МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Кафедра вычислительной математики и программирования

**спецкурс «Параллельные и распределенные вычисления»**

**ОТЧЕТ**

**Лабораторная работа**

**(дополнительное задание)**

**«Сумма квадратов методом scan»**

Выполнил: Сотников И. Д.

Группа: М8О-207М-20

Преподаватель: Семенов С. А.

Москва 2022

Содержание

[1. Постановка задачи 3](#_Toc52570380)

[2. Описание решения 3](#_Toc52570381)

[3. Аппаратное обеспечение и ПО 6](#_Toc52570382)

[4. Основные моменты кода 7](#_Toc52570383)

[5. Результат работы программы 11](#_Toc52570384)

[6. Сравнение скорости выполнения на CPU и GPU 13](#_Toc52570385)

[7. Выводы 15](#_Toc52570386)

[8. Приложения 15](#_Toc52570387)

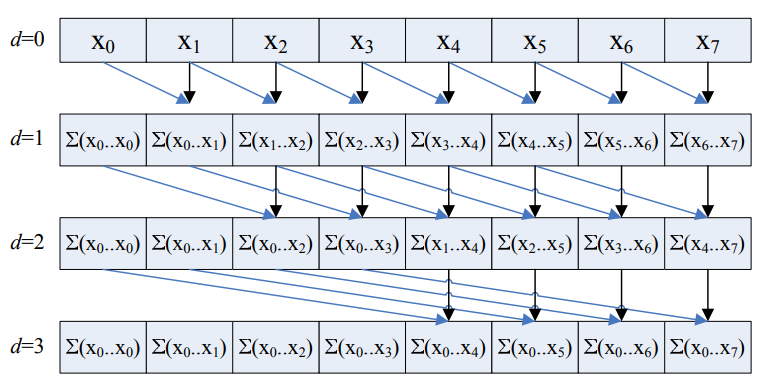
# 1. Постановка задачи

Найти сумму квадратов методом scan. Должны использоваться разные типы памяти.

# 2. Описание решения

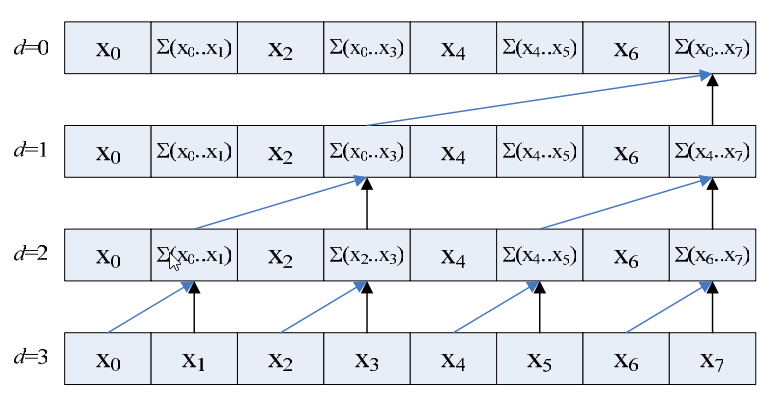
Программа будет предлагать ввести размер вектора, сумму квадратов элементов которого будет считать. Вектор будет заполнен двойками, чтобы не переполнять память при больших размерах вектора и для быстрой проверки правильности результата.

Алгоритм будет реализовывать невключающий scan (exclusive scan), то есть сумма в определенном индексе не будет включать значение в этом индексе. При наивной реализации алгоритма scan параллельно будет совершаться n сложений log(n) раз, то есть вычислительная сложность О(n \* log(n)), в отличии от однопоточного алгоритма, где идет простое последовательное суммирование элементов со сложностью О(n). При больших входных значениях это может быть неэффективно.



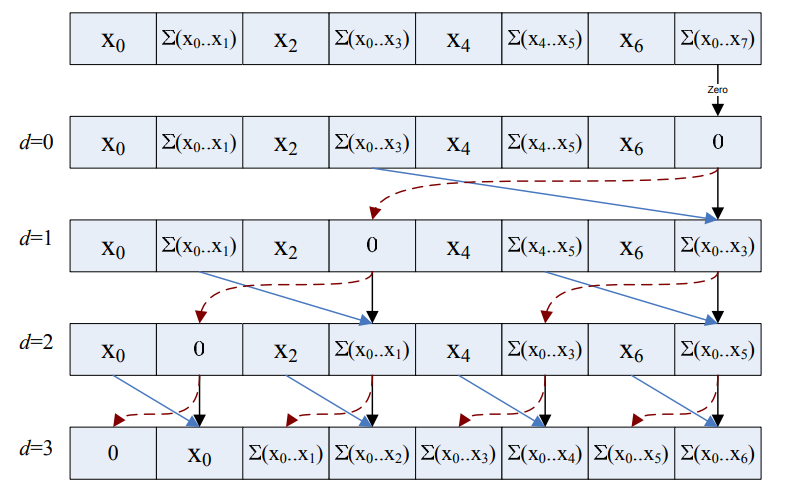
**Рис.** Наивный параллельный алгоритм scan

