



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:

```

1 // Bubble sort on CPU in one thread
2 void BubbleSortNoCUDA(double *array, int N){
3     // Repeat N times
4     for (int i = 0; i < N; i++){
5         // Move "bubble" and swap elements if needed
6         for (int j = 0; j < N-i-1; j++) {
7             if (array[j]<array[j + 1]){
8                 double helper = array[j];
9                 array[j] = array[j + 1];
10                array[j + 1] = helper;
11            }
12        }
13    }
14 }

```

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:

```

1 // One step for bubble sort with CUDA
2 __global__ void BubbleMove(double *array, int N, int step){
3     // Get thread ID
4     int idx = blockDim.x * blockIdx.x + threadIdx.x;
5     // Check that thread belongs to array
6     if (idx < (N-1)) {
7         // Check that "bubble" moved enough for this thread to run and that there are no
            intersections
8         if (step-2 >= idx && (idx - step) % 2 == 0){
9             // Swap elements if need
10            if (array[idx] < array[idx + 1]){
11                double helper = array[idx];
12                array[idx] = array[idx + 1];
13                array[idx + 1] = helper;
14            }
15        }
16    }
17 }
18
19 // Bubble sort on GPU CUDA
20 void BubbleSortCUDA(double *array_host, int N, int blockSize){
21     // Create, allocate memory and copy array to device (GPU)
22     double *array_device;
23     cudaMalloc((void **)&array_device, N * sizeof(double));
24     cudaMemcpy(array_device, array_host, N*sizeof(double), cudaMemcpyHostToDevice);
25     // Calculate needed number of blocks according to block size
26     int nblocks = N / blockSize + 1;
27     // N+N steps for all needed changes sure to be made
28     for (int step = 0; step <= N + N; step++) {
29         // Step of bubble sort
30         BubbleMove<<<nblocks, blockSize>>>(array_device, N, step);
31         // Wait for all threads to finish changes

```

```

32     //cudaThreadSynchronize();
33     cudaDeviceSynchronize();
34 }
35 // Copy array from device to host
36 cudaMemcpy(array_host, array_device, N*sizeof(double), cudaMemcpyDeviceToHost);
37 cudaFree(array_device);
38 }

```

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:

```

1  // Making partition in vector for quick sort algorithm
2  int MakePartition(double* vec, int start, int end){
3      // Taking first element as pivot point
4      double pivot = vec[start];
5      // Finding corret position of pivot element
6      int count = 0;
7      for (int i = start + 1; i <= end; i++) {
8          if (vec[i] >= pivot)
9              count++;
10     }
11     // Giving pivot element its correct position
12     int pivotIndex = start + count;
13     swap(vec[pivotIndex], vec[start]);
14     // Now pivot element is on its true position
15     // and we need to place elements greater than pivot on the left and less on the right
16     int i = start;
17     int j = end;
18     // Number of missplaced elements is even, so we will use pair swaps
19     while (i < pivotIndex && j > pivotIndex) {
20         // looking for missplaced elements on the left of the pivot
21         while (vec[i] >= pivot) {
22             i++;
23         }
24         // looking for missplaced elements on the right of the pivot
25         while (vec[j] < pivot) {
26             j--;
27         }
28         // Swap pair of missplaced elements
29         if (i < pivotIndex && j > pivotIndex) {
30             swap(vec[i++], vec[j--]);
31         }
32     }
33     return pivotIndex;
34 }
35

```

```

36 // QuickSort on CPU, main sorting algorithm
37 void QuickSortCPU(double* vec, int start, int end){
38     // base of the recursion
39     if (start >= end)
40         return;
41
42     // partitioning the array
43     int p = MakePartition(vec, start, end);
44
45     // Sorting the left part
46     QuickSortCPU(vec, start, p - 1);
47
48     // Sorting the right part
49     QuickSortCPU(vec, p + 1, end);
50 }

```

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`):

```

1  //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
2  // Very basic quicksort algorithm, recursively launching the next level.
3  //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
4  __global__ void cdp_simple_quicksort(double* vec, int start, int end, int depth) {
5      // if max depth is reached or array is too small launch selection sort
6      if (depth >= MAX_DEPTH || end - start <= INSERTION_SORT) {
7          selection_sort(vec, start, end);
8          return;
9      }
10
11     double helper;
12
13     // Make partitioning (like in serial mode)
14     // taking first elemt as pivot point
15     double pivot = vec[start];
16     // Finding corret position of pivot element
17     int count = 0;
18     for (int i = start + 1; i <= end; i++) {
19         if (vec[i] >= pivot)
20             count++;
21     }
22     // Giving pivot element its correct position
23     int pivotIndex = start + count;
24     //swap(vec[pivotIndex], vec[start]);
25     helper = vec[pivotIndex];
26     vec[pivotIndex] = vec[start];
27     vec[start] = helper;
28
29     // Now pivot element is on its true position
30     // and we need to place elements greater than pivot on the left and less on the right
31     int i = start;
32     int j = end;
33

