Radix Sort Analyses in Parallel and Serial Way

Ömer Ufuk Efendioğlu
ufukomer@gmail.com

Semih Okan Pehlivan
semihokanp@gmail.com

ABSTRACT

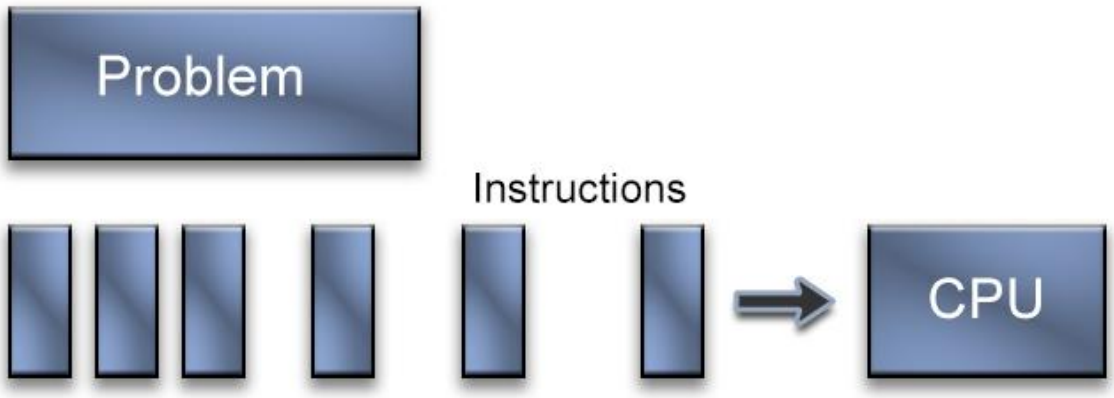
This paper presents a comparative analysis of parallel and serial version of radix sort algorithm: Sorting time and speed variation CPU and GPU execution of these algorithms. For that purpose, we have implemented parallel radix sort that using threads for parallelism in GPU and radix sort that is being executed in CPU as serial execution. Both process implemented in order to run on GTX860M architecture.

KEYWORDS

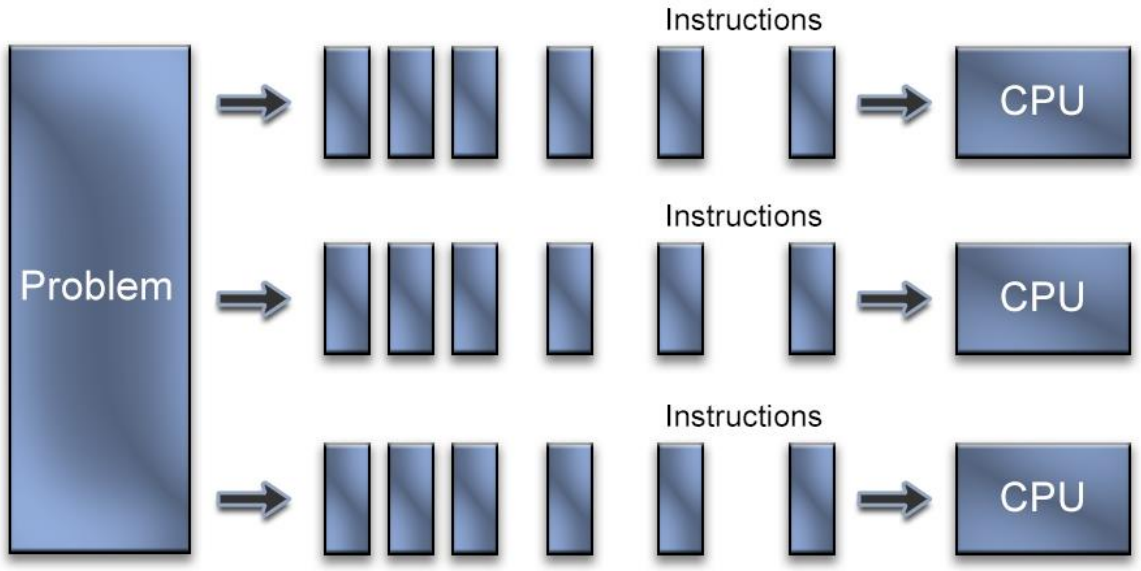
Parallel Computing, Parallel Radix Sort, GPU, CUDA

1. INTRODUCTION

In traditional methods programs or applications are created as serial computations. In this approach program is executed by one processor in a computer and is divided into different command series. Processor executed each command one after another. Therefore, only one command is executed per unit time by processor.