Чтобы достичь алгоритмической сложности О(n), будет использоваться подход с двоичным деревом. Двоичное дерево имеет глубину log(n), каждый уровень х имеет 2х узлов. Если будет выполняться одно сложение в одном узле, то будет выполняться О(n) за один обход дерева. Алгоритм состоит из двух фаз: проход вверх и проход вниз. На первой фазе мы проходим дерево от листьев к корню, вычисляя частичные суммы во внутренних узлах дерева. На этой фазе вычисления производятся в shared memory. Нить обрабатывает по два элемента. Число потоков уменьшается по мере приближения к корню.



**Рис.** Первая фаза прохода

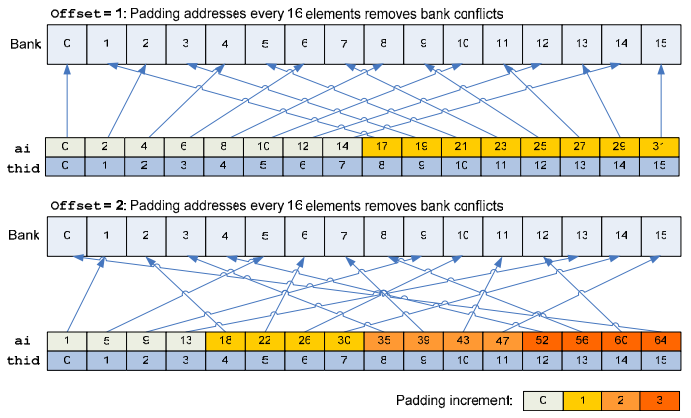
При обходе вниз идет обход от корня к листьям и строиться результирующий вектор из частичных сумм, вычисленных на предыдущем этапе. Результат на каждом уровне записывается из shared memory в global memory.



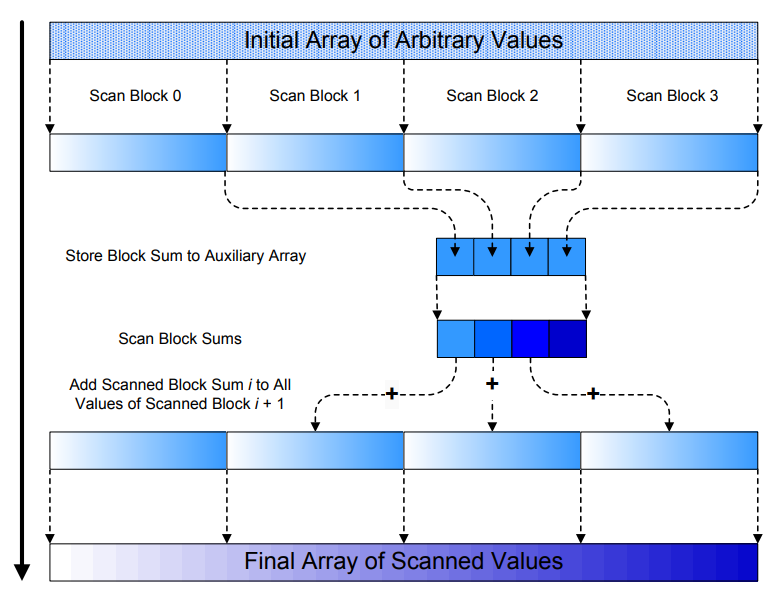
**Рис.** Вторая фаза прохода

Для устранения конфликтов памяти, когда несколько потоков обращается к одной ячейке памяти, установим индекс смещения для каждого потока. Для это добавим к каждому индексу значение индекса, деленное на общее количество банков памяти.

|  |
| --- |
| **Макросы для смещения индексов:** |
| #define SHARED\_MEMORY\_BANKS 32  #define LOG\_MEM\_BANKS 5  #define CONFLICT\_FREE\_OFFSET(n) ((n) >> LOG\_MEM\_BANKS) |



Для векторов большего размера, чем количество нитей в блоке \* 2 (так как каждая нить обрабатывает 2 элемента), используется алгоритм скана частичных сумм. В каждый блок загружается часть вектора, и по нему делается скан. Далее из каждого блока берется сумма всех элементов его части вектора (последний элемент скана), эти суммы собираются в вектор и по нему снова выполняется скан. Результат прибавляется каждому вектору частичных сканов, как показано на рисунке.



