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

Образовательная программа магистратуры
«Прикладные математика и физика»

Лабораторная работа
ЗАДАНИЕ 2
Сортировка и скалярное произведение

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

Санкт-Петербург
2023

СОДЕРЖАНИЕ

ФОРМУЛИРОВКА ЗАДАНИЯ

Дано два одномерных массива $\mathbf{A}$, $\mathbf{B} \in \mathbf{R}^N$, инициализированных случайными числами. Необходимо написать программу с использованием CUDA, которая:

1. Выполняет сортировку обоих массивов по возрастанию.

2. Вычисляет скалярное произведение двух отсортированных массивов.

Алгоритм сортировки выбрать самостоятельно. Выполнить сравнение производительности с вычислениями на хосте.

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

Сперва массивы $\mathbf{A}$ и $\mathbf{B}$ размера `arrayLength` заполняются случайными числами с плавающей точкой (тип `float`), имеющими значение от 0 до 1. Отвечающий инициализации массивов код представлен в Листинге 1.

Листинг 1: Инициализация массивов $\mathbf{A}$ и $\mathbf{B}$ размера `arrayLength` случайными числами с плавающей точкой (тип `float`), имеющими значение от 0 до 1.

```
1 /** Fills array with random values with uniform distribution between 0 and 100
2  * @param a Array of float-typed numbers
3  * @param n Size of the array a
```

```
 */
void fillArray(float* a, size_t n) {
  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(0, 1);
  for (size_t i = 0; i < n; i++) a[i] = dis(gen);
}
```

Далее необходимо отсортировать элементы массивов в порядке возрастания. Для удобства положим, что число элементов в массиве равно степени 2. Для сортировки элементов была выбрана **битоническая сортировка**, так как данный метод сортировки имеет большой потенциал к распараллеливанию и оптимален для использования на графических ускорителях. Свое название данный метод получил из-за того, что основан на понятии битонической последовательности — последовательности чисел, которая сначала возрастает, а затем убывает. Главная идея битонической сортировки — это приведение последовательности к виду битонической последовательности (то есть к такому виду, где первая половина последовательности возрастает, а вторая убывает) с последующим слиянием двух половин в возрастающую последовательность.

Имеются различные подходы к реализации алгоритма: рекурсивный и императивный. Рекурсивный подход основан на том, что исходная последовательность разбивается на половинки, где первая последовательность должна возрастать, а вторая убывать, с последующим их слиянием в монотонную последовательность (слияние представляет собой сравнение элементов из двух половинок и, если результат сравнения не удовлетворяет требуемому порядку, происходит перестановка элементов). К каждой из половинок применима та же логика, что и к исходной последовательности, отсюда и рекурсивность (ограничивается эта рекурсивность тем, что любая последовательность из двух элементов является битонической по определению). Более строго рекурсивный подход можно изложить с помощью псевдокода, который представлен в Листинге 2. Исходно вызывается `sortup(0,N)`.

Листинг 2: Псевдокод рекурсивной версии алгоритма битонической сортировки. Исходно вызывается `sortup(0,N)`.

```
void sortup(int m, int n) {//from m to m+n
    if (n==1) return;
    sortup(m,n/2);
    sortdown(m+n/2,n/2);
    mergeup(m,n/2);
}
void sortdown(int m, int n) {//from m to m+n
    if (n==1) return;
    sortup(m,n/2);
    sortdown(m+n/2,n/2);
```

```
    mergedown(m,n/2);
}
void mergeup(int m, int n) { //Sort in ascending order
    if (n==0) return;
    int i;
    for (i=0;i<n;i++) {
        if (get(m+i)>get(m+i+n)) exchange(m+i,m+i+n);
    }
    mergeup(m,n/2);
    mergeup(m+n,n/2);
}
void mergedown(int m, int n) {//Sort in descending order
    if (n==0) return;
    int i;
    for (i=0;i<n;i++) {
        if (get(m+i)<get(m+i+n)) exchange(m+i,m+i+n);
    }
    mergedown(m,n/2);
    mergedown(m+n,n/2);
}
```

