Student Name: Галактионов Кирилл
Student ID: st067889

Современные технологии программирования
в научных исследованиях II
Лабораторная работа 2

---

1 Постановка задачи

Условие задачи 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 сортировки 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 алгоритма сортировки мною были исследованы алгоритмы сортировки пузырьком и быстрая сортировка.

3.1 Сортировка пузырьком

В последовательной реализации алгоритм заключается в проходе по массиву и попарному сравнению пар чисел, и перемене их местами, при необходимости. Вычислительная сложность: $O(n^2)$ Код:

```
// Bubble sort on CPU in one thread
void BubbleSortNoCUDA(double *array, int N){
    // Repeat N times
    for (int i = 0; i < N; i++){
        // Move "bubble" and swap elements if needed
        for (int j = 0; j < N-i-1; j++) {
            if (array[j]<array[j + 1]){
                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 с шагом в два элемента между соседними пузырьками. Реализация кода нити и алгоритма сортировки представлена ниже:

```
// One step for bubble sort with CUDA
__global__ void BubbleMove(double *array, int N, int step){
    // Get thread ID
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    // Check that thread belongs to array
    if (idx < (N-1)) {
        // Check that "bubble" moved enough for this thread to run and that there are no
            intersections
        if (step-2 >= idx  && (idx - step) % 2 == 0){
            // Swap elements if need
            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){
    // Create, allocate memory and copy array to device (GPU)
    double *array_device;
    cudaMalloc((void **)&array_device, N * sizeof(double));
    cudaMemcpy(array_device, array_host, N*sizeof(double), cudaMemcpyHostToDevice);
    // Calculate needed number of blocks according to block size
    int nblocks = N / blockSize + 1;
    // N+N steps for all needed changes sure to be made
    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();
    }
    // Copy array from device to host
    cudaMemcpy(array_host, array_device, N*sizeof(double), cudaMemcpyDeviceToHost);
    cudaFree(array_device);
}
```

3.2 Быстрая сортировка

Алгоритм быстрой сортировки, это рекурсивный алгоритм, который можно описать следующим образом. На первом шаге выбирается опорный элемент, для которого ищется его истинное место, и все элементы больше него перемещаются на одну сторону от него, элементы меньше - на другую. На следующем шаге выполняется алгоритм сортировки отдельно для левой и правой частей массива. При достижении определенной глубины деления, или достижении минимального размера части выполняется сортировка другими алгоритмами (например сортировка выбором). Вычислительная сложность: $O(n\log n)$. Код последовательной реализации:

```
// Making partition in vector for quick sort algorithm
int MakePartition(double* vec, int start, int end){
  // Taking first element as pivot point
  double pivot = vec[start];
  // Finding corret position of pivot element
  int count = 0;
  for (int i = start + 1; i <= end; i++) {
    if (vec[i] >= pivot)
      count++;
  }
  // Giving pivot element its correct position
  int pivotIndex = start + count;
  swap(vec[pivotIndex], vec[start]);
  // Now pivot element is on its true position
  // and we need to place elements greater than pivot on the left and less on the right
  int i = start;
  int j = end;
  // Number of missplaced elements is even, so we will use pair swaps
  while (i < pivotIndex && j > pivotIndex) {
        // looking for missplaced elements on the left of the pivot
    while (vec[i] >= pivot) {
      i++;
    }
        // looking for missplaced elements on the right of the pivot
    while (vec[j] < pivot) {
      j--;
    }
        // Swap pair of missplaced elements
    if (i < pivotIndex && j > pivotIndex) {
      swap(vec[i++], vec[j--]);
    }
  }
  return pivotIndex;
}

```

```
// QuickSort on CPU, main sorting algorithm
void QuickSortCPU(double* vec, int start, int end){
  // base of the recursion
  if (start >= end)
    return;

  // partitioning the array
  int p = MakePartition(vec, start, end);

  // Sorting the left part
  QuickSortCPU(vec, start, p - 1);

  // Sorting the right part
  QuickSortCPU(vec, p + 1, end);
}
```

Идея использования нитей CUDA в такой сортировке заключается в том, что левую и правую часть массива можно сортировать одновременно, так как они независимы друг от друга, и эту одновременную сортировку могут выполнять разные нити. Реализация с использованием CUDA (компиляция с добавлением ключа -rdc=true):

```
////////////////////////////////////////////////////////////////////////////////
// Very basic quicksort algorithm, recursively launching the next level.
////////////////////////////////////////////////////////////////////////////////
__global__ void cdp_simple_quicksort(double* vec, int start, int end, int depth) {
    // if max depth is reached or array is too small launch selection sort
    if (depth >= MAX_DEPTH || end - start <= INSERTION_SORT) {
        selection_sort(vec, start, end);
        return;
    }

    double helper;