**Рис.** Алгоритм скана частичных сумм.

Если вектор сумм также не помещается в блок, алгоритм повторяется рекурсивно.

# 3. Аппаратное обеспечение и ПО

Процессор: AMD Ryzen 5 3600 (6 core, 12 thread, 3.6 GHz);

RAM: 16 GB;

Графический процессор: Nvidia GeForce GTX 460 SE (

Total amount of global memory: 1024 MB

(6) Multiprocessors, (48) CUDA Cores/MP: 288 CUDA Cores

GPU Max Clock rate: 1460 MHz

Memory Clock rate: 1700 MHz

Memory Bus Width: 256-bit);

CUDA Driver Version / Runtime Version: 8.0 / 8.0;

CUDA Capability Major/Minor version number: 2.1.

# 4. Основные моменты кода

**Ядра**

Ядро вычисления скана вектора произвольной длины (в одном блоке). Вначале идет выделение памяти и копирования двух элементов из глобальной в shared память с учетом смещения. Если это не скан сумм, по условию isSumScan значение возводится в квадрат, так как нам нужно рассчитать скан квадратов (и найти их общую сумму). Далее идет обход вверх и вниз по дереву. Обход вверх работает только с shared памятью, обход вниз записывает результат из shared в глобальную память.

|  |
| --- |
| **Ядро вычисления скана в одном блоке** |
| \_\_global\_\_ void prescan\_arbitrary(int \*output, int \*input, int n, int powerOfTwo, bool isSumScan)  {  extern \_\_shared\_\_ int temp[];// выделение памяти  int threadID = threadIdx.x;  int ai = threadID;  int bi = threadID + (n / 2);  int bankOffsetA = CONFLICT\_FREE\_OFFSET(ai);  int bankOffsetB = CONFLICT\_FREE\_OFFSET(bi);  if (threadID < n) {  int aiInput = input[ai];  int biInput = input[bi];  if (!isSumScan) {  aiInput \*= aiInput;  biInput \*= biInput;  }  temp[ai + bankOffsetA] = aiInput;  temp[bi + bankOffsetB] = biInput;  }  else {  temp[ai + bankOffsetA] = 0;  temp[bi + bankOffsetB] = 0;  }  int offset = 1;  for (int d = powerOfTwo >> 1; d > 0; d >>= 1) // обход вверх  {  \_\_syncthreads();  if (threadID < d)  {  int ai = offset \* (2 \* threadID + 1) - 1;  int bi = offset \* (2 \* threadID + 2) - 1;  ai += CONFLICT\_FREE\_OFFSET(ai);  bi += CONFLICT\_FREE\_OFFSET(bi);  temp[bi] += temp[ai];  }  offset \*= 2;  }  if (threadID == 0) // обнуление последнего элемента, так как prescan {  temp[powerOfTwo - 1 + CONFLICT\_FREE\_OFFSET(powerOfTwo - 1)] = 0;  }  for (int d = 1; d < powerOfTwo; d \*= 2) // обход вниз и вычисление скана  {  offset >>= 1;  \_\_syncthreads();  if (threadID < d)  {  int ai = offset \* (2 \* threadID + 1) - 1;  int bi = offset \* (2 \* threadID + 2) - 1;  ai += CONFLICT\_FREE\_OFFSET(ai);  bi += CONFLICT\_FREE\_OFFSET(bi);  int t = temp[ai];  temp[ai] = temp[bi];  temp[bi] += t;  }  }  \_\_syncthreads();  if (threadID < n) {  output[ai] = temp[ai + bankOffsetA];  output[bi] = temp[bi + bankOffsetB];  }  } |
|  |

Ядро для вычисления скана в нескольких блоках отличается только формированием вектора частичных сумм для каждого блока.

