Ирин Сергей Георгиевич

M19-B01

Отчет по лабораторной работе №5.

Бизнес логика варианта 8:

Сформировать результирующий вектор как среднее по каждой строке исходной квадратной матрицы.

В данной работе рассматривается вариант распараллеливания посредством OpenMPI + CUDA.

Данные генерятся Python скриптом и передаются исполняющей программе по TCP. Результирующий вектор возвращается по сетевому протоколу процессу генератора. В нём проверяется правильность полученного результата, затем данные о времени исполнения выводятся в консоль. Скрипт принимает три параметра: размер генерируемых данных в мегабайтах, порт для отправки данных и порт для получения результирующего вектора со временем исполнения. Сам скрипт приведен ниже.

```
import sys, os, math, socket
import numpy as np

'''
Random matrix stream generator for
Hybrid computing labs
Usage:
<script_name>.py [stream_size_in_MegaBytes] [tcp_port_out] [tcp_port_in]
Produces stream with square matrix of floats
and sends it as byte stream to tcp_port_out
and returns result received from tcp_port_in
'''

NUM_AVERAGES_DISPLAYED = 3

def send_file_to_socket(port, file_path):
    '''
    Sends string as byte stream to an open tcp socket
    '''
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = ('localhost', int(port))
    sock.connect(server_address)

    with open(file_path, 'rb') as input_file:
        sock.sendfile(input_file)

    sock.close()
    os.remove(file_path)
```

```
def get_string_from_socket(port, check_vector):
    '''
    Listens to tcp port until get
    all the stream and then prints it
    '''
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = ('localhost', int(port))
    sock.bind(server_address)
    sock.listen(1)
    connection, _ = sock.accept()

    result = []
    data = connection.recv(256)
    while data:
        result.append(data)
        data = connection.recv(256)
    result_str = ''.join([part.decode() for part in result])
    lines = result_str.split('\n')
    execution_time = lines[-2]
    numbers = lines[0].split(' ')

    assert len(numbers)-1 == len(check_vector)
    for i in range(len(check_vector)):
        assert math.fabs(float(numbers[i]) - check_vector[i]) < 0.0000001

    print(len(numbers)-1)
    print(' '.join(numbers[:NUM_AVERAGES_DISPLAYED]))
    print(execution_time)

    connection.close()
    sock.close()


def generate_stream_with_matrix(sizePorts):
    '''
    Function to generate file with
    random matrix of given size
    '''
    sizeInBytes = float(sizePorts[0]) * (2**20) # 1 Mb = 2^20 bytes
    num_rows_and_cols = int(math.sqrt(sizeInBytes / 11)) # 11 byte chars for each
 number
    matrix = np.random.rand(num_rows_and_cols, num_rows_and_cols)
    print(len(matrix))
    avg_vector = [round(sum(vector) / len(vector), 8) for vector in matrix]
    print(' '.join([str(avg) for avg in avg_vector[:NUM_AVERAGES_DISPLAYED]]))
    matrix_file_path = str(num_rows_and_cols) + '.mtrx'
    np.savetxt(matrix_file_path, matrix, fmt='%.8f', delimiter=' ')
    # matrix_str = np.array2string(matrix, formatter={'float_kind':lambda x: "%.8
f" % x}, separator=' ').replace('[','').replace(']','')
```

```
    # print(matrix_str.split('\n')[0])
    send_file_to_socket(sizePorts[1], matrix_file_path)
    get_string_from_socket(sizePorts[2], avg_vector)


if __name__ == "__main__":
    if len(sys.argv) > 2:
        generate_stream_with_matrix(sys.argv[1:])
    else:
        print("Usage: <script_name>.py [stream_size_in_MegaBytes] [tcp_port_out]
[tcp_port_in]")
```

Описание алгоритма выполнения бизнес-логики:

Программа включает вспомогательные функции:

1) int* listen_to_port(int port) – открывает соединение для переданного порта и возвращает дескрипторы соединения и сокета.
2) void close_connection(int connection, int sfd) – закрывает соединение и сокет.
3) void send_stream_to_port(int port, char* stream) – отправляет массив символов на соединение, открытое по указанному порту.

