SOLUTIONS FOR OPTIMIZING THE RADIX SORT ALGORITHMIC FUNCTION USING THE COMPUTE UNIFIED DEVICE ARCHITECTURE

Alexandru Pîrjan[1]
Dana-Mihaela Petroşanu[2]

ABSTRACT

In this paper, we have researched and developed solutions for optimizing the radix sort algorithmic function using the Compute Unified Device Architecture (CUDA). The radix sort is a common parallel primitive, an essential building block for many data processing algorithms, whose optimization improves the performance of a wide class of parallel algorithms useful in data processing. A particular interest in our research was to develop solutions for optimizing the radix sort algorithmic function that offers optimal solutions over an entire range of CUDA enabled GPUs: Tesla GT200, Fermi GF100 and the latest Kepler GK104 architecture, released on March 2012. In order to confirm the utility of the developed optimization solutions, we have extensively benchmarked and evaluated the performance of the radix sort algorithmic function in CUDA.

Keywords: parallel processing, CUDA, GK104, threads, shared memory.

1. INTRODUCTION

The radix sort is an important algorithmic function, a primitive building block, useful in many algorithms that benefit from massive parallelism [1], [2], [3], [4]. Its huge importance has led to the development of efficient sorting algorithms on a variety of parallel architectures. A wide class of data processing algorithms uses sorting routines, thus creating both an opportunity and a necessity to optimize the sorting operations. The sorting operations are essential in geographic information systems, in computer graphics, in database systems, as they represent the basis for building spatial data structures [5]. Also, efficient sorting routines are of paramount importance in problems regarding web mining [6], in implementing algorithms such as parallel programming models based on MapReduce [7], in sparse matrix multiplication or issues regarding network security improvement [8]. Therefore, it is very important to implement efficient sorting routines on various programming platforms. The continuous evolution of the computer hardware architectures provides an increased computational processing power and therefore it imposes the necessity of the on-going development of efficient sorting techniques on these new architectures [9].

Of all the sorting algorithms, the radix sort is among the oldest, probably the best known algorithm and its sequential variants are one of the most efficient variants in sorting small

[1] Ph D Candidate, Faculty of Computer Science for Business Management, Romanian-American University, 1B, Expozitiei Blvd., district 1, code 012101, Bucharest, Romania, e-mail: alex@pirjan.com
[2] PhD, Department of Mathematics-Informatics I, University Politehnica of Bucharest, 313, Splaiul Independentei, district 6, code 060042, Bucharest, Romania, e-mail: danap@mathem.pub.ro

keys. The elements that are being processed by the radix sort algorithm are usually called "keys" and they can exist independently or can be associated with other data. The algorithm considers the keys as k-digit numbers and sorts one digit at a time. It sorts the data by grouping the individual digits keys that share the same significant position and value. There are two types of radix sort:

- LSD - least significant digit, that processes the elements starting from the least significant digit and moves towards the most significant one;
- MSD - most significant digit, that processes the elements starting from the most significant digit and moves towards the least significant one.

Using specific information related to the nature of the keys, the algorithm performs their sorting. In the case of LSD sorting, we consider $a = (a_1, a_2, \dots, a_n)$ and $b = (b_1, b_2, \dots, b_n)$ two keys, a_i, b_i positive integers, for each $i = 1, \dots, n$. The key a is lower than the key b if there is a natural number k between 1 and n such that $a_1 = b_1$, $a_2 = b_2$,..., but $a_k < b_k$. In this case, the radix sort first takes into account the least significant field, meaning the n-th digit, then the $(n-1)$–th and so on, until it reaches the most significant field, the first one on the left.

For example, when using the ascending LSD radix sorting of the keys 8788, 23469, 6257 and 13, the algorithm uses the following steps:

- the keys are written in the same format (corresponding to the number with the most digits), ie: $08788, 23469, 06257$ and 00013;
- the least significant field is considered the one corresponding to the least significant digit and since we have chosen the LSD sorting, the keys are ordered starting from the last digit, so $3 < 7 < 8 < 9$ determines the order: $00013 < 06257 < 08788 < 23469$;
- passing to the next field, represented by the second digit from right to left, the digits' order is $1 < 5 < 6 < 8$, thus obtaining a reordering: $00013 < 06257 < 23469 < 08788$;
- at the next digit, the order is $0 < 2 < 4 < 7$ and thus the above obtained ordering is kept: $00013 < 06257 < 23469 < 08788$;
- at the next step, $0 < 3 < 6 < 8$, and thus is obtained the reordering: $00013 < 23469 < 06257 < 08788$;
- in the last step, $0 < 2$, and we obtain the order: $00013 < 06257 < 08788 < 23469$.