    // Make partitioning (like in serial mode)
    // taking first elemt as pivot point
    double pivot = vec[start];
  // Finding corret position of pivot element
  int count = 0;
  for (int i = start + 1; i <= end; i++) {
    if (vec[i] >= pivot)
      count++;
  }
  // Giving pivot element its correct position
  int pivotIndex = start + count;
  //swap(vec[pivotIndex], vec[start]);
    helper = vec[pivotIndex];
    vec[pivotIndex] = vec[start];
    vec[start] = helper;

  // Now pivot element is on its true position
  // and we need to place elements greater than pivot on the left and less on the right
  int i = start;
  int j = end;

```

```
  // Number of missplaced elements is even, so we will use pair swaps
  while (i < pivotIndex && j > pivotIndex) {
        // looking for missplaced elements on the left of the pivot
    while (vec[i] >= pivot) {
      i++;
    }
        // looking for missplaced elements on the right of the pivot
    while (vec[j] < pivot) {
      j--;
    }
        // Swap pair of missplaced elements
    if (i < pivotIndex && j > pivotIndex) {
      //swap(vec[i++], vec[j--]);
            helper = vec[i];
            vec[i] = vec[j];
            vec[j] = helper;
            i++;
            j--;
    }
  }

    // Launch a new block to sort the left part.
    cudaStream_t s;
    cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
    cdp_simple_quicksort<<<1, 1, 0, s>>>(vec, start, pivotIndex - 1, depth + 1);
    cudaStreamDestroy(s);


    // Launch a new block to sort the right part.
    cudaStream_t s1;
    cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
    cdp_simple_quicksort<<<1, 1, 0, s1>>>(vec, pivotIndex + 1, end, depth + 1);
    cudaStreamDestroy(s1);
}
////////////////////////////////////////////////////////////////////////////////
// Call the quicksort kernel from the host.
////////////////////////////////////////////////////////////////////////////////
void QuickSortCUDA(double* vec, unsigned int nitems) {
    // Create, allocate memory and copy array to device (GPU)
    double *array_device;
    cudaMalloc((void **)&array_device, nitems * sizeof(double));
    cudaMemcpy(array_device, vec, nitems*sizeof(double), cudaMemcpyHostToDevice);
    // Launch on device
    int left = 0;
    int right = nitems - 1;
    // std::cout << "Launching kernel on the GPU" << std::endl;
    cdp_simple_quicksort<<<1, 1>>>(array_device, left, right, 0);
    cudaDeviceSynchronize();
    // Copy array from device to host
    cudaMemcpy(vec, array_device, nitems*sizeof(double), cudaMemcpyDeviceToHost);
    cudaFree(array_device);
}
```

3.3 Скалярное произведение

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:

```
// Thread code for dot product on CUDA
__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 elements and write into shared array
    helper[threadIdx.x] = deviceLHS[idx] * deviceRHS[idx];

    __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];
    }
}

// Dot product of two arrays on GPU CUDA
double CalculateDotProductCuda(double *hostLHS, double *hostRHS, int N){
    double *hostInterRes; // helper for intermediate results
    double *deviceLHS, *deviceRHS, *deviceRES;
    // Thread blocks per Grid
    int gridDim = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
    // Create vectors
    size_t interResSize = gridDim * sizeof(double);
    hostInterRes = (double*)malloc(interResSize);
    cudaMalloc(&deviceLHS, N * sizeof(double));
    cudaMalloc(&deviceRHS, N * sizeof(double));
    cudaMalloc(&deviceRES, interResSize);
    // Arrays to gpu
    cudaMemcpy(deviceLHS, hostLHS, N * sizeof(double), cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(deviceRHS, hostRHS, N * sizeof(double), cudaMemcpyHostToDevice);

    DotProduct<<<gridDim, THREADS_PER_BLOCK>>>(deviceLHS, deviceRHS, deviceRES, N);

    // 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 9.

5 Результаты измерений

Вывод программы представлен в Приложении 2 в разделе 10 для сортировки пузырьком и в Приложении 3 в разделе 11 для быстрой сортировки.

6 Анализ и обсуждение результатов

Рассмотрим результаты сортировки пузырьком. На рисунке 1 представлена зависимость времени, затраченного на вычисления, от размера массива. Можно заметить, что в логарифмическом мастштабе 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.

Для быстрой сортировки аналогичные зависимости построены на Рис. 3 и 4. Однако, можно заметить, что результаты вычислений с использованием GPU хуже, чем на CPU. Можно предположить, что, так как данный алгоритм имеет меньшую вычислительную сложность, чем сортировка пузырьком, то на данных размерах массива выигрыш в количестве вычислительных потоков не покрывает расходы на пересылку данных.

