Student Name: Галактионов Кирилл
Student ID: st067889

Современные технологии программирования в научных исследованиях II
Лабораторная работа 3

---

1 Постановка задачи

Условие задачи (№5) звучит следующим образом: "Задан вещественный массив. Найти сумму элементов, размещенных в тредах с четными номерами".

2 Описание тестового стенда

Вычисления проводились на выданном вычислительном узле:

- Операционная система: Ubuntu 20.04.4 LTS
- Процессор: Intel(R) Xeon(R) E-2136 CPU @ 3.30GHz, 6 ядер, 2 потока на ядро
- Доступная оперативная память: 62-63 ГБ
- GPU: Quadro P2000
- Компилятор: nvcc (NVIDIA (R) Cuda compiler driver, Cuda compilation tools, release 12.0, V12.0.140)

Время работы алгоритмов вычислялось при помощи команды **clock()** из стандартной библиотеки **time** языка **C++**. Проводилось по три серии экспериментов, затем полученные значения времен усреднялись.

Проверка правильности суммирования осуществлялась сравнением с последовательным вычислением на CPU (время не включается в время выполнения).

3 Описание алгоритма решения задачи

Условие задачи требует вычислить сумму элементов в тредах с четными номерами, что при последовательном размещении массива в памяти соответствует вычислению суммы элементов на четных позициях. В последовательной реализации для данной цели использовался цикл, в котором считалась сумма каждого второго элемента. Код:

```
// Sum elements on even positions on CPU
double SumEvenCPU(double* hostArray, int N){
    double result = 0.0;
    // iterate over half of the array size
    for (int i = 0; i < N / 2; i++){
        // ... end sum every second element
        result+= hostArray[2*i];
    }
    return result;
}
```

Идея использования нитей CUDA в такой сортировке заключается в том, что можно разбить массив на блоки, и средствами нитей вычислять сумму элементов на четных местах сначала в каждом блоке (провести редукцию внутри блока), а а затем на хосте просуммировать результаты поблочных вычислений (финальная редукция). Реализация кода нити и алгоритма представлена ниже:

```
// Thread code for sum of the elements in the even threads
__global__ void SumEven(double* array, int N, double* deviceRes){
    // shared array for threads in the block
    __shared__ double helper[THREADS_PER_BLOCK];

    // Get thread ID
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    // Create thread variable for storing sum
    double sum = 0.0;
    // If thread ID is even
    if (idx % 2 == 0){
        helper[threadIdx.x] = array[idx];
    }else{
        // ... or if odd sum equals zero
        helper[threadIdx.x] = 0.0;
    }

    __syncthreads();

    // Reduction
    int amountOfThreads = blockDim.x / 2;
    while (amountOfThreads > 0){
    if (threadIdx.x < amountOfThreads){
            helper[threadIdx.x] += helper[threadIdx.x + amountOfThreads];
    }

        __syncthreads();

    amountOfThreads /= 2;
  }

    // Save the results for block
    if (threadIdx.x == 0) {
        deviceRes[blockIdx.x] = helper[0];
    }
}

// Sum elements on even positions using CUDA
double SumEvenCUDA(double* hostArray, int N){
    // Make two arrays on device variables
    double *deviceArray, *deviceRes;
    // helper for intermediate results
    double *hostInterRes;
    // Variable to accumulate intermediate results
    double res = 0.0;

    // Thread blocks per Grid
    int gridDim = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
```

```
    size_t interResSize = gridDim * sizeof(double);

    // Create vectors
    hostInterRes = (double*)malloc(interResSize);
    cudaMalloc(&deviceArray, N*sizeof(double));
    cudaMalloc(&deviceRes, interResSize);
    // Copy from host to device
    cudaMemcpy(deviceArray, hostArray, N * sizeof(double), cudaMemcpyHostToDevice);

    // Sum
    SumEven<<<gridDim, THREADS_PER_BLOCK>>>(deviceArray, N, deviceRes);

    // Copy from device to host
    cudaMemcpy(hostInterRes, deviceRes, interResSize, cudaMemcpyDeviceToHost);

    // Free memory
    cudaFree(deviceArray);
    cudaFree(deviceRes);

    // Final reduction
    for (int i = 0; i < gridDim; i++){
        res += hostInterRes[i];
    }
    free(hostInterRes);
    return res;
}
```

4 Программный код

Полный код программы представлен в Приложении 1 в разделе 8.

5 Результаты измерений

Вывод программы представлен в Приложении 2 в разделе 9.