In conclusion, the ascending LSD radix sort of the initial key sequence $8788, 23469, 6257,13$ is: $00013 < 06257 < 08788 < 23469$.

2. DESIGNING AN EFFICIENT RADIX SORT ALGORITHMIC FUNCTION IN CUDA

The graphic processing units (GPUs), based on the Compute Unified Device Architecture (CUDA), provide an increased computational capacity, a major potential in developing high

performance sorting routines. The GPUs excel in processing large data sets that often require sorting or partitioning operations. Multi-GPU configurations in which data and tasks are distributed across multiple nodes offer very important solutions in situations when multiple operations must be performed on the data set: sorting, reduction, compaction and partitioning.

The efficiency of the sorting routines influences the applications' performance, significantly improving the memory bandwidth in those problems that involve pointer addressing or lookup tables. Sorting routines are parallelizable and involve fine computational granularity that facilitates their implementation on the GPUs, but raises significant challenges compared to classic multiple central processing units (CPUs) implementations. The implementation's technical difficulties are mainly generated by the lack of hardware synchronization mechanisms for fine-grain data or the lack of large size memory buffers (cache) that facilitate the uniform access to the memory.

The development of efficient sorting algorithmic functions on GPUs is a solution for a wide range of problems using sorting routines: binary search, finding the closest pair, determining the uniqueness of an item, determining the k-th element of a sequence of sorted elements, identifying outliers in a sequence. Efficient sorting routines are also useful for large-scale problems in distributed systems (eg. crossing graph algorithms in clusters and supercomputers) [10].

Sorting is a basic functional block used in kD trees, octal trees, tree structures on sets of geometric objects. These structures are involved in modelling physical systems such as molecular dynamics, methods for computing the trajectory of a wave or of a particle in a system, collision detection [11], photon mapping, modelling three-dimensional objects, fluid mechanics simulations, n-body systems. Other sorting applications based on GPU units are related to image rendering, including shadow modelling, transparency, graphic rendering, animation and texture compression. GPU radix sorting was implemented in parallel hashing tables, accelerating databases, data processing engines and even in video games engines.

We have designed, depending on the GPU's architecture, a self-adjustable and self-configurable parallel radix sort algorithmic function, that allocates resources like the dimension of the grid blocks, of the blocks of threads and processed elements per thread. We have obtained a solution that has a high level of performance on the most important CUDA enabled graphic processing units (Tesla GT200, Fermi GF100, Kepler GK104 launched in 2008, 2010 and in 2012).

As GPUs use thousands of concurrent execution threads to process tasks, the efficient designing of algorithms running on these processors must take into account the parallelism within these threads. Thus, algorithms must be designed as to properly balance the number of tasks and their granularity.

In order to design at the block level an efficient radix sort algorithmic function in CUDA, we have used the shared memory provided by the GPU's architecture, harnessing its increased speed and reduced latency. We have used the GPU's memory to locally order data for improving the coherence, thus optimizing the memory bandwidth's usage in the radix sort algorithmic function.

In the following, we consider the case when elements are being processed from the least significant digit to the most significant one (LSD). Our research refers to sorting sequences of key-value pairs. In each step of the radix sort, we have used a counting sort or a bucket sort [1]. Each digit is a sequence of bits in the key. For each key's digit, we have computed the number of keys whose digits are lower, plus the number of keys with the same digits that appear earlier in the sequence. This is the index at which the output element will be written, called the element's rank. After having computed each element's rank, the step ends by writing each element in its appropriate position in the output vector. Sorting each digit from the least significant to the most significant one will result in the sorted sequence after having processed all the sorting steps.

The radix sort parallelization is achieved by reducing the counting sort technique used in each step at a parallel prefix sum [12]. This is one of the main optimization solutions that we have implemented in order to improve the performance of the parallel radix sort function in CUDA, because we have used the parallel prefix sum algorithmic function that we have already optimized, benefiting from its high level of performance [13].

The easiest and most intuitive technique based on the parallel prefix sum is to sort in each moment one of the keys' bits and this represents a splitting operation [12]. This approach has the disadvantage that it is not efficient when data arrays are stored in the external DRAM memory [14].

