Student Name: Галактионов Кирилл
Student ID: st067889

**Современные технологии программирования**
**в научных исследованиях II**
**Лабораторная работа 2**

---

# 1 Постановка задачи

Условие задчи звучит следующим образом: "Дано 2 одномерных массива, инициализированных случайными числами. Требуется написать программу с использованием CUDA, которая выполняет сортировку обоих массивов по убыванию и вычисляет скалярное произведение двух отсортированных массивов."

# 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 сотрировки осуществлялась последовательным попарным сравнением элементов массива (время не включается в время выполнения). Проверка правильности скалярного произведения осуществлялась сравнением с последовательным вычислением на CPU.

# 3 Описание алгоритма решения задачи

В качестве алгорима сортировки массивы мною был выбран алгоритм сортировки пузырьком. В последовательной реализации он заключается в проходе по массиву и попарному сравнению пар чисел, и перемене их местами, при необходимости. Код:

```
1  // Bubble sort on CPU in one thread
2  void BubbleSortNoCUDA(double *array, int N){
3    for (int i = 0; i < N; i++){
4      for (int j = 0; j < N-i-1; j++) {
5        if (array[j]<array[j + 1]){
6          double helper = array[j];
```

```
        array[j] = array[j + 1];
        array[j + 1] = helper;
      }
    }
  }
}
```

Идея использования нитей CUDA в такой сортировке заключается в том, что когда "пузырёк" сортировки ушел вперед по массиву, то оставшиеся элементы можно уже сортировать ещё раз, таким образом запуская несколько сортирующих "пузырьков". Реализация кода нити и алгоритма сортировки представлена ниже:

```
__global__ void BubbleMove(double *array, int N, int step){
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < (N-1)) {
        if (step-2>=idx){
            if (array[idx] < array[idx + 1]){
                double helper = array[idx];
                array[idx] = array[idx + 1];
                array[idx + 1] = helper;
            }
        }
    }
}

// Bubble sort on GPU CUDA
void BubbleSortCUDA(double *array_host, int N, int blockSize){
    double *array_device;
    cudaMalloc((void **)&array_device, N * sizeof(double));
    cudaMemcpy(array_device, array_host, N*sizeof(double), cudaMemcpyHostToDevice);
    int nblocks = N / blockSize + 1;
    for (int step = 0; step <= N + N; step++) {
        // Step of bubble sort
        BubbleMove<<<nblocks, blockSize>>>(array_device, N, step);
        // Wait for all threads to finish changes
        //cudaThreadSynchronize();
        cudaDeviceSynchronize();
    }
    cudaMemcpy(array_host, array_device, N*sizeof(double), cudaMemcpyDeviceToHost);
    cudaFree(array_device);
}
```

Алгоритм скалярного произведения с использованием CUDA заключается в том, что каждая нить (блок нитей) скалярно пермножают свои части исходных массивов, а затем производится редукция полученных результатов (складываются полученные частичные скалярные произведения). Реализация представлена ниже:

```
_global__ void DotProduct(double const *deviceLHS, double const *deviceRHS, double *
    deviceRES, int N){
    // Shared array for threads in a block
    __shared__ double helper[THREADS_PER_BLOCK];

    // Get thread ID
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

```
7
8       // Compute dot products for all elemnts belonging to this thread
9       double sum = 0;
10      while (idx < N) {
11          sum += deviceLHS[idx] * deviceRHS[idx];
12          idx += blockDim.x * gridDim.x;
13      }
14      // ... and write into shared array
15      helper[threadIdx.x] = sum;
16
17      __syncthreads();
18
19      // Calculate results for current Block (reduction sum)
20      // Works if THREADS_PER_BLOCK is power of 2
21      int index = blockDim.x / 2;
22      while (index != 0){
23          if (threadIdx.x < index) {
24              helper[threadIdx.x] += helper[threadIdx.x + index];
25          }
26
27          __syncthreads();
28
29          index = index / 2;
30      }
31
32      // Save the results for block
33      if (threadIdx.x == 0) {
34          deviceRES[blockIdx.x] = helper[0];
35      }
36  }
37
38  // Dot product of two arrays on GPU CUDA
39  double CalculateDotProductCuda(double *hostLHS, double *hostRHS, int N){
40      double *hostInterRes; // helper for intermediate results
41      double *deviceLHS, *deviceRHS, *deviceRES;
42
43      // Thread blocks per Grid
44      int gridDim = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
45
46      // Create vectors
47      size_t interResSize = gridDim * sizeof(double);
48      hostInterRes = (double*)malloc(interResSize);
49      cudaMalloc(&deviceLHS, N * sizeof(double));
50      cudaMalloc(&deviceRHS, N * sizeof(double));
51      cudaMalloc(&deviceRES, interResSize);
52
53      // Arrays to gpu
54      cudaMemcpy(deviceLHS, hostLHS, N * sizeof(double), cudaMemcpyHostToDevice);
55      cudaMemcpy(deviceRHS, hostRHS, N * sizeof(double), cudaMemcpyHostToDevice);
56
57
58      DotProduct<<<gridDim, THREADS_PER_BLOCK>>>(deviceLHS, deviceRHS, deviceRES, N);
59
60      // Array to cpu
```

