Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ *Робототехники и комплексной автоматизации*

КАФЕДРА *Системы автоматизированного проектирования (РК-6)*

ОТЧЕТ О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

по дисциплине: «Архитектура параллельных вычислительных систем»

Студент	Косенков Александр Александрович
Группа	РК6-12М
Тип задания	Лабораторная работа №2
Тема лабораторной работы	Программирование CUDA

Студент _____ **Косенков А.А.**
подпись, дата *фамилия, и.о.*

Преподаватель _____ **Спасенов А.Ю.**
подпись, дата *фамилия, и.о.*

Оценка _____

Москва, 2022 г.

Оглавление

Задание на лабораторную работу

Применить технологии CUDA к задаче решения нестационарного уравнения теплопроводности для пластины заданной формы с помощью неявной разностной схемы. Теоретически оценить возможное ускорение программы. Построить зависимость времени работы программы от числа нитей. Оценить накладные расходы и эффективность распараллеливания.

Вариант 69 (Решение нестационарного уравнения теплопроводности с помощью неявной разностной схемы)

Постановка задачи:

С помощью неявной разностной схемы решить нестационарное уравнение теплопроводности для пластины, изображенной на рис., там же указаны размеры сторон.

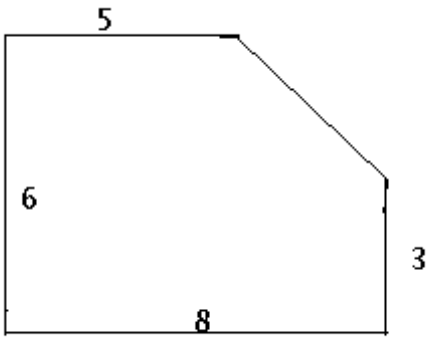

Рис. 1 – Условия задачи

Начальное значение температуры пластины - 0 градусов.

Граничные условия следующие: слева 100 градусов, внизу $\frac{dT}{dn} = T$, справа, на скошенной границе, $\frac{dT}{dn} = 20$ и вверху 50.

При выводе результатов показать динамику изменения температуры (например, с помощью цветовой гаммы).

Выполненные задачи

1. Постановка идеи параллелизма, описание инструментов для реализации программы.
2. Теоретическая оценка возможного ускорения программы.
3. Результаты работы программы. Зависимость времени работы от количества потоков.
4. Оценка накладных расходов и эффективности распараллеливания.
5. Сравнение результатов работы программы с существующим ПО.

В ходе лабораторной работы для программной реализации задач был использован набор инструментов CUDA Toolkit 11.6. Запуск производился на ГП NVIDIA Geforce RTX 2060 в ОС Windows 10.

1. Постановка идеи параллелизма, описание инструментов для реализации программы.

Решение поставленной задачи теплопроводности осуществляется за счет итеративного численного решения СЛАУ вида $A\bar{x} = \bar{b}$, где $\bar{b}$ – вектор значений температур в узлах на k-м временном слое, $\bar{x}$ – вектор неизвестных значений температур на $k+1$ временном слое. Решение данной СЛАУ было осуществлено с помощью метода Гаусса, реализованного в функции *gauss*. Одним из оптимальных мест для внедрения распараллеливания является метод Гаусса, поскольку в ходе прямого хода метода элементарные преобразования для приведения матрицы к треугольному виду являются независимыми в пределах строки. Также возможно распараллеливание обратного хода Гаусса в рамках цикла суммирования элементов с использованием редукции. Распараллеливание итераций решения задачи не представляется возможным, поскольку каждая следующая итерация требует значения предыдущей.

Для реализации параллельных вычислений были использованы следующие инструменты CUDA:

- Функция cudaMalloc, осуществляющая динамическое выделение памяти на устройстве;
- Функция cudaMemcpy, осуществляющая синхронизацию данных между устройством и хостом;
- Функция __syncthreads(), реализующая барьер в контексте нитей;
- Структуры threadIdx, blockDim, предоставляющие интерфейс к текущей конфигурации потоков CUDA;
- Спецификатор __global__, используемый для описания функции-ядра, выполняющейся на устройстве и вызываемой из хоста;
- Спецификатор __shared__, используемый для создания переменных в общей памяти в рамках одного блока CUDA;
- Функция cudaGetErrorString, позволяющая получить информацию об ошибке CUDA.