```

```

34 // Number of misplaced elements is even, so we will use pair swaps
35 while (i < pivotIndex && j > pivotIndex) {
36     // looking for misplaced elements on the left of the pivot
37     while (vec[i] >= pivot) {
38         i++;
39     }
40     // looking for misplaced elements on the right of the pivot
41     while (vec[j] < pivot) {
42         j--;
43     }
44     // Swap pair of misplaced elements
45     if (i < pivotIndex && j > pivotIndex) {
46         //swap(vec[i++], vec[j--]);
47         helper = vec[i];
48         vec[i] = vec[j];
49         vec[j] = helper;
50         i++;
51         j--;
52     }
53 }
54
55 // Launch a new block to sort the left part.
56 cudaStream_t s;
57 cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
58 cdp_simple_quicksort<<<1, 1, 0, s>>>(vec, start, pivotIndex - 1, depth + 1);
59 cudaStreamDestroy(s);
60
61
62 // Launch a new block to sort the right part.
63 cudaStream_t s1;
64 cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
65 cdp_simple_quicksort<<<1, 1, 0, s1>>>(vec, pivotIndex + 1, end, depth + 1);
66 cudaStreamDestroy(s1);
67 }
68 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
69 // Call the quicksort kernel from the host.
70 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
71 void QuickSortCUDA(double* vec, unsigned int nitems) {
72     // Create, allocate memory and copy array to device (GPU)
73     double *array_device;
74     cudaMalloc((void **)&array_device, nitems * sizeof(double));
75     cudaMemcpy(array_device, vec, nitems*sizeof(double), cudaMemcpyHostToDevice);
76     // Launch on device
77     int left = 0;
78     int right = nitems - 1;
79     // std::cout << "Launching kernel on the GPU" << std::endl;
80     cdp_simple_quicksort<<<1, 1>>>(array_device, left, right, 0);
81     cudaDeviceSynchronize();
82     // Copy array from device to host
83     cudaMemcpy(vec, array_device, nitems*sizeof(double), cudaMemcpyDeviceToHost);
84     cudaFree(array_device);
85 }

```

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:

```
1 // Thread code for dot product on CUDA
2 __global__ void DotProduct(double const *deviceLHS, double const *deviceRHS, double *
  deviceRES, int N){
3     // Shared array for threads in a block
4     __shared__ double helper[THREADS_PER_BLOCK];
5
6     // Get thread ID
7     int idx = blockIdx.x * blockDim.x + threadIdx.x;
8
9     // Compute dot products for elements and write into shared array
10    helper[threadIdx.x] = deviceLHS[idx] * deviceRHS[idx];
11
12    __syncthreads();
13
14    // Calculate results for current Block (reduction sum)
15    // Works if THREADS_PER_BLOCK is power of 2
16    int index = blockDim.x / 2;
17    while (index != 0){
18        if (threadIdx.x < index) {
19            helper[threadIdx.x] += helper[threadIdx.x + index];
20        }
21
22        __syncthreads();
23
24        index = index / 2;
25    }
26
27    // Save the results for block
28    if (threadIdx.x == 0) {
29        deviceRES[blockIdx.x] = helper[0];
30    }
31 }
32
33 // Dot product of two arrays on GPU CUDA
34 double CalculateDotProductCuda(double *hostLHS, double *hostRHS, int N){
35     double *hostInterRes; // helper for intermediate results
36     double *deviceLHS, *deviceRHS, *deviceRES;
37     // Thread blocks per Grid
38     int gridDim = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
39     // Create vectors
40     size_t interResSize = gridDim * sizeof(double);
41     hostInterRes = (double*)malloc(interResSize);
42     cudaMalloc(&deviceLHS, N * sizeof(double));
43     cudaMalloc(&deviceRHS, N * sizeof(double));
44     cudaMalloc(&deviceRES, interResSize);
45     // Arrays to gpu
46     cudaMemcpy(deviceLHS, hostLHS, N * sizeof(double), cudaMemcpyHostToDevice);
```

```

47     cudaMemcpy(deviceRHS, hostRHS, N * sizeof(double), cudaMemcpyHostToDevice);
48
49     DotProduct<<<gridDim, THREADS_PER_BLOCK>>>(deviceLHS, deviceRHS, deviceRES, N);
50
51     // Array to cpu
52     cudaMemcpy(hostInterRes, deviceRES, interResSize, cudaMemcpyDeviceToHost);
53     cudaFree(deviceLHS);
54     cudaFree(deviceRHS);
55     cudaFree(deviceRES);
56
57     // Final reduction
58     double res = 0;
59     for (int i = 0; i < gridDim; i++){
60         res += hostInterRes[i];
61     }
62     return res;
63 }

```

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 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 эффективны.

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.