Далее рассматривается так называемый императивный подход. Он представляет собой оптимизированный рекурсивный подход (см. `https://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm`). Императивную реализацию битонической сортировки можно разбить на несколько вложенных циклов: внешний или главный цикл по параметру k, пробегающему значения 2, 4, 8, 16 и так далее до `arraySize`/2; внутренний цикл по параметру j, который пробегает значения k/2, k/4, k/8 и так далее до 1; и самый глубоко вложенный цикл, который называется шагом битонической сортировки. В самом глубоко вложенном цикле (в каждом шаге битонической сортировки) проходит перебор пар элементов i, i + j и их перестановка в том случае, если они лежат не в том порядке. Правильный порядок при этом определяется с помощью k следующим образом: если побитовое "И" `k & i` даёт 0, то порядок должен быть возрастающим, иначе — убывающим. Последовательная реализация битонической сортировки представлена в Листинге 3. Реализация разбита на две части: на головную функцию `bitonicSort`, где происходит итерирование по двум вложенным циклам и вызывается функция шага битонической сортировки `bitonicSortStep`. Не трудно заметить, что каждая итерация самого глубоко вложенного цикла независима, что позволяет в дальнейшем оптимизировать алгоритм с помощью графического ускорителя.

Листинг 3: Последовательная версия битонической сортировки.

```
1 /** Kernel for the bitonic sort step
2  * @param a Sorted array
```

```
 * @param j Parameter of the minor loop of the bitonic sorting
 * @param k Parameter of the main loop of the bitonic sorting
 */
void bitonicSortStep(float* a, size_t j, size_t k) {
  // Loop over all elements
  for (size_t idx = 0; idx < arrayLength; idx++){
    // Current loop step works only with idx-th and (idx xor j)-th elements
    size_t idxXj = idx ^ j;
    // Threads that have idx < idxXj do the step of the bitonic sorting
    if ((idxXj) > idx) {
      bool order = (idx & k);
      // sort in ascending order on threads with order == 0
      if (order == 0) {
        if (a[idx] > a[idxXj]) {
          // Swap two elements
          float temp = a[idx];
          a[idx] = a[idxXj];
          a[idxXj] = temp;
        }
      }
      // sort in descending order on threads with order != 0
      if (order != 0) {
        if (a[idx] < a[idxXj]) {
          // Swap two elements
          float temp = a[idx];
          a[idx] = a[idxXj];
          a[idxXj] = temp;
        }
      }
    }
  }
}

/** Kernel for the bitonic sorting algorithm
 * @param hostArray Sorted array
 */
void bitonicSort(float* hostArray) {
    // Main loop of the sorting algorithm
    size_t j, k;
    for (k = 2; k <= arrayLength; k <<= 1) {
        // Minor loop of the sorting algorithm
        for (j = k >> 1; j > 0; j >>= 1) {
            bitonicSortStep(hostArray, j, k);
        }
    }
}
```

После сортировки элементов массива выполняется скалярное произведение двух массивов — вычисляется сумма произведений элементов двух массивов с одинаковыми

индексами. Последовательная реализация скалярного произведения представлена в Листинге 4.

Листинг 4: Последовательная версия скалярного произведения.

```
/** Computes dot product (CUDA is not used)
 *  @param hostA First array
 *  @param hostB Second array
 *  @param n Size of arrays
 */
float dotProd(float const *hostA, float const *hostB, int n){
  float res = 0;
  for (int i = 0; i < n; i++) {
    res += hostA[i] * hostB[i];
  }
  return res;
}
```

3 ОПТИМИЗАЦИЯ АЛГОРИТМА С ИСПОЛЬЗОВАНИЕМ CUDA

Как и последовательная реализация решения задачи, оптимизированная с помощью CUDA реализация начинается с инициализации массивов, которая проводится таким же самым образом, как и в последовательной версии (см. Листинг 1).

Далее необходимо отсортировать массивы по возрастанию с помощью оптимизированной с помощью CUDA битонической сортировки. Идея оптимизации алгоритма битонной сортировки напрашивается сама собой, ведь, как было показано выше, каждый шаг самого глубоко вложенного цикла императивной реализации битонической сортировки независим от прочих шагов, что позволяет производить их в параллельном режиме. Реализация оптимизированной с помощью CUDA императивной реализации битонической сортировки представлена с Листинге 5.