В свою очередь, функцией-ядром является непосредственно метод Гаусса, предварительно приведенный к необходимому виду для внедрения параллелизма (Листинг 1). Так, матрица A была приведена к виду одномерного массива для оптимизации ее передачи на ГП, результирующий вектор x был вынесен во входной параметр, согласно синтаксису функции-ядра. Помимо этого, по

причине того, что обратный ход Гаусса подразумевает собой вычисление суммы, которое было подвергнуто распараллеливанию – необходимо также предусмотреть редукцию, для реализации которой был передан дополнительный параметр reduction, представляющий собой массив, длина которого соответствует количеству нитей и хранит в себе вычисленные в каждой нити локальные суммы.

Листинг 1. Программная реализация метода Гаусса

```
__global__ void gauss(double *A_matrix, double *y, double *x, size_t nodes_num,
double* reduction) {
    for (int k = 1; k < nodes_num; ++k) { // прямой ход
        for (int j = k + 1 + threadIdx.x; j < nodes_num; j += blockDim.x) {
            double pivot = A_matrix[j * nodes_num + k] / A_matrix[k * nodes_num
+ k];
            for (int i = k; i < nodes_num; i++) {
                A_matrix[j * nodes_num + i] -= pivot * A_matrix[k * nodes_num +
i];
            }

            y[j] = y[j] - pivot * y[k];
        }
    }

    for (int k = nodes_num - 1; k >= 1; k--) { // обратный ход
        __shared__ double total_sum;
        double sum = 0;
        for (int j = k + 1 + threadIdx.x; j < nodes_num; j += blockDim.x) {
            sum += A_matrix[k * nodes_num + j] * x[j];
        }

        reduction[threadIdx.x] = sum;
        __syncthreads();
        for (int size = blockDim.x / 2; size > 0; size /= 2) { //uniform
            if (threadIdx.x < size) {
                reduction[threadIdx.x] += reduction[threadIdx.x + size];
            }
            __syncthreads();
        }
        if (threadIdx.x == 0) {
            total_sum = reduction[0];
        }
        __syncthreads();

        x[k] = (y[k] - total_sum) / A_matrix[k * nodes_num + k];
    }
}
```

2. Теоретическая оценка возможного ускорения программы.

При переходе от последовательной реализации программы к параллельной можно получить ускорение, оценка которого описывается законом Амдала:

$$S \leq \frac{1}{f + \frac{1-f}{p}}, \quad (1)$$

где f – доля последовательных операций $(0 \leq f \leq 1)$, а p – число потоков.

Идеальным случаем является $S = p$ – линейное ускорение работы программы, что возможно при отсутствии доли последовательных операций, что является нереализуемым на практике по причине издержек на пересылку/объединение данных.

В свою очередь следствием из (1) является следующее утверждение:

$$S \leq \frac{1}{f}, \tag{2}$$

которое демонстрирует обратную пропорциональность доли последовательных операций в программе к возможному ускорению. Подобная зависимость также демонстрирует, что заметное ускорение возможно только при условии достаточного количества параллельных операций.

В рамках рассматриваемой задачи для получения оценки (1) необходимо предварительно оценить количество операций прямого и обратного хода Гаусса, которые были подвержены распараллеливанию.

В прямом ходе Гаусса происходит итеративное обнуление элементов в столбце под главным элементом, в ходе которого происходит операция деления главного элемента ведущей строки на множитель и операции умножения и вычитания ведущей строки из обнуляемой. Таким образом, количество операций в одной строке для матрицы, размером N на итерации k составляет $2(N-k)+1$. Данные преобразования выполняются для $N-k$ строк. Откуда количество операций для прямого хода составляет:

$$T_{forward} = \sum_{k=1}^{N-1} 2(N-k)^2 + N - K\,. \tag{3}$$

В свою очередь, обратный ход Гаусса подразумевает $N-k-1$ операций умножения, вычитания, а также одну операцию деления что дает общее количество операций:

$$T_{backward} = \sum_{k=1}^{N-1} 2(N-k-1) + 1\,. \tag{4}$$

Таким образом, получаем следующую оценку количества операций метода Гаусса в зависимости от порядка матрицы N:

$$T_{approx} \cong 2N^2 + 3N. \tag{5}$$

Выражение (5) демонстрирует порядки вычислительной сложности в зависимости от N. Можно наблюдать, что прямой ход Гаусса вносит наибольший вклад в долю операций, поэтому представляет наибольший интерес для

распараллеливания, тогда как распараллеливание обратного хода Гаусса в теории не несет большого прироста на больших размерностях N.