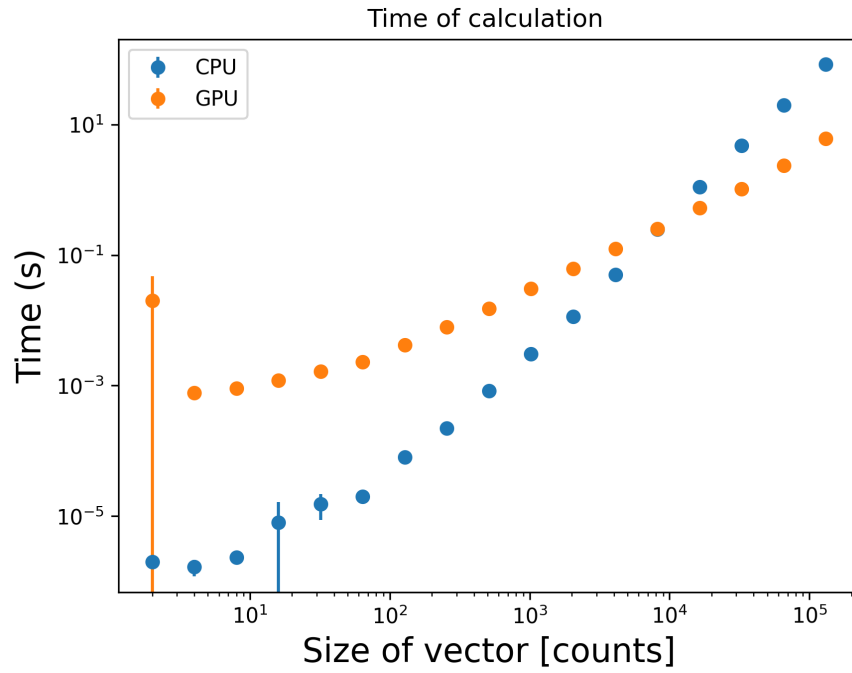


Рис. 1: Зависимость времени вычисления от размера массива. Сортировка пузырьком.

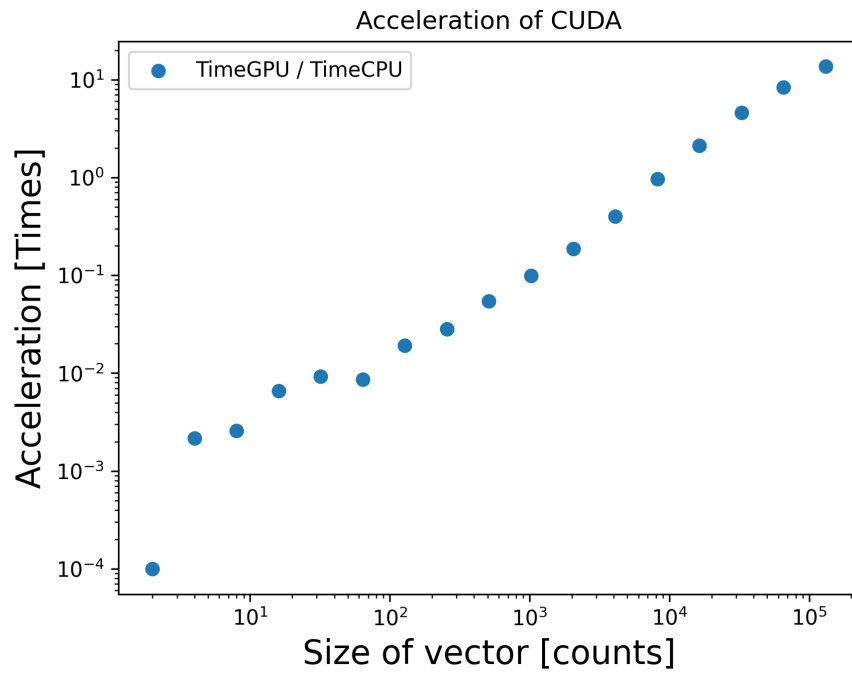


Рис. 2: Зависимость ускорения вычисления от размера массива. Сортировка пузырьком.

