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	Абидоков Рашид Ширамбиевич
Группа	РК6-11М
Тип задания	лабораторная работа
Тема лабораторной работы	Применение технологий OpenMP и CUDA

Студент _____ **Абидоков Р. Ш.**
подпись, дата *фамилия, и.о.*

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

Москва, 2020 г.

Оглавление

Цель выполнения лабораторной работы

Цель выполнения лабораторной работы – получение практических навыков применения технологий OpenMP и CUDA, оценка времени работы программы.

Задание на лабораторную работу

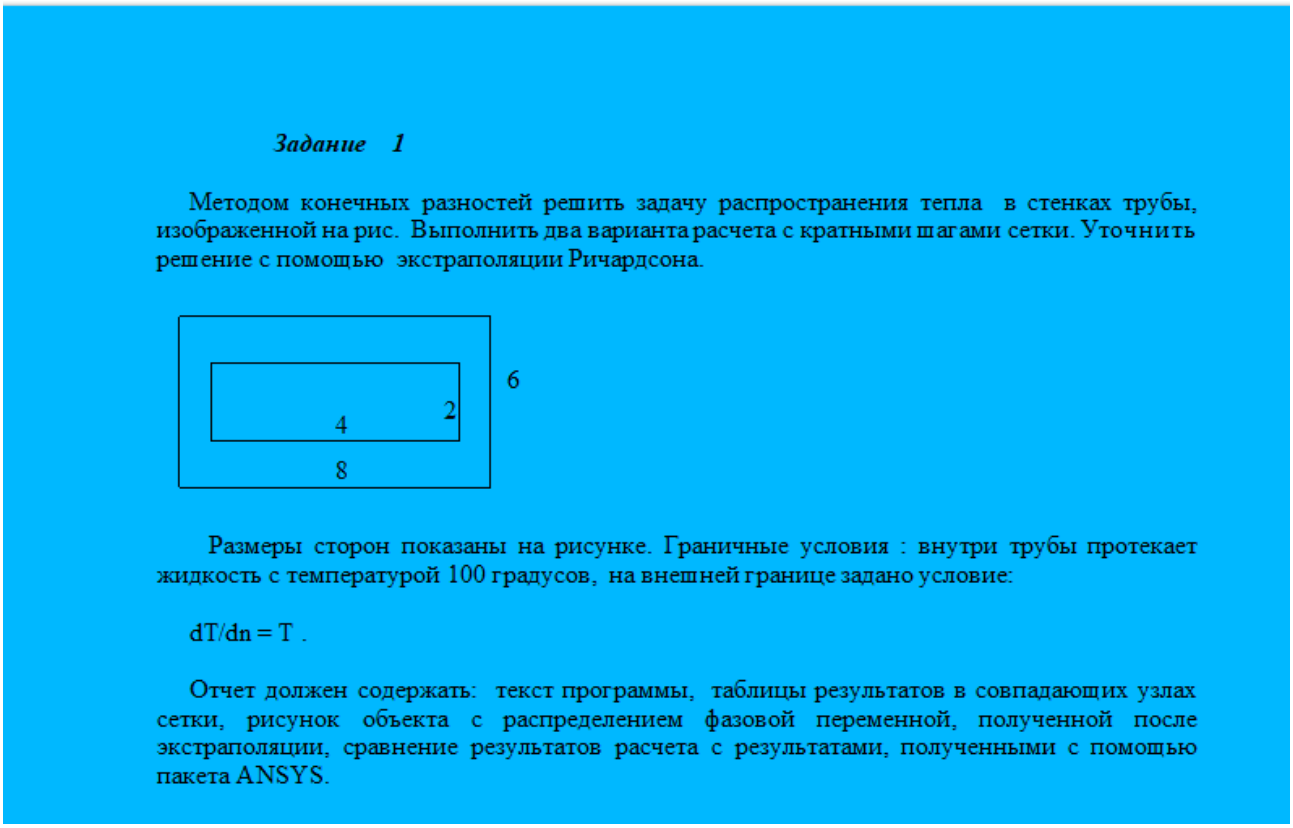

Рис. 1 Исходное задание

Однако для большей наглядности применения методов распараллеливания условие задачи было изменено – требуется решить нестационарную задачу, т.е. получить распределение температуры в зависимости от времени. Начальной температурой пластины считается 20 градусов Цельсия.