3. Результаты работы программы. Зависимость времени работы от количества потоков

В качестве результата работы программы изначально является графическое распределение температур на пластине, представленное на рис. 5. Для анализа эффективности распараллеливания, стандартная версия программы была масштабирована, в ходе чего была внедрена возможность выполнять решение с указанным количеством потоков. Так, задача была решена для 1, 4, 16, 64, 128, 256, 512 и 1024 потоков с замерами времени, требуемого для вычисления результирующего вектора $\bar{x}$. Зависимость времени вычислений от количества потоков представлена на рис.2.

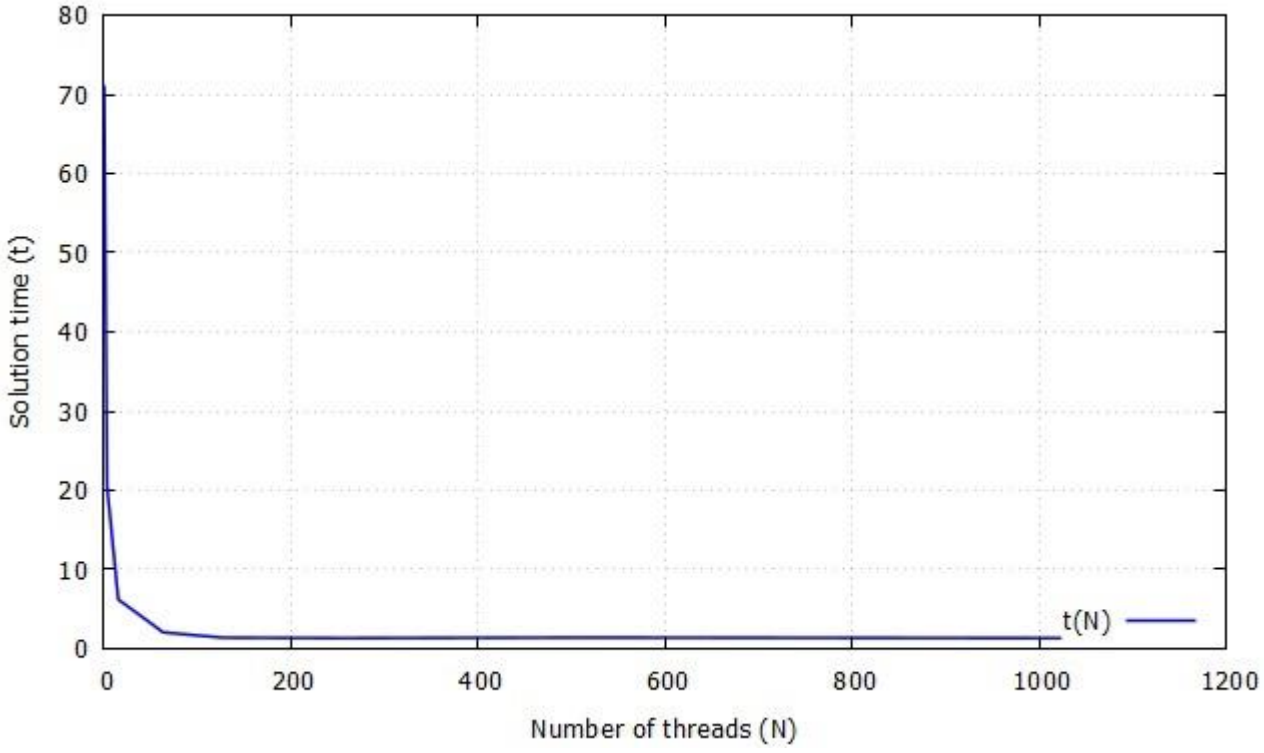

Рис. 2. Зависимость времени решения задачи от количества потоков.

Вычисления были произведены на ГП NVIDIA Geforce RTX 2060 в относительно изолированных условиях (отключены все внешние ПО, потребляющие ресурсы ГП больше, чем на 1%).