For 32-bit keys, 32 reordering operations are required in order to sort the whole sequence. Data transfers from/to external memory consume considerable computational resources, so it is preferable to avoid these data transfers as much as possible. A natural way to reduce the number of reordering operations is to take into account more than 1 bit per each crossing. In order to do this efficiently, each block of threads builds a histogram that counts the occurrences of numbers in its allocated data portion and finally they are processed using the parallel prefix sum [12]. Le Grand and He have implemented in CUDA this kind of techniques [15], [16]. Although they are more efficient than the one that sorts keys one bit at a time, these techniques also have disadvantages, as the external memory bandwidth is not efficiently used. Although the global memory is less accessed, reordering operations are still needed to be applied on the sequence, through which consecutive elements of the sequence need to be written in different memory areas. This procedure causes significant losses, due to the management of the available bandwidth.

In order to optimize the radix sort, we have first divided the sequence into tiles that have been assigned to a number of p thread blocks. Thus, we have efficiently used the memory bandwidth reducing the number of reordering operations in the global memory, allocating more than one bit for each crossing of the sequence, maximizing the reordering coherence,

using the GPU's shared memory in order to locally sort data. Thus, reordered data is written in the GPU's memory that is much faster than the external one (with about 2 magnitude orders).

We have implemented the radix sort algorithmic function in four steps. Between the steps, the global synchronization is required, but this is not possible in CUDA. Therefore, each step corresponds to a separate call of the parallel kernel function [12].

The steps of the radix sort algorithmic function are:

Step 1. Each thread block loads and sorts its data portion in the GPU's memory using b iterations per bit.

Step 2. Each block of threads writes its 2^b histogram entries and its part of sorted data in the global memory.

Step 3. A parallel prefix sum is performed over the $p \cdot 2^b$ histograms to compute the output elements, using the GPU optimized parallel prefix sum described in [13].

Step 4. Using the results obtained in the Step 3, each thread block copies its elements in their correct output position.

In designing the radix sort algorithmic function, we have used $t = 512$ thread blocks, a number chosen after numerous experimental tests so that the function provides maximum performance. Regarding the number of processed elements per thread, the intuitive version is to assign one element per each thread, but handling a larger number of elements per thread is actually more efficient. We have chosen the option to process 4 elements per thread or 2048 elements per block. In **Step3**, the radix sort algorithm processes 8 elements per thread for the GTX 280 GPU, 16 for the GTX 480 GPU and 32 for the GTX 680 GPU, as we have called the GPU optimized parallel prefix sum function [13].

The sequential processing of several independent computations within each thread improves the overall efficiency and offers possibilities to reduce the latency. Since each block processes a portion of 2048 elements, we have chosen the total number of blocks needed to process the entire sequence (having n elements) as follows:

$$p = \begin{cases} \left[\frac{n}{2048}\right] + 1, if\ n\ is\ not\ a\ power\ of\ 2\ or\ if\ n < 2048 \\ \frac{n}{2048}, if\ n\ \ is\ a\ power\ of\ 2\ and\ n \geq\ 2048 \end{cases}.$$

The choice of b (the number of iterations per bit) is determined by two factors. We have used a pre-sorting of each data portion in **Step 1** in order to limit the reordering operations in **Step 4** at only 2^b neighboring blocks. Higher values of b lead to a decreased coherence of the reordering. On the other hand, too small values of b lead to a large number of data rearrangement operations in the global memory.

Each thread block processes a number of elements proportionally to the number of the threads per block. We have determined empirically that choosing $b = 4$, we obtain the best balance between these factors and the best overall performance [9].

We have designed the radix sort algorithmic function to accept the following data types for the keys: integer, unsigned integer, float, double, long long, unsigned long long. Our solution offers the possibility to first set the parameters of a configuration structure that are afterwards passed as calling parameters along with the input vector containing the keys, with the input vector containing the values and with the number of elements.

In the following, we present a series of optimization solutions that we have used in developing the radix sort algorithmic function using the Compute Unified Device Architecture.

3. SOLUTIONS FOR OPTIMIZING THE PERFORMANCE OF THE RADIX SORT ALGORITHMIC FUNCTION IN CUDA

In order to obtain an optimized version of the parallel radix sort algorithmic function, we have developed a series of solutions for optimizing the performance of this function in CUDA.

Solution 1 – using the shared memory. In order to obtain an efficient parallel radix sort algorithmic function, we have used the shared memory provided by the latest generations of CUDA GPUs. In this way, we have locally reordered data to improve the coherence that has optimized the bandwidth usage in the radix sort algorithmic function, taking advantage of the low response time and improved memory bandwidth offered by this type of memory.

