Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Санкт-Петербургский политехнический университет Петра Великого»
—
Институт компьютерных наук и кибербезопасности
Высшая школа кибербезопасности

ЛАБОРАТОРНАЯ РАБОТА №3

РАСЧЁТЫ ПРИ ПОМОЩИ GPU. CUDA.

по дисциплине «Языки программирования»

Выполнили
студенты гр. 5131001/30002 Мишенев Н. С.
Квашенникова В. М.

<*подпись*>

Руководитель
программист Малышев Е. В.

<*подпись*>

Санкт-Петербург
2024г.

СОДЕРЖАНИЕ

ЦЕЛЬ РАБОТЫ 3

ХОД РАБОТЫ 4
1. Разработанный алгоритм на CPU. 4
2. Распараллеленный алгоритм и соответствующее ему разработанное ядро. 4
3. Производительности алгоритмов на CPU и GPU. 7
4. Вычисление количества используемых блоков и потоков. 8

ЦЕЛЬ РАБОТЫ

Цель - необходимо научиться выполнять расчеты при помощи GPU (CUDA, OpenCL). Для выполнения необходимо выбрать любую задачу с матрицами из области линейной алгебры (за исключением задач сложения, умножения, вычитания и деления матриц).

ХОД РАБОТЫ

1. Разработанный алгоритм на CPU.

Для выполнения лабораторной работы, метод Гаусса для приведения матрицы к треугольному виду был реализован на ***CPU***:

```
 95  void triangular_matrix(float **matrix, int size) {
 96      for (int diag = 0; diag < size - 1; diag++) {
 97          for (int str = diag + 1; str < size; str++) {
 98              float k = (-1.0) * matrix[str][diag] / matrix[diag][diag];
 99              for (int column = diag; column < size; column++) {
100                  matrix[str][column] += k * matrix[diag][column];
101              }
102          }
103      }
104  }
```

Рис. 1. Алгоритм приведения матрицы к треугольному виду.

Алгоритм основан на 2-х основных шагах:

1. Вычисление коэффициента ***k*** (коэффициент обнуления ненулевого элемента под диагональным).
2. Сложение строки с диагональным элементом, умноженной на ***k***, со строкой, содержащей первый ненулевой элемент.

Результат работы алгоритма:

```
1  Generated matrix:
2     531.00  -387.00   271.00    49.00   932.00
3     861.00    38.00   381.00   305.00   336.00
4     965.00   893.00   113.00  -805.00  -429.00
5    -836.00   342.00  -798.00  -216.00  -352.00
6      44.00   560.00   768.00   405.00     8.00
7  Transformed matrix:
8     531.00  -387.00   271.00    49.00   932.00
9       0.00   665.51   -58.42   225.55 -1175.21
10      0.00     0.00  -239.37 -1435.05   696.14
11      0.00     0.00     0.00  2318.61  -504.83
12      0.00     0.00     0.00     0.00  2298.22
```

Рис. 2. Результат работы алгоритма приведения матрицы к треугольному виду.

2. Распараллеленный алгоритм и соответствующее ему разработанное ядро.

Использование мульти-поточности в данном алгоритме уместно при

сложении строк, и вычислении ***k*** для каждого числа под диагональным элементом. Для реализации данной идеи было разработано 2 ядра.

Первое, для вычисления ***k*** для каждой строки. Для каждого элемента под диагональным вычисляется ***k*** и внутри этого ядра вызывается второе ядро, для параллельного сложения строк. В данном ядре реализован первый шаг алгоритма.

```
19  __global__ void compute_k(float* dev_matrix, int diag, int size) {
20      dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
21      dim3 blocksPerGrid((size + threadsPerBlock.x - 1) / threadsPerBlock.x);
22      int row = diag + 1 + (blockIdx.x * blockDim.x + threadIdx.x);
23
24      if (row < size) {
25          float k = (- 1.0) * dev_matrix[row*size+diag] / dev_matrix[diag*size+diag];
26          __syncthreads();
27          add_to_row<<<blocksPerGrid, threadsPerBlock>>>(dev_matrix, row, diag, k, size);
28      }
29  }
```

Рис. 3. Ядро для вычисления k.

