МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Кафедра вычислительной математики и программирования

**спецкурс «Параллельные и распределенные вычисления»**

**ОТЧЕТ**

**Лабораторная работа**

**(дополнительное задание)**

**«Битоническая сортировка»**

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 4](#_Toc52570382)

[4. Основные моменты кода 5](#_Toc52570383)

[5. Результат работы программы 7](#_Toc52570384)

[6. Сравнение скорости выполнения на CPU и GPU 9](#_Toc52570385)

[7. Выводы 10](#_Toc52570386)

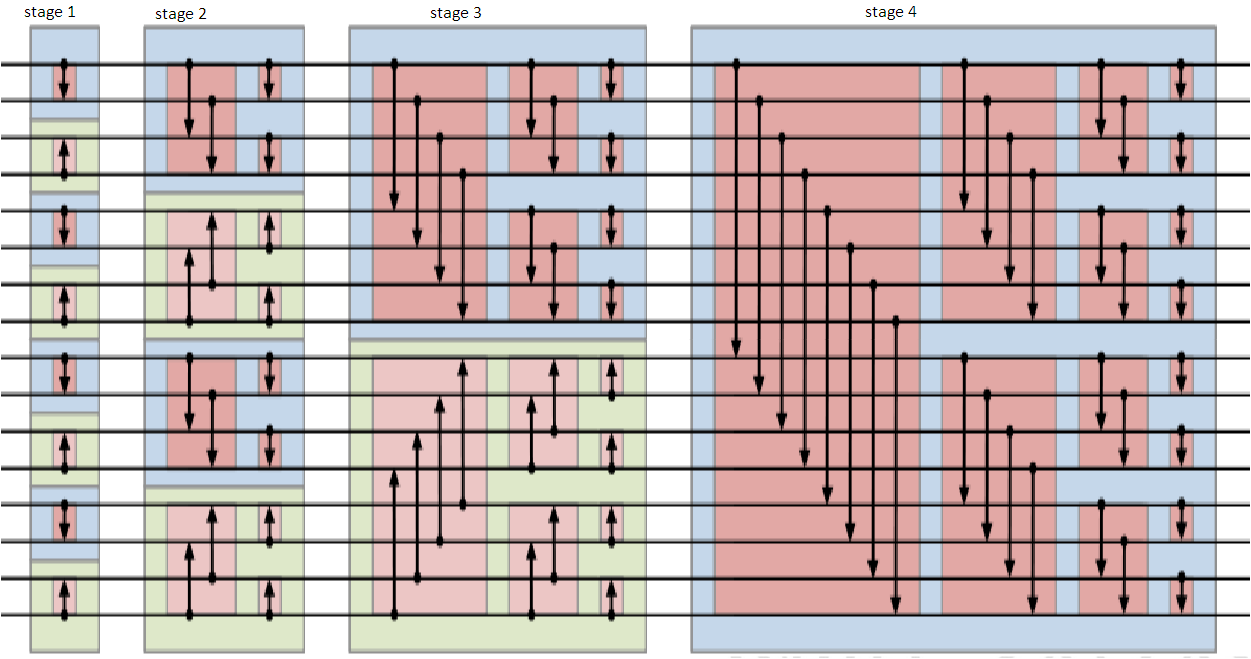
[8. Приложения 10](#_Toc52570387)

# 1. Постановка задачи

Отсортировать массив с помощью битонической сортировки.

# 2. Описание решения

Битоническая сортировка основана на понятии битонической последовательности. Битонической последовательностью называется конечный упорядоченный массив чисел, в котором они сначала монотонно возрастают, а затем монотонно убывают. Все параллельные алгоритмы можно представить в виде сети сортировки или компараторов. Алгоритм будет реализовывать данную сеть:



Здесь мы видим четыре этапа, каждый из которых имеет переменное количество колонок. Каждый узел битонической сортировочной сети идентифицируется тремя целыми числами, которые являются этапом, столбцом внутри этапа и строкой узла. Каждый столбец имеет N/2 сравнений. Таким образом, для параллельного выполнения этого алгоритма нам потребуется N/2 нитей.

Имея неотсортированную последовательность, построим битонические последовательности размера 4 (этап 3) и применим битоническое слияние, т.е. сортировку маленьких битонических последовательностей таким образом, чтобы массив распался на битонические последовательности двойного размера. На каждом шаге битоническая последовательность делится на две маленькие битонические последовательности половинной длины так, чтобы все элементы одной последовательности были меньше всех элементов другой последовательности. Если мы применим эту операцию log2(N) раз, то размер битонической последовательности станет 2\*N, а ее ось будет находиться в конце массива, т.е. значения от начала последовательности к концу идут в порядке возрастания, таким образом, массив будет отсортирован.