In parallel computing or parallel programming, more than one computation resource is used while solving a problem. More than one processors are used so that problem is divided into pieces. Each piece is divided into different instruction then instructions are executed by different processors.



Reducing the execution time of problem is a mandatory if the problem has so many instructions that cause its execution took very long time like hours, days and maybe weeks. According to this issue, a platform that give us ability to program a GPU is essential, which is NVDIAs CUDA.

CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU). With millions of CUDA-enabled GPUs sold to date, software developers, scientists and researchers are finding broad-ranging uses for GPU computing with CUDA. [1]

CUDA programming enable programmer to use GPUs' CUDA cores to perform parallel computing. Thus, the GPU architecture should be considered before starting coding on CUDA platform, GTX860M in this project. The architecture is extremely enough to satisfy the needs that is essential for parallel sorting algorithms. Specifications are depicted as following, both CUDA core and microprocessor counts:

GTX860M Specifications	
CUDA Cores	640
Graphics Clock (MHz)	1019
Memory Data Rate (MHz)	5010
Memory Interface	128-bit
Memory Bandwidth (GB/s)	80.16
Total Graphics Memory (MB)	4096
OpenGL	4.5
OpenCL	1.1

2. SORTING ALGORITHMS

Sorting on GPU require transferring data from main (host) memory to GPU (device) memory. To calculate execution time on this part of the execution is out of favour because it is not part of analysing sorting algorithm instead it is arrangement for GPU allocation.

Synchronization of threads is one aspect that reduces the execution time, but it is a necessity for example if it is needed to calculate maximum number, namely reduction process in GPU. Putting data into shared memory increases the performance. Because it is sharing data among threads in a block so that they use this data from shared memory, therefore, this approach reduces the memory access.

The application we've created for parallel and serial radix sort performance test firstly creates an array of given size and fills the array with integer value from 0 to 1024. Most part of the main application that accomplish array creation with its filling process and the way of starting serial radix sort algorithm same with the main application of parallel radix sort algorithm, although they are running in the same main thread.