7 Nvidia Visual Profiler

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

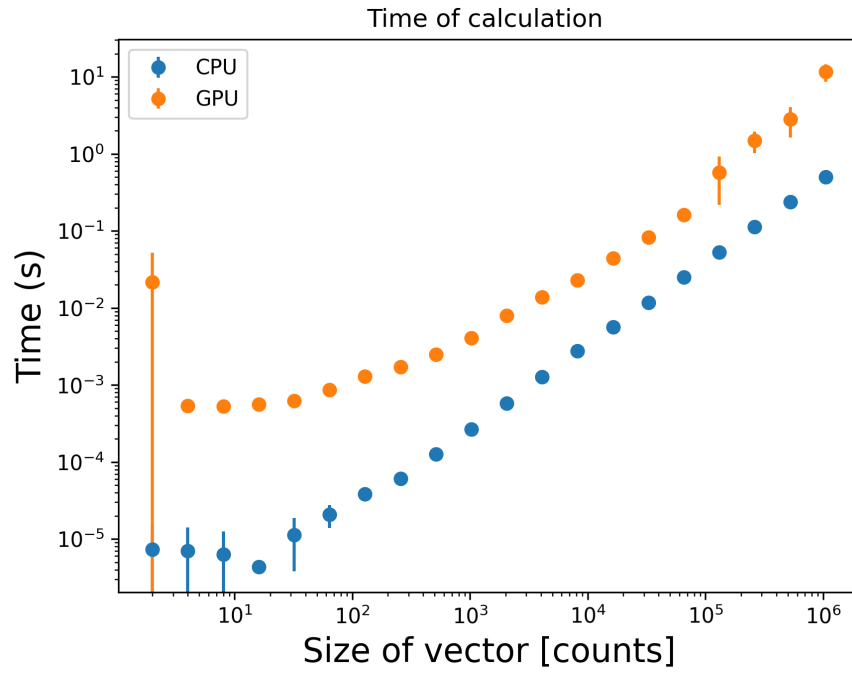


Рис. 3: Зависимость времени вычисления от размера массива. Быстрая сортировка.

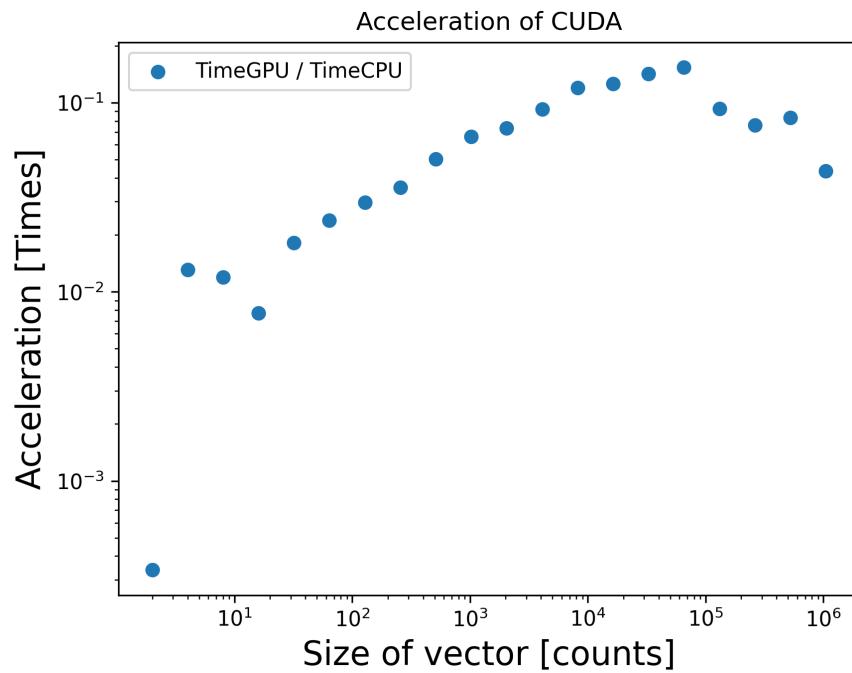


Рис. 4: Зависимость ускорения вычисления от размера массива. Быстрая сортировка.

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.