Второе, для сложения строк несколькими процессами. В данном ядре реализован второй шаг алгоритма.

```
11  __global__ void add_to_row(float* dev_matrix, int row, int diag, float k, int size) {
12      int col = diag + (blockIdx.x * blockDim.x + threadIdx.x);
13
14      if (col < size) {
15          dev_matrix[row*size+col] += k * dev_matrix[diag*size+col];
16      }
17  }
```

Рис. 4. Ядро для сложения строк.

Оба этих ядра вызываются из основной функции ***triangular_matrix()***, которая и производит приведение матрицы в памяти устройства в треугольную форму.

```
31  void triangular_matrix(float* dev_matrix, int size) {
32      for (int diag = 0; diag < size-1; diag++) {
33          compute_k<<<1, size - (diag + 1)>>>(dev_matrix, diag, size);
34      }
35  }
```

Рис. 5. Функция triangular_matrix().

Результаты работы функции на ***GPU*** с разными размерами матриц:

```
Enter size of square matrix, which will be transformed into a triangular >>> 5
allocation... - allocated.
filling... - filled.
Generated matrix:
    0.34    0.63    0.38    0.79    0.67
    0.74    0.72    0.29    0.60    0.07
    0.29    0.62    0.72    0.53    0.00
    0.51    0.99    0.41    0.95    0.59
    0.95    0.77    0.58    0.43    0.52

transforming... - transformed
Transformed matrix:
    0.34    0.63    0.38    0.79    0.67
   -0.00   -0.65   -0.53   -1.12   -1.40
   -0.00    0.00    0.33   -0.29   -0.76
   -0.00    0.00    0.00   -0.48   -0.97
    0.00   -0.00    0.00    0.00    1.11

Time to execute - 97.101000 ms
```

Рис. 6. Работа функции преобразования для матрицы со стороной 5.

```
Enter size of square matrix, which will be transformed into a triangular >>> 10
allocation... - allocated.
filling... - filled.
Generated matrix:
    0.02    0.79    0.36    0.47    0.32    0.66    0.04    0.33    0.11    0.16
    0.26    1.00    0.12    0.96    0.51    0.60    0.13    0.28    0.31    0.92
    0.21    0.87    0.92    0.96    0.47    0.15    0.46    0.93    0.99    0.66
    0.87    0.33    0.90    0.96    0.79    0.24    0.02    0.76    0.02    0.70
    0.57    0.14    0.40    0.23    0.93    0.21    0.53    0.54    0.75    0.89
    0.34    0.08    0.22    0.87    0.49    0.71    0.67    0.42    0.29    0.28
    0.67    0.71    0.55    0.74    0.28    0.79    0.77    0.84    0.68    0.80
    0.91    0.99    0.30    0.87    0.93    0.68    0.23    0.61    0.24    0.16
    0.58    0.11    0.91    0.76    0.66    0.72    0.64    0.70    0.47    0.95
    0.99    0.75    0.20    0.97    0.07    0.71    0.74    0.56    0.67    0.57

transforming... - transformed
Transformed matrix:
    0.02    0.79    0.36    0.47    0.32    0.66    0.04    0.33    0.11    0.16
   -0.00  -10.98   -5.28   -6.22   -4.29   -9.38   -0.49   -4.69   -1.34   -1.50
    0.00   -0.00    0.78    0.16    0.04   -0.39    0.35    0.67    0.74   -0.07
   -0.00   -0.00    0.00   -0.96    0.18    1.77   -1.17   -0.57   -2.48   -1.77
    0.00    0.00   -0.00   -0.00    0.40   -0.54    0.98    0.40    1.71    1.02
   -0.00    0.00   -0.00   -0.00    0.00    2.03   -0.71   -0.40   -2.35   -1.80
   -0.00    0.00    0.00   -0.00    0.00   -0.00    2.07    0.82    2.51    0.95
    0.00    0.00    0.00    0.00    0.00    0.00   -0.00    0.09   -0.16   -1.41
    0.00   -0.00    0.00    0.00    0.00   -0.00   -0.00    0.00   -0.23   -2.28
   -0.00    0.00   -0.00   -0.00    0.00   -0.00   -0.00    0.00    0.00   -1.94

Time to execute - 93.350000 ms
```

Рис. 7. Работа функции преобразования для матрицы со стороной 10.