2.1 Serial Radix Sort

Since we want to calculate performance it push us through to accept using so many loops, according to analyse step, with constant array element count and different element values. This algorithm also run on GPU with 1 block and 1 thread sizes which makes it serial execution.

```
1.  #define WSIZE 32
2.  #define LOOPS 3000000
3.  #define UPPER_BIT 10
4.  #define LOWER_BIT 0
5.
6.  __device__ int ddata_s[WSIZE];
7.
8.  __device__ int getMax(int arr[], int n) {
9.      int mx = arr[0];
10.     for (int i = 1; i < n; i++)
11.         if (arr[i] > mx) mx = arr[i];
12.     return mx;
13. }
14.
15. __device__ void countSort(int arr[], int n, int exp) {
16.     int * output; // Output array
17.     int i, count[10] = { 0 };
18.     // Store count of occurrences in count[]
19.     output = (int * ) malloc(sizeof(int) * n);
20.     for (i = 0; i < n; i++) count[(arr[i] / exp) % 10]++;
21.     // Change count[i] so that count[i] now contains actual
22.     // position of this digit in output[]
23.     for (i = 1; i < 10; i++) {
24.         count[i] += count[i - 1];
25.     } // Build the output array
26.     for (i = n - 1; i >= 0; i--) {
27.         output[count[(arr[i] / exp) % 10] - 1] = arr[i];
28.         count[(arr[i] / exp) % 10]--;
29.     }
30.     // Copy the output array to arr[], so that arr[] now
31.     // contains sorted numbers according to current digit
32.     for (i = 0; i < n; i++) arr[i] = output[i];
33. }
34. __device__ void radixsort(int arr[], int n) {
35.     // Find the maximum number to know number of digits
36.     int m = getMax(arr, n); /
37.     // Do counting sort for every digit. Note that instead
38.     // of passing digit number, exp is passed. exp is 10^i
39.     // where i is current digit number
40.     for (int exp = 1; m / exp > 0; exp *= 10) countSort(arr, n, exp);
41. }
42.
43. __global__ void serialRadix() {
44.     radixsort(ddata_s, WSIZE);
45. }
```

```
46.
47. int main() {
48.
49.     unsigned int hdata_s[WSIZE];
50.     totalTime = 0;
51.
52.     for (int lcount = 0; lcount < LOOPS; lcount++) {
53.     // Array elements have value in range of 1024
54.         unsigned int range = 1 U << UPPER_BIT;
55.         // Fill array with random elements
56.         // Range = 1024
57.         for (int i = 0; i < WSIZE; i++) {
58.             hdata_s[i] = rand() % range;
59.         }
60.
61.         // Copy data from host to device
62.         cudaMemcpyToSymbol(ddata_s, hdata_s, WSIZE * sizeof(unsigned int));
63.         // Execution time measurement, that point starts the clock
64.         high_resolution_clock::time_point t1 = high_resolution_clock::now();
65.         serialRadix << < 1, 1 >>> ();
66.         // Make kernel function synchronous
67.         cudaDeviceSynchronize();
68.         // Execution time measurement, that point stops the clock
69.         high_resolution_clock::time_point t2 = high_resolution_clock::now();
70.         // Execution time measurement, that is the result
71.         auto duration = duration_cast < milliseconds > (t2 - t1).count();
72.         // Summination of each loops' execution time
73.         totalTime += duration;
74.         // Copy data from device to host
75.         cudaMemcpyFromSymbol(hdata_s, ddata_s, WSIZE * sizeof(unsigned int));
76.     }
77.
78.     printf("\nSerial Radix Sort:\n");
79.     printf("Array size = %d\n", WSIZE * LOOPS);
80.     printf("Time elapsed = %gmilliseconds\n\n", totalTime);
81.
82.     return 0;
83. }
```

In the serial radix sort, device functions which declared using __device__ prefix are can be called only from GPU. Thus, algorithm uses `cudaMemcpyToSybol` instead `cudaMemcpy`. The reason is that `cudaMemcpy` cannot do the same thing as `cudaMemcpyToSymbol` without an additional API call. Consider device memory variable.

```
1.  __device__ int ddata_s[WSIZE];
```

To copy values to this array using cudaMemcpyToSymbol is as shown below:

```
1.  cudaMemcpyToSymbol(ddata_s,  hdata_s,  WSIZE  *  sizeof(unsigned  int));
```

To do same with cudaMemcpy requires following process:

```
1.  float * d_ddata_s;
2.  cudaGetSymbolAddress((void * * ) & d_ddata_s, ddata_s);
3.  cudaMemcpy(d_ddata_s, hdata_s, WSIZE * sizeof(float));
```

As a result, this algorithm performs serial radix sorting of given random array on GPU using 1 block and 1 thread size, and then calculates execution time in milliseconds.

2.2 Parallel Radix Sort

In the parallel radix sort, each element will be sorted digit by digit, from least significant to most significant. For each digit, algorithm will move the elements so that those digits are in increasing order.

Let's examine sorting logic of parallel radix algorithm:

Let's sort 4 elements which have 4 binary digits, 1, 2, 3 and 4.

ELEMENT	**1**	**2**	**3**	**4**
VALUE	7	14	4	1
BINARY	0111	1110	0100	0001

Firstly the algorithm considers bit 0:

ELEMENT	**1**	**2**	**3**	**4**
VALUE	7	14	4	1
BINARY	0111	1110	0100	0001
BIT 0	1	0	0	1

Radix sort algorithm says we must move the elements in such a way that all the zeroes are on the left, namely in the beginning of array.