6 Анализ и обсуждение результатов

Рассмотрим затраты времени при проверке алгоритма на векторах разного размера. На рисунке 1 представлена зависимость времени, затраченного на вычисления, от размера массива. При вычислениях на CPU зависимость является линейной ($O(N)$), что наблюдается на всём интервале размеров массива (с учетом погрешностей усреднения). При вычислениях на GPU же, до размера массива 10^5 элементов, затраты времени практически постоянные и много больше времени для CPU, что связано с затратами на создание нитей, пересылку данных. При увеличении размера массива, зависимость становится так же линейной, причем угол наклона меньше, чем для CPU, что говорит о некотором ускорении. И действительно, начиная с размера массива в 10^8 элементов, вычисления на GPU начинают выигрывать во времени у CPU.

Аналогичные тренды можно наблюдать и на графике ускорения GPU 2. Рост ускорения на первом участке до 10^5 элементов обусловлен тем, что вычисления на CPU замедлялись,

а на GPU былим постоянными по времени. На оставшемся участке наблюдается истинное увеличение ускорения, обусловленное эффективностью использования GPU.

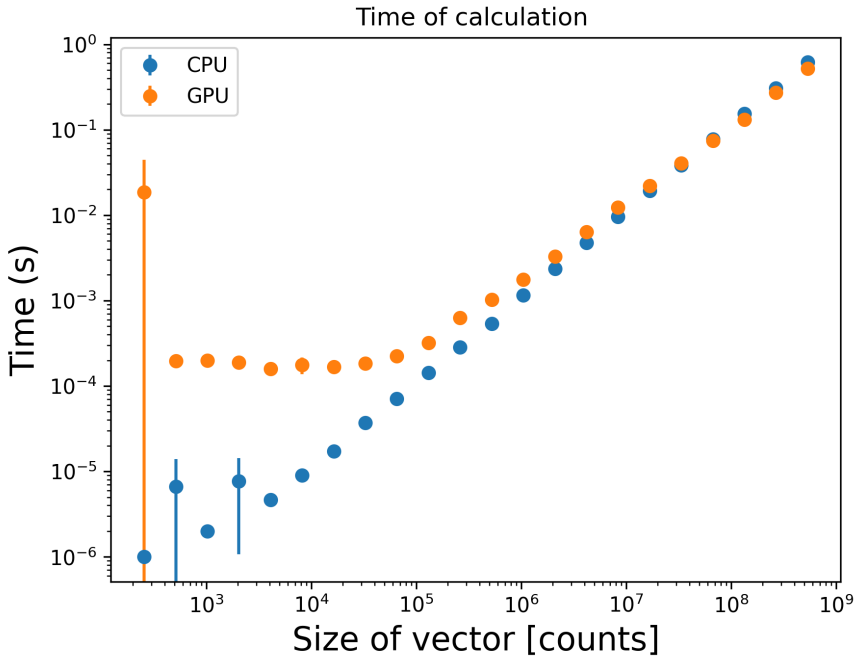

Рис. 1: Зависимость времени вычисления от размера массива.

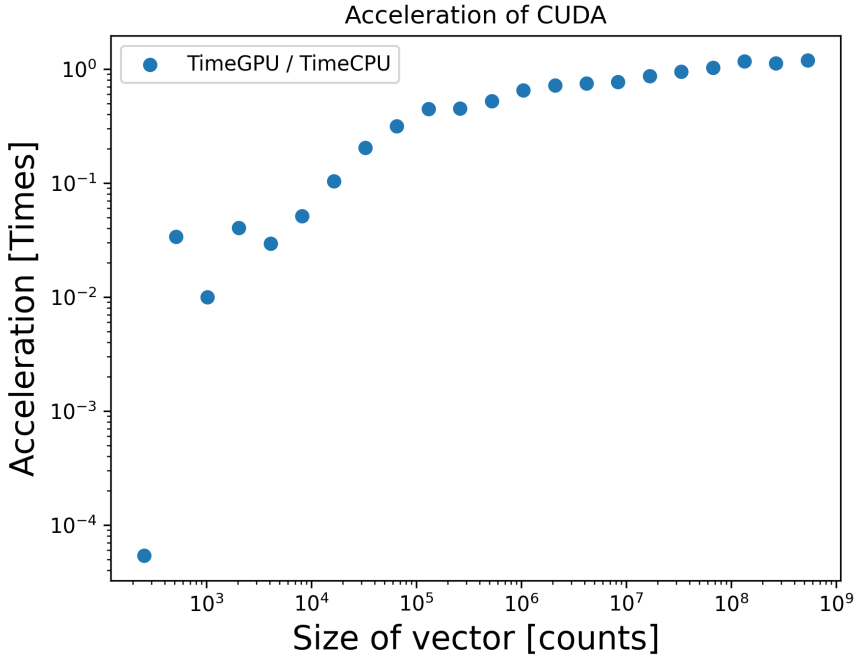

Рис. 2: Зависимость ускорения вычисления от размера массива.