3. Производительности алгоритмов на CPU и GPU.

В качестве входных данных возьмем размер входной матрицы, который будем изменять от 10 до 10000 элементов и проверим производительность алгоритмов на CPU и на GPU заполним таблицу и построим графики.

N, элем	CPU, мСек	GPU, мСек
10	~0	90
100	2	91
300	23	100
500	95	102
1000	711	108
2000	5643	119
5000	85904	150
10000	682263	158

Рис. 9. Таблица зависимости времени исполнения от количества элементов.

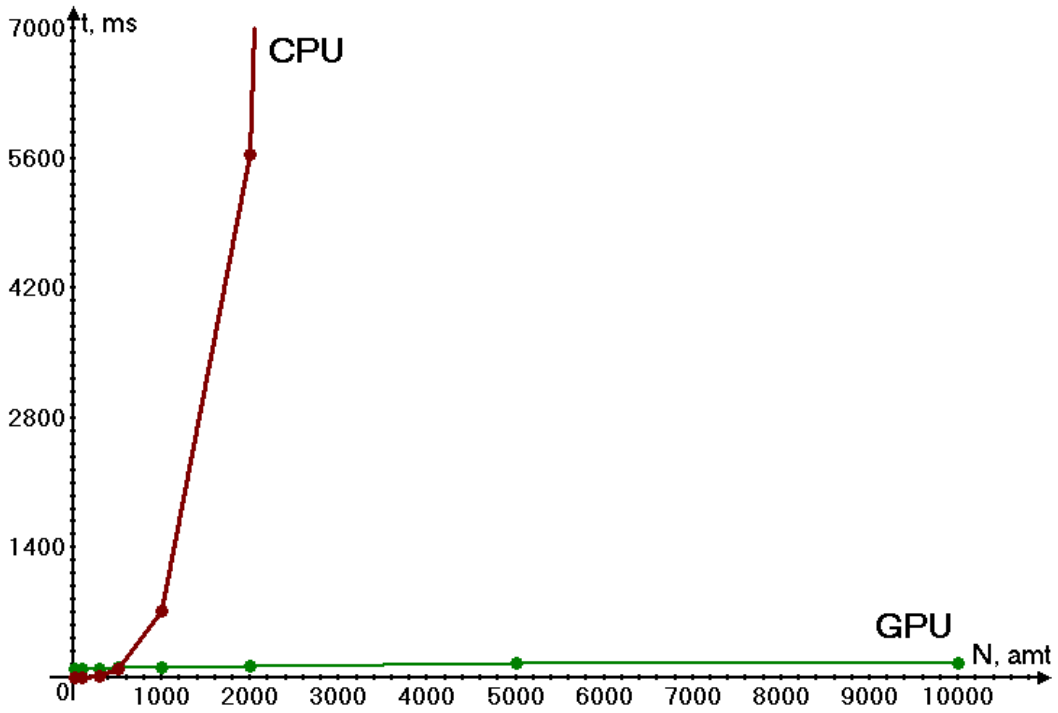

Рис. 10. График зависимости времени исполнения от количества элементов.

