ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(СПБГУ)

Образовательная программа магистратуры
«Прикладные математика и физика»

Лабораторная работа
ЗАДАНИЕ 3*
Вычисление l-нормы матрицы и профилирование

Выполнил студент
1 курса магистратуры
(группа 22.М21-фз)
Козлов Александр

Санкт-Петербург
2023

СОДЕРЖАНИЕ

ФОРМУЛИРОВКА ЗАДАНИЯ 3

1 ОПИСАНИЕ ПРОГРАММНО-АППАРАТНОЙ КОНФИГУРАЦИИ ТЕСТОВОГО СТЕНДА 3

2 ОПИСАНИЕ ПОСЛЕДОВАТЕЛЬНОГО АЛГОРИТМА РЕШЕНИЯ ЗАДАЧИ 3

3 ОПТИМИЗАЦИЯ АЛГОРИТМА С ИСПОЛЬЗОВАНИЕМ CUDA 3

4 ВЕРИФИКАЦИЯ РЕАЛИЗАЦИЙ 5

5 ИЗМЕРЕНИЕ ВРЕМЕНИ И РЕЗУЛЬТАТЫ 5

6 ПРОФИЛИРОВАНИЕ 7

7 ИНТЕРПРЕТАЦИЯ РЕЗУЛЬТАТОВ, ВЫВОДЫ 9

ПРИЛОЖЕНИЕ 10

ФОРМУЛИРОВКА ЗАДАНИЯ

Дана матрица $A \in \mathbb{R}^{m \times n}$, составленная из элементов $a_{i,j}$, i=1..m, j=1..n. Необходимо вычислить l-норму матрицы по следующей формуле:

$$\|A\| = \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{i,j}|. \tag{1}$$

1 ОПИСАНИЕ ПРОГРАММНО-АППАРАТНОЙ КОНФИГУРАЦИИ ТЕСТОВОГО СТЕНДА

В качестве тестового стенда выступала вычислительная машина, доступ к которой был предоставлен преподавателем. Краткое описание программно-аппаратной конфигурации тестового стенда приведено в Таблице 1.

ОС	Ubuntu 20.04.4 LTS
Число ядер	6
Число потоков	12
Модель процессора	Intel(R) Xeon(R) E-2136 CPU @ 3.30GHz
ОЗУ	62 Гб
Вычислетельная способность девайса	61
Имя девайса	Quadro P2000
Общий размер глобальной памяти девайса	5290131456 байт
Размер разделяемой памяти на блок девайса	49152 байт
Число регистров на один блок девайса	65536
Размер варпа девайса	32
Максимальное число потоков на блок девайса	1024
Общий размер константной памяти девайса	65536 байт

Таблица 1: Программно-аппаратная конфигурация тестового стенда.

2 ОПИСАНИЕ ПОСЛЕДОВАТЕЛЬНОГО АЛГОРИТМА РЕШЕНИЯ ЗАДАЧИ

Матрица A формы m×n заполняется случайными числами типа `float`, которые подчиняются равномерному распределению на отрезке значений от `A_MIN` до `A_MAX` (в программной реализации `A_MIN` $= -5$ и `A_MAX` $= 5$). Далее вычисляются l-нормы столбцов матрицы $\sum_{i=1}^{m} |a_{i,j}|$, j=1..n, которые формируют одномерный массив `temp` длины n. После этого ищется наибольший элемент массива `temp` (найденное значение и будет l-нормой исходной матрицы A). Код программной реализации последовательной версии алгоритма решения задачи приведен в Листинге 2

3 ОПТИМИЗАЦИЯ АЛГОРИТМА С ИСПОЛЬЗОВАНИЕМ CUDA

В функции `main` происходит инициализация матрицы A формы m×n заполняется случайными числами типа `float`, которые подчиняются равномерному распределению на

отрезке значений от `A_MIN` до `A_MAX` (в программной реализации `A_MIN` $= -5$ и `A_MAX` $= 5$). Далее вызывается функция `matrixLNormCUDA`, которая принимает на вход в качестве аргументов числа m и n, характеризующие форму матрицы, саму матрицу A и размер блока CUDA-нитей, а возвращает l-норму матрицы A. Внутри данной функции сперва вычисляется l-норма столбцов матрицы A, что делается параллельно, так как вычисление l-норм различных столбцов происходит независимо. Параллельное вычисление l-норм столбцов матрицы A выполняется согласно следующему алгоритму:

1. На хосте декларируется два одномерных массива длины n `hostV1` и `hostV2`;
2. `hostV2` заполняется нулями;
3. Выделяется память на девайсе два одномерных массива длины n `deviceV1` и `deviceV2`;
4. В `deviceV2` копируется `hostV2`;
5. В цикле по i от 0 до m-1 включительно с шагом 1:
   (a) `hostV1` заполняется значениями i-ой строки матрицы A;
   (b) В `deviceV1` копируется `hostV1`;
   (c) Каждый элемент `deviceV2` увеличивается на абсолютное значение соответствующего элемента `deviceV1`, для чего вызывается ядро `increseInV2byAbsV1Kernel`, код которого приведён в Листинге 1;
6. В `hostV2` копируется `deviceV2`;
7. С помощью функции `cublasIsamax` из библиотеки `CUBLAS` отыскивается индекс наибольшего элемента в массиве `deviceV2` `max_idx` (при этом `CUBLAS` индексирует элементы массивов с 1, а не с 0, как принято в C/C++);
8. Результатом является `hostV2[max_idx-1]`.

Полная реализация оптимизированной с помощью CUDA версии алгоритма приведена в Листинге 3.

Листинг 1: Реализация ядра `increseInV2byAbsV1Kernel`.

```
__global__ void increseInV2byAbsV1Kernel(int n, float *V2, float *V1){
  // Linear index of the current thread
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  // Adding absolute values of all elements that lie on this thread
  while (idx < n) {
    V2[idx] += std::abs(V1[idx]);
    idx += blockDim.x * gridDim.x;
  }
}
```

4 ВЕРИФИКАЦИЯ РЕАЛИЗАЦИЙ

Для верификации реализаций была взята форма матриц 3×4 и размер CUDA-блока 512, на Рис. 1 представлен результат работы последовательной и оптимизированной с использованием CUDA реализаций алгоритма, при этом выведены сгенерированные матрицы A, что позволяет проверить правильность ответа. Нетрудно убедиться, что ответы в обоих случаях верные.

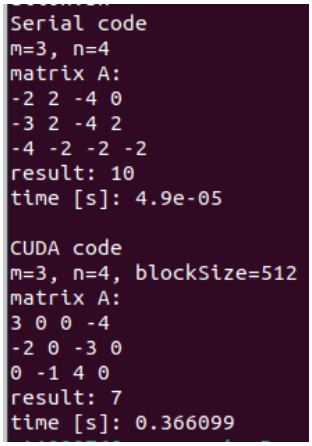


Рис. 1: Результат работы последовательной и оптимизированной с использованием CUDA реализаций алгоритма при m = 3, n = 4 и `blockSize` = 512.

5 ИЗМЕРЕНИЕ ВРЕМЕНИ И РЕЗУЛЬТАТЫ

Время работы программы измерялась с помощью функции `clock()`. Время работы печаталось в командной строке в результате работы программ. Для автоматизации проведения численных экспериментов были написаны сценарии на языке `bash` отдельно для измерения зависимости времени работы последовательной реализации от вычислительной сложности и для измерения зависимости времени работы оптимизированной с помощью CUDA реализации от вычислительной сложности задачи. Код сценария для последовательной версии приведен в Листинге 4, а для оптимизированной — в Листинге 5. В ходе тестирования было выявлено, что свободный параметр CUDA-версии алгоритма blockSize слабо влияет на время работы программы, поэтому тут и далее он полагается равным 1024. Было проведено 3 эксперимента, в ходе каждого m пробегало значения 500, 1000, 5000, 10000, 50000, 100000, а n — значения 500, 1000, 5000, 10000.

Результаты измерений времени представлены на Рис. 7 и 6.