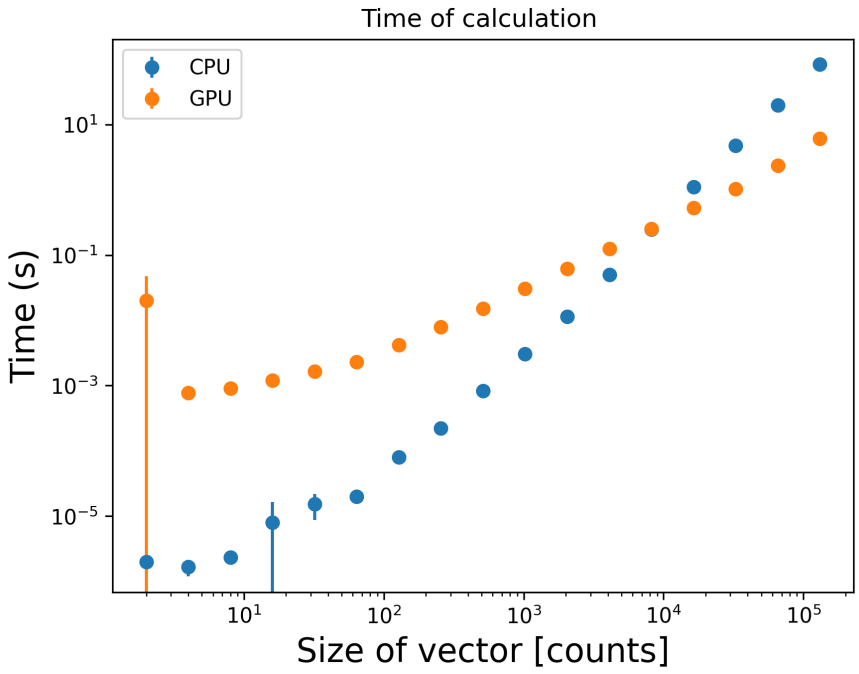

Рис. 1: Зависимость времени вычисления от размера массива. Сортировка пузырьком.

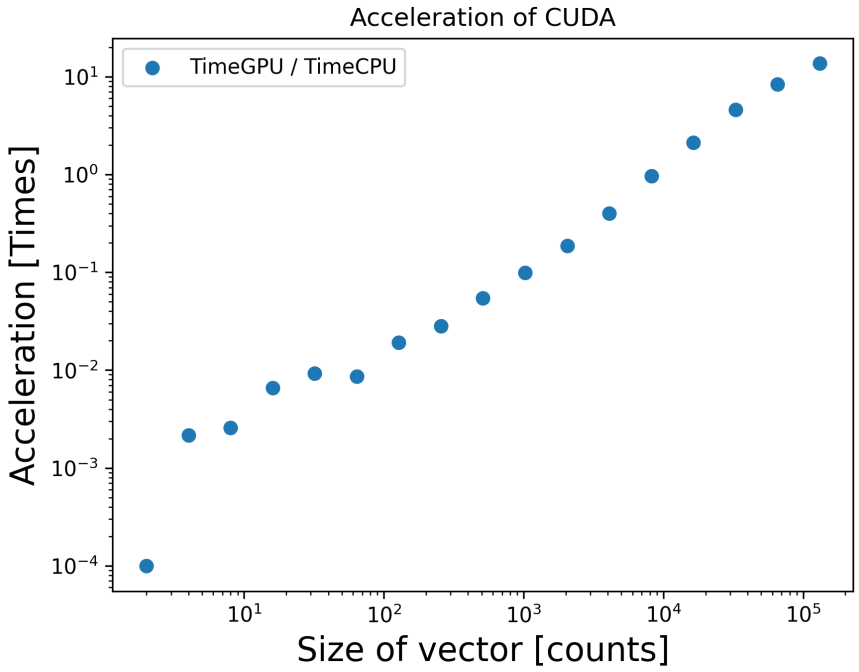

Рис. 2: Зависимость ускорения вычисления от размера массива. Сортировка пузырьком.

7 Nvidia Visual Profiler

На примере вычисления с использованием сортировкой пузырьком также был исследован инструмент визуальной профилировки программы NVVP (Nvidia Visual Profiler). Запускается из

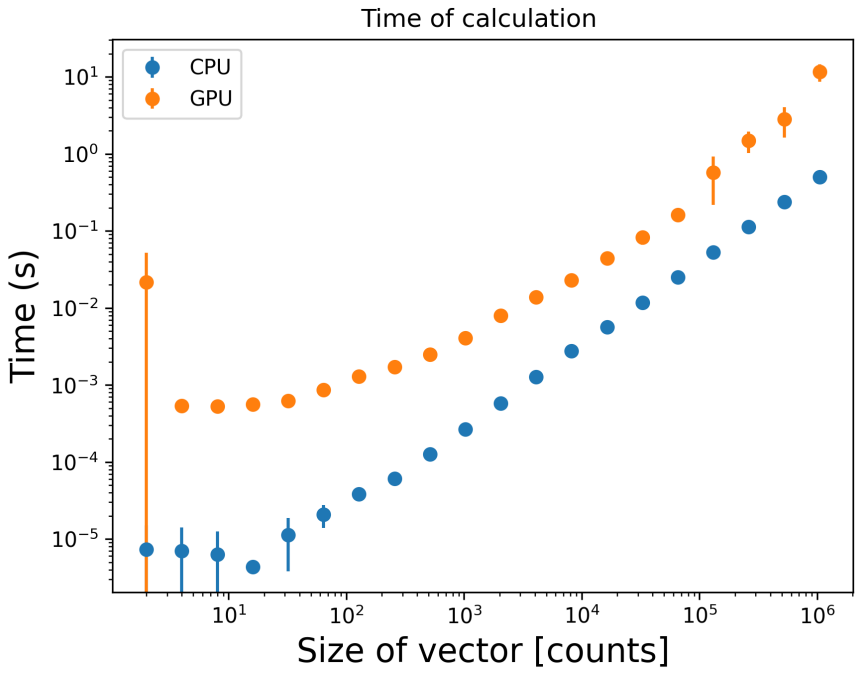

Рис. 3: Зависимость времени вычисления от размера массива. Быстрая сортировка.