График зависимости на рис.2 демонстрирует ускорение работы при увеличении количества потоков, что говорит о корректности применения распараллеливания в рассматриваемой задачи.

Конкретные числовые данные приведены на рис. 3. Конфигурация ГП была также выведена в процессе работы программы и представлена на рис. 4.

```
[Performance] threads: '1': 71.137000 secs
[Performance] threads: '4': 20.589000 secs
[Performance] threads: '16': 6.213000 secs
[Performance] threads: '64': 2.059000 secs
[Performance] threads: '128': 1.395000 secs
[Performance] threads: '256': 1.342000 secs
[Performance] threads: '512': 1.404000 secs
[Performance] threads: '1024': 1.345000 secs
```

Рис. 3. Время выполнения программы для разного количества нитей.

```
Device name: NVIDIA GeForce RTX 2060
Total global memory: 2147024896
Shared memory per block: 49152
Registers per block: 65536
Warp size: 32
Memory pitch: 2147483647
Max threads per block: 1024
Max threads dimensions: x = 1024, y = 1024, z = 64
Max grid size: x = 2147483647, y = 65535, z = 65535
Clock rate: 1770000
Total constant memory: 65536
Compute capability: 7.5
Texture alignment: 512
Device overlap: 1
Multiprocessor count: 30
Kernel execution timeout enabled: true
```

Рис. 4. Конфигурация ГП.

4. Оценка накладных расходов и эффективности распараллеливания.

Накладные расходы представляют собой время на осуществление дополнительных операций, необходимых для работы параллельной программы, такие как пересылка и объединение данных. Вычисление подобных расходов осуществляется по следующей формуле:

$$T_0 = pT_p - T_1, \quad (3)$$

где p – количество процессоров, T_p – время выполнения распараллеленой программы, T_1 – время выполнения последовательной программы.

В свою очередь, ускорение работы программы при переходе с системы из одного процессора на систему из p процессоров – можно вычислить по формуле:

$$S = \frac{T_1}{T_p}. \quad (4)$$

Эффективность распараллеливания программы вычисляется как:

$$E = \frac{S}{p}. \quad (5)$$

Для полученной дискретной зависимости (рис. 2) результатов работы программы были вычислены перечисленные характеристики и представлены в Таблице 1:

Таблица 1. Сводные данные эффективности распараллеливания.

p	T_p	T_0	S	E
1	71.137	0	1	1
4	20.589	11.219	3.46	0.865
16	6.213	28.271	11.45	0.716
64	2.059	60.639	34.549	0.54
128	1.395	107.423	50.994	0.398
256	1.342	272.415	53.008	0.207
512	1.404	647.711	50.667	0.099
1024	1.345	1306.143	52.89	0.0516

Анализ результатов в Таблице 1 демонстрирует падение эффективности с увеличением количества потоков, что сопровождается ростом накладных расходов. Использование 16 потоков показывает большой разрыв в эффективности и накладных расходах по причине архитектуры ЦП, где реальных физических ядер всего 8, в то время как остальные 8 потоков –

виртуальные. Рост накладных расходов демонстрирует затраты ресурсов на коммуникацию между потоками, синхронизацию между ними.

5. Сравнение результатов работы программы с существующим ПО.

Для проверки достоверности непосредственных результатов работы программы (графическое распределение температуры по пластине – рис.3) поставленная задача была также решена средствами программного пакета ANSYS (рис. 4).