ELEMENT	**2**	**3**	**1**	**4**
VALUE	14	4	7	1
BINARY	1110	0100	0111	0001
BIT 0	0	0	1	1

The first step of parallel radix algorithm is done. Now it should consider the next bit, i.e. bit 1:

ELEMENT	**3**	**2**	**1**	**4**
VALUE	4	14	7	1
BINARY	0100	1110	0111	0001
BIT 1	0	1	1	0

And order them again:

ELEMENT	**3**	**4**	**2**	**1**
VALUE	4	1	14	7
BINARY	0100	0001	1110	0111
BIT 1	0	0	1	1

Then it moves to the next higher bit again:

ELEMENT	**3**	**4**	**2**	**1**
VALUE	4	1	14	7
BINARY	0100	0001	1110	0111
BIT 2	1	0	1	1

And order them again:

ELEMENT	4	3	2	1
VALUE	1	4	14	7
BINARY	0001	0100	1110	0111
BIT 2	0	1	1	1

Then it moves to the last significant bit:

ELEMENT	4	3	2	1
VALUE	1	4	14	7
BINARY	0001	0100	1110	0111
BIT 3	0	0	1	0

And final move to get ordered array:

ELEMENT	4	3	1	2
VALUE	1	4	7	14
BINARY	0001	0100	0111	1110
BIT 3	0	0	0	1

Eventually values are now sorted. This is the basic description of parallel radix sort. The algorithm runs on GPU in 1 blocks and 32 threads which is equal to constant array size.

```
#define WSIZE 32
#define LOOPS 1
#define UPPER_BIT 10
#define LOWER_BIT 0

__device__ unsigned int ddata[WSIZE];

__global__ void parallelRadix() {

    // This data in shared memory
    __shared__ volatile unsigned int sdata[WSIZE * 2];

    // Load from global into shared variable
    sdata[threadIdx.x] = ddata[threadIdx.x];

    unsigned int bitmask = 1 << LOWER_BIT;
    unsigned int offset = 0;
    // -1, -2, -4, -8, -16, -32, -64, -128, -256,...
    unsigned int thrmask = 0xFFFFFFFF U << threadIdx.x;
    unsigned int mypos; //  For each LSB to MSB

    for (int i = LOWER_BIT; i <= UPPER_BIT; i++) {
        unsigned int mydata = sdata[((WSIZE - 1) - threadIdx.x) + offset];
        unsigned int mybit = mydata & bitmask;
        // Get population of ones and zeroes
        unsigned int ones = __ballot(mybit);
        unsigned int zeroes = ~ones;
        offset ^= WSIZE; // Switch ping-pong buffers

        // Do zeroes, then ones
        if (!mybit) {
            mypos = __popc(zeroes & thrmask);
        } else { // Threads with a one bit
            // Get my position in ping-pong buffer
            mypos = __popc(zeroes) + __popc(ones & thrmask);
```

```
        } // Move to buffer
        sdata[mypos - 1 + offset] = mydata;
        // Repeat for next bit
        bitmask <<= 1;
    }
    // Put results to global
    ddata[threadIdx.x] = sdata[threadIdx.x + offset];
}

int main() {

    unsigned int hdata[WSIZE];
    float totalTime = 0;

    for (int lcount = 0; lcount < LOOPS; lcount++) {
        // Array elements have value in range of 1024
        unsigned int range = 1 U << UPPER_BIT;
        // Fill array with random elements
        // Range = 1024
        for (int i = 0; i < WSIZE; i++) {
            hdata[i] = rand() % range;
        } // Copy data from host to device

        cudaMemcpyToSymbol(ddata, hdata, WSIZE * sizeof(unsigned int));
        // Execution time measurement, that point starts the clock
        high_resolution_clock::time_point t1 = high_resolution_clock::now();
        parallelRadix << < 1, WSIZE >>> ();
        // Make kernel function synchronous
        cudaDeviceSynchronize();
        // Execution time measurement, that point stops the clock
        high_resolution_clock::time_point t2 = high_resolution_clock::now();
        // Execution time measurement, that is the result
        auto duration = duration_cast < milliseconds > (t2 - t1).count();
        // Summination of each loops' execution time
        totalTime += duration;
        // Copy data from device to host
        cudaMemcpyFromSymbol(hdata, ddata, WSIZE * sizeof(unsigned int));
    }

    printf("Parallel Radix Sort:\n");
    printf("Array size = %d\n", WSIZE * LOOPS);
    printf("Time elapsed = %gmilliseconds\n", totalTime);

    return 0;
}
```

As a result, this algorithm performs parallel radix sorting on given random array on GPU using 1 blocks and 32 threads for each loop, and then calculates execution time in milliseconds.