На Рис. 2 представлено время работы последовательной и оптимизированной с помощью CUDA версий реализации алгоритма решения задачи в зависимости от формы матрицы A m и n. Видно, что при малых значениях m и n CUDA-версия работает сильно дольше. Чтобы оценить эффект от оптимизации алгоритма, было рассчитано ускорение

$$\texttt{speedup}_{m,n} = \frac{(t_{\text{serial}})_{m,n}}{(t_{\text{CUDA}})_{m,n}} \tag{2}$$

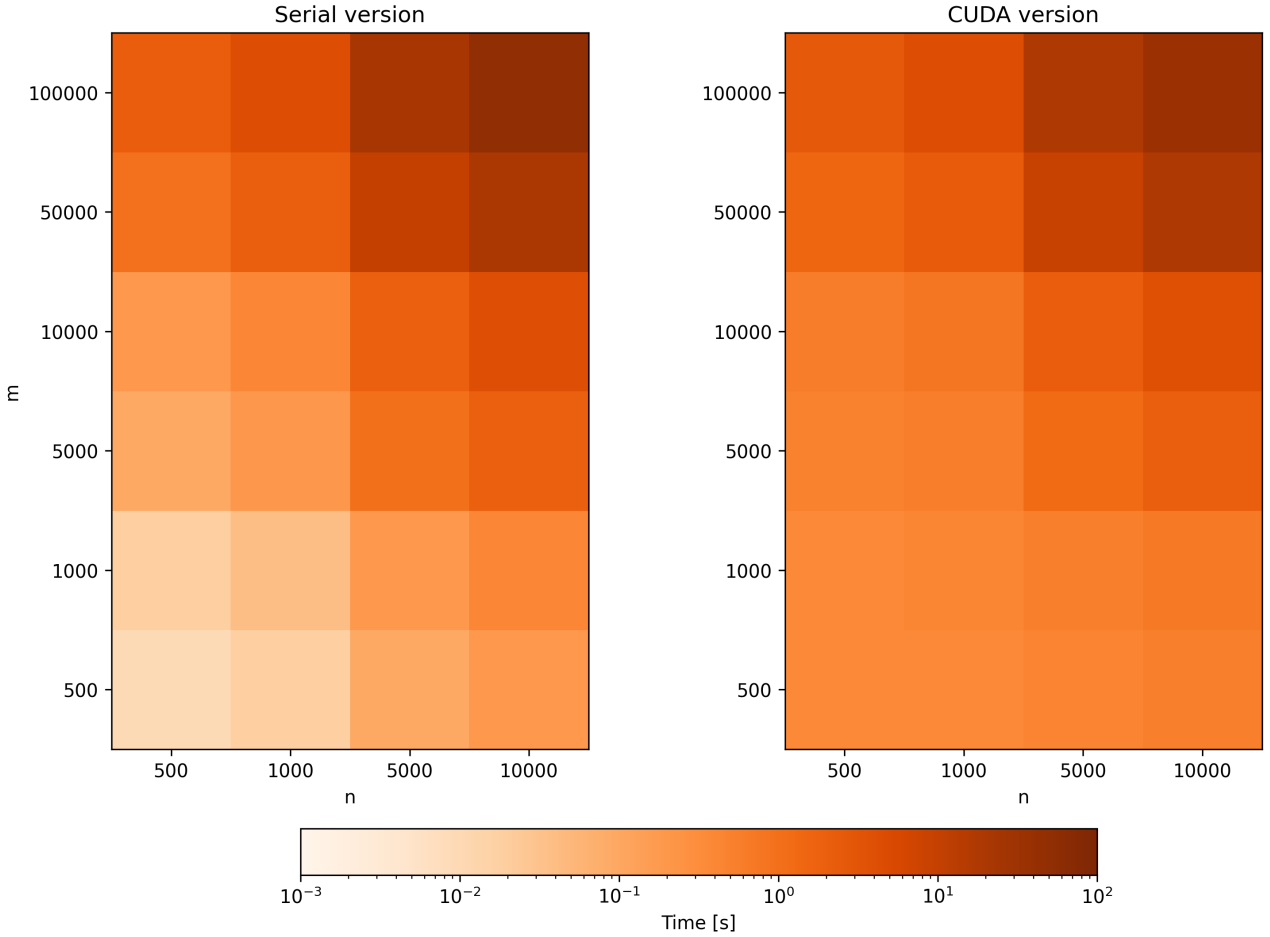


Рис. 2: Время работы последовательной (Serial version) и оптимизированной с помощью CUDA (CUDA version) версий реализации алгоритма решения задачи в зависимости от формы матрицы A m и n.