```
    cudaMemcpy(hostInterRes, deviceRES, interResSize, cudaMemcpyDeviceToHost);


    cudaFree(deviceLHS);
    cudaFree(deviceRHS);
    cudaFree(deviceRES);

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

    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

На рисунке 1 представлена зависимость времени, затраченного на вычисления, от размера массива. Можно заметить, что в логарифмическом масштабе зависимость близка к линейной, что говорит о степенном характере зависимости времени вычисления, при этом, угол наклона, а соответственно и показатель степени, у вычислений на GPU ниже, чем у вычислений на CPU.

При малых размерах массива затраты на пересылку данных и выделение нитей выше, чем требуемые вычислительные ресурсы, поэтому вычисления на CPU дают лучший результат. Но, начиная с размера массива в $10^4$ элементов, вычисления на GPU становятся более эффектывны.

Кроме того, был построен график зависимости ускорения вычислений на GPU, которое считалось как отношение времен затраченных на GPU к времени, затраченному на CPU. Зависимость ускорения от размера массива представлена на рисунке 2. С увеличением размера массива растет ускорение, достигая 10 раз на $10^5$ элементах.

## 7 Выводы

Поставленные задачи были выполнены. Было проведено сравнение работы алгоритмов на CPU и GPU и наблюдалось ускорение работы, которое может обеспечить GPU.

## 8 Приложение 1. Код программы

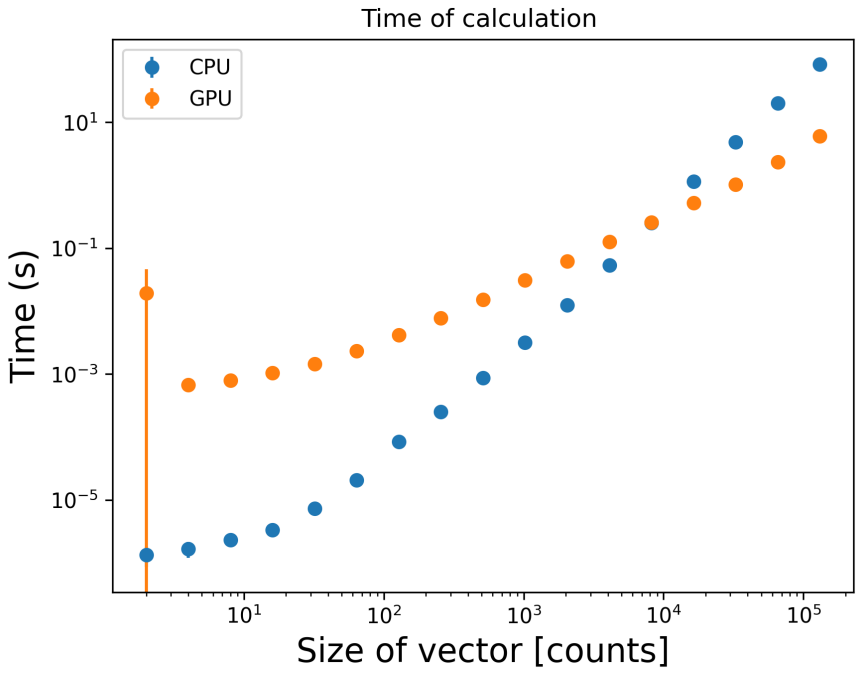


Рис. 1: Зависимость времени вычисления от размера массива.