Логика получения по сети матрицы воплощена в функции int get_stream_matrix(int connection, std::vector<std::vector<float>> &matrix). В ней передаваемая матрица считывается и записывается в двумерный массив.

Логика получения результирующего вектора воплощена в функции-ядре CUDA __global__ void get_avg_vector(float *matrix, float *result, int row_size, int num_rows). Она принимает в качестве аргументов матрицу в виде одномерного массива чисел уже скопированного в память устройства GPU, одномерный массив чисел, размеченный в памяти устройства, в котором мы получим результаты, а также количество чисел в каждой строке и количество строк в данном куске матрицы. В самом ядре воплощен обход всех чисел переданной матрицы с шагом по гриде, каждое число суммируется к ячейке результирующего массива соответствующей номеру строки матрицы.

Логика распараллеливания воплощена в функции int main(int argc, char* argv[]). В качестве параметров командной строки получаются порты для получения данных матрицы и отправки результата. Далее происходит инициализация MPI и в бесконечном цикле исполняется следующий алгоритм: процесс с рангом 0 создает соединение по порту ввода и ждёт данные матрицы, считывает их, когда они появляются, затем проводит расчёты каким образом разделить полученную матрицу при имеющемся количестве процессов, остальные процессы ждут завершения всех

вышеописанных операций с помощью функции MPI_Barrier(MPI_COMM_WORLD). Далее первый процесс с помощью MPI_Bcast передает всем остальным данные, которые им понадобятся для правильной обработки матрицы, так как она была расплюснута в одномерный массив перед отправкой им. Отправка строк матрицы, средние значения для которых будет вычислять процесс, передаются с помощью функции MPI_Scatterv. Далее каждый процесс вычисляет свой кусок результирующего вектора средних. Кроме того, вычисляются параметры отступов данных и их количества для последующей склейки. Куски результирующего вектора передаются функции MPI_Gatherv, которая их склеивает в один. Далее процесс с рангом 0 добавляет к результирующей строке данные о времени выполнения и размере матрицы в байтах, после чего отправляет результат на порт вывода. Время выполнения измеряется с помощью функции MPI_Wtime и засекается с момента вызова MPI_Scatterv до получения первым процессом результирующего вектора. Здесь же в функции main воплощена логика резервирования памяти на устройстве CUDA и вызов функции-ядра, в котором производятся вычисления средних по куску матрицы.

В качестве эксперимента производились запуски программы на одном, двух и четырех процессах с матрицами размером от одного мегабайта до одного гигабайта (1мб, 5мб, 10мб, 100мб, 1000мб). Наблюдается прирост производительности в среднем на 107% при исполнении на двух процессах по сравнению с одним и прирост на 92% на четырех по сравнению с одним. На четырех процессах по сравнение с двумя расчеты происходят быстрее почти на 52%.

Код программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <string.h>
#include <time.h>
#include <vector>
#include <math.h>
#include <mpi.h>
// CUDA runtime
#include <cuda_runtime.h>
// helper functions and utilities to work with CUDA
#include <helper_functions.h>
#include <helper_cuda.h>

#define WORD_SIZE 11
#define MAX_SIZE 12

int* listen_to_port(int port)
```

```
{
    int sfd, connection;

    int* opts = new int[1]{ 1 };

    if ((sfd = socket(AF_INET, SOCK_STREAM, 0)) == 0
        || setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, opts,
sizeof(int)))
        exit(EXIT_FAILURE);

    struct sockaddr_in address;
    int addrlen = sizeof(address);
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(port);

    if (bind(sfd, (struct sockaddr*)&address,
        sizeof(address)) < 0 || listen(sfd, 3) < 0
        || (connection = accept(sfd, (struct sockaddr*)&address,
(socklen_t*)&addrlen)) < 0)
        exit(EXIT_FAILURE);