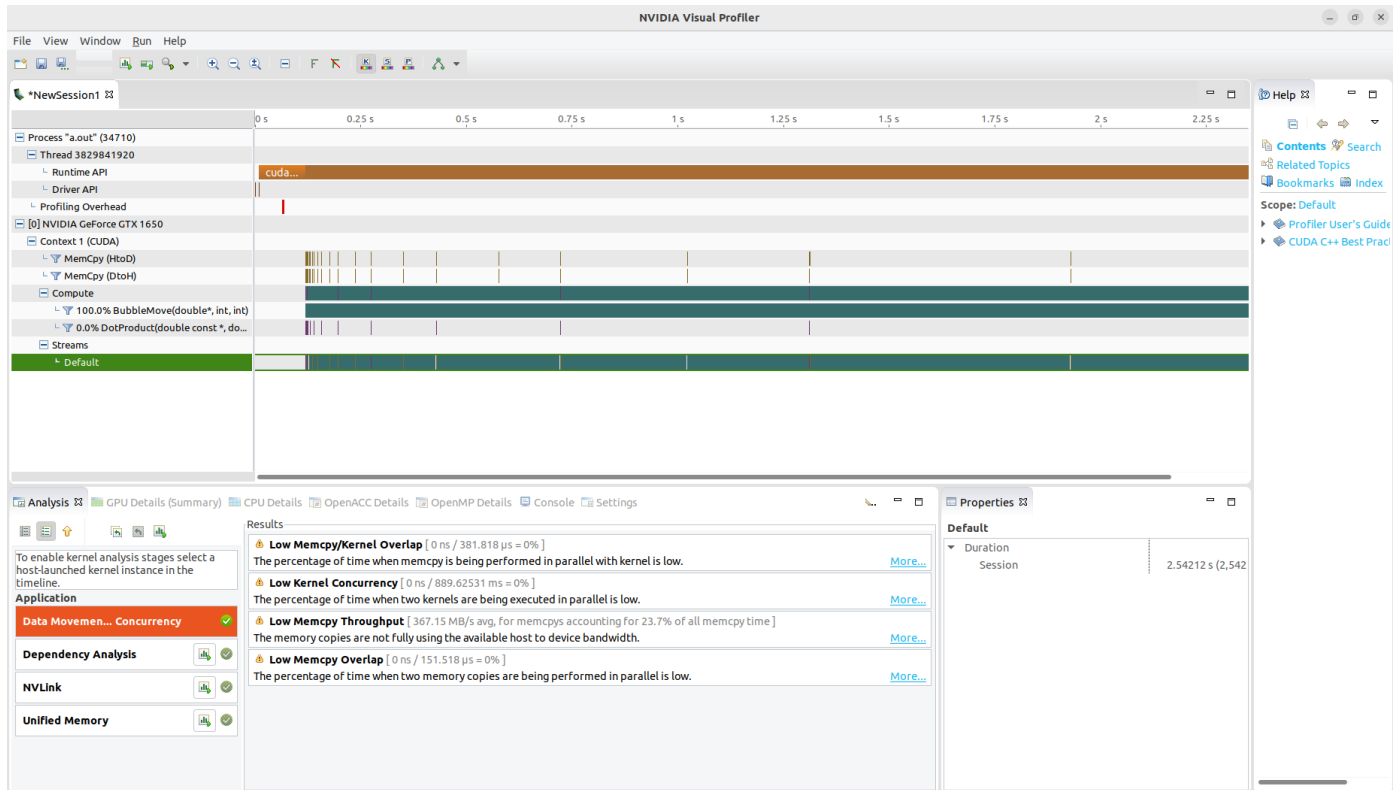


Рис. 5: Результаты построения таймлайна выполнения программы в NVVP.

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

```
1 #include <iostream>
2 #include <stdio.h>
3 #include <time.h>
4 #include <iomanip>
5 #define THREADS_PER_BLOCK 1024
6 #define MAX_DEPTH 16           // max depth for quick sort layers
7 #define INSERTION_SORT 32     // smaller than these - selection sort instead of quick
8
9 using namespace std;
10
11 __global__ void BubbleMove(double* array, int N, int step);
12
13 __global__ void DotProduct(double const *deviceLHS, double const *deviceRHS, double *
    deviceRES, int N);
14
15 __global__ void cdp_simple_quicksort(double* vec, int start, int end, int depth);
16
17 __device__ void selection_sort(double* data, int start, int end);
18
19 void BubbleSortCUDA(double *array_host, int N, int blockSize);
20
21 void PrintArray(const double* array, const int n);
22
23 void BubbleSortNoCUDA(double *array, int N);
24
25 bool IsSorted(double *array, int N);
26
27 void measureNoCUDA(int N);
28
29 void measureCUDA(int N);
30
31 double CalculateDotProductNoCuda(double *lhs, double *rhs, int N);
32
33 double CalculateDotProductCuda(double *hostLHS, double *hostRHS, int N);
34
35 int MakePartition(double* vec, int start, int end);
36
37 void QuickSortCPU(double* vec, int start, int end);
38
39 void QuickSortCUDA(double* vec, unsigned int nitems);
40
41
42 int main (int argc, char * argv []){
43     srand(static_cast <unsigned> (time(0)));
44
45     cout << "CPU:" << endl;
46     cout << "N      Time1      Time2      Time3" << endl;
47     for (int n = 2; n < 1000000; n*=2 ){
48         cout<< setw(6) << n << " ";
49         for (int i = 0; i < 3; i++){
50             measureNoCUDA(n);
```

```

51         cout << "    ";
52     }
53     cout << endl;
54 }
55
56 cout << "GPU:" << endl;
57 cout << "N      Time1      Time2      Time3" << endl;
58 //cout << "N      Time" << endl;
59 for (int n = 2; n < 1000000; n*=2 ){
60     cout<< setw(6) << n << "    ";
61     for (int i = 0; i < 3; i++){
62         measureCUDA(n);
63         cout << "    ";
64     }
65     cout << endl;
66 }
67 cout << endl;
68 return 0;
69 }
70
71 // One step for bubble sort with CUDA
72 __global__ void BubbleMove(double *array, int N, int step){
73     // Get thread ID
74     int idx = blockDim.x * blockIdx.x + threadIdx.x;
75     // Check that thread belongs to array
76     if (idx < (N-1)) {
77         // Check that "bubble" moved enough for this thread to run and that there are no
78         // intersections
79         if (step-2 >= idx && (idx - step) % 2 == 0){
80             // Swap elements if need
81             if (array[idx] < array[idx + 1]){
82                 double helper = array[idx];
83                 array[idx] = array[idx + 1];
84                 array[idx + 1] = helper;
85             }
86         }
87     }
88
89     // Thread code for dot product on CUDA
90     __global__ void DotProduct(double const *deviceLHS, double const *deviceRHS, double *
91     deviceRES, int N){
92         // Shared array for threads in a block
93         __shared__ double helper[THREADS_PER_BLOCK];
94
95         // Get thread ID
96         int idx = blockIdx.x * blockDim.x + threadIdx.x;
97
98         // Compute dot products for elements and write into shared array
99         helper[threadIdx.x] = deviceLHS[idx] * deviceRHS[idx];
100
101         __syncthreads();
102
103         // Calculate results for current Block (reduction sum)