Solution 2 - balancing tasks using the fine-grain parallelism. In order to successfully use the thousands of parallel threads provided by the CUDA architecture, in designing the parallel radix sort function, we have implemented the fine-grain parallelism that results in a proper balance of the tasks and their granularity. The tasks' decomposition generates a corresponding computational load so that the potential of all the computational cores is fully utilized during the execution.

Solution 3 - calling the optimized parallel prefix sum function developed in CUDA. In designing an efficient radix sort algorithmic function, we have parallelized the counting sort technique used in the Step 3 of the algorithm by reducing it to a parallel prefix sum. Therefore, in the **Step 3** of the radix sort algorithmic function, we have used the optimized parallel prefix sum algorithmic function developed in CUDA [13]. This is the most important optimization technique applied in designing the radix sort algorithmic function and therefore we have obtained a performance improvement of up to 14% compared to the case when processing this step sequentially on a central processing unit.

Solution 4 – the sequential processing of several independent computations within each execution thread. By applying this solution in designing the parallel radix sort algorithmic function, we have improved the overall efficiency of parallel computations, thus providing more opportunities to reduce the latency.

Solution 5 - minimizing data transfers between the host and the device. Given that data transfers between the host and the device consume considerable computational resources,

we have designed the radix sort algorithmic function so that data transfers between the host and the device are minimal.

Solution 6 – optimizing the memory bandwidth. To design an efficient radix sort algorithmic function, we have first divided the input data into smaller fragments allocated to the thread blocks. The main purpose for which we have made this partitioning is the efficient usage of the memory bandwidth. Therefore, we have obtained:

- the reduction of the global memory calls, allocating more than one bit to each crossing of the sequence
- maximizing the reordering coherence using the GPU's shared memory in order to locally sort data, that replaces the writing of reordered data from the external memory with writing reordered data in the GPU's memory, that is about two orders of magnitude faster.

The designing techniques that we have used in this section, in the context of running on GPUs, are also applicable to other graphic processors with more execution cores than the ones we had available in our benchmarks.

In the following, we analyze the performance of the radix sort algorithmic function that we have developed and optimized in CUDA using the above presented solutions.

4. THE EXPERIMENTAL RESULTS AND THE PERFORMANCE ANALYSIS OF THE RADIX SORT ALGORITHMIC FUNCTION IN CUDA

For analyzing the performance of the parallel radix sort algorithmic function, we have used:

- the Windows 7 64-bit operating system;
- the Intel i7-2600K CPU, overclocked at 4.6 GHz;
- 2x4 GB DDR3 dual channel RAM memory, running at a frequency of 1333 MHz;
- the GeForce GTX 280 from the GT200 architecture, the GTX 480 from the Fermi architecture and the GTX 680 from the Kepler architecture
- the CUDA toolkit 4.1 for programming the GPUs
- for the GTX 280 and GTX 480 GPUs, the driver version 270.81
- for the GTX 680 GPU, the driver version 300.1.

The complexity of each developed application influences the transfer times between the CPU and the GPU. This is the reason why we have measured only the time corresponding to the data processing and we have not included the transfer times. In this context we have used the CUDA API (application programming interface) in order to measure the average execution time that the GPU spends for processing the data during the execution of the radix sort algorithmic function. Other methods, like those based on the operating system's timers, have the disadvantage of including in their measurements variations and latency

from different sources that would have compromised the results. In addition, computations can be asynchronously performed on the host while the GPU kernel runs and the only way to measure the necessary time for the host computations is to use the CPU or the operating system timing mechanism. In this way, we get a reliable measurement of the execution time for computing the above described radix sort algorithmic function [13].

We have first evaluated the execution times obtained by applying the radix sort algorithmic function on key-value pairs of various sizes (35-60,000,000) and float type elements. Using a random number generator, we have obtained the input data, consisting of 26 vectors of key-value pairs. The keys are of float data type, allocated in an ascending order. In order to obtain accurate results, we have computed the average of 10,000 iterations of the parallel radix sort algorithmic function run on the three GPUs and on the CPU.

For each input vector's size and each of the four processors, we have computed the execution time and the memory bandwidth (**Table 1, Table 2**). We have measured the execution time in milliseconds (ms) and the memory bandwidth in GB/s.