    return new int[2]{ connection, sfd };
}

void close_connection(int connection, int sfd)
{
    shutdown(connection, SHUT_RDWR);
    close(connection);
    shutdown(sfd, SHUT_RDWR);
    close(sfd);
}

void send_stream_to_port(int port, char* stream)
{
    int sock = 0;
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        exit(EXIT_FAILURE);

    struct sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(port);

    if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) <
0)
        exit(EXIT_FAILURE);

    send(sock, stream, strlen(stream), 0);

    shutdown(sock, SHUT_WR);
    close(sock);
}

int get_stream_matrix(int connection, std::vector<std::vector<float>>&
matrix)
{
    int buffer_size = 1;
    char* buffer = new char[buffer_size];

    std::vector<char> number;
```

```
    std::vector<float> line;
    int stream_size = 0;
    while (read(connection, buffer, buffer_size) > 0)
    {
        stream_size++;
        int num_bytes = 1;

        while (buffer[0] != '\n' && num_bytes > 0)
        {
            if (buffer[0] == ' ')
            {
                line.push_back(atof(&number[0]));
                number.clear();
            }
            else
                number.push_back(buffer[0]);

            num_bytes = read(connection, buffer, buffer_size);
            stream_size++;
        }
        line.push_back(atof(&number[0]));
        number.clear();

        matrix.push_back(line);

        line.clear();
    }

    return stream_size;
}

__global__ void get_avg_vector(float *matrix, float *result, int row_size,
int num_rows)
{
    int row_idx = blockIdx.x * blockDim.x + threadIdx.x;
    int max_idx = row_size * num_rows;

    if (row_idx < max_idx)
    {
        for (int i = row_idx; i < max_idx; i += blockDim.x * gridDim.x)
        {
            atomicAdd(&result[i / row_size], matrix[i] / row_size);
        }
    }
}

int main(int argc, char* argv[])
{
    if (argc < 3)
    {
        printf("Need input and output streams ports as parameters!");
        return -1;
    }

    int input_port = atoi(argv[1]);
    int output_port = atoi(argv[2]);

    int devId = findCudaDevice(argc, (const char **)argv);

    MPI_Init(&argc, &argv);
```

```
    for (;;) // forever and ever
    {
        int rank, numtasks, stream_size, num_elements, row_size;

        std::vector<std::vector<float>> matrix;
        std::vector<float> flat_matrix;
        std::vector<float> matrix_row;

        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

        int* sendcounts = new int[numtasks];
        int* displs = new int[numtasks];

        if (rank == 0) // only the first process communicates with
external
        {
            int* connection = listen_to_port(input_port);

            stream_size = get_stream_matrix(connection[0], matrix);

            close_connection(connection[0], connection[1]);

            int total_size = matrix.size() * matrix.size();

            for (int i = 0; i < matrix.size(); i++)
                for (int j = 0; j < matrix[i].size(); j++)
                    flat_matrix.push_back(matrix[i][j]);

            row_size = (int)matrix.size();
            num_elements = (int)floor(total_size / numtasks);
            num_elements -= num_elements % row_size;

            // calculate send counts and displacements
            int sum = 0;
            for (int i = 0; i < numtasks; i++) {
                sendcounts[i] = num_elements;

                if (i == numtasks - 1)
                    sendcounts[i] = total_size - sum;

                displs[i] = sum;
                sum += sendcounts[i];
            }
        }

        MPI_Barrier(MPI_COMM_WORLD);

        MPI_Bcast(&num_elements, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(&row_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(sendcounts, numtasks, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(displs, numtasks, MPI_INT, 0, MPI_COMM_WORLD);

        int num_rows = sendcounts[rank] / row_size;

        std::vector<float> partial_matrix;
        partial_matrix.resize(row_size * row_size);

        double start;
        if (rank == 0) start = MPI_Wtime(); // only the first process
controls the timing
```

