Федеральное государственное бюджетное образовательное учреждение высшего образования «Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Кафедра прикладной математики и кибернетики

РАСЧЕТНО-ГРАФИЧЕСКОЕ ЗАДАНИЕ

по дисциплине «Программирование графических процессоров»

Выполнил:

студент гр. И\*–\*4\*

Некто Н.Н.

Проверил:Старший преподаватель Кафедры ПМиК

Милешко А.В.

Новосибирск 2020 г.

**Задание**

Провести анализ производительности программ, реализующих алгоритмы линейной алгебры с использованием библиотек Thrust, cuBLAS и «сырого» кода на CUDA C.

В данном расчётно-графическом задании для реализации выбран алгоритм матрично-векторного умножения.

**Выполнение**

Необходимо реализовать три версии программы матрично-векторного умножения:

1. «Сырая» версия на Си *CUDA*;
2. С использованием библиотеки *Thrus*t;
3. С использованием библиотеки *cuBLAS*.

Умножение матрицы на вектор производится по правилу «строка на столбец». В результате умножения матрицы размера *N* \* *M* на вектор-строку размером *N* на выходе получаем вектор-столбец размером *M*. Число столбцов в матрице должно совпадать с числом столбцов в векторе-строке.

Рисунок 1 – Матрично-векторное умножение

Технология *CUDA* — это программно-аппаратная вычислительная архитектура *NVidia*, основанная на расширении языка Си, которая даёт возможность организации доступа к набору инструкций графического ускорителя и управления его памятью при организации параллельных вычислений [3].

*Thrust* - это библиотека шаблонов для *CUDA*, основанная на стандартной библиотеке шаблонов (*STL*) языка Си++. Тяга позволяет реализовать высокопроизводительные параллельные приложения с минимальными усилиями программирования через высокоуровневый интерфейс, который полностью совместим с *CUDA* Си [2].

Библиотека *cuBLAS* является реализацией библиотеки *BLAS* для программ, исполняемых на видеокартах *NVidia*, поддерживающих технологию *CUDA*. Она позволяет программисту получить доступ к вычислительным ресурсам GPU [1].

По заданию необходимо исследовать производительность трёх версий программ. Замеры проводились на квадратных матрицах размером *N* \* *N* и векторах размером *N*.

Таблица 1 – Замеры времени выполнения программы

|  |  |  |  |
| --- | --- | --- | --- |
| Количество элементов (N) | Время выполнения | | |
| Си CUDA | Thrust | cuBLAS |
| 100 | 0,095261 | 0,337365 | 0,093516 |
| 200 | 0,102473 | 1,203294 | 0,092172 |
| 300 | 0,103429 | 18,972449 | 0,093888 |
| 400 | 0,115121 | 33,822699 | 0,093172 |
| 500 | 0,109679 | 52,559943 | 0,093516 |
| 600 | 0,113992 | 76,459078 | 0,093059 |
| 700 | 0,115656 | 102,693482 | 0,093130 |
| 800 | 0,116946 | 134,003864 | 0,092905 |
| 900 | 0,118503 | 181,893446 | 0,093975 |
| 1000 | 0,122163 | 223,751513 | 0,092724 |

Исходя из данных, полученных в ходе замеров, было выявлено, что время выполнения программы на Си *CUDA* и программы, использующей библиотеку *cuBLAS* примерно равно константе , в то время как время выполнения программы, использующей библиотеку *Thrust*, сильно зависит от размера перемножаемых матрицы и вектора.

**Список использованных источников**

1. cuBLAS Library [Электронный ресурс]. – URL: https://docs.nvidia.com/cuda/pdf/

CUBLAS\_Library.pdf (Дата обращения: 03.05.2020)

2. Thrust :: CUDA Toolkit Documentation [Электронный ресурс]. – URL: https://docs.nvidia.com/cuda/thrust/index.html (Дата обращения: 03.05.2020)

3. Nvidia CUDA ? неграфические вычисления на графических процессорах [Электронный ресурс]. – Электрон. текстовые данные. – URL: https://www.ixbt.

com/video3/cuda-1.shtml (Дата обращения: 03.05.2020)

**Исходный код проекта**

#include <thrust/device\_vector.h>

#include <thrust/transform.h>

#include "cublas\_v2.h"

#include <cuda.h>

#include <time.h>

#include <random>

using namespace std;

//Определяем индекс матрицы

#define IDX2C(i, j, ld)(((j) \*(ld)) +(i))

void fill\_rand(float \*pointer, const int size)

{

random\_device random;

mt19937 mt(random());

uniform\_real\_distribution<float> dist(-10.0, 10.0);

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

pointer[i] = dist(mt);

}

}

\_\_global\_\_ void kernel(const float \*device\_M, const float \*device\_A, float \*device\_Result, const int rows, const int cols)

{

unsigned x = blockIdx.x \* blockDim.x + threadIdx.x;

device\_Result[x] = 0;

for(int j = 0; j < cols; ++j) {

device\_Result[x] += device\_M[IDX2C(x, j, rows)] \* device\_A[j];

}

}

//1)"сырой" код на Си CUDA

void basic\_cuda(const float \*Matrix, const float \*Vector\_A, float \*Result, const int rows, const int cols)