и его погрешность

$$\texttt{speedup error}_{m,n} = \frac{\texttt{std}\Big\{(t_{\text{serial}})_{m,n}\Big\}(t_{\text{CUDA}})_{m,n} + (t_{\text{serial}})_{m,n}\texttt{std}\Big\{(t_{\text{CUDA}})_{m,n}\Big\}}{(t_{\text{CUDA}})^2_{m,n}}. \quad (3)$$

Данные величины приведены на Рис. 3. Видно, что ускорение превышает 1 лишь при

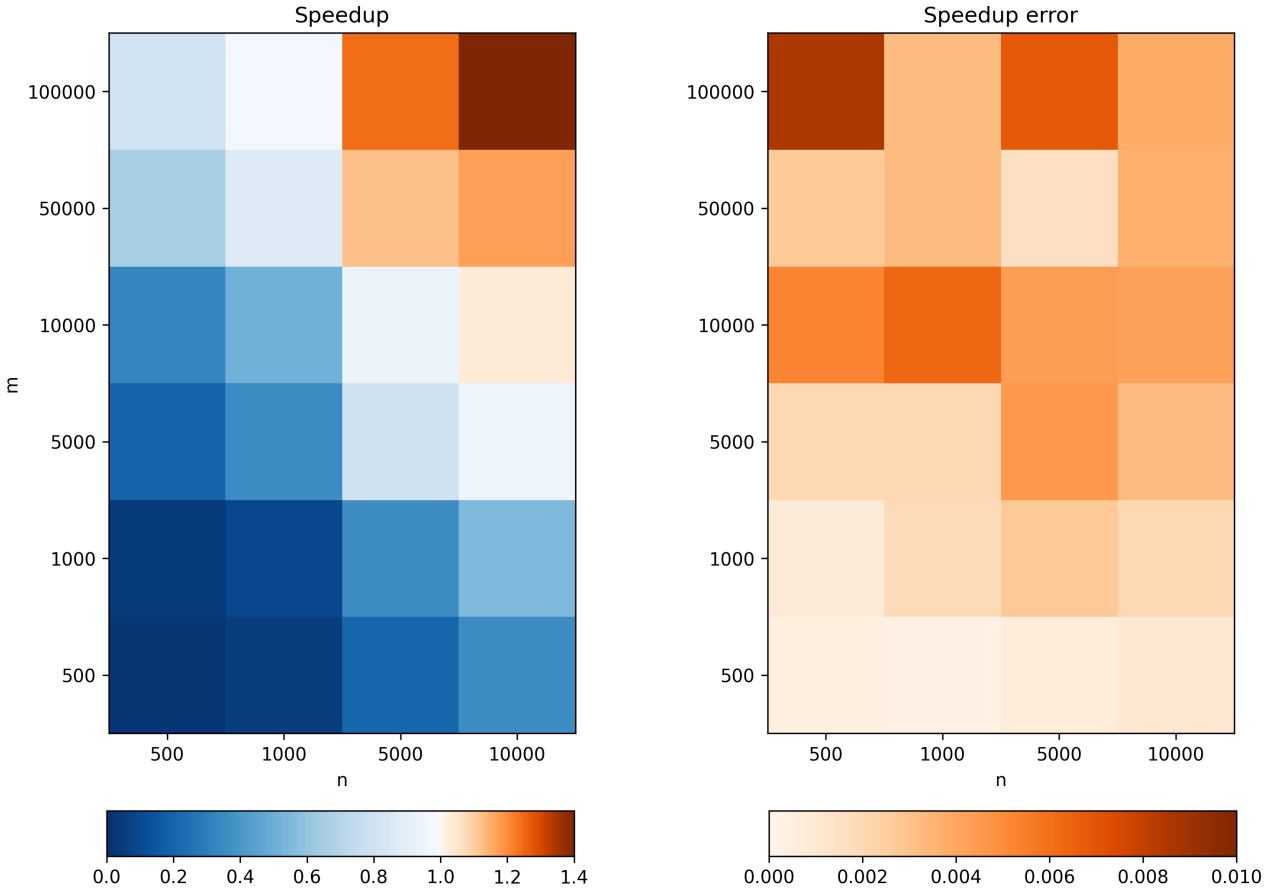
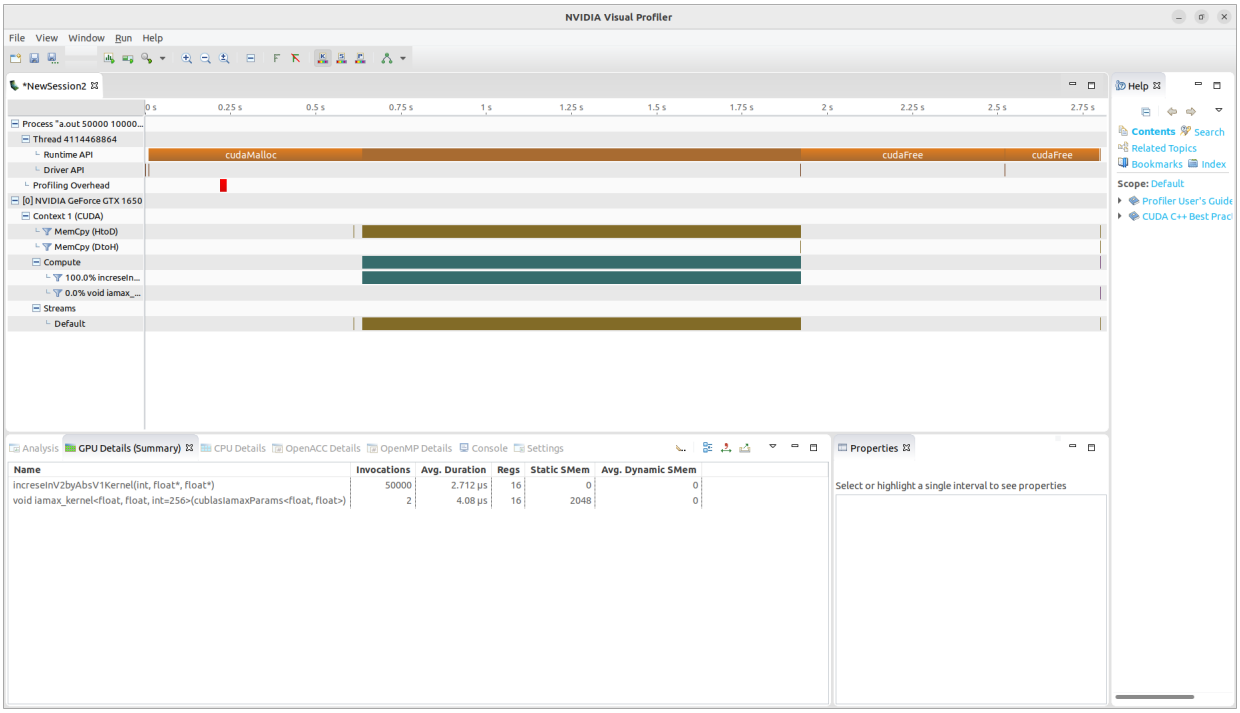