7 Выводы

Поставленные задачи были выполнены. Было проведено сравнение работы алгоритмов на CPU и GPU и наблюдалось ускорение работы, которое может обеспечить GPU, и был определен размер массива, для которого использование GPU становится оправданным.

8 Приложение 1. Код программы

```
#include <iostream>
#include <stdio.h>
#include <time.h>
#include <iomanip>
#define THREADS_PER_BLOCK 1024

using namespace std;


__global__ void SumEven(double* array, int N, double* deviceRes);


double SumEvenCUDA(double* hostArray, int N);

double SumEvenCPU(double* hostArray, int N);

void PrintArray(const double* array, const int n);

void measureNoCUDA(int N);

void measureCUDA(int N);


int main (int argc, char *  argv []){
    srand(static_cast <unsigned> (time(0)));

    cout << "CPU:" << endl;
    cout << "N        Time1     Time2      Time3" << endl;
    for (int n = 256; n < 1000000000; n*=2 ){
        cout<< setw(9) <<  n << "   ";
        for (int i = 0; i < 3; i++){
            measureNoCUDA(n);
            cout << "   ";
        }
        cout << endl;
    }

    cout << "GPU:" << endl;
    cout << "N        Time1     Time2      Time3" << endl;
    //cout << "N      Time" << endl;
    for (int n = 256; n < 1000000000; n*=2 ){
        cout<< setw(9) <<  n << "   ";
        for (int i = 0; i < 3; i++){
            measureCUDA(n);
```

```
            cout << "    ";
        }
        cout << endl;
    }
    cout << endl;
    return 0;
}

// Thread code for sum of the elements in the even threads
__global__ void SumEven(double* array, int N, double* deviceRes){
    // shared array for threads in the block
    __shared__ double helper[THREADS_PER_BLOCK];

    // Get thread ID
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    // Create thread variable for storing sum
    double sum = 0.0;
    // If thread ID is even
    if (idx % 2 == 0){
        helper[threadIdx.x] = array[idx];
    }else{
        // ... or if odd sum equals zero
        helper[threadIdx.x] = 0.0;
    }

    __syncthreads();

    // Reduction
    int amountOfThreads = blockDim.x / 2;
    while (amountOfThreads > 0){
    if (threadIdx.x < amountOfThreads){
            helper[threadIdx.x] += helper[threadIdx.x + amountOfThreads];
    }

        __syncthreads();

    amountOfThreads /= 2;
  }

    // Save the results for block
    if (threadIdx.x == 0) {
        deviceRes[blockIdx.x] = helper[0];
    }
}

// Sum elements on even positions using CUDA
double SumEvenCUDA(double* hostArray, int N){
    // Make two arrays on device variables
    double *deviceArray, *deviceRes;
    // helper for intermediate results
    double *hostInterRes;
    // Variable to accumulate intermediate results
    double res = 0.0;