1. Реализация с помощью OpenMP

Поскольку практически всё время работы программы занимает решение СЛАУ методом Гаусса, распараллеливанию как в данном пункте, так и при использовании CUDA подвергался именно метод Гаусса. Исходный код реализующей его функции приведен в Листинге 1.

Листинг 1.

```
template <typename T>
vector<T>& gauss(const Matrix<T>& l_matr, const vector<T>& r_vect) {
  unsigned size_ = r_vect.size();
  if (l_matr.get_cols() != l_matr.get_rows()) {
    cout << "not square matrix in gauss" << endl;
    throw ("not square matrix in gauss");
  }
  if (size_ != l_matr.get_rows()) {
    cout << "l_matr size != r_vect size in gauss" << endl;
    throw ("l_matr size != r_vect size in gauss");
  }
  Matrix<T> l_matr_ = l_matr;
  vector<T> r_vect_ = r_vect;
  vector<T>& ret_vect = *(new vector<T>(size_));

  // Прямой ход
  // Внешний нельзя, важен порядок
  for (unsigned i = 0; i < size_; i++) {
    #pragma omp parallel for
    for (unsigned j = 0; j < i; j++) {
      #pragma omp atomic
      r_vect_[i] -= r_vect_[j] * l_matr_[i][j];
      l_matr_.plus_row(i, j, -l_matr_[i][j]);
    }
    r_vect_[i] /= l_matr_[i][i];
    l_matr_.set_diag_to_one_r(i);
  }

  // Обратный ход
  for (unsigned i = 0; i < size_; i++) {
    T sum = 0;
    #pragma omp parallel for
    // Здесь нельзя трогать наружный цикл
    for (unsigned j = 0; j < i; j++) {
      #pragma omp atomic
      sum += l_matr_[size_-i-1][size_-j-1] * ret_vect[size_-j-1];
    }
    ret_vect[size_-i-1] = r_vect_[size_-i-1] - sum;
  }
```

В явном виде количество потоков задается в функции main, фрагмент которой приведен в Листинге 2.

Листинг 2.

```
int main(int argc, char *argv[]) {
  // Выставляем количество потоков
  omp_set_dynamic(0);
  omp_set_num_threads(4);
```

2. Реализация с помощью CUDA

Поскольку прямой ход метода Гаусса имеет сложность $O(n^3)$, а обратный, в свою очередь, $O(n^2)$, при большом количестве узлов (когда распараллеливание является эффективным) наибольшее влияние на время выполнения оказывает именно прямой ход. Исходный код функции, реализующей метод Гаусса, приведен в Листинге 3.

