Московский авиационный институт
(национальный исследовательский университет)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №4 по курсу «Программирование графических процессоров»

Работа с матрицами. Метод Гаусса.

Студент: Л. Я. Вельтман
Преподаватель: К. Г. Крашенинников
А. Ю. Морозов
Группа: М8О-407Б
Дата:
Оценка:
Подпись:

Москва, 2020

Условие

Цель работы: Использование объединения запросов к глобальной памяти. Реализация метода Гаусса с выбором главного элемента по столбцу. Ознакомление с библиотекой алгоритмов для параллельных расчетов Thrust.

Вариант 3: Решение квадратной СЛАУ.
Необходимо решить систему уравнений Ax = b, где A – квадратная матрица n x n, b – вектор-столбец свободных коэффициентов длиной n, x – вектор неизвестных.
Входные данные: На первой строке задано число n – размер матрицы. В следующих n строках, записано по n вещественных чисел – элементы матрицы. Далее записываются n элементов вектора свободных коэффициентов. $n \leqslant 10^4$.

Выходные данные: Необходимо вывести n значений, являющиеся элементами вектора неизвестных x.

Программное и аппаратное обеспечение:
Device Number: 0
Device name: GeForce GT 545
TotalGlobalMem: 3150381056
Const Mem : 65536
Max shared mem for blocks 49152
Max regs per block 32768
Max thread per block 1024
multiProcessorCount : 3
maxThreadsDim 1024 1024 64
maxGridSize 65535 65535 65535
OS: macOS Catalina version 10.15.5
Text Editor: Sublime Text 3

1 Описание

Метод решения

Метод Гаусса состоит из двух этапов. Первый этап - это прямой ход, в результате которого расширенная матрица системы путём элементарных преобразований (перестановка уравнений системы, умножение уравнений на число, отличное от нуля, и сложение уравнений) приводится к верхнетреугольному виду. На втором этапе (обратный ход) происходит вычисление значений неизвестных.

Описание программы и метод решения

Для того, чтобы проще было искать максимальный элемент по столбцам, храним матрицу в транспонированном виде. Вектор b записываем в конец матрицы.

Прямой ход метода Гаусса состоит в последовательном исключении переменных по одной до тех пор, пока не останется только одно уравнение с одной переменной в левой части. Затем это уравнение решается относительно единственной переменной. Таким образом, систему уравнений приводят к треугольной (ступенчатой) форме. Для этого среди элементов первого столбца матрицы выбираем максимальный элемент с помощью методов библиотеки thrust и перемещаем его на крайнее верхнее положение перестановкой строк. Затем нормируем все уравнение, разделив его на коэффициент ai1, где i – номер столбца.

Затем вычитают получившуюся после перестановки строку из остальных строк, стоящих ниже. После всех преобразований процесс продолжается для следующей строки, причем предыдущую уже не трогаем, пока не останется уравнение с одной неизвестной.

Обратный ход последовательно проходит по всем строкам в порядке убывания номера итерации, на которой они были ведущими. Для каждой такой строки производится вычисление соответствующей переменной по следующим формулам:

$$x_{n-1} = b_{n-1} / a_{n-1,n-1},$$
$$x_i = (b_i - \sum_{j=i+1}^{n-1} a_{ij} x_j) / a_{ii}, \ i = n-2, n-3, \ldots, 0$$