```

```

103 // Works if THREADS_PER_BLOCK is power of 2
104 int index = blockDim.x / 2;
105 while (index != 0){
106     if (threadIdx.x < index) {
107         helper[threadIdx.x] += helper[threadIdx.x + index];
108     }
109
110     __syncthreads();
111
112     index = index / 2;
113 }
114
115 // Save the results for block
116 if (threadIdx.x == 0) {
117     deviceRES[blockIdx.x] = helper[0];
118 }
119 }
120
121 // Bubble sort on GPU CUDA
122 void BubbleSortCUDA(double *array_host, int N, int blockSize){
123     // Create, allocate memory and copy array to device (GPU)
124     double *array_device;
125     cudaMalloc((void **)&array_device, N * sizeof(double));
126     cudaMemcpy(array_device, array_host, N*sizeof(double), cudaMemcpyHostToDevice);
127     // Calculate needed number of blocks according to block size
128     int nblocks = N / blockSize + 1;
129     // N+N steps for all needed changes sure to be made
130     for (int step = 0; step <= N + N; step++) {
131         // Step of bubble sort
132         BubbleMove<<<nblocks, blockSize>>>(array_device, N, step);
133         // Wait for all threads to finish changes
134         //cudaThreadSynchronize();
135         cudaDeviceSynchronize();
136     }
137     // Copy array from device to host
138     cudaMemcpy(array_host, array_device, N*sizeof(double), cudaMemcpyDeviceToHost);
139     cudaFree(array_device);
140 }
141
142 // Bubble sort on CPU in one thread
143 void BubbleSortNoCUDA(double *array, int N){
144     // Repeat N times
145     for (int i = 0; i < N; i++){
146         // Move "bubble" and swap elements if needed
147         for (int j = 0; j < N-i-1; j++) {
148             if (array[j]<array[j + 1]){
149                 double helper = array[j];
150                 array[j] = array[j + 1];
151                 array[j + 1] = helper;
152             }
153         }
154     }
155 }
156

```

```

157 // check if and array is sorted
158 bool IsSorted(double *array, int N){
159     // Iterate over all elements
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

```

```

211 // Check that array is sorted
212 if (IsSorted(array2, N)){
213     //cout << timeArray2Sorted << " ";
214 }
215 else {
216     free(array1);
217     free(array2);
218     cout << "ERROR, ARRAY 2 NOT SORTED" << endl;
219     return;
220 }
221
222 // Calculate dot product of sorted arrays
223 start_time = clock();
224 resCPU = CalculateDotProductNoCuda(array1, array2, N);
225 end_time = clock();
226
227 // Time taken
228 timeDotProduct = (float)(end_time - start_time) / CLOCKS_PER_SEC;
229
230 // Free memory
231 free(array1);
232 free(array2);
233 // Print time info
234 cout << setw(10) << timeArray1Sorted + timeArray2Sorted + timeDotProduct;
235 }
236
237 // measure time taken for all steps on GPU CUDA
238 void measureCUDA(int N){
239     // Create arrays
240     double *array1 = (double *)malloc(N * sizeof(double));
241     double *array2 = (double *)malloc(N * sizeof(double));
242     // Create variables for saving time results
243     clock_t start_time, end_time;
244     float timeArray1Sorted, timeArray2Sorted, timeDotProduct;
245     // Create variables for saving results of all calculations
246     double resGPU = 0.0;
247     double resCPU = 0.0;
248     // Fill arrays with random numbers
249     for (int i = 0; i < N; i++) {
250         array1[i] = static_cast<double>(rand()) / static_cast<double>(RAND_MAX);
251         array2[i] = static_cast<double>(rand()) / static_cast<double>(RAND_MAX);
252     }
253
254     // Sort first array
255     start_time = clock();
256     BubbleSortCUDA(array1, N, THREADS_PER_BLOCK);
257     // QuickSortCUDA(array1, N);
258     end_time = clock();
259
260     // Time taken
261     timeArray1Sorted = (float)(end_time - start_time) / CLOCKS_PER_SEC;
262
263     // Check that array is sorted
264     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