3. PERFORMANCE ANALYSIS

Several data inputs end up with various results in radix algorithms. Increasing the array size over time and deducing from examination of the result data is determined the outcome.

3.1 Test Results

The test result shows that sorting times of arrays that filled with random values between 0 and 1024.

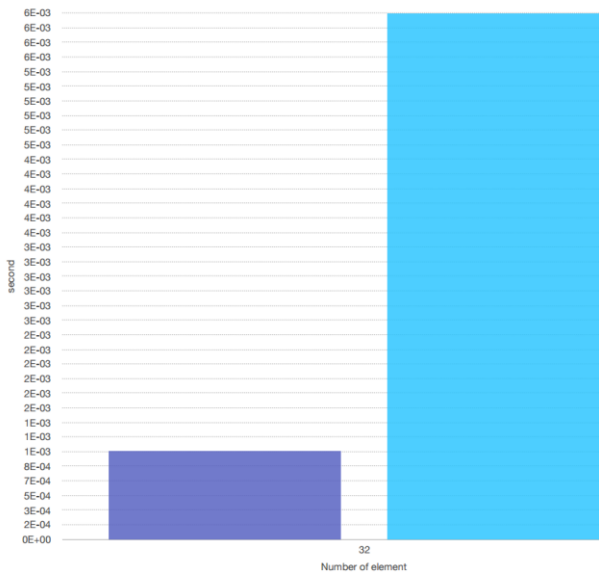
We've also tested arrays that have values linearly increasing, i.e. sorted values, and the result was same for parallel radix sort algorithm because even if algorithm sorts different arrays that consist of different values it always do the same thing. Compare the binary equivalent of the numbers, then

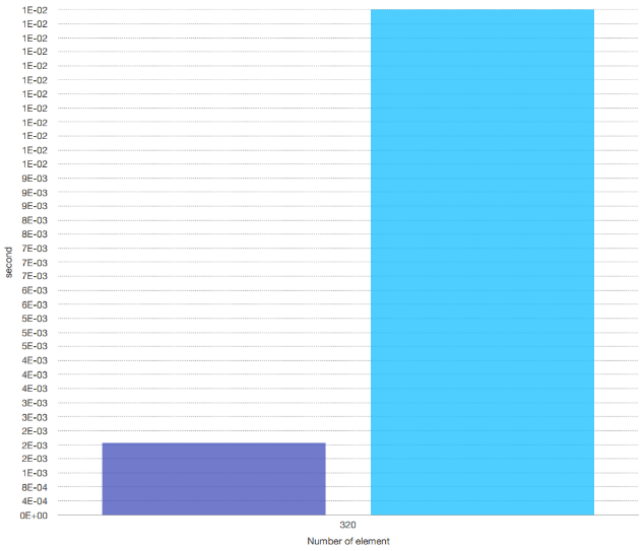
increase the index which separated for this value and etc. Although, the serial code was more efficient, took less time, with the sorted arrays because of little differences between our serial and parallel algorithms. Using sorted arrays reduced the assignments for serial radix algorithm. Since giving sorted arrays as an input does not give evident conception for our goal, we have not save those results in the graphs.

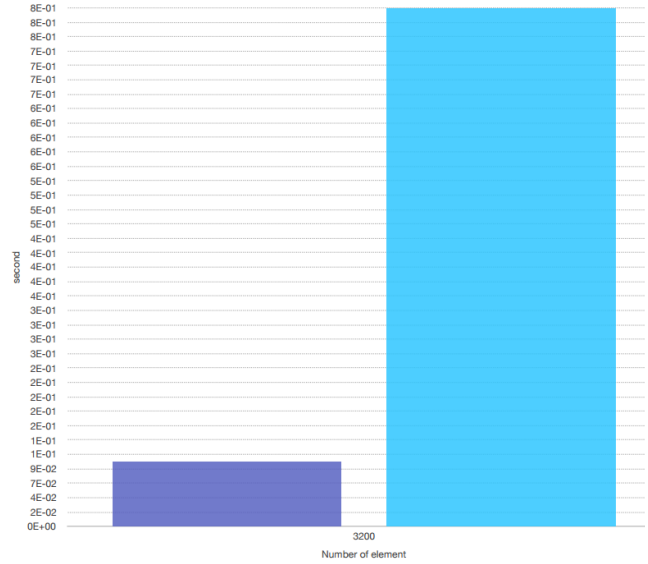
The results below are obtained after very long test periods especially for ones that have very large array sizes.