Допустим, у нас есть N/2 нитей, каждый из которых имеет индекс от 0 до N/2-1. Определим, какая нить имеет дело с каким элементом на этапе k и шаге j: допустим, tid - это индекс нити, тогда два элемента, которые сравниваются нитью на этапе g и шаге t, находятся в позициях e1 и e2:

|  |
| --- |
|  |
| e1 = map  e2 = map + (2(j - 1))  pos = (map / 2k) % 2 |

Где map = (tid / (2(j - 1)))\*(2j) + (tid % (2(t - 1)))

pos определяет порядок сравнения (если равно 0 – по возрастанию, иначе по убыванию).

# 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

В ядре используются функции atomicMin и atomicMax для сравнения и замены элементов атомарной операцией:

|  |
| --- |
| **Ядро** |
| \_\_global\_\_ void bitonicSortKernel(int\* d\_arr, int n, int k, int j) {  int tid = blockIdx.x \* blockDim.x + threadIdx.x,  map = (tid / (1 << (j - 1)))\*(1 << j) + (tid % (1 << (j - 1))),  pos = (map / (1 << k)) % 2,  e1 = (pos == 0) ? map : (map + (1 << (j - 1))),  e2 = (pos == 0) ? (map + (1 << (j - 1))) : map;  atomicMin(&d\_arr[e1], atomicMax(&d\_arr[e2], d\_arr[e1]));  \_\_syncthreads();  } |

Ядро выполняется на каждом шаге в этапе:

|  |
| --- |
| **Функция сортировки на GPU:** |
| void bitonicSortGpu(int\* arr, int N) {  int\* d\_arr;  int logn2 = (int)(log(N) / log(2));  cudaMalloc((void \*\*)&d\_arr, sizeof(int)\*N);  cudaMemcpy(d\_arr, arr, sizeof(int) \* N, cudaMemcpyHostToDevice);  int blocks = ((N / 2) + THREADS\_PER\_BLOCK - 1) / THREADS\_PER\_BLOCK;  for (int k = 1; k <= logn2; k++) {  for (int j = k; j > 0; j--) {  bitonicSortKernel<<<blocks, THREADS\_PER\_BLOCK >>>(d\_arr, N, k, j);  }  }  cudaDeviceSynchronize();  cudaMemcpy(arr, d\_arr, N \* sizeof(int), cudaMemcpyDeviceToHost);  cudaFree(d\_arr);  } |

Функция сортировки на CPU идентична ядру. Вместо параллельной сортировки на каждом шаге, выполняется сортировка в цикле от 0 до N/2 (так как каждый обход цикла сравнивает два значения):

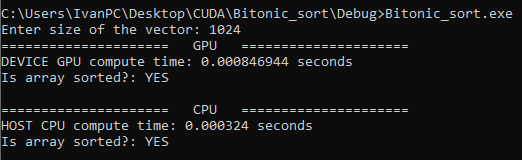
|  |
| --- |
| **Функция сортировки на CPU:** |
| void bitonicSortCpu(int\* arr, int N) {  int logn2 = (int)(log(N) / log(2));  for (int k = 1; k <= logn2; k++) {  for (int j = k; j > 0; j--) {  for (int i = 0; i < N/2; i++)  {  int map = (i / (1 << (j - 1)))\*(1 << j) + (i % (1 << (j - 1))),  pos = (map / (1 << k)) % 2,  e1 = (pos == 0) ? map : (map + (1 << (j - 1))),  e2 = (pos == 0) ? (map + (1 << (j - 1))) : map;  if (arr[e1] > arr[e2]) {  float temp = arr[e1];  arr[e1] = arr[e2];  arr[e2] = temp;  }  }  }  }  } |

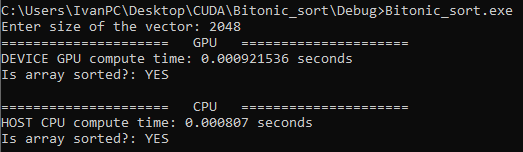
|  |
| --- |
| **Функция проверки сортировки:** |
| bool isSorted(int\* arr, int N) {  for (int i = 1; i < N; i++)  {  if (arr[i] < arr[i - 1])  return false;  }  return true;  } |

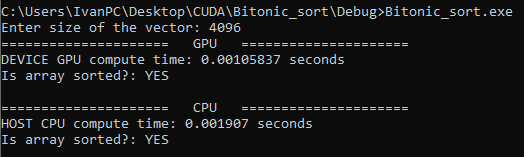
|  |
| --- |
| **Функция main:** |
| int main() {  int N;  std::cout << "Enter size of the vector: ";  std::cin >> N;  // start timer  cudaEvent\_t start, stop;  cudaEventCreate(&start);  cudaEventCreate(&stop);  cudaEventRecord(start);  int\* arr = fillArray(N);  bitonicSortGpu(arr, N);  // end timer  cudaEventRecord(stop);  cudaEventSynchronize(stop);  float gpuTime = 0;  cudaEventElapsedTime(&gpuTime, start, stop);  std::cout << "===================== GPU =====================\n";  std::cout << "DEVICE GPU compute time: " << double(gpuTime) / pow(10, 3) << " seconds\n";  printf("Is array sorted?: %s", isSorted(arr, N) ? "YES\n\n" : "NO\n\n");  using micro = std::chrono::microseconds;  auto start2 = std::chrono::high\_resolution\_clock::now();  delete[] arr;  arr = fillArray(N);  bitonicSortCpu(arr, N);  auto stop2 = std::chrono::high\_resolution\_clock::now();  auto cpuTime = std::chrono::duration\_cast<micro>(stop2 - start2).count();  std::cout << "===================== CPU =====================\n";  std::cout << "HOST CPU compute time: " << double(cpuTime) / pow(10, 6) << " seconds\n";  printf("Is array sorted?: %s", isSorted(arr, N) ? "YES\n\n" : "NO\n\n");  return 0;  } |