Листинг 3.

```
vector<double>& gauss_cuda(const Matrix& l_matr, const vector<double>& r_vect) {
    size_t size = r_vect.size();
    // Делаем копию вектора правой части в виде массива
    double* r_vec = new double[size];
    for (size_t i = 0; i < size; i++) {
        r_vec[i] = r_vect[i];
    }
    // Количество блоков и тредов
    dim3 N_threads1(8);
    dim3 N_blocks1(size / 8 + 1);
    dim3 N_threads_once(1);
    dim3 N_blocks_once(1);
    // Выделяем память на девайсе
    double* l_matr_dev;
    double* r_vec_dev;
    double* ret_vec_dev;
    size_t l_matr_size = size * size * sizeof(double);
    size_t vec_size = size * sizeof(double);
    cudaMalloc((void**) &l_matr_dev, l_matr_size);
    cudaMalloc((void**) &r_vec_dev, vec_size);
    cudaMalloc((void**) &ret_vec_dev, vec_size);
    // Зануляем правый вектор
    fillVecKernel<<<N_blocks1, N_threads1>>>(ret_vec_dev, size, 0);
    // Копируем туда данные
    cudaMemcpy(l_matr_dev, l_matr.get_arr(), l_matr_size, cudaMemcpyHostToDevice);
    cudaMemcpy(r_vec_dev, r_vec, vec_size, cudaMemcpyHostToDevice);
```

```
    // Прямой ход
    // Приводим матрицу к верхнетреугольной
    double* mcoeff;
    cudaMalloc((void**)&mcoeff, sizeof(double));
    for (size_t i = 0; i < size; i++) {
        for (size_t j = 0; j < i; j++) {
            calculateMCoeffKernel<<<N_threads_once, N_blocks_once >>>(l_matr_dev, size, i, j, mcoeff);
            plusRowKernel<<<N_blocks1, N_threads1>>>(l_matr_dev, r_vec_dev, size, i, j, mcoeff);
        }
     }
    cudaFree(mcoeff);
    cudaDeviceSynchronize();
    // Выводим единицы на главной диагонали
    // Вытаскиваем диагональные элементы, чтобы модифицировать матрицу
    double* diag_vec_dev;
    cudaMalloc((void**)&diag_vec_dev, vec_size);
    diagElemsToVecKernel <<<N_blocks1, N_threads1 >>> (l_matr_dev, diag_vec_dev, size);
    // Диагональный элементы строк в единицы
    for (size_t i = 0; i < size; i++) {
        setDiagToOneKernel <<<N_blocks1, N_threads1 >>> (l_matr_dev, r_vec_dev, diag_vec_dev, size, i);
    }
    // И правый вектор
    rvecDiagDivKernel<<<N_blocks1, N_threads1 >>>(r_vec_dev, diag_vec_dev, size);
    cudaFree(diag_vec_dev);
    // Обратный ход
    double* l_matr_host = new double[size*size];
    double* r_vec_host = new double[size];
    cudaMemcpy(l_matr_host, l_matr_dev, l_matr_size, cudaMemcpyDeviceToHost);
    cudaMemcpy(r_vec_host, r_vec_dev, vec_size, cudaMemcpyDeviceToHost);
    vector<double>& ret_vect = *(new vector<double>(size));
    for (size_t i = 0; i < size; i++) {
        double sum = 0;
        for (size_t j = 0; j < i; j++) {
            sum += l_matr_host[(size - i - 1)*size + (size - j - 1)] * ret_vect[size - j - 1];
        }
        ret_vect[size - i - 1] = r_vec_host[size - i - 1] - sum;
    }
```

Используемые Kernel-функции в Листингах 4 и 5.

Листинг 4.

```
__global__ void fillVecKernel(double* vec, size_t size, double val) {
    size_t idx = threadIdx.x + blockIdx.x *blockDim.x;
    if (idx < size) {
        vec[idx] = val;
    }
}
__global__ void calculateMCoeffKernel(double* l_matr, size_t size, size_t i, size_t j, double* mcoeff) {
    *mcoeff = l_matr[i * size + j] / l_matr[j * size + j];
}

__global__ void plusRowKernel(double* l_matr, double* r_vec, size_t size,
                              size_t mr_idx, size_t pr_idx, double* mcoeff) {
    size_t idx = threadIdx.x + blockIdx.x *blockDim.x;
    if (idx < size) {
        l_matr[mr_idx * size + idx] -= l_matr[pr_idx * size + idx] * (*mcoeff);
    }
}
```

Листинг 5.

```
__global__ void diagElemsToVecKernel(double* l_matr, double* diag_vec, size_t size) {
    size_t idx = threadIdx.x + blockIdx.x *blockDim.x;
    if (idx < size) {
        diag_vec[idx] = l_matr[idx*size + idx];
    }
}

__global__ void setDiagToOneKernel(double* l_matr, double* r_vec, double* diag_vec, size_t size,
                                   size_t mr_idx) {
    size_t idx = threadIdx.x + blockIdx.x *blockDim.x;
    if (idx < size) {
        l_matr[mr_idx * size + idx] /= diag_vec[mr_idx];
    }
}

__global__ void rvecDiagDivKernel(double* r_vec, double* diag_vec, double size) {
    size_t idx = threadIdx.x + blockIdx.x *blockDim.x;
    if (idx < size) {
        r_vec[idx] /= diag_vec[idx];
    }
}
```

3. Сравнение времени работы

Сравнение времени работы однопоточной и многопоточных реализаций проводилось на компьютере с процессором Intel Core i7-6500u, имеющем 2 ядра с 4 потоками, и видеокартой gtx-960m. Для автоматизации использовался скрипт, написанный на языке python, код которого приведен в Листинге 6.