```



```

319     cout << setw(10) << timeArray1Sorted + timeArray2Sorted + timeDotProduct;
320 }
321
322 // Print double array into console
323 void PrintArray(const double* array, const int n){
324     for(int i = 0; i < n; i++){
325         cout << array[i] << ' ';
326     }
327     cout << endl;
328 }
329
330 // Dot product of two arrays on CPU
331 double CalculateDotProductNoCuda(double *lhs, double *rhs, int N){
332     double result = 0.0;
333     for (int i = 0; i < N; i++){
334         result += lhs[i] * rhs[i];
335     }
336     return result;
337 }
338
339 // Dot product of two arrays on GPU CUDA
340 double CalculateDotProductCuda(double *hostLHS, double *hostRHS, int N){
341
342     double *hostInterRes; // helper for intermediate results
343     double *deviceLHS, *deviceRHS, *deviceRES;
344
345     // Thread blocks per Grid
346     int gridDim = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
347
348     // Create vectors
349     size_t interResSize = gridDim * sizeof(double);
350     hostInterRes = (double*)malloc(interResSize);
351     cudaMalloc(&deviceLHS, N * sizeof(double));
352     cudaMalloc(&deviceRHS, N * sizeof(double));
353     cudaMalloc(&deviceRES, interResSize);
354
355     // Arrays to gpu
356     cudaMemcpy(deviceLHS, hostLHS, N * sizeof(double), cudaMemcpyHostToDevice);
357     cudaMemcpy(deviceRHS, hostRHS, N * sizeof(double), cudaMemcpyHostToDevice);
358
359
360     DotProduct<<<gridDim, THREADS_PER_BLOCK>>>(deviceLHS, deviceRHS, deviceRES, N);
361
362     // Array to cpu
363     cudaMemcpy(hostInterRes, deviceRES, interResSize, cudaMemcpyDeviceToHost);
364
365
366     cudaFree(deviceLHS);
367     cudaFree(deviceRHS);
368     cudaFree(deviceRES);
369
370     // Final reduction
371     double res = 0;
372     for (int i = 0; i < gridDim; i++){

```

```

373         res += hostInterRes[i];
374     }
375
376     return res;
377 }
378
379 // Selection sort for final recursion steps of quick sort
380 // only for threads
381 __device__ void selection_sort(double* data, int start, int end) {
382     for (int i = start; i <= end; ++i) {
383         double max_val = data[i];
384         int max_idx = i;
385
386         // Find the biggest value in the range [start, end].
387         for (int j = i + 1; j <= end; ++j) {
388             double val_j = data[j];
389
390             if (val_j > max_val) {
391                 max_idx = j;
392                 max_val = val_j;
393             }
394         }
395
396         // Swap the values.
397         if (i != max_idx) {
398             data[max_idx] = data[i];
399             data[i] = max_val;
400         }
401     }
402 }
403
404
405 ///////////////////////////////////////////////////
406 // Very basic quicksort algorithm, recursively launching the next level.
407 ///////////////////////////////////////////////////
408 __global__ void cdp_simple_quicksort(double* vec, int start, int end, int depth) {
409     // if max depth is reached or array is too small launch selection sort
410     if (depth >= MAX_DEPTH || end - start <= INSERTION_SORT) {
411         selection_sort(vec, start, end);
412         return;
413     }
414
415     double helper;
416
417     // Make partitioning (like in serial mode)
418     // taking first elemt as pivot point
419     double pivot = vec[start];
420     // Finding corret position of pivot element
421     int count = 0;
422     for (int i = start + 1; i <= end; i++) {
423         if (vec[i] >= pivot)
424             count++;
425     }
426     // Giving pivot element its correct position

```

```

427     int pivotIndex = start + count;
428     //swap(vec[pivotIndex], vec[start]);
429     helper = vec[pivotIndex];
430     vec[pivotIndex] = vec[start];
431     vec[start] = helper;
432
433     // Now pivot element is on its true position
434     // and we need to place elements greater than pivot on the left and less on the right
435     int i = start;
436     int j = end;
437
438     // Number of misplaced elements is even, so we will use pair swaps
439     while (i < pivotIndex && j > pivotIndex) {
440         // looking for misplaced elements on the left of the pivot
441         while (vec[i] >= pivot) {
442             i++;
443         }
444         // looking for misplaced elements on the right of the pivot
445         while (vec[j] < pivot) {
446             j--;
447         }
448         // Swap pair of misplaced elements
449         if (i < pivotIndex && j > pivotIndex) {
450             //swap(vec[i++], vec[j--]);
451             helper = vec[i];
452             vec[i] = vec[j];
453             vec[j] = helper;
454             i++;
455             j--;
456         }
457     }
458
459     // Launch a new block to sort the left part.
460     cudaStream_t s;
461     cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
462     cdp_simple_quicksort<<<1, 1, 0, s>>>(vec, start, pivotIndex - 1, depth + 1);
463     cudaStreamDestroy(s);
464
465
466     // Launch a new block to sort the right part.
467     cudaStream_t s1;
468     cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
469     cdp_simple_quicksort<<<1, 1, 0, s1>>>(vec, pivotIndex + 1, end, depth + 1);
470     cudaStreamDestroy(s1);
471 }
472
473 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
474 // Call the quicksort kernel from the host.
475 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
476 void QuickSortCUDA(double* vec, unsigned int nitens) {
477     // Create, allocate memory and copy array to device (GPU)
478     double *array_device;
479     cudaMalloc((void **)&array_device, nitens * sizeof(double));
480     cudaMemcpy(array_device, vec, nitens*sizeof(double), cudaMemcpyHostToDevice);