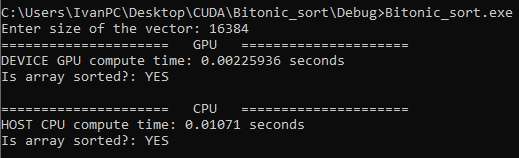
# 5. Результат работы программы

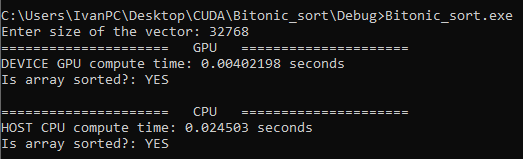
На вход программы подается размер вектора для сортировки. Значения генерируются в убывающем порядке, так что каждый раз меряется время в худшем случае:

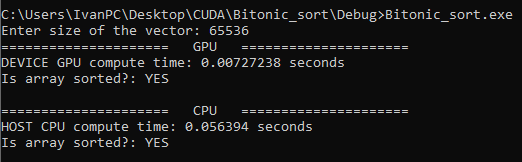


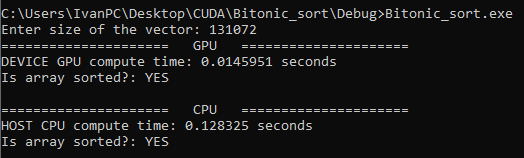


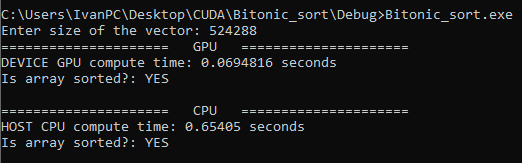










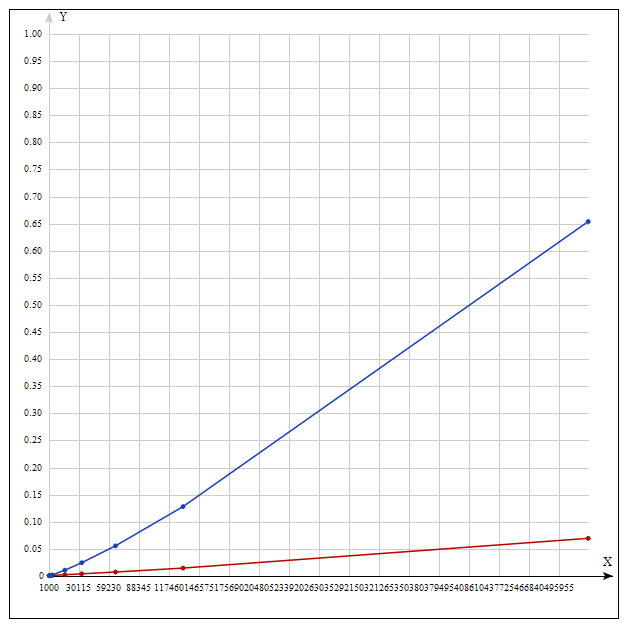


# 6. Сравнение скорости выполнения на CPU и GPU

При запуске программы с различными значениями *N* видно, что с повышением *N* время вычисления на процессоре сильно возрастает, когда как на видеокарте увеличивается не сильно.

Время выполнения программы при различных значениях *N*:

|  |  |  |
| --- | --- | --- |
| **N** | **GPU**  **время выполнения, сек** | **CPU**  **время выполнения, сек** |
| 1024 | 8.469 \* 104 | 3.24 \* 104 |
| 2048 | 9.215 \* 104 | 8.07 \* 104 |
| 4096 | 1.058 \* 103 | 1.907 \* 103 |
| 16384 | 2.259 \* 103 | 1.071 \* 102 |
| 32768 | 4.022 \* 103 | 2.45 \* 102 |
| 65536 | 7.272 \* 103 | 5.564 \* 102 |
| 131072 | 1.46 \* 102 | 0.128 |
| 524288 | 6.948 \* 102 | 0.654 |



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

# 7. Выводы

В Лабораторной работе написана программа сортировки, выполняющаяся на CPU и GPU соответственно. Был проведен анализ времени работы и сравнение двух реализаций битонической сортировки на CPU и GPU.

# 8. Приложения

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