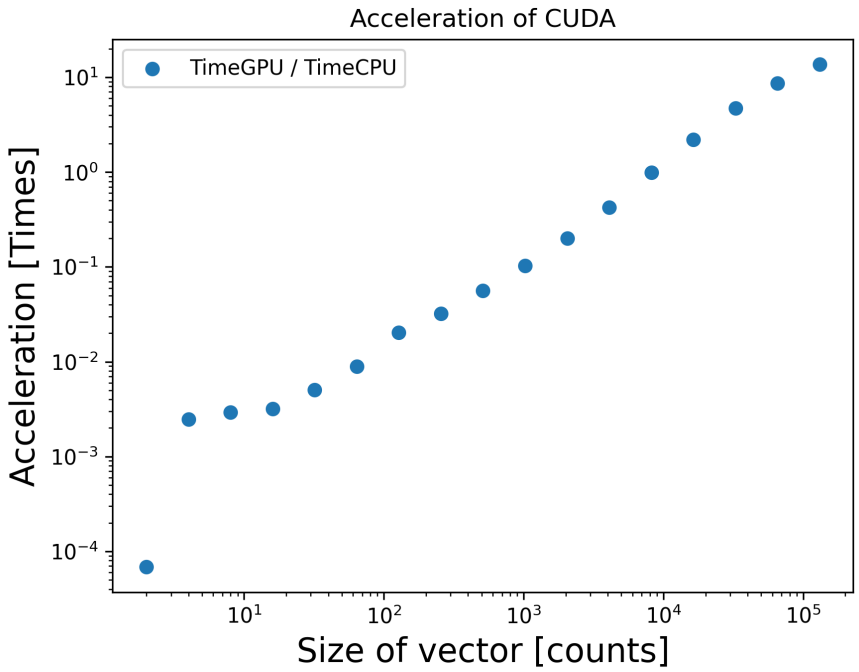


Рис. 2: Зависимость ускорения вычисления от размера массива.

```
1 #include <iostream>
2 #include <stdio.h>
3 #include <time.h>
```

```
#define THREADS_PER_BLOCK 1024

using namespace std;

__global__ void BubbleMove(double* array, int N, int step);

__global__ void DotProduct(double const *deviceLHS, double const *deiceRHS, double *
    deviceRES, int N);



void BubbleSortCUDA(double *array_host, int N, int blockSize);

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

void BubbleSortNoCUDA(double *array, int N);

bool IsSorted(double *array, int N);

void measureNoCUDA(int N);

void measureCUDA(int N);

double CalculateDotProductNoCuda(double *lhs, double *rhs, int N);

double CalculateDotProductCuda(double *hostLHS, double *hostRHS, 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 = 2; n < 100000; n*=2 ){
        cout<< n << "    ";
        for (int i = 0; i < 3; i++){
            measureNoCUDA(n);
            cout << "    ";
        }
        cout << endl;
    }

    cout << "GPU:" << endl;
    cout << "N      Time1     Time2      Time3" << endl;
    for (int n = 2; n < 100000; n*=2 ){
        cout<< n << "    ";
        for (int i = 0; i < 3; i++){
            measureCUDA(n);
            cout << "    ";
        }
        cout << endl;
    }
    cout << endl;
    return 0;
```

```
}

__global__ void BubbleMove(double *array, int N, int step){
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < (N-1)) {
        if (step-2>=idx){
            if (array[idx] < array[idx + 1]){
                double helper = array[idx];
                array[idx] = array[idx + 1];
                array[idx + 1] = helper;
            }
        }
    }
}

__global__ void DotProduct(double const *deviceLHS, double const *deviceRHS, double *
    deviceRES, int N){
    // Shared array for threads in a block
    __shared__ double helper[THREADS_PER_BLOCK];

    // Get thread ID
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Compute dot products for all elemnts belonging to this thread
    double sum = 0;
    while (idx < N) {
        sum += deviceLHS[idx] * deviceRHS[idx];
        idx += blockDim.x * gridDim.x;
    }
    // ... and write into shared array
    helper[threadIdx.x] = sum;

    __syncthreads();

    // Calculate results for current Block (reduction sum)
    // Works if THREADS_PER_BLOCK is power of 2
    int index = blockDim.x / 2;
    while (index != 0){
        if (threadIdx.x < index) {
            helper[threadIdx.x] += helper[threadIdx.x + index];
        }

        __syncthreads();

        index = index / 2;
    }

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

// Bubble sort on GPU CUDA
```

```
110 void BubbleSortCUDA(double *array_host, int N, int blockSize){
111     double *array_device;
112     cudaMalloc((void **)&array_device, N * sizeof(double));
113     cudaMemcpy(array_device, array_host, N*sizeof(double), cudaMemcpyHostToDevice);
114     int nblocks = N / blockSize + 1;
115     for (int step = 0; step <= N + N; step++) {
116         // Step of bubble sort
117         BubbleMove<<<nblocks, blockSize>>>(array_device, N, step);
118         // Wait for all threads to finish changes
119         //cudaThreadSynchronize();
120         cudaDeviceSynchronize();
121     }
122     cudaMemcpy(array_host, array_device, N*sizeof(double), cudaMemcpyDeviceToHost);
123     cudaFree(array_device);
124 }
125
126 // Bubble sort on CPU in one thread
127 void BubbleSortNoCUDA(double *array, int N){
128   for (int i = 0; i < N; i++){
129     for (int j = 0; j < N-i-1; j++) {
130       if (array[j]<array[j + 1]){
131         double helper = array[j];
132         array[j] = array[j + 1];
133         array[j + 1] = helper;
134       }
135     }
136   }
137 }
138
139 // check if and array is sorted
140 bool IsSorted(double *array, int N){
141     for (int i = 0; i < N-1; i++){
142         if (array[i] < array[i+1]) {
143             return false;
144         }
145     }
146     return true;
147 }
148
149 // measure time taken for all steps on CPU CUDA
150 void measureNoCUDA(int N) {
151     //cout << N << "   ";
152     double *array1 = (double *)malloc(N * sizeof(double));
153     double *array2 = (double *)malloc(N * sizeof(double));
154     clock_t start_time, end_time;
155     float timeArray1Sorted, timeArray2Sorted, timeDotProduct;
156     double resCPU = 0.0;
157     for (int i = 0; i < N; i++) {
158         array1[i] = static_cast <double> (rand()) / static_cast <double> (RAND_MAX);
159         array2[i] = static_cast <double> (rand()) / static_cast <double> (RAND_MAX);
160     }
161
162     start_time = clock();
163     BubbleSortNoCUDA(array1, N);
```

```
    end_time = clock();

    timeArray1Sorted = (float)(end_time - start_time) / CLOCKS_PER_SEC;