|  |
| --- |
| **Ядро для вычисления скана в нескольких блоках** |
| \_\_global\_\_ void prescan\_large(int \*output, int \*input, int n, int \*sums, bool isSumScan) {  extern \_\_shared\_\_ int temp[];  int blockID = blockIdx.x;  int threadID = threadIdx.x;  int blockOffset = blockID \* n;  int ai = threadID;  int bi = threadID + (n / 2);  int bankOffsetA = CONFLICT\_FREE\_OFFSET(ai);  int bankOffsetB = CONFLICT\_FREE\_OFFSET(bi);  int aiInput = input[blockOffset + ai];  int biInput = input[blockOffset + bi];  if (!isSumScan) {  aiInput \*= aiInput;  biInput \*= biInput;  }  temp[ai + bankOffsetA] = aiInput;  temp[bi + bankOffsetB] = biInput;  int offset = 1;  for (int d = n >> 1; d > 0; d >>= 1) {  \_\_syncthreads();  if (threadID < d)  {  int ai = offset \* (2 \* threadID + 1) - 1;  int bi = offset \* (2 \* threadID + 2) - 1;  ai += CONFLICT\_FREE\_OFFSET(ai);  bi += CONFLICT\_FREE\_OFFSET(bi);  temp[bi] += temp[ai];  }  offset \*= 2;  }  \_\_syncthreads();  if (threadID == 0) {  sums[blockID] = temp[n - 1 + CONFLICT\_FREE\_OFFSET(n - 1)];  temp[n - 1 + CONFLICT\_FREE\_OFFSET(n - 1)] = 0;  }  for (int d = 1; d < n; d \*= 2) {  offset >>= 1;  \_\_syncthreads();  if (threadID < d)  {  int ai = offset \* (2 \* threadID + 1) - 1;  int bi = offset \* (2 \* threadID + 2) - 1;  ai += CONFLICT\_FREE\_OFFSET(ai);  bi += CONFLICT\_FREE\_OFFSET(bi);  int t = temp[ai];  temp[ai] = temp[bi];  temp[bi] += t;  }  }  \_\_syncthreads();  output[blockOffset + ai] = temp[ai + bankOffsetA];  output[blockOffset + bi] = temp[bi + bankOffsetB];  } |

Ядро для инкрементирования частичных сканов значениями скана сумм, для реализации скана вектора произвольного размера.

|  |
| --- |
| **Ядра для добавления элементов скана сумм к частичным сканам** |
| \_\_global\_\_ void add(int \*output, int length, int \*incr) {  int blockOffset = blockIdx.x \* length;  output[blockOffset + threadIdx.x] += incr[blockIdx.x];  } |

**Функции на host**

Функция скана на GPU. Если элементов во входном векторе больше чем THREADS\_PER\_BLOCK \* 2 (ELEMENTS\_PER\_BLOCK) то выполняется функция скана большого вектора, иначе маленького.

|  |
| --- |
| **Функция скана на GPU** |
| void gpuScan(int \*output, int \*input, int length) {  int \*d\_out, \*d\_in;  const int arraySize = length \* sizeof(int);  cudaMalloc((void \*\*)&d\_out, arraySize);  cudaMalloc((void \*\*)&d\_in, arraySize);  cudaMemcpy(d\_out, output, arraySize, cudaMemcpyHostToDevice);  cudaMemcpy(d\_in, input, arraySize, cudaMemcpyHostToDevice);  if (length > ELEMENTS\_PER\_BLOCK) {  scanLargeDeviceArray(d\_out, d\_in, length, false);  }  else {  scanSmallDeviceArray(d\_out, d\_in, length, false);  }  cudaMemcpy(output, d\_out, arraySize, cudaMemcpyDeviceToHost);  cudaFree(d\_out);  cudaFree(d\_in);  } |

Функция скана большого вектора. Если количество элементов кратно ELEMENTS\_PER\_BLOCK то выполняется скан без остатка. Иначе выполняется скан кратного кол-ва элементов, а затем скан остатка.

|  |
| --- |
| **Функция скана большого вектора** |
| void scanLargeDeviceArray(int \*d\_out, int \*d\_in, int length, bool isSumScan) {  int remainder = length % (ELEMENTS\_PER\_BLOCK);  if (remainder == 0) {  scanLargeEvenDeviceArray(d\_out, d\_in, length, isSumScan);  }  else {  // скан кратного кол-ва элементов  int lengthMultiple = length - remainder;  scanLargeEvenDeviceArray(d\_out, d\_in, lengthMultiple, isSumScan);  // скан остальных элементов и добавление (включительно) последнего элемента «большого» скана  int \*startOfOutputArray = &(d\_out[lengthMultiple]);  scanSmallDeviceArray(startOfOutputArray, &(d\_in[lengthMultiple]), remainder, isSumScan);  add<<<1, remainder>>>(startOfOutputArray, remainder, &(d\_in[lengthMultiple - 1]), &(d\_out[lengthMultiple - 1]));  }  } |