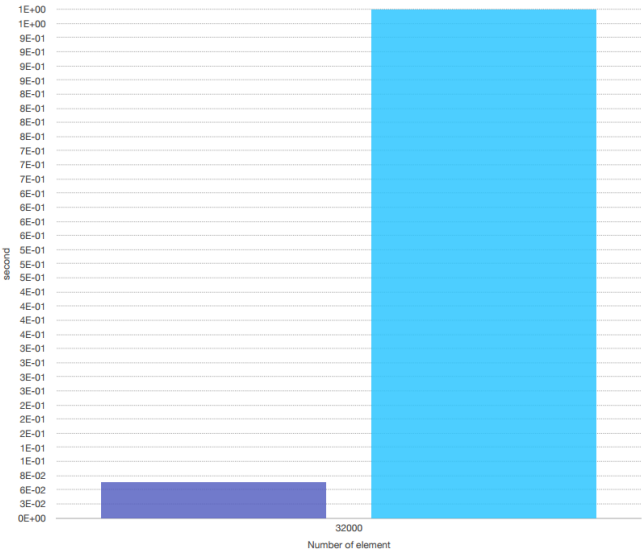
3.2 Graphical Representation of Result

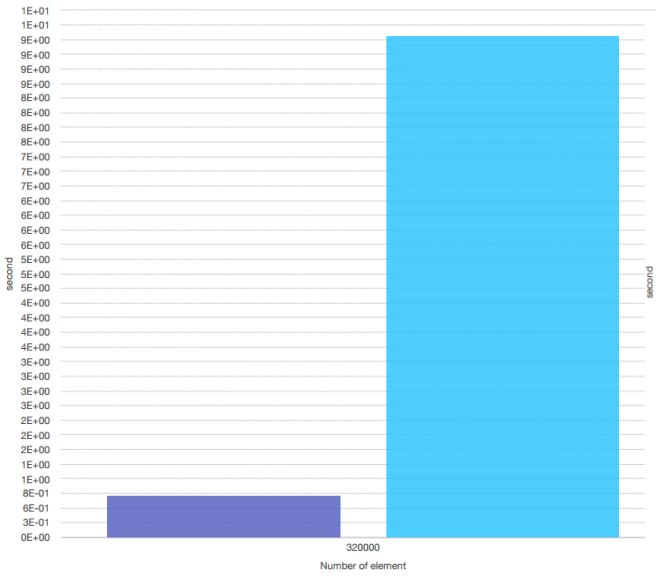
The results gained using GTX860M architecture depicted by following graphics:

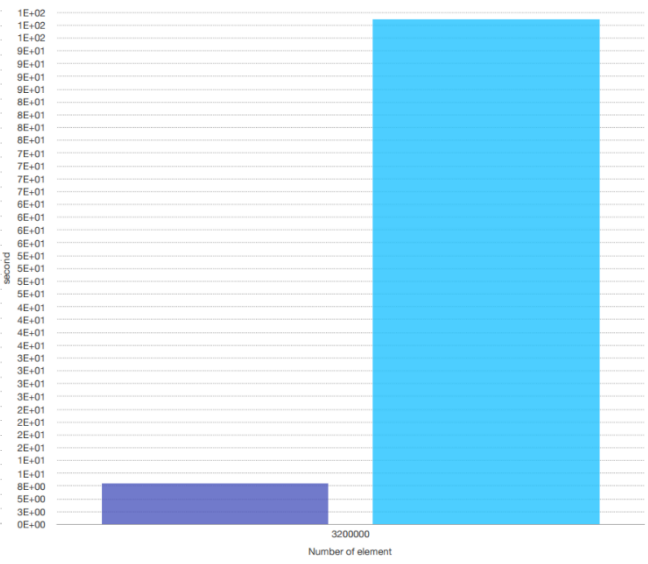

second
6E-03
5E-03
4E-03
3E-03
2E-03
1E-03
8E-04
7E-04
5E-04
3E-04
2E-04
0E+00
32
Number of element