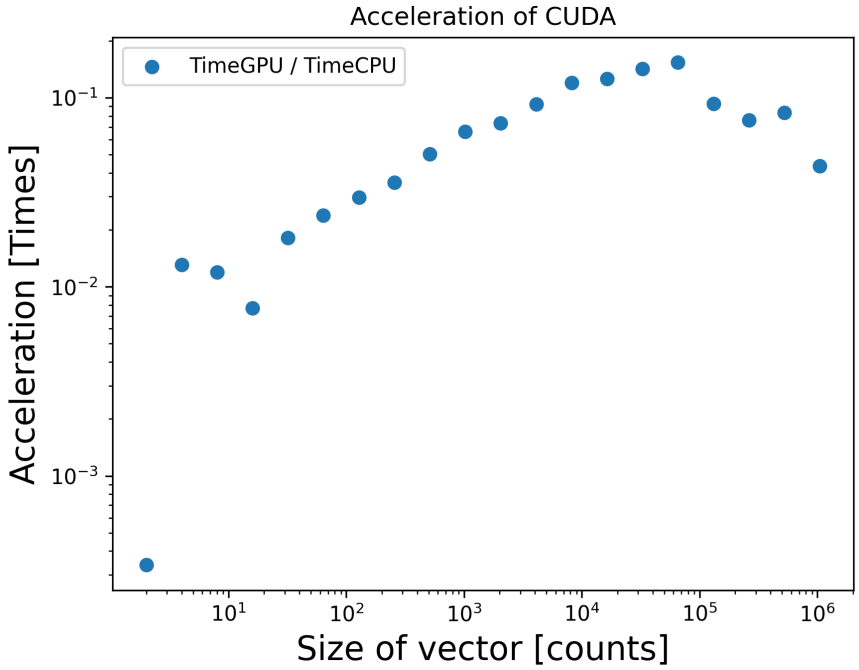

Рис. 4: Зависимость ускорения вычисления от размера массива. Быстрая сортировка.

терминала, командой nvvp, затем создается сессия с указанием профилируемой программы, и проводится её анализ. Данный инструмент позволяет строить таймлайн исполнения программы и анализировать затраты времени на разные части программы. Как можно заметить по

рисунку 5, в моей программе практически всё время идет исполнение сортировки пузырьком, а скалярное произведение занимает лишь малую долю от общего времени.