Функция скана длинного вектора, кратного кол-ву элементов в блоке. Если вектор сумм не помещается в один блок, функция выполняется еще раз рекурсивно, но с флагом isSumScan = true, таким образом суммы не возводятся в квадрат.

|  |
| --- |
| **Функция скана большого кратного вектора** |
| void scanLargeEvenDeviceArray(int \*d\_out, int \*d\_in, int length, bool isSumScan) {  const int blocks = length / ELEMENTS\_PER\_BLOCK;  const int sharedMemArraySize = ELEMENTS\_PER\_BLOCK \* sizeof(int);  int \*d\_sums, \*d\_incr;  cudaMalloc((void \*\*)&d\_sums, blocks \* sizeof(int));  cudaMalloc((void \*\*)&d\_incr, blocks \* sizeof(int));  prescan\_large<<<blocks, THREADS\_PER\_BLOCK, 2 \* sharedMemArraySize>>>(d\_out, d\_in, ELEMENTS\_PER\_BLOCK, d\_sums, isSumScan);  const int sumsArrThreadsNeeded = (blocks + 1) / 2;  if (sumsArrThreadsNeeded > THREADS\_PER\_BLOCK) {  // perform a large scan on the sums arr  scanLargeDeviceArray(d\_incr, d\_sums, blocks, true);  }  else {  // only need one block to scan sums arr so can use small scan  scanSmallDeviceArray(d\_incr, d\_sums, blocks, true);  }  add<<<blocks, ELEMENTS\_PER\_BLOCK>>>(d\_out, ELEMENTS\_PER\_BLOCK, d\_incr);  cudaFree(d\_sums);  cudaFree(d\_incr);  } |

Функция скана вектора меньшего размера чем кол-во элементов в блоке.

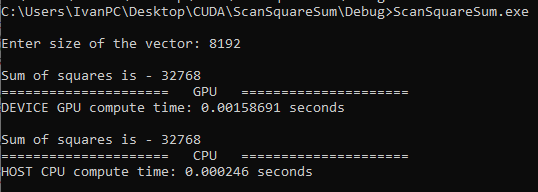
|  |
| --- |
| **Функция скана маленького вектора** |
| void scanSmallDeviceArray(int \*d\_out, int \*d\_in, int length, bool isSumScan) {  int powerOfTwo = nextPowerOfTwo(length);  prescan\_arbitrary<<<1, (length + 1) / 2, 2 \* powerOfTwo \* sizeof(int)>>>(d\_out, d\_in, length, powerOfTwo, isSumScan);  } |

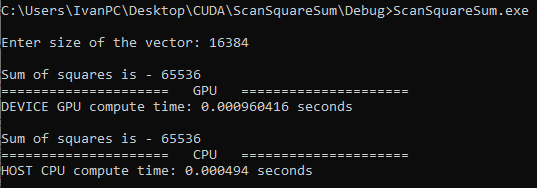
Функция скана на CPU.

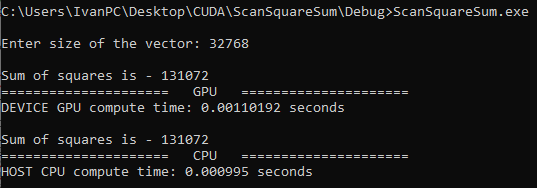
|  |
| --- |
| **Функция скана на CPU** |
| void cpuScan(int\* output, int\* input, int length) {  output[0] = pow(input[0], 2);  for (int j = 1; j < length; ++j)  {  output[j] = pow(input[j - 1], 2) + output[j - 1];  }  } |

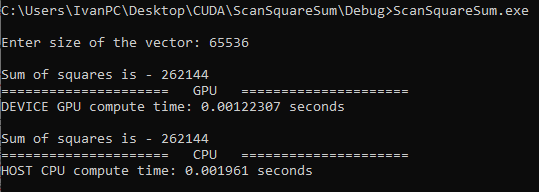
# 5. Результат работы программы

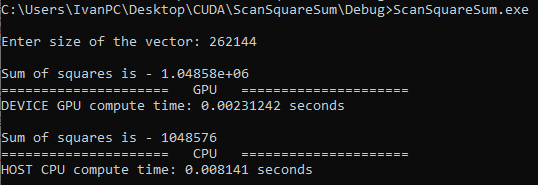
На вход программы подается размер вектора, сумму квадратов которого нужно вычислить:

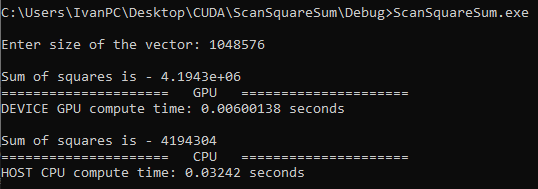


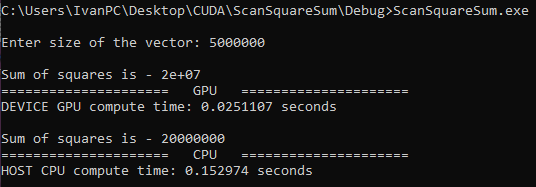


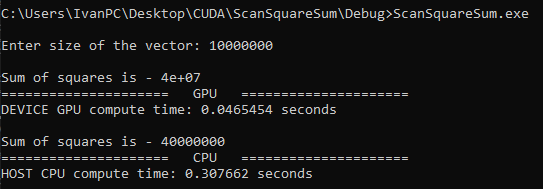










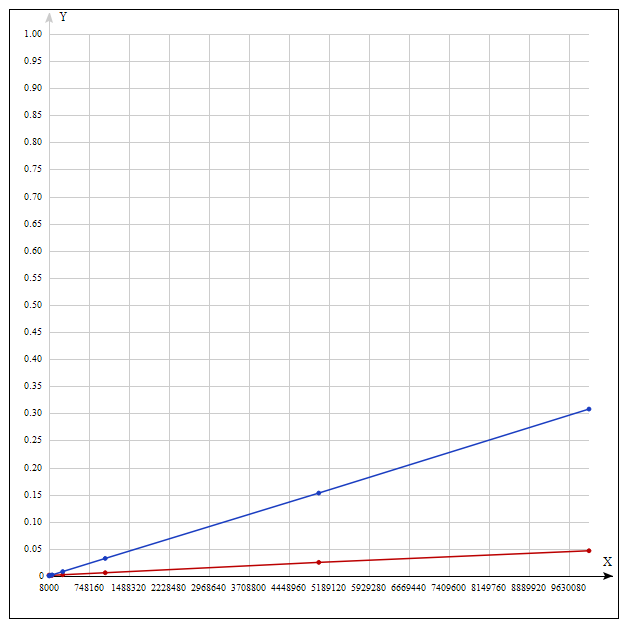


# 6. Сравнение скорости выполнения на CPU и GPU

При запуске программы с различными значениями *N* видно, что сначала вычисления быстрее на процессоре, но по мере увеличения *N* отрыв по времени увеличивается в пользу видеокарты. На больших значениях время выполнения различается значительно.

Время выполнения программы при различных значениях *N*:

|  |  |  |
| --- | --- | --- |
| **N** | **GPU**  **время выполнения, сек** | **CPU**  **время выполнения, сек** |
| 8192 | 1.587 \* 10-3 | 2.46 \* 10-4 |
| 16384 | 9.6 \* 10-4 | 4.94 \* 10-4 |
| 32768 | 1.101 \* 10-3 | 9.95 \* 10-4 |
| 65536 | 1.223 \* 10-3 | 1.961 \* 10-3 |
| 262144 | 2.312 \* 10-3 | 8.141 \* 10-3 |
| 1048576 | 6 \* 10-3 | 3.242 \* 10-2 |
| 5000000 | 2.511 \* 10-2 | 0.153 |
| 10000000 | 4.654 \* 10-2 | 0.308 |



**Рис.** График зависимости времени выполнения программы от размера входных данных *N*, (красный цвет – GPU, синий цвет – CPU)

# 7. Выводы

В Лабораторной работе написана программа расчета суммы квадратов вектора с помощью алгоритма scan, выполняющаяся на CPU и GPU соответственно. Был проведен анализ времени работы и сравнение двух реализаций подсчета на CPU и GPU.

# 8. Приложения

Проект Visual Studio: https://github.com/L0F1/CUDA\_labs