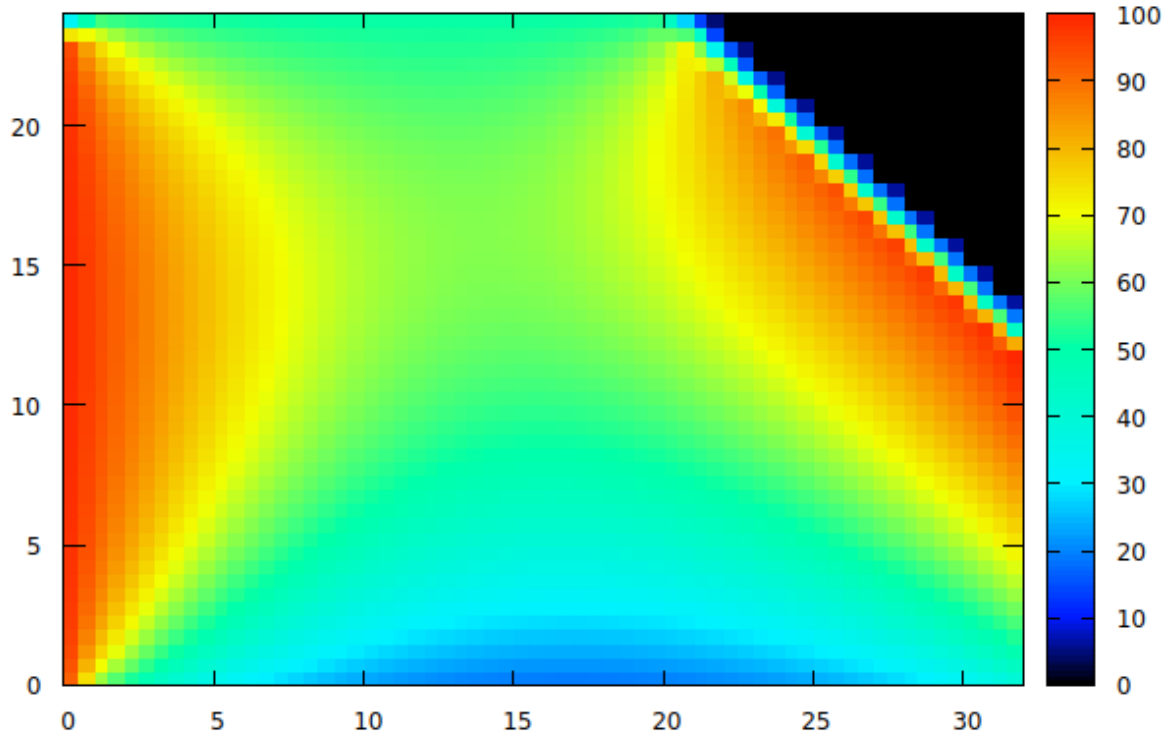

Рис. 5. Результаты работы программы для момента времени $t = 15$ с. Визуализация решения средствами gnuplot.