2 Исходный код

```
#include <iostream>
#include <iomanip>
#include <stdio.h>
#include <stdlib.h>
#include <thrust/extrema.h>
#include <thrust/device_vector.h>


#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)



struct comparator {
    __host__ __device__ double fabs(double a){
        return a < 0.0 ? -a : a;
    }

    __host__ __device__ bool operator()(double a, double b)
    {
        return fabs(a) < fabs(b);
    }
};



__host__ void Printer(double* matrix, int height, int width)
{
    std::cout << "Printer\n";
    for (int i = 0; i < width; ++i)
    {
        for (int j = 0; j < height; ++j)
        {
          printf("a[i=%d, j=%d->%d] = %.1f ", i, j, j * width + i, matrix[j * width + i
              ]);
        }
        printf("\n");
    }

}
```

```
47
48
49  __global__ void SwapGPU(double* matrix, int width, int height, int row, int rowWithMax
        )
50  {
51      int idx = blockDim.x * blockIdx.x + threadIdx.x;
52      int xOffset = gridDim.x * blockDim.x;
53      double tmp;
54
55      for (int i = idx + row; i < height; i += xOffset)
56      {
57          tmp = matrix[i * width + row];
58          matrix[i * width + row] = matrix[i * width + rowWithMax];
59          matrix[i * width + rowWithMax] = tmp;
60      }
61  }
62
63
64  __global__ void Normalization(double* matrix, int width, int height, int row)
65  {
66      int idx = blockDim.x * blockIdx.x + threadIdx.x;
67      int xOffset = gridDim.x * blockDim.x;
68
69      for (int i = idx + row + 1; i < height; i += xOffset)
70      {
71          matrix[i * width + row] /= matrix[row * width + row];
72      }
73  }
74
75
76  __global__ void ForwardGauss(double* matrix, int width, int height, int row)
77  {
78      int idx = blockDim.x * blockIdx.x + threadIdx.x;
79      int idy = blockDim.y * blockIdx.y + threadIdx.y;
80
81      int xOffset = gridDim.x * blockDim.x;
82      int yOffset = gridDim.y * blockDim.y;
83
84      for (int i = idx + row + 1; i < width; i += xOffset)
85      {
86          for (int j = idy + row + 1; j < height; j += yOffset)
87          {
88              matrix[j * width + i] -= matrix[j * width + row] * matrix[row * width + i];
89          }
90      }
91  }
92
93
94  __global__ void BackwardGauss(double* matrix, double* x, int size, int row)
```

```
95  {
96      int idx = blockDim.x * blockIdx.x + threadIdx.x;
97      int xOffset = gridDim.x * blockDim.x;
98
99      for (int i = row - 1 - idx; i >= 0; i -= xOffset)
100     {
101         x[i] -= matrix[row * size + i] * x[row];
102     }
103 }
104
105
106
107 int main(int argc, const char* argv[])
108 {
109     std::ios_base::sync_with_stdio(false);
110     std::cin.tie(nullptr);
111
112     int size;
113     std::cin >> size;
114     int height = size + 1;
115     int width = size;
116     double* matrix = new double[height * width];
117
118       for (int i = 0; i < size; ++i)
119     {
120         for (int j = 0; j < size; ++j)
121         {
122             std::cin >> matrix[j * width + i];
123         }
124     }
125
126     for (int i = 0; i < size; ++i)
127     {
128         std::cin >> matrix[size * size + i];
129     }
130
131     double* matrixGPU;
132     CSC(cudaMalloc(&matrixGPU, sizeof(double) * height * width));
133     CSC(cudaMemcpy(matrixGPU, matrix, sizeof(double) * height * width,
        cudaMemcpyHostToDevice));
134
135     int xThreadCount = 32;
136     int yThreadCount = 32;
137
138     int xBlockCount = 32;
139     int yBlockCount = 32;
140
141     comparator comp;
142     thrust::device_ptr<double> ptr, ptrMax;
```

```
143     int rowWithMax;
144     for (int row = 0; row < size - 1; ++row)
145     {
146         ptr = thrust::device_pointer_cast(matrixGPU + row * size);
147         ptrMax = thrust::max_element(ptr + row, ptr + size, comp);
148         rowWithMax = ptrMax - ptr;
149 
150         if (rowWithMax != row)
151         {
152             SwapGPU<<<dim3(xBlockCount * yBlockCount), dim3(xThreadCount * yThreadCount
                )>>>(matrixGPU, width, height, row, rowWithMax);
153             CSC(cudaGetLastError());
154         }
155         Normalization<<<dim3(xBlockCount * yBlockCount), dim3(xThreadCount *
            yThreadCount)>>>(matrixGPU, width, height, row);
156         CSC(cudaGetLastError());
157 
158         ForwardGauss<<<dim3(xBlockCount, yBlockCount), dim3(xThreadCount, yThreadCount)
            >>>(matrixGPU, width, height, row);
159         CSC(cudaGetLastError());
160     }
161     CSC(cudaMemcpy(matrix, matrixGPU, sizeof(double) * width * height,
        cudaMemcpyDeviceToHost));
162 
163     double* x = new double[size];
164 
165     for (int i = 0; i < size; ++i)
166     {
167         x[i] = matrix[width * width + i];
168     }
169     x[size - 1] /= matrix[(width - 1) * width + (width - 1)];
170 
171     double* xGPU;
172     CSC(cudaMalloc(&xGPU, sizeof(double) * size));
173     CSC(cudaMemcpy(xGPU, x, sizeof(double) * size, cudaMemcpyHostToDevice));
174 
175     for (int row = size - 1; row > 0; --row)
176     {
177         BackwardGauss<<<dim3(xBlockCount * yBlockCount), dim3(xThreadCount *
            yThreadCount)>>>(matrixGPU, xGPU, size, row);
178         CSC(cudaGetLastError());
179     }
180 
181     CSC(cudaMemcpy(x, xGPU, sizeof(double) * size, cudaMemcpyDeviceToHost));
182 
183     const int accuracy = 10;
184 
185     for (int i = 0; i < size - 1; ++i)
186     {
```

```
187         std::cout << std::scientific << std::setprecision(accuracy) << x[i] << " ";
188     }
189     std::cout << std::scientific << std::setprecision(accuracy) << x[size - 1];
190 
191     CSC(cudaFree(matrixGPU));
192     CSC(cudaFree(xGPU));
193 
194     delete[] matrix;
195     delete[] x;
196 
197     return 0;
198 }
```

3 Результаты

Маленький тест. Матрица 5x5.

CPU
time = 0.001

blocks	threads	time
blocks = (4, 4)	threads = (4, 4)	0.600128
blocks = (16, 16)	threads = (16, 16)	0.539264
blocks = (32, 32)	threads = (32, 32)	1.821376

Средний тест. Матрица 500x500.

CPU
time = 902826

blocks	threads	time
blocks = (4, 4)	threads = (4, 4)	169649.406250
blocks = (16, 16)	threads = (16, 16)	33430.527344
blocks = (32, 32)	threads = (32, 32)	29694.294922

Большой тест. Матрица 5000x5000.

CPU
time = 756259

blocks	threads	time
blocks = (4, 4)	threads = (4, 4)	161281.531250
blocks = (16, 16)	threads = (16, 16)	33424.562500
blocks = (32, 32)	threads = (32, 32)	29849.796875

4 Выводы

Для выполнения данной лабораторной работы нужно было реализовать алгоритм решения квадратной СЛАУ методом Гаусса. Реализованный многопоточный алгоритм на GPU во многом превосходит по производительности этот же алгоритм, но работающий на CPU. Для этого понадобилось изучить, как правильно обращаться к глобальной памяти с использованием объединения запросов. Также я познакомилась с библиотекой thrust, с помощью которой можно производить эффективные и безопасные вычисления. Выполнение данной лабораторной заняло очень много времени, но оказалось, что большая часть моих ошибок была из-за невнимательности.