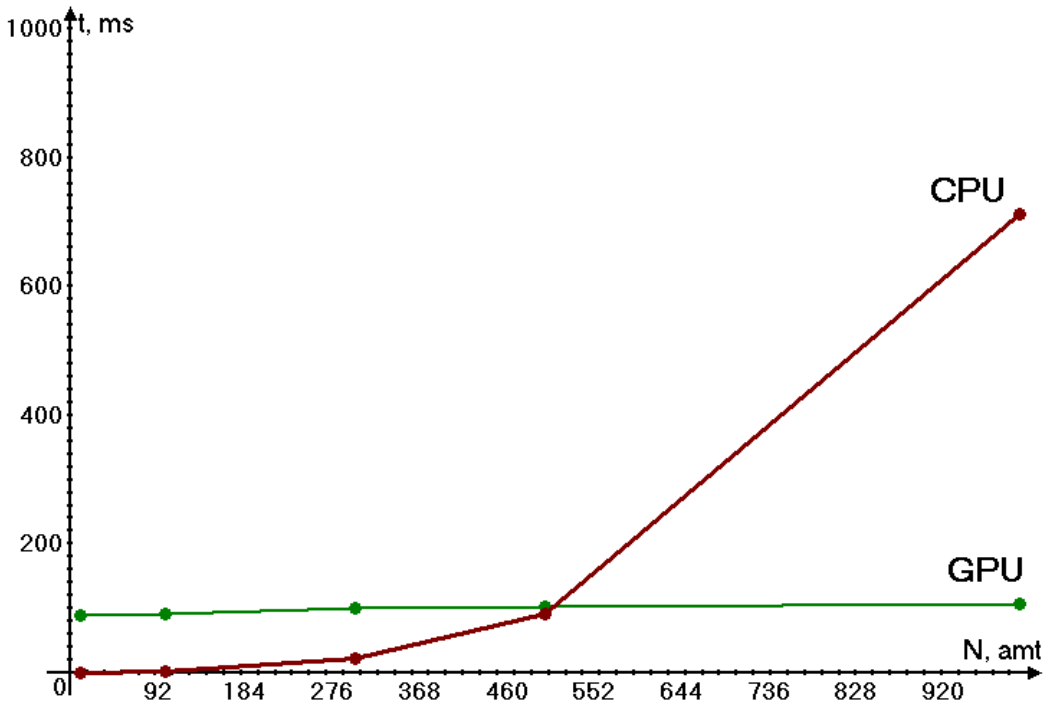

Рис. 11. График вблизи начала координат.

Из анализа приведенных графиков можно сделать вывод о том, что для маленького количества исходных данных, алгоритм на ***CPU*** показывает себя лучше, однако с ростом размерности входных данных, рост затрачиваемого ***CPU*** времени переходит в экспоненциальную форму, в то время как рост на ***GPU*** чрезвычайно медленен.

4. Вычисление количества используемых блоков и потоков.

Количество используемых блоков и потоков вычисляется следующим образом. Рассмотрим вызов ядра подсчёта коэффициентов ***k***, для которого количество потоков в блоке взято константным. Вызывается 1 блок, количество потоков в нём равно количеству элементов под диагональным элементом.

```
compute_k<<<1, size - (diag + 1)>>>(dev_matrix, diag, size);
```

Рис. 12. Вызов ядра подсчёта k.

Рассмотрим вызов ядра для сложения строк. Количество потоков в блоке константно и равно 256 потокам, так как такое количество потоков в блоке (128-256) рекомендовано в документации Nvidia. Количество блоков же вычисляется динамически в зависимости от размерности матрицы так,

чтобы количество блоков покрывало полностью необходимое количество вызовов.

```
dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 blocksPerGrid((size + threadsPerBlock.x - 1) / threadsPerBlock.x);
...
add_to_row<<<blocksPerGrid, threadsPerBlock>>>(dev_matrix, row, diag, k, size);
```

Рис. 13. Вызов ядра сложения строк.

ВЫВОД

В ходе лабораторной работы, были изучены методы выполнения расчетов при помощи GPU (CUDA, OpenCL). Для выполнения была выбрана задача приведения матрицы к треугольному виду методом Гаусса. Было разработано два алгоритма, для CPU и для GPU с использованием CUDA.

ПРИЛОЖЕНИЕ А

Репозиторий с файлами лабораторной работы - репозиторий на GitHub