Рис. 3: Ускорение (Speedup) и его погрешность (Speedup error) в зависимости от формы матрицы A m и n.

больших размерах матрицы A — при $m \geq 50000$ и $m \geq 10000$, погрешность при этом крайне мала и в максимуме достигает 0.01. Таким образом ощутимый эффект от CUDA-оптимизации удается достичь лишь при большой вычислительной трудоемкости задачи.

6 ПРОФИЛИРОВАНИЕ

Оптимизированная с помощью CUDA версия алгоритма была проанализирована с помощью Nvidia Visual Profiler при параметрах задачи m = 500000, n = 10000 и blockSize = 1024. Временная диаграмма и прочие основные результаты профилирования представлены на Рис. 4 и Рис. 5. Из рисунков следует, что большая часть времени выполнения алгоритма приходится на вызов и последующие выполнение функции `increseInV2byAbsV1Kernel` (а именно 100.0% времени, надо полагать такое значение является приближенным). То есть почти всё время выполнения программы уходит на суммирование модулей элементов



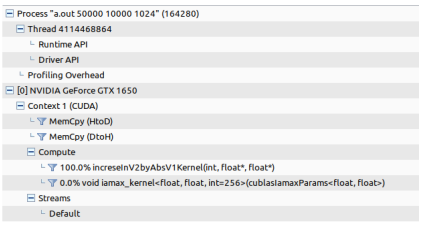


Рис. 5: Результаты анализа оптимизированной с помощью CUDA версии алгоритма решения задачи с помощью Nvidia Visual Profiler. Левая колонка приближенная.

столбцов. А поиск максимума в получившемся массиве с помощью функции `cublasIsamax` занимает пренебрежимо малую часть времени исполнения программы.