Table 1. The experimental results for the radix sort algorithmic function (execution time)

Test No.	Number of elements	Execution time (ms)			
		CPU	GTX 280	GTX 480	GTX 680
1	35	0.000308	3.271094	0.911673	0.852586
2	128	0.000519	3.237406	0.980564	0.806586
3	256	0.001008	3.331148	0.943952	0.781448
4	260	0.001373	3.323932	0.945672	0.762190
5	512	0.001992	3.286394	1.039153	0.826186
6	1000	0.004389	3.401922	0.95746	0.801103
7	1024	0.004836	3.348141	1.103351	0.838858
8	1030	0.003997	5.159959	1.28986	1.077253
9	32768	0.144484	6.353939	1.197392	0.952060
10	45555	0.197665	6.469057	1.188689	1.147154
11	65536	0.305246	6.507369	1.179774	1.068354
12	131072	0.632163	6.684911	1.588619	1.292114
13	262144	1.348589	8.600079	1.940587	1.480067
14	500111	2.696119	9.268106	2.310846	1.407021
15	524288	2.738219	9.308323	2.264077	1.357708
16	1048555	8.196775	10.551018	3.057081	1.960797
17	1048576	8.586647	10.574672	2.92803	1.929766
18	1048581	8.622486	10.573973	2.981098	1.964001
19	2097152	28.618340	13.03183	4.3827	2.867482
20	2097999	28.238203	13.059888	4.41974	2.968492
21	4194334	65.020836	17.939398	7.238373	4.911704
22	8388600	137.395126	27.824905	12.848075	9.042197
23	16000000	269.111176	47.711851	24.198442	15.201030
24	32000000	549.668030	85.512920	45.815244	29.554805

25	48000000	827.618652	119.112119	66.763672	43.802071
26	60000000	1055.027710	163.922578	83.702705	54.588300
Total execution time – 10.000 tests (h)		8.317	1.670	0.773	0.512
The system's power (kW)		0.198	0.306	0.358	0.307
Total energy consumption (kWh)		1.647	0.511	0.277	0.157
The GPU's consumption compared to the CPU's			3 times lower	5 times lower	10 times lower

Table 2. The experimental results for the radix sort algorithmic function (memory bandwidth)

Test No.	Number of elements	The memory bandwidth (GB/s)			
		CPU	GTX 280	GTX 480	GTX 680
1	35	0.4545	0.0001	0.0002	0.0002
2	128	0.9865	0.0002	0.0005	0.0006
3	256	1.0159	0.0003	0.0011	0.0013
4	260	0.7575	0.0003	0.0011	0.0014
5	512	1.0281	0.0006	0.0020	0.0025
6	1000	0.9114	0.0012	0.0042	0.0050
7	1024	0.8470	0.0012	0.0037	0.0049
8	1030	1.0308	0.0008	0.0032	0.0038
9	32768	0.9072	0.0206	0.1095	0.1377
10	45555	0.9219	0.0282	0.1533	0.1588
11	65536	0.8588	0.0403	0.2222	0.2454
12	131072	0.8294	0.0784	0.3300	0.4058
13	262144	0.7775	0.1219	0.5403	0.7085
14	500111	0.7420	0.2158	0.8657	1.4218
15	524288	0.7659	0.2253	0.9263	1.5446
16	1048555	0.5117	0.3975	1.3720	2.1390
17	1048576	0.4885	0.3966	1.4325	2.1735
18	1048581	0.4864	0.3967	1.4070	2.1356
19	2097152	0.2931	0.6437	1.9140	2.9254
20	2097999	0.2972	0.6426	1.8988	2.8270
21	4194334	0.2580	0.9352	2.3178	3.4158
22	8388600	0.2442	1.2059	2.6116	3.7109
23	16000000	0.2378	1.3414	2.6448	4.2102
24	32000000	0.2329	1.4968	2.7938	4.3309
25	48000000	0.2320	1.6119	2.8758	4.3834
26	60000000	0.2275	1.4641	2.8673	4.3965

For each of the 26 dimensions of the input vectors we have computed the total execution time of the 10,000 iterations. Using an energy consumption meter device (Voltcraft Energy Logger 4000) we have measured the system's power (kW) and the total energy consumption

in each of the four analyzed cases (running the tests on the CPU and on the three GPUs). Analysing the obtained results, we have noticed that running the test suite on the GTX 280 GPU, determines a system power consumption 3 times lower than when running the suite on the central processing unit. When running the suite on the GTX 480, the power consumption is 5 times lower, on the GTX 680 the consumption is 10 times lower than when running the benchmark suite on the central processing unit. The obtained experimental results reflect an increased economic efficiency through the reduced energy consumption and highlight a high level of performance through the reduced execution times when running the test suite of the parallel radix sort algorithmic function on graphics processors compared to the sequential version run on the central processing unit.