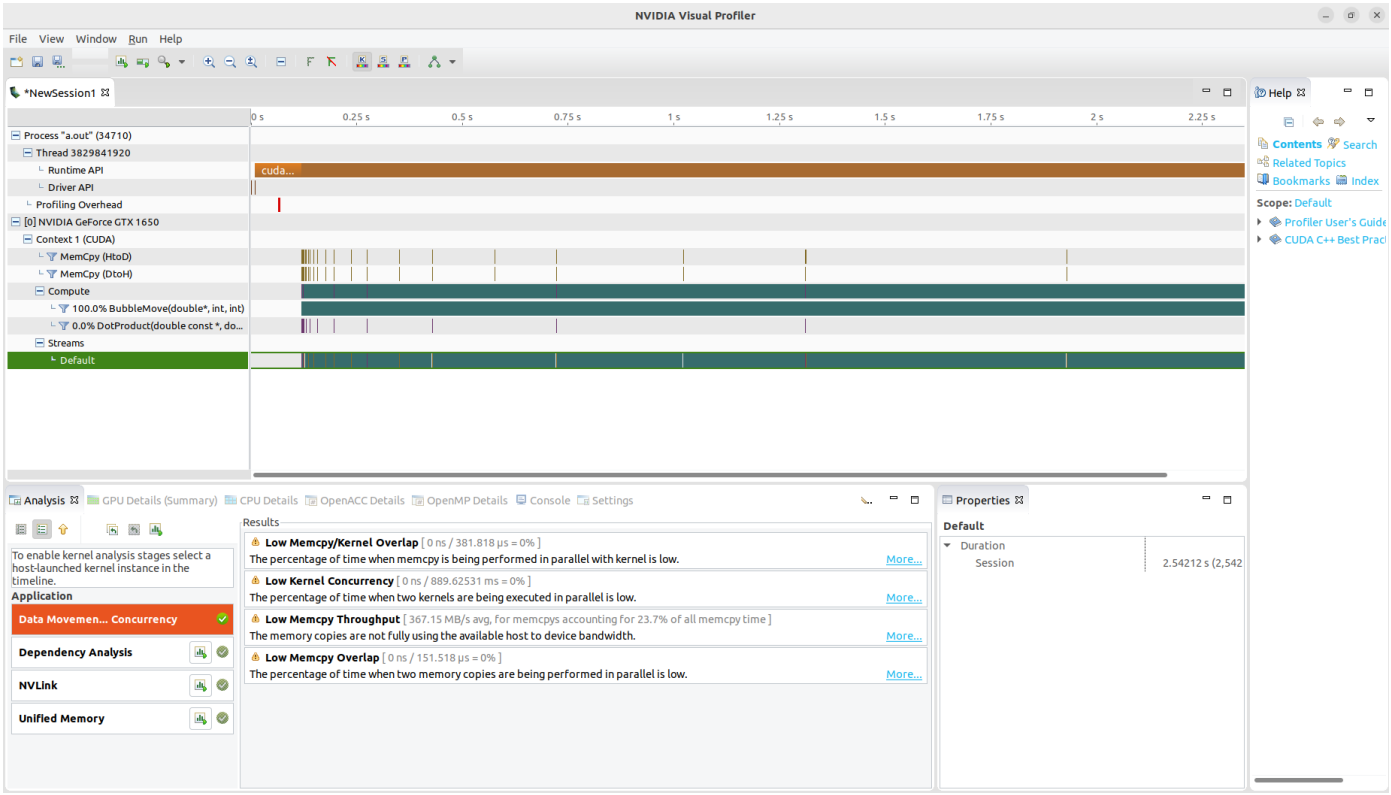

Рис. 5: Результаты построения таймлайна выполнения программы в NVVP.

8 Выводы

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

9 Приложение 1. Код программы

```
#include <iostream>
#include <stdio.h>
#include <time.h>
#include <iomanip>
#define THREADS_PER_BLOCK 1024
#define MAX_DEPTH 16            // max depth for quick sort layers
#define INSERTION_SORT 32       // smaller than these - selection sort instead of quick

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);

__global__ void cdp_simple_quicksort(double* vec, int start, int end, int depth);

__device__ void selection_sort(double* data, int start, int end);

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 MakePartition(double* vec, int start, int end);

void QuickSortCPU(double* vec, int start, int end);

void QuickSortCUDA(double* vec, unsigned int nitems);


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 < 1000000; n*=2 ){
        cout<< setw(6) <<  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 = 2; n < 1000000; n*=2 ){
        cout<< setw(6) <<  n << "    ";
        for (int i = 0; i < 3; i++){
            measureCUDA(n);
            cout << "    ";
        }
        cout << endl;
    }
    cout << endl;
    return 0;
}

// One step for bubble sort with CUDA
__global__ void BubbleMove(double *array, int N, int step){
    // Get thread ID
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    // Check that thread belongs to array
    if (idx < (N-1)) {
        // Check that "bubble" moved enough for this thread to run and that there are no
            intersections
        if (step-2 >= idx  && (idx - step) % 2 == 0){
            // Swap elements if need
            if (array[idx] < array[idx + 1]){
                double helper = array[idx];
                array[idx] = array[idx + 1];
                array[idx + 1] = helper;
            }
        }
    }
}

// Thread code for dot product on CUDA
__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 elements and write into shared array
    helper[threadIdx.x] = deviceLHS[idx] * deviceRHS[idx];

    __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
void BubbleSortCUDA(double *array_host, int N, int blockSize){
    // Create, allocate memory and copy array to device (GPU)
    double *array_device;
    cudaMalloc((void **)&array_device, N * sizeof(double));
    cudaMemcpy(array_device, array_host, N*sizeof(double), cudaMemcpyHostToDevice);
    // Calculate needed number of blocks according to block size
    int nblocks = N / blockSize + 1;
    // N+N steps for all needed changes sure to be made
    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();
    }
    // Copy array from device to host
    cudaMemcpy(array_host, array_device, N*sizeof(double), cudaMemcpyDeviceToHost);
    cudaFree(array_device);
}

// Bubble sort on CPU in one thread
void BubbleSortNoCUDA(double *array, int N){
    // Repeat N times
    for (int i = 0; i < N; i++){
        // Move "bubble" and swap elements if needed
        for (int j = 0; j < N-i-1; j++) {
            if (array[j]<array[j + 1]){
                double helper = array[j];
                array[j] = array[j + 1];
                array[j + 1] = helper;
            }
        }
    }
}