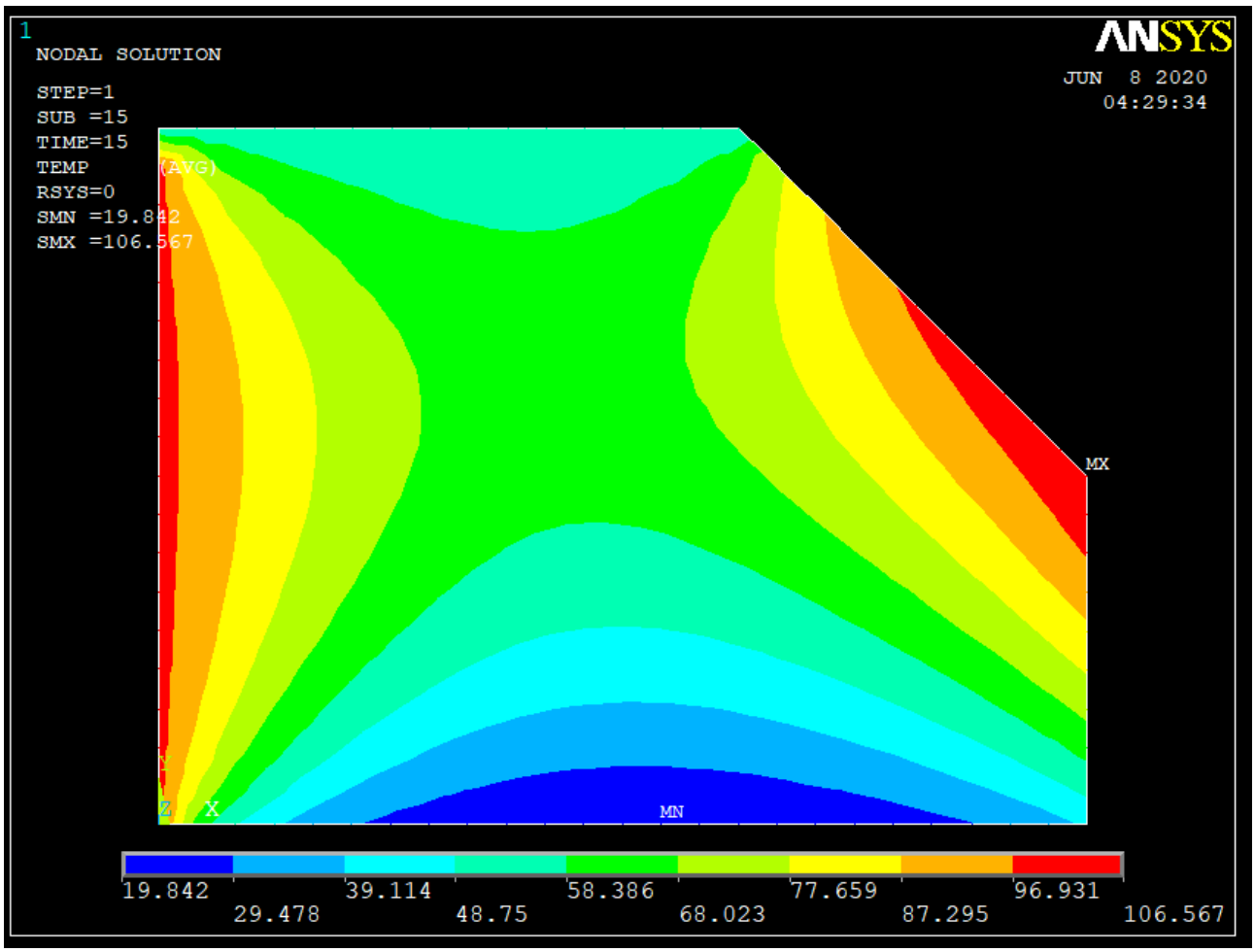

Рис.6. Распределение температур в пластине для момента времени t=15с, полученное с помощью программы ANSYS.

На основе сравнения распределения температур решеной задачи теплопроводности пластины на рис. 3 и 4 был сделан вывод о корректности работы программы.

Заключение

В ходе лабораторной работы было успешно применено распараллеливание исходной программы средствами CUDA. Была проведена оценка накладных расходов, а также анализ влияния количества потоков на эффективность работы программы, в ходе которого было установлено, что увеличение количества потоков до оптимального способствует выгодному ускорению программы при небольших накладных затратах.

Текст программы

Файл *main.cu*:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "solution.h"
#include "utils.h"

#define PERFORMANCE_RESULTS_FILENAME "performance.dat"
#define SOLUTION_FILENAME "solution.dat"

#define TIME_END 15
#define HEIGHT 6
#define WIDTH 8
#define RIGHT_BORDER 3
#define TOP_BORDER 5
#define STEP 0.8

#define ALPHA 1
#define TEMP_INIT 0


int main() {
    size_t y_nodes = (int) (HEIGHT / STEP + 1);
    size_t x_nodes = (int) (WIDTH / STEP + 1);

    size_t nodes_amount = (int) x_nodes * y_nodes;
    printf("Nodes amount: %llu\n", nodes_amount);

    double *temperatures = (double *) calloc(sizeof(double), nodes_amount);
    double *matrix = (double *) calloc(sizeof(double), nodes_amount *
nodes_amount);

    struct solution_params params{};
    params.time_end = TIME_END;
    params.step = STEP;
    params.height = HEIGHT;
    params.width = WIDTH;
    params.right_border = RIGHT_BORDER;
    params.top_border = TOP_BORDER;
    params.temp_init = TEMP_INIT;
    params.alpha = ALPHA;

    print_cuda_device_info();

    initialise(&temperatures, &matrix, &params);

    double *performance_stats = (double *) malloc(sizeof(double) * 8);
    size_t threads[] = {1, 4, 16, 64, 128, 256, 512, 1024};

    double *x = nullptr;

    for (int i = 0; i < 8; ++i) {
        free(x);
        x = solution(temperatures, matrix, &params, threads[i],
&(performance_stats[i]));
    }

//    x = solution(temperatures, matrix, &params, SOLUTION_CUDA_THREADS,
&(performance_stats[0]));
```