```

```

481     // Launch on device
482     int left = 0;
483     int right = nitems - 1;
484     // std::cout << "Launching kernel on the GPU" << std::endl;
485     cdp_simple_quicksort<<<1, 1>>>(array_device, left, right, 0);
486     cudaDeviceSynchronize();
487     // Copy array from device to host
488     cudaMemcpy(vec, array_device, nitems*sizeof(double), cudaMemcpyDeviceToHost);
489     cudaFree(array_device);
490 }
491
492 // Making partition in vector for quick sort algorithm
493 int MakePartition(double* vec, int start, int end){
494     // Taking first element as pivot point
495     double pivot = vec[start];
496     // Finding corret position of pivot element
497     int count = 0;
498     for (int i = start + 1; i <= end; i++) {
499         if (vec[i] >= pivot)
500             count++;
501     }
502     // Giving pivot element its correct position
503     int pivotIndex = start + count;
504     swap(vec[pivotIndex], vec[start]);
505     // Now pivot element is on its true position
506     // and we need to place elements greater than pivot on the left and less on the right
507     int i = start;
508     int j = end;
509     // Number of missplaced elements is even, so we will use pair swaps
510     while (i < pivotIndex && j > pivotIndex) {
511         // looking for missplaced elements on the left of the pivot
512         while (vec[i] >= pivot) {
513             i++;
514         }
515         // looking for missplaced elements on the right of the pivot
516         while (vec[j] < pivot) {
517             j--;
518         }
519         // Swap pair of missplaced elements
520         if (i < pivotIndex && j > pivotIndex) {
521             swap(vec[i++], vec[j--]);
522         }
523     }
524     return pivotIndex;
525 }
526
527 // QuickSort on CPU, main sorting algorithm
528 void QuickSortCPU(double* vec, int start, int end){
529     // base of the recursion
530     if (start >= end)
531         return;
532
533     // partitioning the array
534     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

1	CPU:			
2	N	Time1	Time2	Time3
3	2	3e-06	1.8e-05	1e-06
4	4	2e-06	1.7e-05	2e-06
5	8	2e-06	2e-06	1.5e-05
6	16	4e-06	4e-06	5e-06
7	32	2.2e-05	6e-06	6e-06
8	64	2.6e-05	1.1e-05	2.5e-05
9	128	4.1e-05	3.7e-05	3.7e-05
10	256	6.6e-05	6.6e-05	5e-05
11	512	0.000126	0.000131	0.00012
12	1024	0.000267	0.000263	0.000273
13	2048	0.000585	0.000576	0.000583
14	4096	0.001271	0.001284	0.001268
15	8192	0.002768	0.002764	0.002792
16	16384	0.005908	0.005520	0.005429
17	32768	0.011763	0.011668	0.011603
18	65536	0.024622	0.025111	0.025155
19	131072	0.053309	0.052946	0.052943
20	262144	0.113118	0.112153	0.113690
21	524288	0.236008	0.236645	0.237462
22	1048576	0.515007	0.498812	0.498667
23				
24	GPU:			
25	N	Time1	Time2	Time3
26	2	0.064100	0.000500	0.000499
27	4	0.000562	0.000539	0.000504
28	8	0.000566	0.000509	0.000513
29	16	0.000582	0.000533	0.000572
30	32	0.000658	0.000608	0.000608
31	64	0.000913	0.000874	0.000807
32	128	0.001216	0.001283	0.001376
33	256	0.001708	0.001882	0.001524
34	512	0.002508	0.002304	0.002655
35	1024	0.003550	0.003886	0.004721
36	2048	0.007340	0.009466	0.006943
37	4096	0.012505	0.017194	0.011700
38	8192	0.021236	0.024641	0.023384
39	16384	0.036863	0.054128	0.042480
40	32768	0.097180	0.060684	0.088445
41	65536	0.164778	0.188296	0.134376
42	131072	0.254523	1.064770	0.391670
43	262144	2.038400	1.521510	0.902341
44	524288	1.983840	4.567120	1.964410
45	1048576	14.262500	7.465550	12.941900