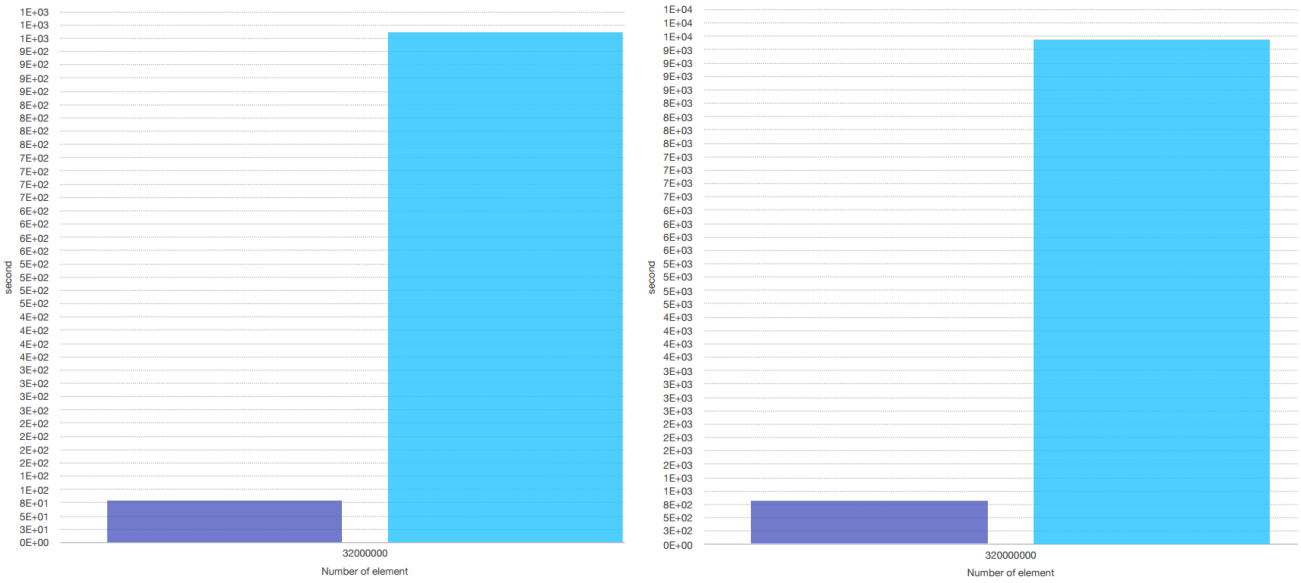

second
1E-02
9E-03
8E-03
7E-03
6E-03
5E-03
4E-03
3E-03
2E-03
1E-03
8E-04
4E-04
0E+00
320
Number of element


second
8E-01
7E-01
6E-01
5E-01
4E-01
3E-01
2E-01
1E-01
9E-02
7E-02
4E-02
2E-02
0E+00
3200
Number of element


second
1E+00
9E-01
8E-01
7E-01
6E-01
5E-01
4E-01
3E-01
2E-01
1E-01
8E-02
6E-02
3E-02
0E+00
32000
Number of element