Листинг 6.

```
import numpy as np
import pandas as pd
import subprocess


single_path = "single\\out\\main.exe"
openmp_path = "openmp\\out\\main.exe"
cuda_path = "cuda\\x64\\Release\\cuda.exe"

# Сравниваем при различных разбиениях при постоянном количестве шагов интегрирования
t_end = 10
h_t = 0.2
N_x_list = np.append(np.array([4]), [np.arange(8, 48, 8)])
N_y_list = np.append(np.array([3]), [np.arange(6, 36, 6)])
N_list = np.array([N_x_list, N_y_list]).T
print(N_list)

```

```python
N_nodes_data = pd.DataFrame(columns=['N_nodes', 'cuda_total', 'cuda_avg'])
for i, [N_x, N_y] in enumerate(N_list):
    print(i, N_x, N_y)
    # Однопоточное решение
    single_curr = subprocess.run([single_path, str(t_end), str(h_t), str(N_x), str(N_y), '-d'], stdout=subprocess.PIPE).stdout
    n_nodes, single_total, single_avg = [float(s) for s in single_curr.split()]
    print('single_total', single_total)
    print('single_avg', single_avg)
    # OpenMP решение
    openmp_curr = subprocess.run([openmp_path, str(t_end), str(h_t), str(N_x), str(N_y), '-d'], stdout=subprocess.PIPE).stdout
    n_nodes, openmp_total, openmp_avg = [float(s) for s in openmp_curr.split()]
    print('openmp_total', openmp_total)
    print('openmp_avg', openmp_avg)
    # CUDA решение
    cuda_curr = subprocess.run([cuda_path, str(t_end), str(h_t), str(N_x), str(N_y), '-d'], stdout=subprocess.PIPE).stdout
    n_nodes, cuda_total, cuda_avg = [float(s) for s in cuda_curr.split()]
    print('cuda_total', cuda_total)
    print('cuda_avg', cuda_avg)
    print('\n')
    N_nodes_data.loc[i] = [n_nodes, single_total, single_avg, openmp_total, openmp_avg, cuda_total, cuda_avg]

print(N_nodes_data)
N_nodes_data.to_csv("N_nodes_compare.csv")
```

Всё время указано в мс, столбцы "шаг" – общее время, деленное на количество шагов интегрирования

Таблица 1.

N узлов	Однопоточн. всего	Однопоточн. шаг	OpenMP всего	OpenMP шаг	CUDA всего	CUDA шаг
40	31.0	0.6	299.0	6.0	130.0	2.6
160	1354.0	27.08	1794.0	35.88	594.0	9.2
360	14783.0	295.66	10096.0	201.92	1975.0	35.22
640	82487.0	1649.74	48314.0	966.28	4955.0	92.08
1000	287670.0	5753.4	179644.0	3592.88	11382.0	214.68

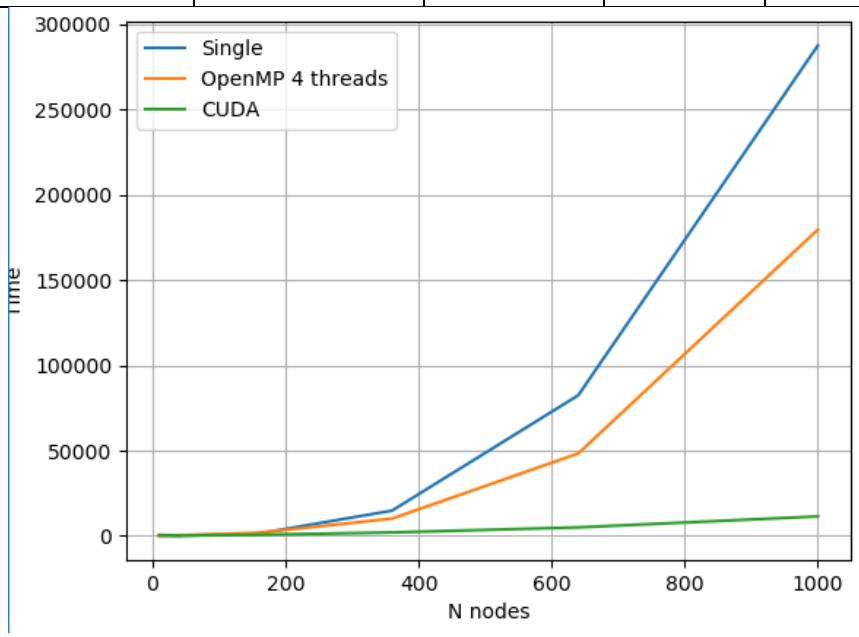

Рис 2.