Листинг 5: Оптимизированная с помощью CUDA императивная реализация битонической сортировки.

```
/** Kernel for the bitonic sort step
 * @param a Sorted array
 * @param j Parameter of the minor loop of the bitonic sorting
 * @param k Parameter of the main loop of the bitonic sorting
 */
__global__ void bitonicSortStep(float* a, size_t j, size_t k) {
  // Thread linear id
  size_t idx = blockDim.x * blockIdx.x + threadIdx.x;
  // Current thread works only with idx-th and (idx xor j)-th elements
  size_t idxXj = idx ^ j;
  // Threads that have idx < idxXj do the step of the bitonic sorting
  if ((idxXj) > idx) {
    bool order = (idx & k);
    // sort in ascending order on threads with order == 0
```

```
15      if (order == 0) {
16        if (a[idx] > a[idxXj]) {
17          // Swap two elements
18          float temp = a[idx];
19          a[idx] = a[idxXj];
20          a[idxXj] = temp;
21        }
22      }
23      // sort in descending order on threads with order != 0
24      if (order != 0) {
25        if (a[idx] < a[idxXj]) {
26          // Swap two elements
27          float temp = a[idx];
28          a[idx] = a[idxXj];
29          a[idxXj] = temp;
30        }
31      }
32    }
33  }
34
35  /** Kernel for the bitonic sorting algorithm
36   * @param hostArray Sorted array
37   */
38  void bitonicSort(float* hostArray) {
39    // Allocate memory for the deviceArray
40    float* deviceArray;
41    cudaMalloc(&deviceArray, arrayLength * sizeof(float));
42    // Copy host vector to a device memory
43    cudaMemcpy(deviceArray, hostArray, arrayLength * sizeof(float),
        cudaMemcpyHostToDevice);
44    // Main loop of the sorting algorithm
45    size_t j, k;
46    for (k = 2; k <= arrayLength; k <<= 1) {
47      // Minor loop of the sorting algorithm
48      for (j = k >> 1; j > 0; j >>= 1) {
49        bitonicSortStep<<<GRID_SIZE_SORT, BLOCK_SIZE_SORT>>>(deviceArray, j, k);
50      }
51    }
52    cudaMemcpy(hostArray, deviceArray, arrayLength * sizeof(float),
        cudaMemcpyDeviceToHost);
53    cudaFree(deviceArray);
54  }
```

Заключительной частью задачи является вычисление скалярного произведения отсортированных массивов. Идея оптимизации с помощью CUDA алгоритма вычисления скалярного произведения была взята с сайта[1]. Вычисление может быть разбито на два этапа. На первом этапе вычисляются элементы массива **C** размера `gridDim`, содержащего

[1] `http://cuda-programming.blogspot.com/2013/01/vector-dot-product-in-cuda-c.html`

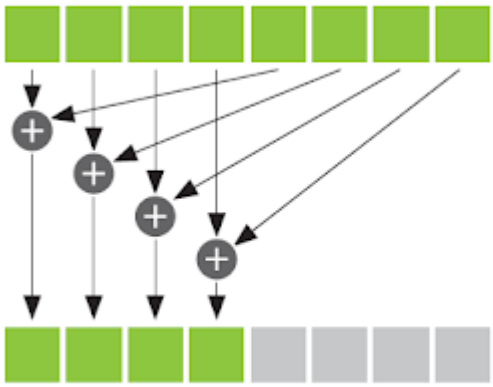

Figure 5.4 One step of a summation reduction

Рис. 1: Один шаг сложения элементов массива `cache[blockSize]`.

промежуточный результат вычисления скалярного произведения. В каждом блоке декларируется в разделяемой памяти массив `cache[blockSize]`. Элемент массива с индексом `idx` (с индексом, равным линейному индексу данной нити) инициализируется нулём, затем к нему прибавляют произведения $\mathbf{A}[i] \cdot \mathbf{B}[i]$ для всех i таких, что элементы массивов с индексом i, хранятся в памяти данной нити, то есть i = `idx`, `idx` + `blockDim.x * gridDim.x` и так далее. Когда в каждом из блоков посчитан массив `cache[blockSize]`, необходимо выполнить его суммирование. Для данной цели используется следующая схема, состоящая из нескольких шагов. Полагаем, что размер блока является степенью 2, тогда за первый шаг можно сложить в параллельном режиме две половинки массива (для определённости будем хранить результаты сложений в левой половинке), на следующем шаге мы уже складываем две половинки от первой половинки и так далее до тех пор, пока не дойдем до «половинки», состоящей из единственного элемента. Один шаг такой схемы суммирования можно проиллюстрировать Рис. 1. Результат суммирования элементов массива `cache[blockSize]` для данного блока нитей будет лежать в элементе массива `cache[0]`. Если сложить элементы `cache[0]` всех блоков, то мы получим ответ! Однако, последний этап вычисления скалярного произведения оптимальнее производить на хосте, поэтому сохраним промежуточный результат в массив $\mathbf{C}$ следующим образом: `C[blockIdx.x] = cache[0]` (чтобы избежать конфликтов запись–запись, такую операцию необходимо выполнять на одной нити с одного блока).