{

float \*device\_M, \*device\_A, \*device\_Result;

cudaMalloc(&device\_M, cols \* rows \* sizeof(float));

cudaMemcpy(device\_M, Matrix, cols \* rows \* sizeof(float), cudaMemcpyHostToDevice);

cudaMalloc(&device\_A, cols \* sizeof(float));

cudaMemcpy(device\_A, Vector\_A, cols \* sizeof(float), cudaMemcpyHostToDevice);

cudaMalloc(&device\_Result, rows \* sizeof(float));

dim3 threads(rows, 1, 1);

dim3 grid(1, 1, 1);

kernel<<<grid, threads>>>(device\_M, device\_A, device\_Result, cols, rows);

cudaDeviceSynchronize();

cudaMemcpy(Result, device\_Result, rows \* sizeof(float), cudaMemcpyDeviceToHost);

cudaFree(device\_M);

cudaFree(device\_A);

cudaFree(device\_Result);

}

//2)с использованием библиотеки "Thrust"

void thrust\_cuda(const float \*Matrix, const float \*Vector\_A, float \*Result\_THRUST, const int rows, const int cols)

{

thrust::device\_vector<float> device\_M(Matrix, Matrix + (cols \* rows));

thrust::device\_vector<float> device\_Buffer(cols);

thrust::device\_vector<float> device\_A(Vector\_A, Vector\_A + cols);

thrust::device\_vector<float> device\_Result(cols);

float reduce = 0.0f;

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

for(int j = 0; j < cols; ++j) {

device\_Buffer[j] = device\_M[IDX2C(i, j, rows)];

}

thrust::transform(device\_Buffer.begin(), device\_Buffer.end(), device\_A.begin(), device\_Result.begin(), thrust::multiplies<float>());

for(int j = 0; j < cols; ++j) {

reduce = thrust::reduce(device\_Result.begin(), device\_Result.end());

}

Result\_THRUST[i] = reduce;

reduce = 0.0f;

}

}

//3)с использованием библиотеки "CUDA BLAS"

void blas\_cuda(const float \*Matrix, const float \*Vector\_A, float \*Result\_BLAS, const int cols, const int rows)

{

float \*device\_M, \*device\_A, \*device\_Result;

cudaMalloc(&device\_M, cols \* rows \* sizeof(float));

cudaMemcpy(device\_M, Matrix, cols \* rows \* sizeof(float), cudaMemcpyHostToDevice);

cudaMalloc(&device\_A, cols \* sizeof(float));

cudaMemcpy(device\_A, Vector\_A, cols \* sizeof(float), cudaMemcpyHostToDevice);

cudaMalloc(&device\_Result, rows \* sizeof(float));

cublasHandle\_t handle;

cublasCreate(&handle);

cublasSetMatrix(rows, cols, sizeof(float), Matrix, rows, device\_M, rows);

cublasSetVector(cols, sizeof(float), Vector\_A, 1, device\_A, 1);

cublasSetVector(rows, sizeof(float), Result\_BLAS, 1, device\_Result, 1);

float al = 1.0f;

float bet = 1.0f;

cublasSgemv(handle, CUBLAS\_OP\_N, rows, cols, &al, device\_M, rows, device\_A, 1, &bet, device\_Result, 1);

cublasGetVector(rows, sizeof(float), device\_Result, 1, Result\_BLAS, 1);

cudaFree(device\_M);

cudaFree(device\_A);

cudaFree(device\_Result);

cublasDestroy(handle);

}

//Матрично-векторное умножение

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

{

int cols, rows;

if(argc < 3) {

printf("Ошибка! Некорректные аргументы.\n\nПример ввода: ./main <rows> <cols>");

return -1;

}

rows = atoi(argv[1]);

cols = atoi(argv[2]);

float \*Matrix, \*Vector\_A, \*Result\_CUDA, \*Result\_THRUST, \*Result\_BLAS;

Matrix = static\_cast<float \*>(malloc(cols \* rows \* sizeof(float)));

Vector\_A = static\_cast<float \*>(malloc(cols \* sizeof(float)));

Result\_CUDA = static\_cast<float \*>(calloc(rows, sizeof(float)));

Result\_THRUST = static\_cast<float \*>(calloc(rows, sizeof(float)));

Result\_BLAS = static\_cast<float \*>(calloc(rows, sizeof(float)));

//Заполнение матрицы

fill\_rand(Matrix, cols \* rows);

//Заполнение вектора

fill\_rand(Vector\_A, cols);

/\*BASIC CUDA\*/

unsigned int start\_time = clock();

basic\_cuda(Matrix, Vector\_A, Result\_CUDA, rows, cols);

unsigned int end\_time = clock();

double time = (end\_time - start\_time) / ((double)CLOCKS\_PER\_SEC);

printf("\n[BASIC CUDA] Время выполнения: %lf секунд.\n", time);

/\*THRUST CUDA\*/

start\_time = clock();

thrust\_cuda(Matrix, Vector\_A, Result\_THRUST, rows, cols);

end\_time = clock();

time = (end\_time - start\_time) / ((double)CLOCKS\_PER\_SEC);

printf("[THRUST CUDA] Время выполнения: %lf секунд.\n", time);

/\*BLAS CUDA\*/

start\_time = clock();

blas\_cuda(Matrix, Vector\_A, Result\_BLAS, rows, cols);

end\_time = clock();

time = (end\_time - start\_time) / ((double)CLOCKS\_PER\_SEC);

printf("[BLAS CUDA] Время выполнения: %lf секунд.\n", time);

free(Matrix);

free(Vector\_A);

free(Result\_CUDA);

free(Result\_THRUST);

free(Result\_BLAS);

}