Харламов А.А. Основы работы с технологией CUDA. Глава 5. Реализация на CUDA базовых операций над массивами – reduce, scan, построения гистограмм и сортировки / А.В. Боресков, А.А. Харламов – ДМК Пресс, 2012 (дата обращения: 21.12.2024).