```

```
    // Thread blocks per Grid
    int gridDim = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
    size_t interResSize = gridDim * sizeof(double);

    // Create vectors
    hostInterRes = (double*)malloc(interResSize);
    cudaMalloc(&deviceArray, N*sizeof(double));
    cudaMalloc(&deviceRes, interResSize);
    // Copy from host to device
    cudaMemcpy(deviceArray, hostArray, N * sizeof(double), cudaMemcpyHostToDevice);

    // Sum
    SumEven<<<gridDim, THREADS_PER_BLOCK>>>(deviceArray, N, deviceRes);

    // Copy from device to host
    cudaMemcpy(hostInterRes, deviceRes, interResSize, cudaMemcpyDeviceToHost);

    // Free memory
    cudaFree(deviceArray);
    cudaFree(deviceRes);

    // Final reduction
    for (int i = 0; i < gridDim; i++){
        res += hostInterRes[i];
    }
    free(hostInterRes);
    return res;
}

// Sum elements on even positions on CPU
double SumEvenCPU(double* hostArray, int N){
    double result = 0.0;
    // iterate over half of the array size
    for (int i = 0; i < N / 2; i++){
        // ... end sum every second element
        result+= hostArray[2*i];
    }
    return result;
}

// measure time taken for all steps on CPU CUDA
void measureNoCUDA(int N){
    // Create array
    double *array = (double *)malloc(N * sizeof(double));
    // Create variables for saving time results
    clock_t start_time, end_time;
    float timeResult;
    double resCPU = 0.0;
    // Fill array with random numbers
    for (int i = 0; i < N; i++) {
        array[i] = static_cast <double> (rand()) / static_cast <double> (RAND_MAX);
    }

    // Calculation for the array
```

```
153     start_time = clock();
154     resCPU = SumEvenCPU(array, N);
155     end_time = clock();
156     // Time taken
157     timeResult = (float)(end_time - start_time) / CLOCKS_PER_SEC;
158 
159     // Free memory
160     free(array);
161     // Print time info
162     cout << setw(10) << timeResult;
163 }
164 
165 // measure time taken for algorithm on GPU CUDA
166 void measureCUDA(int N){
167     // Create array
168     double *array = (double *)malloc(N * sizeof(double));
169     // Create variables for saving time results
170     clock_t start_time, end_time;
171     float timeResult;
172     double resGPU = 0.0, resCPU = 0.0;
173     // Fill array with random numbers
174     for (int i = 0; i < N; i++) {
175         array[i] = static_cast <double> (rand()) / static_cast <double> (RAND_MAX);
176     }
177 
178     // Calculation for the array
179     start_time = clock();
180     resGPU = SumEvenCUDA(array, N);
181     end_time = clock();
182     // Time taken
183     timeResult = (float)(end_time - start_time) / CLOCKS_PER_SEC;
184 
185     // compare to the CPU result
186     resCPU = SumEvenCPU(array, N);
187     //cout << "CPU : " << resCPU << endl;
188     //cout << "GPU : " << resGPU << endl;
189     if (abs(resCPU - resGPU) < 1e-2){
190         //cout << timeDotProduct << "    ";
191     }
192     else {
193         free(array);
194         cout << "ERROR, RESULT DOESN'T MATCH CPU" << endl;
195         return;
196     }
197 
198     // Free memory
199     free(array);
200     // Print time info
201     cout << setw(10) << timeResult;
202 }
203 
204 // Print double array into console
205 void PrintArray(const double* array, const int n){
206     for(int i = 0; i < n; i++){
```

```
        cout << array[i] <<  ' ';
    }
    cout << endl;
}
```

9 Приложение 2. Результаты

```
CPU:
        N        Time1        Time2        Time3
      256        1e-06        1e-06        1e-06
      512      1.7e-05        1e-06        2e-06
     1024        2e-06        2e-06        2e-06
     2048        3e-06        3e-06      1.7e-05
     4096        5e-06        4e-06        5e-06
     8192        9e-06        9e-06        9e-06
    16384      1.8e-05      1.7e-05      1.7e-05
    32768      3.3e-05      4.6e-05      3.3e-05
    65536      7.1e-05      7.1e-05      7.1e-05
   131072     0.000142     0.000143     0.000142
   262144     0.000286     0.000284     0.000283
   524288     0.000559     0.000532     0.000527
  1048576     0.001163     0.001142     0.001151
  2097152     0.002417     0.002354     0.002350
  4194304     0.004758     0.004843     0.004733
  8388608     0.009491     0.009605     0.009665
 16777216     0.019017     0.019326     0.019348
 33554432     0.038119     0.038327     0.038573
 67108864     0.077164     0.076971     0.077040
134217728     0.153913     0.153270     0.153367
268435456     0.306564     0.309114     0.306612
536870912     0.614487     0.615426     0.614708

GPU:
        N        Time1        Time2        Time3
      256     0.055382     0.000203     0.000208
      512     0.000184     0.000203     0.000204
     1024     0.000200     0.000200     0.000199
     2048     0.000202     0.000184     0.000182
     4096     0.000166     0.000155     0.000156
     8192     0.000231     0.000149     0.000147
    16384     0.000161     0.000160     0.000179
    32768     0.000196     0.000180     0.000174
    65536     0.000231     0.000221     0.000220
   131072     0.000325     0.000329     0.000305
   262144     0.000634     0.000626     0.000629
   524288     0.001025     0.001019     0.001038
  1048576     0.001774     0.001744     0.001770
  2097152     0.003340     0.003301     0.003262
  4194304     0.006390     0.006367     0.006392
  8388608     0.012403     0.012343     0.012417
 16777216     0.021943     0.022332     0.022116
 33554432     0.040394     0.040898     0.040242
```

```
 67108864     0.076952     0.074762     0.072239
134217728     0.132258     0.131437     0.131514
268435456     0.288641     0.270241     0.258923
536870912     0.515918     0.517701     0.517307
```