```
    if (write_solution(x, x_nodes, y_nodes, SOLUTION_FILENAME) < 0) {
        fprintf(stderr, "solution writing error");
        return -1;
    }

    if (write_performance(threads, performance_stats, 8,
PERFORMANCE_RESULTS_FILENAME) < 0) {
        fprintf(stderr, "performance writing error");
        return -1;
    }

    if (plot_solution(SOLUTION_FILENAME) < 0) {
        fprintf(stderr, "solution plotting error");
        return -1;
    }

    if (plot_performance(PERFORMANCE_RESULTS_FILENAME) < 0) {
        fprintf(stderr, "performance stats plotting error");
        return -1;
    }

    free(temperatures);
    free(matrix);
    free(x);

    return 0;
}
```

Файл *solution.cu*:

```
#include "solution.h"
#include "utils.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <ctime>


__global__ void gauss(double *A_matrix, double *y, double *x, size_t nodes_num,
double* reduction) {
    for (int k = 1; k < nodes_num; ++k) { // прямой ход
        for (int j = k + 1 + threadIdx.x; j < nodes_num; j += blockDim.x) {
            double pivot = A_matrix[j * nodes_num + k] / A_matrix[k * nodes_num
+ k];
            for (int i = k; i < nodes_num; i++) {
                A_matrix[j * nodes_num + i] -= pivot * A_matrix[k * nodes_num +
i];
            }

            y[j] = y[j] - pivot * y[k];
        }
    }

    for (int k = nodes_num - 1; k >= 1; k--) { // обратный ход
        __shared__ double total_sum;
        double sum = 0;
        for (int j = k + 1 + threadIdx.x; j < nodes_num; j += blockDim.x) {
            sum += A_matrix[k * nodes_num + j] * x[j];
        }

        reduction[threadIdx.x] = sum;
```

```
        __syncthreads();
        for (int size = blockDim.x / 2; size > 0; size /= 2) { //uniform
            if (threadIdx.x < size) {
                reduction[threadIdx.x] += reduction[threadIdx.x + size];
            }
            __syncthreads();
        }
        if (threadIdx.x == 0) {
            total_sum = reduction[0];
        }
        __syncthreads();

        x[k] = (y[k] - total_sum) / A_matrix[k * nodes_num + k];
    }
}

double *solution(double *y, double *A, struct solution_params *params, size_t
threads_num, double *performance_stats) {
    size_t y_nodes = (int) (params->height / params->step + 1);
    size_t x_nodes = (int) (params->width / params->step + 1);

    size_t nodes_amount = (int) x_nodes * y_nodes;

    double *x = (double *) calloc(sizeof(double), nodes_amount);
    double *x_device = nullptr;
    double *y_device = nullptr;
    double *A_device = nullptr;

    // Массив для суммирования в обратном ходе Гаусса (в случае параллелизма
необходима редукция)
    double *reduction_arr = nullptr;

    HANDLE_ERROR(cudaMalloc((void **) &x_device, nodes_amount *
sizeof(double)));
    HANDLE_ERROR(cudaMalloc((void **) &y_device, nodes_amount *
sizeof(double)));
    HANDLE_ERROR(cudaMalloc((void **) &reduction_arr, threads_num *
sizeof(double)));
    HANDLE_ERROR(cudaMalloc((void **) &A_device, nodes_amount * nodes_amount *
sizeof(double)));

    clock_t t = clock();
    for (int k = 0; k < params->time_end; ++k) {
        for (int i = 0; i < y_nodes; ++i) {
            for (int j = 0; j < x_nodes; ++j) {
                if (j > 0 && i > 0 && i < y_nodes - 1 && i > (j - params-
>top_border / params->step) && j < (x_nodes - 1)) {
                    y[i * x_nodes + j] = x[i * x_nodes + j];
                }
            }
        }

        HANDLE_ERROR(cudaMemcpy((void *) y_device, (void *) y, nodes_amount *
sizeof(double), cudaMemcpyHostToDevice));
        HANDLE_ERROR(cudaMemcpy((void *) x_device, (void *) x, nodes_amount *
sizeof(double), cudaMemcpyHostToDevice));
        HANDLE_ERROR(cudaMemcpy((void *) A_device, (void *) A, nodes_amount *
nodes_amount * sizeof(double), cudaMemcpyHostToDevice));

        gauss<<<1, threads_num>>>(A_device, y_device, x_device, nodes_amount,
reduction_arr);