Более наглядным является график кубического корня времени от количества узлов. Т.к. сложность алгоритма $O(n^3)$, графики похожи на прямые линии. Видно, что при малом количестве узлов (и, как следствие, малой сложности) параллельные реализации проигрывают в скорости однопоточной, что связано с накладными расходами при распараллеливании.

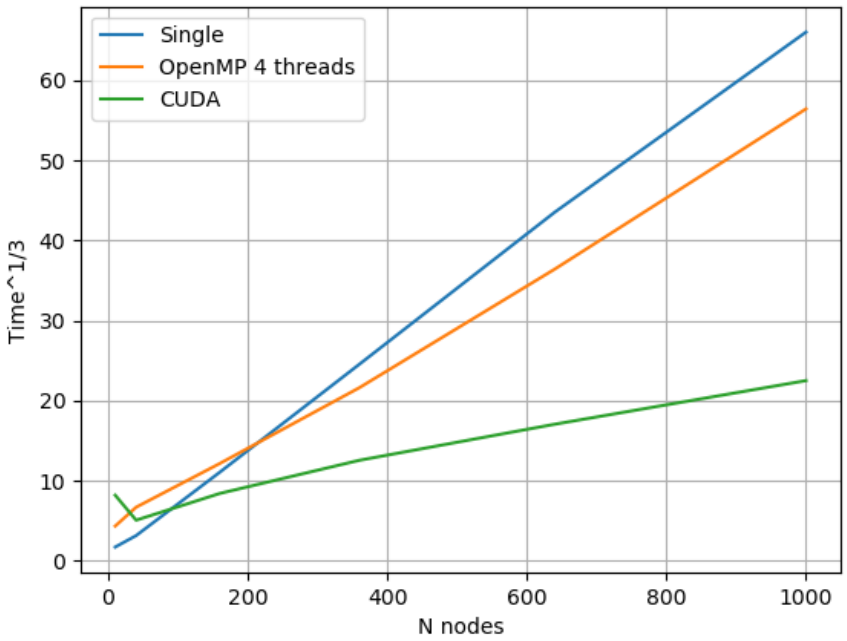

Рис 3.

Графики отношения времени однопоточной реализации к времени многопоточных

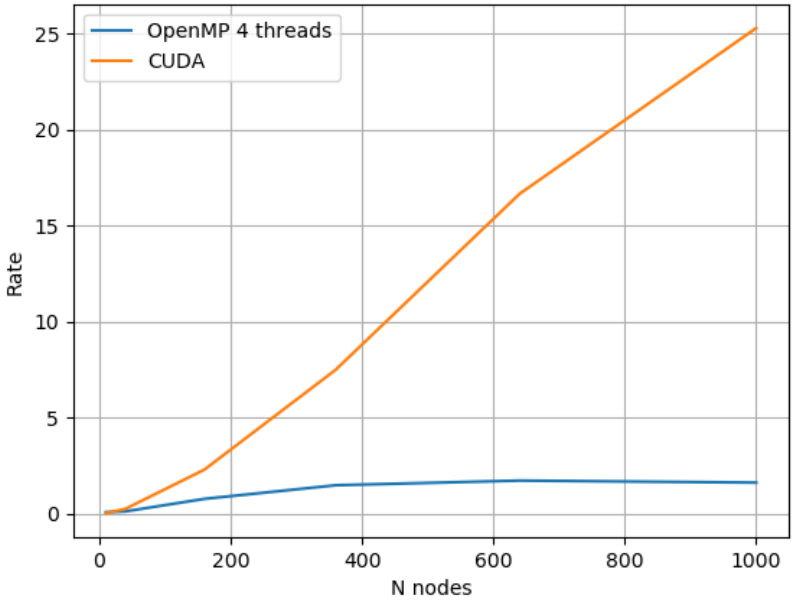

Рис 4.

Заключение

Было реализовано распараллеливание метода Гаусса с использованием технологий OpenMP и CUDA. Проведено сравнение времени работы однопоточной и многопоточных реализаций, показано, что даже при локальном применении данных технологий прирост производительности является существенным.

При этом в случае, если вычислительная сложность задачи не очень велика, распараллеливание может не только не привести к ускорению работы программы, но даже замедлить её, что связано с возникающими накладными расходами, связанными с управлением памятью и синхронизацией потоков.