```

```
157 // check if and array is sorted
158 bool IsSorted(double *array, int N){
159     // Iterate over all elemtnts
160     for (int i = 0; i < N-1; i++){
161         // ... and make sure that they are sorted properly
162         if (array[i] < array[i+1]) {
163             return false;
164         }
165     }
166     return true;
167 }
168 
169 // measure time taken for all steps on CPU CUDA
170 void measureNoCUDA(int N) {
171     // Create arrays
172     double *array1 = (double *)malloc(N * sizeof(double));
173     double *array2 = (double *)malloc(N * sizeof(double));
174     // Create variables for saving time results
175     clock_t start_time, end_time;
176     float timeArray1Sorted, timeArray2Sorted, timeDotProduct;
177     double resCPU = 0.0;
178     // Fill arrays with random numbers
179     for (int i = 0; i < N; i++) {
180         array1[i] = static_cast <double> (rand()) / static_cast <double> (RAND_MAX);
181         array2[i] = static_cast <double> (rand()) / static_cast <double> (RAND_MAX);
182     }
183 
184     // Sort first array
185     start_time = clock();
186     BubbleSortNoCUDA(array1, N);
187     end_time = clock();
188 
189     // Time taken
190     timeArray1Sorted = (float)(end_time - start_time) / CLOCKS_PER_SEC;
191 
192     // Check that array is sorted
193     if (IsSorted(array1, N)){
194         //cout << timeArray1Sorted << "   ";
195     }
196     else {
197         free(array1);
198         free(array2);
199         cout << "ERROR, ARRAY 1 NOT SORTED" << endl;
200         return;
201     }
202 
203     // Sort second array
204     start_time = clock();
205     BubbleSortNoCUDA(array2, N);
206     end_time = clock();
207 
208     // Time taken
209     timeArray2Sorted = (float)(end_time - start_time) / CLOCKS_PER_SEC;
210 
```

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

    // Calculate dot product of sorted arrays
    start_time = clock();
    resCPU = CalculateDotProductNoCuda(array1, array2, N);
    end_time = clock();

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

    // Free memory
    free(array1);
    free(array2);
    // Print time info
    cout << setw(10) << timeArray1Sorted + timeArray2Sorted + timeDotProduct;
}

// measure time taken for all steps on GPU CUDA
void measureCUDA(int N){
    // Create arrays
    double *array1 = (double *)malloc(N * sizeof(double));
    double *array2 = (double *)malloc(N * sizeof(double));
    // Create variables for saving time results
    clock_t start_time, end_time;
    float timeArray1Sorted, timeArray2Sorted, timeDotProduct;
    // Create variables for saving results of all calculations
    double resGPU = 0.0;
    double resCPU = 0.0;
    // Fill arrays with random numbers
    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);
    }

    // Sort first array
    start_time = clock();
    BubbleSortCUDA(array1, N, THREADS_PER_BLOCK);
    // QuickSortCUDA(array1, N);
    end_time = clock();

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

    // Check that array is sorted
    if (IsSorted(array1, N)){
```

```
265         //cout << timeArray1Sorted << "   ";
266     }
267     else {
268         free(array1);
269         free(array2);
270         cout << "ERROR, ARRAY 1 NOT SORTED" << endl;
271         return;
272     }
273 
274     // Sort second array
275     start_time = clock();
276     BubbleSortCUDA(array2, N, THREADS_PER_BLOCK);
277     // QuickSortCUDA(array2, N);
278     end_time = clock();
279 
280     // Time taken
281     timeArray2Sorted = (float)(end_time - start_time) / CLOCKS_PER_SEC;
282 
283     // Check that array is sorted
284     if (IsSorted(array2, N)){
285         //cout << timeArray2Sorted << "   ";
286     }
287     else {
288         free(array1);
289         free(array2);
290         cout << "ERROR, ARRAY 2 NOT SORTED" << endl;
291         return;
292     }
293 
294 
295     // Calculate dot product of sorted arrays
296     start_time = clock();
297     resGPU = CalculateDotProductCuda(array1, array2, N);
298     end_time = clock();
299 
300     // Time taken
301     timeDotProduct = (float)(end_time - start_time) / CLOCKS_PER_SEC;
302 
303     // Compare to the CPU result
304     resCPU = CalculateDotProductNoCuda(array1, array2, N);
305     if (abs(resCPU - resGPU) < 1e-2){
306         //cout << timeDotProduct << "   ";
307     }
308     else {
309         free(array1);
310         free(array2);
311         cout << "ERROR, DOT PRODUCT DOESN'T MATCH CPU" << endl;
312         return;
313     }
314 
315     // Free memory
316     free(array1);
317     free(array2);
318     // Print time info
```

```
    cout << setw(10) << timeArray1Sorted + timeArray2Sorted + timeDotProduct;
}

// Print double array into console
void PrintArray(const double* array, const int n){
    for(int i = 0; i < n; i++){
        cout << array[i] <<  ' ';
    }
    cout << endl;
}

// Dot product of two arrays on CPU
double CalculateDotProductNoCuda(double *lhs, double *rhs, int N){
    double result = 0.0;
    for (int i = 0; i < N; i++){
        result += lhs[i] * rhs[i];
    }
    return result;
}

// Dot product of two arrays on GPU CUDA
double CalculateDotProductCuda(double *hostLHS, double *hostRHS, int N){

    double *hostInterRes; // helper for intermediate results
    double *deviceLHS, *deviceRHS, *deviceRES;