Кроме того, следует заметить, сколь много времени уходит на выделение памяти на GPGPU (`cudaMalloc`, тратится примерно 0.6 секунд) и на её освобождение (`cudaFree`, тратится примерно 1.1 секунд). Это время сопоставимо с временем выполнения полезных вычислительным операций.

7 ИНТЕРПРЕТАЦИЯ РЕЗУЛЬТАТОВ, ВЫВОДЫ

Было реализовано две версии алгоритма решения предлагаемой задачи — последовательный и оптимизированный с помощью CUDA. Было измерено время работы каждой из версий алгоритма при различной форме матрицы A и установлено, что оптимизация с помощью CUDA даёт выигрыш во времени исполнения лишь при большой вычислительной сложности задачи, ускорение получилось больше 1 лишь при $n = 5000$ и $m \geq 50000$, а так же при $n = 10000$ и $m \geq 100000$. Максимальное ускорение составляет 1.4 и достигается при размере матрицы A 100000×10000. Таким образом, оптимизация с помощью CUDA оптимально лишь при больших формах матрицы A и не сокращает время работы алгоритма существенно (максимум в 1.4 раза).

ПРИЛОЖЕНИЕ

Листинг 2: Код программной реализации последовательной версии алгоритма решения задачи приведен.

```
#include <cstdlib>
#include <cmath>
#include <ctime>
#include <iostream>
#include <random>

#define A_MIN -5
#define A_MAX 5
#define PRINT false

int main(int argc, char **argv){
  // First point of the time measurement
  clock_t t = clock();
  // Shape of the matrix A (m x n)
  int m = std::atoi(argv[1]);
  int n = std::atoi(argv[2]);
  if (PRINT) std::cout << "m=" << m << ", n=" << n << std::endl;
  float **a; // Matrix A declaration
  a = new float *[m];
  for(int i = 0; i < m; i++)
    a[i] = new float[n];
  // Filling this matrix with random values
  std::random_device rd;  // Will be used to obtain a seed for the random number
    engine
  std::mt19937 gen(rd()); // Standard mersenne_twister_engine seeded with rd()
  std::uniform_real_distribution<float> dis(A_MIN, A_MAX);
  if (PRINT) std::cout << "matrix A:" << std::endl;
  for(int i = 0; i < m; i++){
    for(int j = 0; j < n; j++){
      a[i][j] = int(dis(gen));
      if (PRINT) std::cout << a[i][j] << " ";
    }
    if (PRINT) std::cout << std::endl;
  }
  // Calculation of the each column l-norm
  float *temp;
  temp = (float*)malloc(n*sizeof(float));
  for(int j = 0; j < n; j++){
    temp[j] = 0.;
    for(int i = 0; i < m; i++){
      temp[j] += std::abs(a[i][j]);
    }
    //std::cout << temp[j] << std::endl;
  }
```

```
  // Finding the maximum value of the temp array
  float temp_max = 0.;
  for(int j = 0; j < n; j++){
    if(temp_max < temp[j]){
      temp_max = temp[j];
    }
  }
  if (PRINT) std::cout << "result: " << temp_max << std::endl;
  // Last point of the time measurement
  if (PRINT) std::cout << "time [s]: ";
  std::cout << float(clock() - t) / float(CLOCKS_PER_SEC) << std::endl;
  return 0;
}
```