We have first analysed the obtained experimental results by running the radix sort algorithmic function, regarding the execution time and memory bandwidth, corresponding to each of the four processors, when the input array has a relatively low dimension (35-262,144 elements). By analysing and comparing the experimental results, we have found that for an input vector containing a number of 35-262,144 elements, the best results (lower execution time, higher bandwidth) are obtained when running the radix sort function on the central processing unit, then on GTX 680, GTX 480 and GTX 280. This happens because in this case it has not been generated an enough computational load in order to use the huge parallel processing capacity of the GPUs (**Figure 1, Figure 2**).

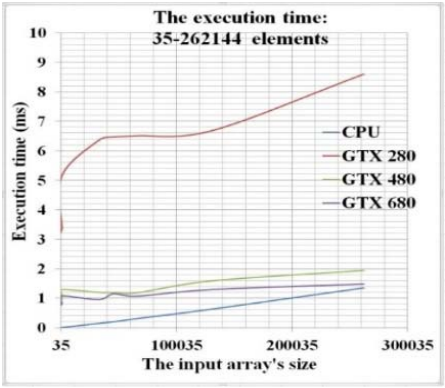


Figure 1. The execution time: 35-262,144 elements of the input array

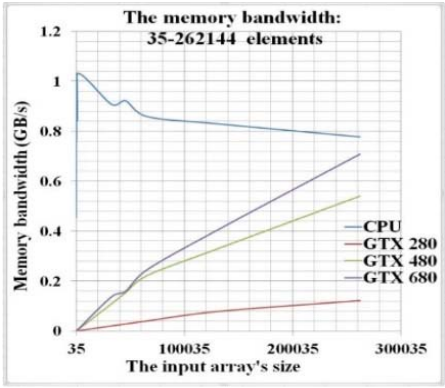


Figure 2. The memory bandwidth: 35-262,144 elements of the input array

When the number of key-value pairs of the input vector is between 500,111 and 60,000,000 elements, the best results (lower execution time, higher bandwidth) are obtained when running the radix sort algorithmic function on the GTX 680, then on the GTX 480, on the GTX 280 and on the CPU. This time, it has been generated a sufficient computational load that fully employs the huge parallel processing capacity of the GPUs (**Figure 3, Figure 4**).

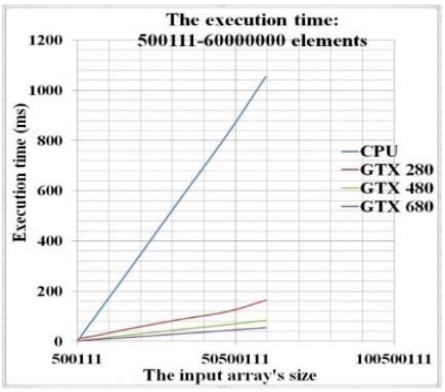


Figure 3. The execution time: 500,111-60,000,000 elements of the input array

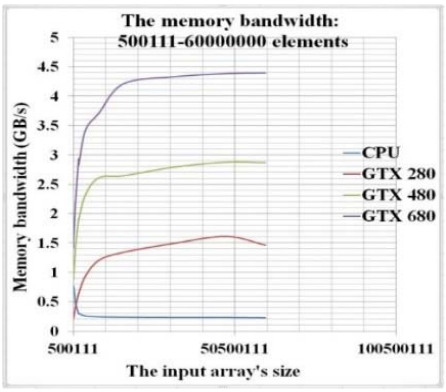
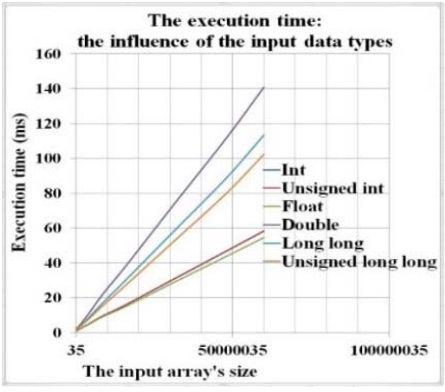


Figure 4. The memory bandwidth: 500,111-60,000,000 elements of the input array

In the next set of tests, we have evaluated the influence of the generated data types on the performance of our radix sort algorithmic function, run on the GTX 680 GPU. We have designed the algorithm to allow the selection of the data type of the input array components, which can be integer, unsigned integer, float, double, long long or unsigned long long. We have first highlighted the execution time variation depending on the input vector's data types (**Figure 5**). For measuring accurate results, we have computed the average of 10,000 iterations. We have obtained similar performance in the cases when the keys are of integer, unsigned integer or float data type, the execution time ranging from 0.798116 ms to 58.405197 ms. If the keys are of double, long long or unsigned long long type, the execution times are generally higher than in the three cases mentioned above, ranging from 1.42692 ms to 140.848443 ms. This is justified taking into account the amount of necessary memory required by the analysed data types. We have then highlighted the bandwidth variation depending on the input vector's data types (**Figure 6**). Regarding the bandwidth, we have noticed that the performance is comparable in all the six considered cases, reflecting the efficiency of the optimization solutions applied to the parallel radix sort algorithmic function that provides a high data processing speed, regardless of the considered data type.