    // check that array is sorted
    if (IsSorted(array1, N)){
        //cout << timeArray1Sorted << "   ";
    }
    else {
        free(array1);
        free(array2);
        cout << "ERROR, ARRAY 1 NOT SORTED" << endl;
        return;
    }

    start_time = clock();
    BubbleSortNoCUDA(array2, N);
    end_time = clock();

    timeArray2Sorted = (float)(end_time - start_time) / CLOCKS_PER_SEC;

    // check that array is sorted
    if (IsSorted(array2, N)){
        //cout << timeArray2Sorted << "   ";
    }
    else {
        free(array1);
        free(array2);
        cout << "ERROR, ARRAY 2 NOT SORTED" << endl;
        return;
    }



    start_time = clock();
    resCPU = CalculateDotProductNoCuda(array1, array2, N);
    end_time = clock();

    timeDotProduct = (float)(end_time - start_time) / CLOCKS_PER_SEC;
    //cout << timeDotProduct << "   ";


    free(array1);
    free(array2);
    cout << timeArray1Sorted + timeArray2Sorted + timeDotProduct;
}

// measure time taken for all steps on GPU CUDA
void measureCUDA(int N){
    //cout << N << "   ";
    double *array1 = (double *)malloc(N * sizeof(double));
    double *array2 = (double *)malloc(N * sizeof(double));
    clock_t start_time, end_time;
    float timeArray1Sorted, timeArray2Sorted, timeDotProduct;
```

```
    double resGPU = 0.0;
    double resCPU = 0.0;
    for (int i = 0; i < N; i++) {
        array1[i] = static_cast <double> (rand()) / static_cast <double> (RAND_MAX);
        array2[i] = static_cast <double> (rand()) / static_cast <double> (RAND_MAX);
    }

    start_time = clock();
    BubbleSortCUDA(array1, N, THREADS_PER_BLOCK);
    end_time = clock();

    timeArray1Sorted = (float)(end_time - start_time) / CLOCKS_PER_SEC;

    // check that array is sorted
    if (IsSorted(array1, N)){
        //cout << timeArray1Sorted << "   ";
    }
    else {
        free(array1);
        free(array2);
        cout << "ERROR, ARRAY 1 NOT SORTED" << endl;
        return;
    }

    start_time = clock();
    BubbleSortCUDA(array2, N, THREADS_PER_BLOCK);
    end_time = clock();

    timeArray2Sorted = (float)(end_time - start_time) / CLOCKS_PER_SEC;

    // check that array is sorted
    if (IsSorted(array2, N)){
        //cout << timeArray2Sorted << "   ";
    }
    else {
        free(array1);
        free(array2);
        cout << "ERROR, ARRAY 2 NOT SORTED" << endl;
        return;
    }



    start_time = clock();
    resGPU = CalculateDotProductCuda(array1, array2, N);
    end_time = clock();

    timeDotProduct = (float)(end_time - start_time) / CLOCKS_PER_SEC;

    // compare to the CPU result
    resCPU = CalculateDotProductNoCuda(array1, array2, N);
    if (abs(resCPU - resGPU) < 1e-2){
        //cout << timeDotProduct << "   ";
    }
```

```
272     else {
273         free(array1);
274         free(array2);
275         cout << "ERROR, DOT PRODUCT DOESN'T MATCH CPU" << endl;
276         return;
277     }
278 
279 
280     free(array1);
281     free(array2);
282 
283     cout << timeArray1Sorted + timeArray2Sorted + timeDotProduct;
284 }
285 
286 // Print double array into console
287 void PrintArray(const double* array, const int n){
288     for(int i = 0; i < n; i++){
289         cout << array[i] <<  ' ';
290     }
291     cout << endl;
292 }
293 
294 // Dot product of two arrays on CPU
295 double CalculateDotProductNoCuda(double *lhs, double *rhs, int N){
296     double result = 0.0;
297     for (int i = 0; i < N; i++){
298         result += lhs[i] * rhs[i];
299     }
300     return result;
301 }
302 
303 // Dot product of two arrays on GPU CUDA
304 double CalculateDotProductCuda(double *hostLHS, double *hostRHS, int N){
305     double *hostInterRes; // helper for intermediate results
306     double *deviceLHS, *deviceRHS, *deviceRES;
307 
308     // Thread blocks per Grid
309     int gridDim = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
310 
311     // Create vectors
312     size_t interResSize = gridDim * sizeof(double);
313     hostInterRes = (double*)malloc(interResSize);
314     cudaMalloc(&deviceLHS, N * sizeof(double));
315     cudaMalloc(&deviceRHS, N * sizeof(double));
316     cudaMalloc(&deviceRES, interResSize);
317 
318     // Arrays to gpu
319     cudaMemcpy(deviceLHS, hostLHS, N * sizeof(double), cudaMemcpyHostToDevice);
320     cudaMemcpy(deviceRHS, hostRHS, N * sizeof(double), cudaMemcpyHostToDevice);
321 
322 
323     DotProduct<<<gridDim, THREADS_PER_BLOCK>>>(deviceLHS, deviceRHS, deviceRES, N);
324 
325     // Array to cpu
```

```
    cudaMemcpy(hostInterRes, deviceRES, interResSize, cudaMemcpyDeviceToHost);


    cudaFree(deviceLHS);
    cudaFree(deviceRHS);
    cudaFree(deviceRES);

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

    return res;
}
```