Листинг 3: Реализация оптимизированной с помощью CUDA версии алгоритма.

```
#include <cstdlib>
#include <cmath>
#include <ctime>
#include <iostream>
#include <random>
#include "cublas_v2.h"

#define A_MIN -5
#define A_MAX 5
#define PRINT false

__global__ void increseInV2byAbsV1Kernel(int n, float *V2, float *V1){
  // Linear index of the current thread
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  // Adding absolute values of all elements that lie on this thread
  while (idx < n) {
    V2[idx] += std::abs(V1[idx]);
    idx += blockDim.x * gridDim.x;
  }
}

float matrixLNormCUDA(const int m, const int n, float **a, const int blockSize){
  // Number of thread blocks in grid
  int gridSize = (int)ceil((float)n/blockSize);
  // Previous result ?
  float *hostV1, *hostV2;
  hostV1 = (float*)malloc(n*sizeof(float));
  hostV2 = (float*)malloc(n*sizeof(float));
  for(int j = 0; j < n; j++){
    hostV2[j] = 0.;
  }
  // Declaration of the device arrrays
  float *deviceV1, *deviceV2;
  cudaMalloc(&deviceV1, n*sizeof(float));
  cudaMalloc(&deviceV2, n*sizeof(float));
  // Copy host vectors to device
  cudaMemcpy(deviceV2, hostV2, n*sizeof(float), cudaMemcpyHostToDevice);
  // Loop over the all rows of the matrix A
  for(int i = 0; i < m; i++){
    // Filling hostV1 array with values of the current row of the matrix A
    for(int j = 0; j < n; j++){
      hostV1[j] = a[i][j];
    }
    // Copy hostV1 to deviceV1
    cudaMemcpy(deviceV1, hostV1, n*sizeof(float), cudaMemcpyHostToDevice);
    // Add absolute values of the current row to the deviceV2 array
    increseInV2byAbsV1Kernel<<<gridSize, blockSize>>>(n, deviceV2, deviceV1);
```

```
48    }
49    // Copy deviceV2 to host
50    cudaMemcpy(hostV2, deviceV2, n*sizeof(float), cudaMemcpyDeviceToHost);
51    // Finding maximum element of the deviceV2 array with the help of the CUBLAS
52    cublasHandle_t handle;
53    cublasStatus_t stat;
54    cublasCreate(&handle);
55    int max_idx;
56    stat = cublasIsamax(handle, n, deviceV2, 1, &max_idx);
57    if (stat != CUBLAS_STATUS_SUCCESS)
58      std::cout << "Max failed" << std::endl;
59    cublasDestroy(handle);
60    // Result (CUBLAS indexates elements from 1)
61    float result = hostV2[max_idx-1];
62    // Release memory
63    free(hostV2);
64    cudaFree(deviceV1);
65    cudaFree(deviceV2);
66    // Return result
67    return result;
68  }
69
70  int main(int argc, char **argv){
71    // First point of the time measurement
72    clock_t t = clock();
73    // Shape of the matrix A (m x n)
74    int m = std::atoi(argv[1]);
75    int n = std::atoi(argv[2]);
76    int blockSize = std::atoi(argv[3]);
77    if (PRINT) std::cout << "m=" << m << ", n=" << n << ", blockSize=" <<
       blockSize << std::endl;
78    float **a; // Matrix A declaration
79    a = new float *[m];
80    for(int i = 0; i < m; i++)
81      a[i] = new float[n];
82    // Filling this matrix with random values
83    std::random_device rd;  // Will be used to obtain a seed for the random number
       engine
84    std::mt19937 gen(rd()); // Standard mersenne_twister_engine seeded with rd()
85    std::uniform_real_distribution<float> dis(A_MIN, A_MAX);
86    if (PRINT) std::cout << "matrix A:" << std::endl;
87    for(int i = 0; i < m; i++){
88      for(int j = 0; j < n; j++){
89        a[i][j] = int(dis(gen));
90        if (PRINT) std::cout << a[i][j] << " ";
91      }
92      if (PRINT) std::cout << std::endl;
93    }
94    float temp_max = matrixLNormCUDA(m,n,a,blockSize);
```

```
95    if (PRINT) std::cout << "result: " << temp_max << std::endl;
96    // Last point of the time measurement
97    if (PRINT) std::cout << "time [s]: ";
98    std::cout << float(clock() - t) / float(CLOCKS_PER_SEC) << std::endl;
99    return 0;
100 }
```

Листинг 4: Сценарий на языке `bash` для измерения времени работы последовательной реализации алгоритма решения задачи.

```
#!/bin/bash
cppcode=main.cpp
cppout=main_cpp.out
g++ $cppcode -o $cppout
df=../data/serial_experiments.csv
echo "experiment,m,n,time [s]" > $df
ms=(500 1000 5000 10000 50000 100000)
ns=(500 1000 5000 10000)
experiments=(1 2 3)
for experiment in ${experiments[@]}
do
  for m in ${ms[@]}
  do
    for n in ${ns[@]}
    do
      ./$cppout $m $n > t
      read t < "t"
      echo "$experiment,$m,$n,$t" >> $df
      echo "$experiment,$m,$n,$t"
    done
  done
done
# cleaning
rm *.out
rm t
```