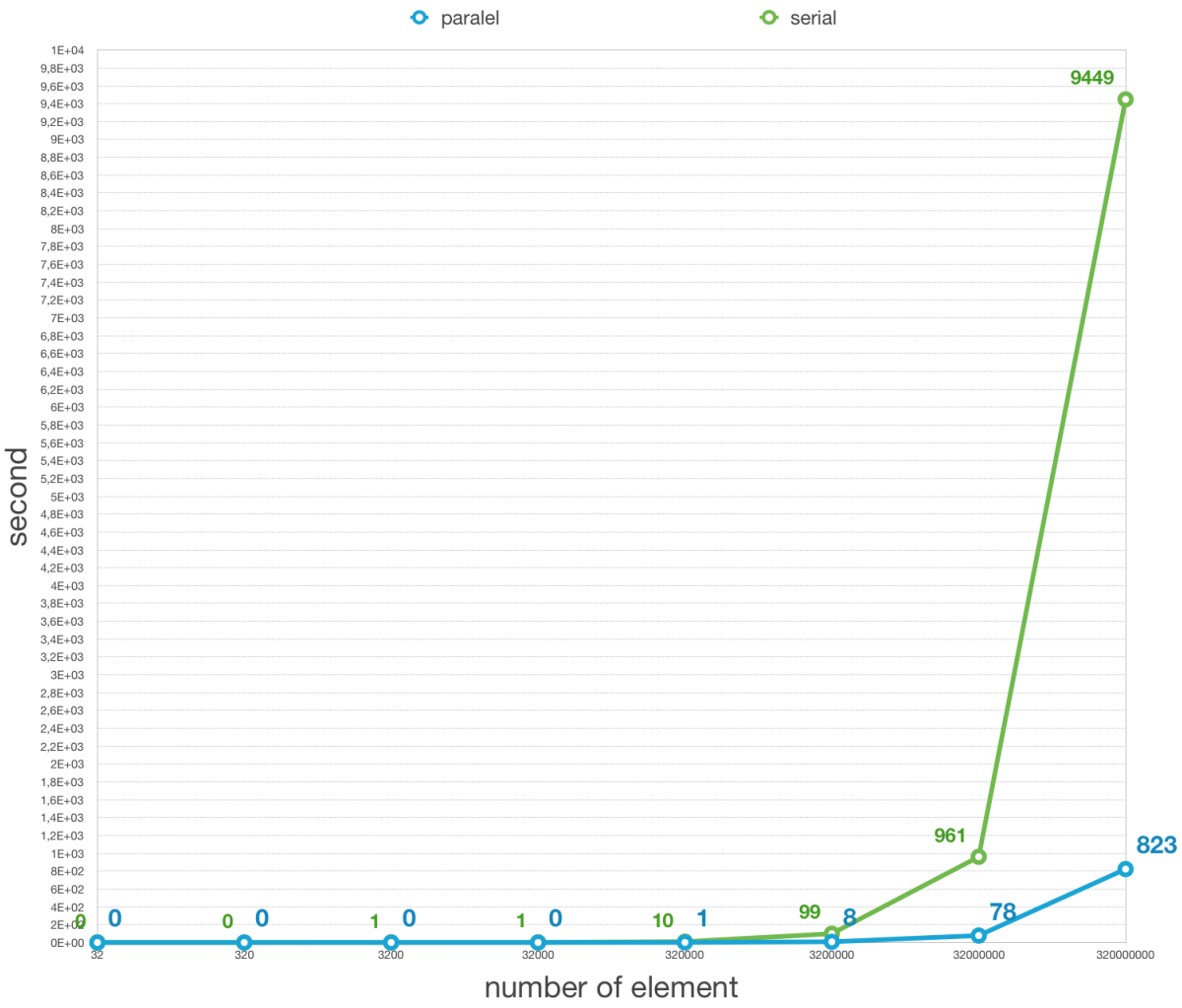

second
1E+01
9E+00
8E+00
7E+00
6E+00
5E+00
4E+00
3E+00
2E+00
1E+00
8E-01
6E-01
3E-01
0E+00
320000
Number of element


second
1E+02
9E+01
8E+01
7E+01
6E+01
5E+01
4E+01
3E+01
2E+01
1E+01
8E+00
5E+00
3E+00
0E+00
3200000
Number of element



4. CONCLUSION

We tested performance of parallel and serial radix sort algorithms on GPU. Test result comparison showed us that performance is affected mainly by two things: array size and parallelism, i.e. parallel thread counts.



The final results for different array sizes proved that parallel computing is much more efficient than serial one.

REFERENCES

[1] NVIDIA - What is CUDA?

[2] GPU Gems – Chapter 39. Parallel Prefix Sum (Scan) with CUDA

[3] Fast Parallel Sorting Algorithms on GPUs