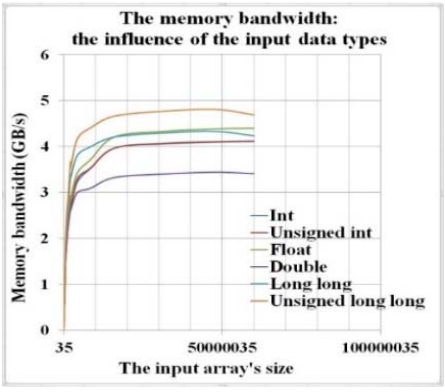


Figure 5. The influence of the data typeson the execution time

Figure 6. The influence of the data types on the memory bandwidth

The above described experimental results emphasize that our solutions for optimizing the parallel radix sort algorithmic function are efficient and thus, the function provides optimum results in different situations, proving to be useful in a wide range of data processing applications and algorithms.

5. PERSONAL CONTRIBUTIONS AND CONCLUSIONS

In this paper, we have analysed, developed and designed the radix sort algorithmic function, depicting first the algorithm's steps, then its CUDA implementation and a series of optimization solutions for improving the performance of the radix sort algorithmic function. These optimization solutions are:

- using the shared memory;
- balancing tasks using the fine-grain parallelism;
- calling the optimized parallel prefix sum function developed in CUDA;
- the sequential processing of several independent computations within each execution thread;
- minimizing data transfers between the host and the device;
- optimizing the memory bandwidth.

We have analyzed the performance of the radix sort algorithmic function in CUDA, using a series of experimental tests and we have compared it with an alternative approach run on the central processing unit. In order to compute the average execution time of the graphic processing unit, we have used the CUDA application programming interface (API). After having analyzed the experimental results obtained by using the solutions for optimizing the performance of the radix sort algorithmic function, we have noticed the following:

- Measuring the total energy consumption in each of the four analysed cases (when running the tests on the CPU and on the three GPUs), we have noticed that when running the radix sort algorithmic function on the GTX 280 GPU, the system consumes 3 times less energy than when the function is run on the central processing unit i7-2600K. For the GTX 480, the power consumption is 5 times lower and for the GTX 680 is 10 times lower than for the i7-2600K central processing unit.
- After running the test suite, we have recorded a high level of performance reflected by the low execution times and we have obtained an increased level of economic efficiency confirmed by the reduced energy consumption of the radix sort algorithmic function, run on graphics processors, compared to the sequential version run on the central processing unit.
- When running the radix sort algorithmic function on the graphics processing units GTX 280, GTX 480, GTX 680 and on the central processing unit i7-2600K, for a number of key-value pairs of the input vector between 35 and 262,144 elements, the best execution time was recorded on the central processing unit, then on the

GTX 680 GPU, because the amount of data does not generate an enough computational load in order to fully use the huge parallel processing capacity of the GPUs.

- We have obtained improved execution times and larger bandwidth when sorting large dimension arrays of float data types on the GTX 680 graphics processing unit (500,111-60,000,000 elements) than when using the traditional central processing unit. We have recorded on the GTX 680 an improvement of up to 19.32x in both execution time (54.5883 ms vs 1055.02771 ms) and bandwidth (0.2275 GB/s vs 4.3965 GB/s) compared to the i7-2600K processor. This time the huge parallel processing capacity of the GTX 680 GPU has been fully used since there were required much more computations than in the previous cases. Based on these results, we have concluded that in order to obtain the best performance of the radix sort algorithmic function, it is necessary to use a hybrid solution: a solution that sorts data using the CPU if the number of key-value pairs of the input vector ranges between 35 and 262,144, and using the GPU if the number of key-value pairs of the input vector ranges between 500,111 and 60,000,000.
- When running the radix sort algorithmic function on the GTX 680 graphic processor, using a variable dimension input array (35-60,000,000 elements) of integer, unsigned integer or float input data-types, the performance is comparable. When the keys are of double, long long or unsigned long long data types, the performance is comparable but the execution times are up to 2.58x higher (54.5883 ms vs 140.848443 ms) then in the cases when the input data are of integer, unsigned integer or float data type. Regarding the memory bandwidth, the performance is comparable in all the six analysed cases. These results reflect a high level of performance for our parallel radix sort algorithmic function, regardless of the processed data types and confirm the efficiency of our optimization solutions used in developing the function.