Листинг 5: Сценарий на языке `bash` для измерения времени работы оптимизированной с помощью CUDA реализации алгоритма решения задачи.

```
#!/bin/bash
cudacode=main.cu
cudaout=main_cuda.out
nvcc $cudacode -o $cudaout -lcublas
df=../data/cuda_experiments.csv
echo "experiment,m,n,time [s]" > $df
ms=(500 1000 5000 10000 50000 100000)
ns=(500 1000 5000 10000)
blockSize=1024
experiments=(1 2 3)
for experiment in ${experiments[@]}
do
  for m in ${ms[@]}
  do
    for n in ${ns[@]}
    do
      ./$cudaout $m $n $blockSize > t
      read t < "t"
      echo "$experiment,$m,$n,$t" >> $df
      echo "$experiment,$m,$n,$t"
    done
  done
done
# cleaning
rm *.out
rm t
```

m	n	Время [с]		
		Эксперимент 1	Эксперимент 2	Эксперимент 3
500	500	9.213E-03	9.230E-03	9.212E-03
500	1000	1.843E-02	1.844E-02	1.852E-02
500	5000	9.378E-02	9.372E-02	9.366E-02
500	10000	1.893E-01	1.894E-01	1.891E-01
1000	500	1.779E-02	1.794E-02	1.834E-02
1000	1000	3.670E-02	3.643E-02	3.705E-02
1000	5000	1.906E-01	1.903E-01	1.914E-01
1000	10000	3.802E-01	3.824E-01	3.809E-01
5000	500	9.510E-02	9.443E-02	9.582E-02
5000	1000	1.973E-01	1.973E-01	1.969E-01
5000	5000	9.825E-01	9.815E-01	9.804E-01
5000	10000	1.970E+00	1.973E+00	1.967E+00
10000	500	1.907E-01	1.915E-01	1.962E-01
10000	1000	3.922E-01	3.906E-01	3.947E-01
10000	5000	1.967E+00	1.967E+00	1.969E+00
10000	10000	3.944E+00	3.944E+00	3.959E+00
50000	500	9.671E-01	9.731E-01	9.726E-01
50000	1000	2.000E+00	2.006E+00	2.009E+00
50000	5000	1.011E+01	1.013E+01	1.012E+01
50000	10000	2.058E+01	2.048E+01	2.059E+01
100000	500	2.151E+00	2.098E+00	2.124E+00
100000	1000	4.170E+00	4.165E+00	4.150E+00
100000	5000	2.214E+01	2.190E+01	2.197E+01
100000	10000	4.869E+01	4.889E+01	4.887E+01

Рис. 6: Время работы последовательной версии алгоритма в различных экспериментах при различной вычислительной сложности задачи.

m	n	Время [с]		
		Эксперимент 1	Эксперимент 2	Эксперимент 3
500	500	3.721E-01	3.752E-01	3.600E-01
500	1000	3.709E-01	3.727E-01	3.744E-01
500	5000	4.374E-01	4.386E-01	4.404E-01
500	10000	5.220E-01	5.249E-01	5.237E-01
1000	500	3.741E-01	3.741E-01	3.775E-01
1000	1000	3.945E-01	3.930E-01	4.046E-01
1000	5000	5.250E-01	5.253E-01	5.309E-01
1000	10000	6.968E-01	6.986E-01	6.966E-01
5000	500	4.731E-01	4.778E-01	4.754E-01
5000	1000	5.600E-01	5.552E-01	5.536E-01
5000	5000	1.230E+00	1.217E+00	1.219E+00
5000	10000	2.072E+00	2.077E+00	2.083E+00
10000	500	5.945E-01	5.951E-01	5.900E-01
10000	1000	7.663E-01	7.626E-01	7.772E-01
10000	5000	2.100E+00	2.098E+00	2.080E+00
10000	10000	3.818E+00	3.800E+00	3.799E+00
50000	500	1.487E+00	1.483E+00	1.482E+00
50000	1000	2.310E+00	2.305E+00	2.299E+00
50000	5000	9.005E+00	8.996E+00	9.006E+00
50000	10000	1.755E+01	1.752E+01	1.753E+01
100000	500	2.618E+00	2.617E+00	2.616E+00
100000	1000	4.270E+00	4.263E+00	4.257E+00
100000	5000	1.765E+01	1.768E+01	1.766E+01
100000	10000	3.475E+01	3.475E+01	3.468E+01

Рис. 7: Время работы оптимизированной с помощью CUDA версии алгоритма в различных экспериментах при различной вычислительной сложности задачи.