# 9 Приложение 2. Результаты

```
CPU:
N           Time1       Time2       Time3
2           1e-06       1e-06       2e-06
4           1e-06       2e-06       2e-06
8           3e-06       2e-06       2e-06
16          3e-06       3e-06       4e-06
32          8e-06       7e-06       7e-06
64          1.9e-05     2.2e-05     2.1e-05
128         0.000105    8.2e-05     6.8e-05
256         0.000281    0.000235    0.000237
512         0.000894    0.000848    0.000848
1024        0.003152    0.003177    0.003204
2048        0.012514    0.012566    0.012473
4096        0.053377    0.053778    0.053673
8192        0.25175     0.252643    0.251596
16384       1.14409     1.14356     1.14078
32768       4.86105     4.86254     4.86399
65536       20.2512     20.2415     20.2577
131072      83.0369     83.1161     83.1233

GPU:
N           Time1       Time2       Time3
2           0.057244    0.000603    0.000603
4           0.00066     0.000672    0.00068
8           0.000797    0.000803    0.000798
16          0.001041    0.001044    0.001051
32          0.001512    0.001424    0.001424
64          0.002335    0.00233     0.002326
128         0.004164    0.004183    0.00418
256         0.00783     0.007823    0.007822
512         0.015337    0.01537     0.01535
1024        0.030781    0.030888    0.030949
2048        0.062373    0.062143    0.062205
4096        0.126013    0.125975    0.126324
```

| | | | |
|---|---|---|---|
| 8192 | 0.255572 | 0.25552 | 0.255696 |
| 16384 | 0.526794 | 0.519356 | 0.519172 |
| 32768 | 1.06806 | 1.00689 | 1.00526 |
| 65536 | 2.33396 | 2.32828 | 2.32841 |
| 131072 | 6.08528 | 6.0879 | 6.09131 |