Оптимизированный с помощью CUDA код может быть найден в Листинге 6, где функции с именами, имеющими на конце `CUDA` или `Kernel` относятся к оптимизированной версии алгоритма.

Листинг 6: Оптимизированная с помощью CUDA версия вычисления скалярного произведения.

```
/** Computes intermediate result of dot product
 *  @param blockSize Size of a CUDA-block
 *  @param deviceA First vector
 *  @param deviceB Second vector
```

```
 *  @param deviceC Intermediate result
 *  @param n Size of the first and the second vectors
 */
template<int blockSize>
__global__ void dotProdKernel(float const *deviceA, float const *deviceB, float
    *deviceC, int n){
  // shared array for cache (this is a shared array for the whole threads in the
     current block)
  __shared__ float cache[blockSize];
  // Get our thread linear ID
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  float sum = 0;
  // Computation of sum of all elements that lie on this thread
  while (idx < n) {
    sum += deviceA[idx] * deviceB[idx];
    idx += blockDim.x * gridDim.x;
  }
  cache[threadIdx.x] = sum; // Filling cache
  __syncthreads(); // Barier synchronisation
  // Summation the cache (on the current block)
  int i = blockDim.x / 2;
  while (i != 0){
    if (threadIdx.x < i) cache[threadIdx.x] += cache[threadIdx.x + i];
    __syncthreads(); // Barier synchronisation
    i /= 2;
  }
  // Sum of the cache on the current block is at cache[0]
  // Saving this result for each blocks
  if (threadIdx.x == 0) deviceC[blockIdx.x] = cache[0];
}

/** Computes dot product (CUDA is used)
 *  @param hostA First array
 *  @param hostB Second array
 *  @param n Size of arrays
 */
float dotProdCUDA(float const *hostA, float const *hostB, int n){
  float *hostC; // declaration of vectors
  float *deviceA, *deviceB, *deviceC;
  // Number of thread blocks per grid
  int gridDimDotProd = (n + blockSizeDotProd - 1) / blockSizeDotProd;
  // Size in bytes of the vector C
  size_t bytesC = gridDimDotProd * sizeof(float);
  // Vectors allocation
  hostC = (float*)malloc(bytesC);
  cudaMalloc(&deviceA, n * sizeof(float));
  cudaMalloc(&deviceB, n * sizeof(float));
  cudaMalloc(&deviceC, bytesC);
  // Copy host vectors to device
```

```
  cudaMemcpy(deviceA, hostA, n * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(deviceB, hostB, n * sizeof(float), cudaMemcpyHostToDevice);
  // Execute the kernel
  dotProdKernel<blockSizeDotProd><<<gridDimDotProd, blockSizeDotProd>>>(deviceA,
     deviceB, deviceC, n);
  // Copy array back to host
  cudaMemcpy(hostC, deviceC, bytesC, cudaMemcpyDeviceToHost);
  // Release device memory
  cudaFree(deviceA);
  cudaFree(deviceB);
  cudaFree(deviceC);
  // Finish up on the host
  float res = 0;
  for (int i = 0; i < gridDimDotProd; i++){
    res += hostC[i];
  }
  return res;
}
```

Следует заметить, что в оптимизированной с помощью CUDA версии вычисления скалярного произведения имеется один свободный параметр — размер блока нитей `blockSizeDotProd`.

4 ВЕРИФИКАЦИЯ РЕАЛИЗАЦИЙ

Для проверки верности работы реализаций решений задачи был проведен пробный запуск с малым размером массивов **A** и **B**. Результат такого запуска приведён на Рис. 2. Легко проверить, что реализации работают правильно.

5 ИЗМЕРЕНИЕ ВРЕМЕНИ И РЕЗУЛЬТАТЫ

Время работы программы измерялась с помощью функции `clock()`. Гиперпараметр `BLOCK_SIZE_DOTPROD` не влиет на время исполнения и полагается равным 1024, были рассмотрены размеры массивов 1024, 4096, 16384, 65536, 262144, 1048576, 4194304. Важно, чтобы число CUDA-потоков всегда было меньше либо равно числу элементов массивов, поэтому рассматривались `GRID_SIZE_SORT` = 4096 и `BLOCK_SIZE_SORT` = 1024. Результаты измерений времени при различной вычислительной сложности задачи приведены на правой панели Рис. 3. Чтобы оценить эффект от оптимизации алгоритма, было рассчитано ускорение

$$\texttt{speedup}_n = \frac{(t_{\text{serial}})_n}{(t_{\text{CUDA}})_n}, \tag{1}$$