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

    // Create vectors
    size_t interResSize = gridDim * sizeof(double);
    hostInterRes = (double*)malloc(interResSize);
    cudaMalloc(&deviceLHS, N * sizeof(double));
    cudaMalloc(&deviceRHS, N * sizeof(double));
    cudaMalloc(&deviceRES, interResSize);

    // Arrays to gpu
    cudaMemcpy(deviceLHS, hostLHS, N * sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(deviceRHS, hostRHS, N * sizeof(double), cudaMemcpyHostToDevice);


    DotProduct<<<gridDim, THREADS_PER_BLOCK>>>(deviceLHS, deviceRHS, deviceRES, N);

    // 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;
}

// Selection sort for final recursion steps of quick sort
// only for threads
__device__ void selection_sort(double* data, int start, int end) {
    for (int i = start; i <= end; ++i) {
        double max_val = data[i];
        int max_idx = i;

        // Find the biggest value in the range [start, end].
        for (int j = i + 1; j <= end; ++j) {
            double val_j = data[j];

            if (val_j > max_val) {
                max_idx = j;
                max_val = val_j;
            }
        }

        // Swap the values.
        if (i != max_idx) {
            data[max_idx] = data[i];
            data[i] = max_val;
        }
    }
}


////////////////////////////////////////////////////////////////////////////////
// Very basic quicksort algorithm, recursively launching the next level.
////////////////////////////////////////////////////////////////////////////////
__global__ void cdp_simple_quicksort(double* vec, int start, int end, int depth) {
    // if max depth is reached or array is too small launch selection sort
    if (depth >= MAX_DEPTH || end - start <= INSERTION_SORT) {
        selection_sort(vec, start, end);
        return;
    }

    double helper;

    // Make partitioning (like in serial mode)
    // taking first elemt as pivot point
    double pivot = vec[start];
  // Finding corret position of pivot element
  int count = 0;
  for (int i = start + 1; i <= end; i++) {
    if (vec[i] >= pivot)
      count++;
  }
  // Giving pivot element its correct position
```

```
  int pivotIndex = start + count;
  //swap(vec[pivotIndex], vec[start]);
    helper = vec[pivotIndex];
    vec[pivotIndex] = vec[start];
    vec[start] = helper;

  // Now pivot element is on its true position
  // and we need to place elements greater than pivot on the left and less on the right
  int i = start;
  int j = end;

  // Number of missplaced elements is even, so we will use pair swaps
  while (i < pivotIndex && j > pivotIndex) {
        // looking for missplaced elements on the left of the pivot
    while (vec[i] >= pivot) {
      i++;
    }
        // looking for missplaced elements on the right of the pivot
    while (vec[j] < pivot) {
      j--;
    }
        // Swap pair of missplaced elements
    if (i < pivotIndex && j > pivotIndex) {
      //swap(vec[i++], vec[j--]);
            helper = vec[i];
            vec[i] = vec[j];
            vec[j] = helper;
            i++;
            j--;
    }
  }

    // Launch a new block to sort the left part.
    cudaStream_t s;
    cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
    cdp_simple_quicksort<<<1, 1, 0, s>>>(vec, start, pivotIndex - 1, depth + 1);
    cudaStreamDestroy(s);


    // Launch a new block to sort the right part.
    cudaStream_t s1;
    cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
    cdp_simple_quicksort<<<1, 1, 0, s1>>>(vec, pivotIndex + 1, end, depth + 1);
    cudaStreamDestroy(s1);
}

////////////////////////////////////////////////////////////////////////////////
// Call the quicksort kernel from the host.
////////////////////////////////////////////////////////////////////////////////
void QuickSortCUDA(double* vec, unsigned int nitems) {
    // Create, allocate memory and copy array to device (GPU)
    double *array_device;
    cudaMalloc((void **)&array_device, nitems * sizeof(double));
    cudaMemcpy(array_device, vec, nitems*sizeof(double), cudaMemcpyHostToDevice);
```

```
    // Launch on device
    int left = 0;
    int right = nitems - 1;
    // std::cout << "Launching kernel on the GPU" << std::endl;
    cdp_simple_quicksort<<<1, 1>>>(array_device, left, right, 0);
    cudaDeviceSynchronize();
    // Copy array from device to host
    cudaMemcpy(vec, array_device, nitems*sizeof(double), cudaMemcpyDeviceToHost);
    cudaFree(array_device);
}