//        HANDLE_ERROR(cudaDeviceSynchronize());
```

```
        HANDLE_ERROR(cudaMemcpy((void *) x, (void *) x_device, nodes_amount *
sizeof(double), cudaMemcpyDeviceToHost));
    }

    double total_time = (double) (clock() - t) / CLOCKS_PER_SEC;
    *performance_stats = total_time;
    printf("[Performance] threads: '%llu': %f secs\n", threads_num,
*performance_stats);
    return x;
}

void initialise(double **vector, double **matrix, struct solution_params
*params) {
    size_t y_nodes = (int) (params->height / params->step + 1);
    size_t x_nodes = (int) (params->width / params->step + 1);

    size_t nodes_amount = (int) x_nodes * y_nodes;

    for (int i = 0; i < y_nodes; i++) {
        for (int j = 0; j < x_nodes; j++) {
            if (i == 0 && j <= params->top_border / params->step) { // Верхняя
граница - ГУ 1-го рода
                (*vector)[i * x_nodes + j] = 50; // Само значение в векторе
температур
                (*matrix)[(i * x_nodes + j) * nodes_amount + (i * x_nodes + j)]
= 1; // Указание этого уравнения в СЛАУ
            } else if (j == 0) { // Левая граница - ГУ 1-го рода
                (*vector)[i * x_nodes + j] = 100;
                (*matrix)[(i * x_nodes + j) * nodes_amount + (i * x_nodes + j)]
= 1;
            } else if (i == y_nodes - 1) { // Нижняя граница - ГУ 3-го рода (-1
т.к. индексация с нуля)
                (*matrix)[(i * x_nodes + j) * nodes_amount + (i * x_nodes + j)]
= -1 / params->step - 1;
                (*matrix)[(i * x_nodes + j) * nodes_amount + ((i - 1) * x_nodes
+ j)] = 1 / params->step;
                (*vector)[i * x_nodes + j] = params->temp_init; // начальное
условие нестационарной задачи
            } else if (j >= params->top_border / params->step && i == (j -
params->top_border / params->step)) { // Правая скошенная граница - ГУ 2-го рода
                double cos_45 = sqrt(2) / 2;
                (*matrix)[(i * x_nodes + j) * nodes_amount + (i * x_nodes + j)]
= 1 / (sqrt(2) * params->step);
                (*matrix)[(i * x_nodes + j) * nodes_amount + ((i + 1) * x_nodes
+ j - 1)] = -1 / (sqrt(2) * params->step);
                (*vector)[i * x_nodes + j] = 20;
            } else if (i * params->step <= (params->height - params-
>right_border) && j >= params->top_border / params->step &&
                       i <= (j - params->top_border / params->step)) { // Пустая
область
                (*matrix)[(i * x_nodes + j) * nodes_amount + (i * x_nodes + j)]
= 1;
                (*vector)[i * x_nodes + j] = 0;
            } else if (j == x_nodes - 1) { // Правая граница - ГУ 2-го рода
                (*matrix)[(i * x_nodes + j) * nodes_amount + (i * x_nodes + j)]
= 1 / params->step;
                (*matrix)[(i * x_nodes + j) * nodes_amount + (i * x_nodes + j -
1)] = -1 / params->step;
                (*vector)[i * x_nodes + j] = 20;
            } else if (j < x_nodes - 1) { // Остальные (внутренние) узлы
                (*matrix)[(i * x_nodes + j) * nodes_amount + (i * x_nodes + j)]
= 1 + 2 / (params->step * params->step) + 2 / (params->step * params->step);
```

```
                (*matrix)[(i * x_nodes + j) * nodes_amount + (i * x_nodes + j +
1)] = -1 / (params->step * params->step);
                (*matrix)[(i * x_nodes + j) * nodes_amount + (i * x_nodes + j -
1)] = -1 / (params->step * params->step);
                (*matrix)[(i * x_nodes + j) * nodes_amount + ((i + 1) * x_nodes
+ j)] = -1 / (params->step * params->step);
                (*matrix)[(i * x_nodes + j) * nodes_amount + ((i - 1) * x_nodes
+ j)] = -1 / (params->step * params->step);
            }
        }
    }
}
```