где n — размер массива, и его погрешность

$$\texttt{speedup error}_n = \frac{\texttt{std}\{(t_{\text{serial}})_n\}(t_{\text{CUDA}})_n + (t_{\text{serial}})_n \texttt{std}\{(t_{\text{CUDA}})_n\}}{(t_{\text{CUDA}})_n^2}. \tag{2}$$

Рассчитанное ускорение изображено на левой панели Рис. 3. Видно, что при малых n быст-

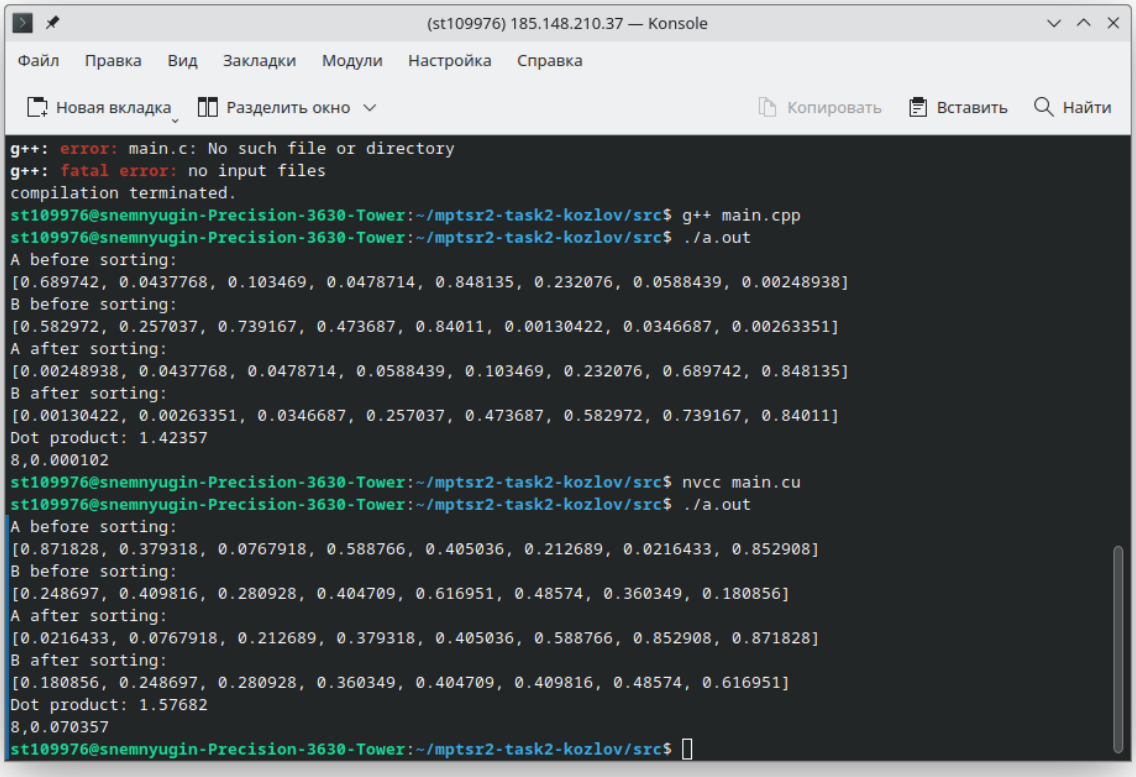

Рис. 2: Результат запуска последовательной и оптимизированной с помощью CUDA реализаций алгоритма.

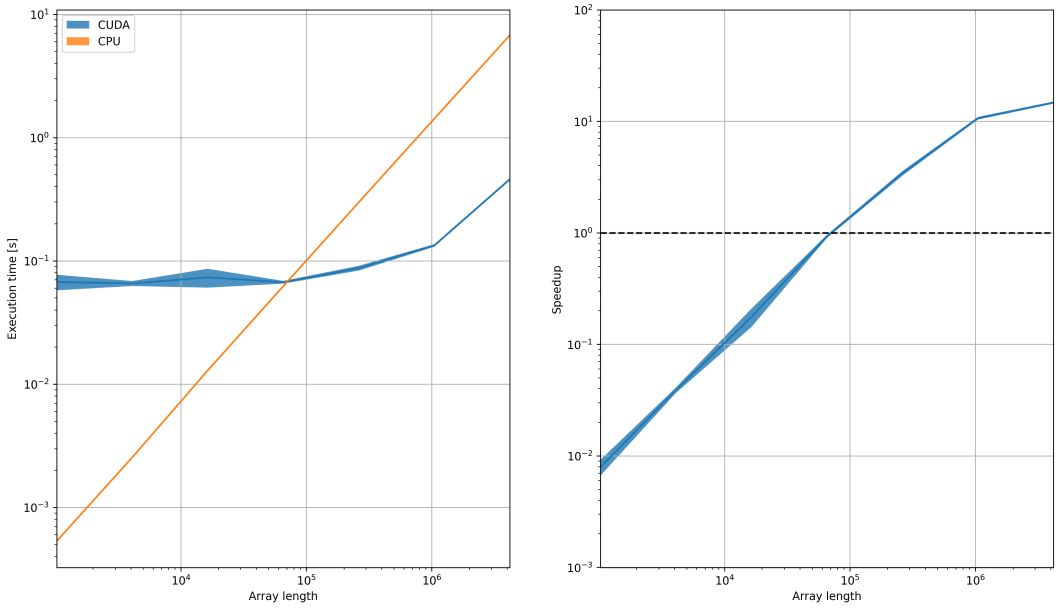

Рис. 3: На правой панели изображена зависимость времени исполнения от размера векторов. На левой панели представлена зависимость ускорения от вычислительной сложности задачи.

рее оказывается последовательная версия, но при `arraySize` $>$ 262144 более эффективной оказывается алгоритм, оптимизированный с помощью CUDA. Максимальное значение ускорения достигается при `arraySize` $= 4194304$ и составляет 14.73 ± 0.33.

6 ИНТЕРПРЕТАЦИЯ РЕЗУЛЬТАТОВ, ВЫВОДЫ

В данной работе рассматривается задача, состоящая из двух частей. На первом шаге необходимо отсортировать 2 массива по убыванию, на втором — вычислить их скалярное произведение. В качестве алгоритма сортировки был выбран **алгоритм битонической сортировки**. Были реализованы две версии решения задачи — последовательная и оптимизированная с помощью CUDA, кроме того была проведено измерение времени работы реализаций при различной вычислительной трудоёмкости. В результате чего было установлено, что при малой вычислительной трудоёмкости (при `arraySize` $<$ 262144) быстрее оказывается последовательная версия. При большой вычислительной сложности задачи (при `arraySize` $>$ 262144) время работы оптимизированной версии становится меньше времени исполнения последовательного алгоритма, что можно связать с тем, что при увеличения вычислительной сложности все меньшую часть времени исполнения занимают транзакции между GPGPU и CPU и всё большая часть времени работы уходит под параллельную часть программы. Максимальное значение ускорения достигается при `arraySize` $= 4194304$ и составляет 14.73 ± 0.33.

ПРИЛОЖЕНИЕ

Длина массивов	Время исполнения [c]				
	Эксперимент 1	Эксперимент 2	Эксперимент 3	Эксперимент 4	Эксперимент 5
1024	0.000535	0.000521	0.000547	0.000534	0.000531
4096	0.002560	0.002563	0.002555	0.002559	0.002504
16384	0.013074	0.012768	0.012906	0.013138	0.012777
65536	0.062862	0.062067	0.061808	0.062108	0.062848
262144	0.298426	0.298829	0.298501	0.298396	0.300047
1048576	1.414440	1.428220	1.426100	1.424030	1.422680
4194304	6.763430	6.773680	6.769610	6.769310	6.767140

Таблица 2: Время работы последовательной реализации решения задачи при различной вычислительной сложности задачи.

Длина массивов	Время исполнения [c]				
	Эксперимент 1	Эксперимент 2	Эксперимент 3	Эксперимент 4	Эксперимент 5
1024	0.084370	0.063336	0.063639	0.065823	0.059983
4096	0.067229	0.068427	0.064370	0.067249	0.060883
16384	0.062902	0.094018	0.071070	0.063067	0.076440
65536	0.067157	0.068036	0.069291	0.066234	0.064913
262144	0.086871	0.088072	0.083595	0.093580	0.084718
1048576	0.130118	0.136219	0.136118	0.134351	0.129931
4194304	0.476043	0.460958	0.456771	0.452885	0.450355

Таблица 3: Время работы оптимизированной с помощью CUDA реализации решения задачи при различной вычислительной сложности задачи.