// Making partition in vector for quick sort algorithm
int MakePartition(double* vec, int start, int end){
  // Taking first element as pivot point
  double pivot = vec[start];
  // Finding corret position of pivot element
  int count = 0;
  for (int i = start + 1; i <= end; i++) {
    if (vec[i] >= pivot)
      count++;
  }
  // Giving pivot element its correct position
  int pivotIndex = start + count;
  swap(vec[pivotIndex], vec[start]);
  // Now pivot element is on its true position
  // and we need to place elements greater than pivot on the left and less on the right
  int i = start;
  int j = end;
  // Number of missplaced elements is even, so we will use pair swaps
  while (i < pivotIndex && j > pivotIndex) {
        // looking for missplaced elements on the left of the pivot
    while (vec[i] >= pivot) {
      i++;
    }
        // looking for missplaced elements on the right of the pivot
    while (vec[j] < pivot) {
      j--;
    }
        // Swap pair of missplaced elements
    if (i < pivotIndex && j > pivotIndex) {
      swap(vec[i++], vec[j--]);
    }
  }
  return pivotIndex;
}

// QuickSort on CPU, main sorting algorithm
void QuickSortCPU(double* vec, int start, int end){
  // base of the recursion
  if (start >= end)
    return;

  // partitioning the array
  int p = MakePartition(vec, start, end);
```

```
535
536   // Sorting the left part
537   QuickSortCPU(vec, start, p - 1);
538
539   // Sorting the right part
540   QuickSortCPU(vec, p + 1, end);
541 }
```

10 Приложение 2. Результаты с использованием сортировки пузырьком

```
1  CPU:
2  N          Time1          Time2       Time3
3       2         2e-06          2e-06         2e-06
4       4         2e-06          1e-06         2e-06
5       8         3e-06          2e-06         2e-06
6      16         2e-05          2e-06         2e-06
7      32         6e-06       1.9e-05       2.1e-05
8      64       2.1e-05          2e-05       1.9e-05
9     128       9.5e-05       7.9e-05       6.5e-05
10    256       0.00023      0.000215      0.000221
11    512      0.000812      0.000826      0.000835
12   1024      0.003036      0.003034      0.003072
13   2048      0.011849      0.011414      0.011220
14   4096      0.050121      0.049833      0.049526
15   8192      0.246187      0.246981      0.246672
16  16384      1.104490      1.111950      1.118400
17  32768      4.715060      4.712280      4.746010
18  65536     19.569600     19.606000     19.619100
19 131072     83.036900     83.116100     83.123300
20
21 GPU:
22 N           Time1         Time2         Time3
23      2      0.058873      0.000688      0.000686
24      4      0.000783      0.000783      0.000743
25      8      0.000909      0.000905      0.000898
26     16      0.001229      0.001193      0.001206
27     32      0.001768      0.001784      0.001388
28     64      0.002343      0.002286      0.002307
29    128      0.004188      0.004102      0.004207
30    256      0.007896      0.007927      0.007834
31    512      0.015202      0.015232      0.015223
32   1024      0.030685      0.030746      0.030602
33   2048      0.062144      0.061682      0.061839
34   4096      0.125046      0.125299      0.125089
35   8192      0.254650      0.254290      0.253483
36  16384      0.527156      0.524305      0.521602
37  32768      1.062670      1.003710      1.002440
38  65536      2.357570      2.350200      2.348180
39 131072      6.085280      6.087900      6.091310
```

11 Приложение 3. Результаты с использованием быстрой сортировки

```
CPU:
N           Time1        Time2        Time3
      2        3e-06       1.8e-05        1e-06
      4        2e-06       1.7e-05        2e-06
      8        2e-06         2e-06       1.5e-05
     16        4e-06         4e-06        5e-06
     32       2.2e-05        6e-06        6e-06
     64       2.6e-05       1.1e-05       2.5e-05
    128       4.1e-05       3.7e-05       3.7e-05
    256       6.6e-05       6.6e-05         5e-05
    512      0.000126      0.000131       0.00012
   1024      0.000267      0.000263      0.000273
   2048      0.000585      0.000576      0.000583
   4096      0.001271      0.001284      0.001268
   8192      0.002768      0.002764      0.002792
  16384      0.005908      0.005520      0.005429
  32768      0.011763      0.011668      0.011603
  65536      0.024622      0.025111      0.025155
 131072      0.053309      0.052946      0.052943
 262144      0.113118      0.112153      0.113690
 524288      0.236008      0.236645      0.237462
1048576      0.515007      0.498812      0.498667

GPU:
N           Time1        Time2        Time3
      2      0.064100     0.000500     0.000499
      4      0.000562     0.000539     0.000504
      8      0.000566     0.000509     0.000513
     16      0.000582     0.000533     0.000572
     32      0.000658     0.000608     0.000608
     64      0.000913     0.000874     0.000807
    128      0.001216     0.001283     0.001376
    256      0.001708     0.001882     0.001524
    512      0.002508     0.002304     0.002655
   1024      0.003550     0.003886     0.004721
   2048      0.007340     0.009466     0.006943
   4096      0.012505     0.017194     0.011700
   8192      0.021236     0.024641     0.023384
  16384      0.036863     0.054128     0.042480
  32768      0.097180     0.060684     0.088445
  65536      0.164778     0.188296     0.134376
 131072      0.254523     1.064770     0.391670
 262144      2.038400     1.521510     0.902341
 524288      1.983840     4.567120     1.964410
1048576     14.262500     7.465550    12.941900
```