In order to harness the huge computational power of the CUDA architecture, we have designed the parallel radix sort algorithmic function so that it dynamically configures the optimal parameters for each of the used graphics cards. The obtained experimental results reflect a high level of performance on all the three GPUs (whether they were launched in 2008, 2010 or 2012), exceeding the performance obtained on the last generation Sandy Bridge i7-2600K CPU, even if we had overclocked it at 4.6 GHz.

The experimental results confirm the tremendous capacity of scalability and self-adjustability of our parallel radix sort algorithmic function implemented in CUDA, designed to employ the full potential of the GPUs' architectures. The solutions for optimizing the radix sort algorithmic function using the Compute Unified Device Architecture prove their efficiency when the function runs on GPUs from different generations and thus the function is useful and applicable in different scenarios and situations, in a variety of data processing applications that require the parallel radix sort algorithm.

REFERENCES

1. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C., Introduction to Algorithms, Second ed., MIT Press, 2001.
2. Wassenberg J., Sanders P., Engineering a multi-core radix sort, Proceedings of the 17th international conference on Parallel processing - Volume Part II , Springer-Verlag, August 2011.
3. Bandyopadhyay S., Sahni S., Sorting Large Multifield Records on a GPU, Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems, IEEE Computer Society Washington, pp. 149-156, 2011.
4. Satish N., Kim C., Chhugani J., Nguyen A.D., LeeV. W., Daehyun K., Dubey P., Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort, Proceedings of the 2010 international conference on Management of data, ACM New York, pp. 351-362, 2010.
5. Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D., Fast BVH construction on GPUs, in Proceedings of the Eurographics Symposium on Rendering, Eurographics and ACM/SIGGRAPH, Mar. 2009.
6. Tăbușcă A., The new "universal truth" of the World Wide Web, Journal of Information Systems & Operations Management, Vol.5, No.1/2011, pp. 108-115, 2011.
7. Dean J., Ghemawat S., MapReduce: Simplified Data Processing on Large Clusters, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, pp. 137–150, 2004.
8. Tăbușcă A., Enăceanu A., Network security increase by using extended validation secure socket layer certificates for avoiding the phishing threats, Annals of DAAAM for 2009 & Proceedings of 20th DAAAM International Symposium "Intelligent Manufacturing & Automation: Theory, Practice & Education", Austria, Vienna, pp. 0817-0818, 2009.
9. Pîrjan A., Optimization Techniques for Data Sorting Algorithms, Annals of DAAAM for 2011 & Proceedings of the 22nd International DAAAM Symposium, Editor B. Katalinic, DAAAM International, Austria, Viena, pp. 1065-1066, 2011.
10. Nguyen H., GPU Gems 3, Ed. Addison-Wesley Professional, Jul. 2007.
11. Kolb A., Latta L., Rezk-Salama C., Hardware-based simulation and collision detection for large particle systems, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics hardware, ACM New York, pp. 123 – 131, 2004.
12. Satish N., Harris M., Garland M., Designing efficient sorting algorithms for manycore GPUs, Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, Rome, pp. 1-10, Satish, N. (Ed.), IEEE Publisher, Washington, 23-29 May 2009.
13. Lungu I., Petroşanu D., Pîrjan A., Solutions For Optimizing The Data Parallel Prefix Sum Algorithm Using The Compute Unified Device Architecture, Journal of Information Systems & Operations Management, Vol. 5 No. 2.1, pp. 465-477, 2011.
14. Harris M., Sengupta S., Owens J. D., Parallel Prefix Sum (Scan) with CUDA, in GPU Gems 3, H. Nguyen, Ed. Addison-Wesley Professional, Jul. 2007.
15. Le Grand S., Broad-phase collision detection with CUDA, in GPU Gems 3, H. Nguyen, Ed. Addison-Wesley Professional, ch. 32, pp. 697–721, Jul. 2007.
16. He B., Govindaraju N. K., Luo Q., Smith B., Efficient gather and scatter operations on graphics processors, in Proceedings of ACM/IEEE Conference on Supercomputing, pp. 1–12, 2007.