```
        MPI_Scatterv(flat_matrix.data(), sendcounts, displs, MPI_FLOAT,
partial_matrix.data(), sendcounts[numtasks-1], MPI_FLOAT, 0,
MPI_COMM_WORLD);

        // CUDA kernel data initialization
        std::vector<float> lines_matrix =
std::vector<float>(partial_matrix.begin(), partial_matrix.begin() +
(num_rows * row_size));
        int matrix_size = lines_matrix.size();

        int row_float_size = num_rows * sizeof(float);
        int float_size = matrix_size * sizeof(float);

        float *a_matrix = (float*)malloc(float_size);
        for (int i = 0; i < matrix_size; i++)
            a_matrix[i] = lines_matrix[i];

        float *d_matrix;
        float *d_result;

        cudaMalloc(&d_matrix, float_size);
        cudaMemcpy(d_matrix, a_matrix, float_size,
cudaMemcpyHostToDevice);

        cudaMalloc(&d_result, row_float_size);
        cudaMemset(d_result, 0, row_float_size);

        int numSMs;
        cudaDeviceGetAttribute(&numSMs, cudaDevAttrMultiProcessorCount,
devId);

        cudaDeviceProp props;
        cudaGetDeviceProperties(&props, devId);

        get_avg_vector<<<numSMs, 256>>>(d_matrix, d_result, row_size,
num_rows);

        float *float_result = (float*)malloc(row_float_size);
        cudaMemcpy(float_result, d_result, row_float_size,
cudaMemcpyDeviceToHost);

        std::vector<char> partial_result;
        partial_result.reserve(num_rows);

        for (int i = 0; i < num_rows; i++)
        {
            char* avg_str = new char[MAX_SIZE];
            sprintf(avg_str, "%.8f ", float_result[i]);
            partial_result.insert(partial_result.end(), avg_str, avg_str +
strlen(avg_str));
        }

        cudaFree(d_result);
        cudaFree(d_matrix);
        free(float_result);
        free(a_matrix);

        // CUDA finished

        std::vector<char> parallel_result;
```

```
        if (rank == 0) parallel_result.reserve(WORD_SIZE * row_size);

        int* recvcounts = new int[numtasks];
        int* rdispls = new int[numtasks];
        int sum = 0;
        int char_size = num_elements / row_size * WORD_SIZE;
        for (int i = 0; i < numtasks; i++) {
            recvcounts[i] = char_size;

            if (i == numtasks - 1)
                recvcounts[i] = (row_size * WORD_SIZE) - sum;

            rdispls[i] = sum;
            sum += recvcounts[i];
        }

        MPI_Gatherv(partial_result.data(), (int)partial_result.size(),
MPI_CHAR, parallel_result.data(), recvcounts, rdispls, MPI_CHAR, 0,
MPI_COMM_WORLD);

        if (rank == 0) // only the first process communicates with
external
        {
            char* result = new char[sum + 2];
            snprintf(result, sum + 1, "%s", &parallel_result[0]);

            double end = MPI_Wtime();
            double execution_time_in_seconds = (double)(end - start) *
1000;

            char* buffer = new char[sum + 3 + 256];
            snprintf(buffer, sum + 2 + 256, "%s\n%d bytes in %.9f
milliseconds\n", result, stream_size, execution_time_in_seconds);
            printf("%s\n", buffer);

            send_stream_to_port(output_port, buffer);
        }
    }

    MPI_Finalize();

    return 0;
}
```

Проект целиком с Makefile’ом можно посмотреть онлайн здесь: https://github.com/RinSer/MephiHybridComputing/tree/master/Lab1/Lab5

График зависимости времени выполнения от размера данных (по горизонтали отложены мегабайты, по вертикали – время исполнения в миллисекундах), синий график – исполнение на одном треде, жёлтый – параллельное на двух тредах, красный – параллельное на четырех: