A Version of Parallel Odd-Even Sorting Algorithm Implemented in CUDA Paradigm

Jaumin Ajdari, Bujar Raufi, Xhemal Zenuni, Florije Ismaili

Faculty of Contemporary Sciences and Technologies
South East European University
Tetovo, Macedonia

Abstract
Sorting data is an important problem for many applications. Parallel sorting is a way to improve sorting performance using more nodes or threads e.g. dividing data in more nodes and perform sorting in each node simultaneously or including more threads in process of sorting. It was experimented with one type of those sorting algorithms, namely the well-known sorting algorithms called Odd-Even sort. This paper describes a modification of the above mentioned algorithm. Namely, the algorithm modification consists in the ability to work with the blocks of elements instead of working with individual elements. This modification is done with the idea to make it in a closer form for use of the CUDA technology. Both theoretical and experimental analysis of Odd-Even sort algorithm together with its parallel implementation is done. For experimental purpose, a GeForce GT 645M with 2 GB memory is used. The programming language C++ with CUDA 7.0 paradigm is utilized to implement Odd-Even algorithm and the results indicated that sorting of integers in CUDA environment are dozens of times faster.
Keywords: *Parallel sorting; Odd-Even sort; shared memory; CUDA.*

1. Introduction

Sorting problem is very important in computer science and other disciplines. There are many related work on the issue together with many investigated properties [6, 13]. Nowadays, different sorting algorithms have been developed including such as sequential and parallel [1, 3, 10, 11, 12, 19]. Some of them are implemented in sorting machines as well [18]. In recent years, a lot of investigations of the sorting problem are focused in GPU technology [2, 9, 17] and CUDA [4, 5, 7, 14, 15, 20, 21]. In this paper we have analyzed a simple odd-even sorting algorithm implemented by use of the CUDA paradigm techniques. The odd-even algorithm is used in modified form and instead of comparing the pair of neighbor elements (as in standard odd-even algorithm) we use merge of the subsequences of successive elements. The idea is to separate the sequence of the elements into k subsequences and continue in two steps. The first step is local sort which is between the subsequences by the use of any sorting algorithm and the second step is the merge of the subsequences, and by this modification we achieved $O(n^2/k)$ as computation complexity. This modification is quite convenient to use CUDA paradigm and GPU technology. The algorithm is adapted for CUDA paradigm use and the parallel implementation is done. With parallel implementation, $O(n^2/k^2)$ is achieved as computation complexity. The purpose of this paper was to measure and compare the execution time of the modified algorithm implemented and executed in both CPU and GPU, and to highlight the time speedup in case of GPU use.

The paper is organized in seven sections. It starts with the introduction, in the second section some related works are given, odd-even algorithm and modification are analyzed in the third section, in the fourth section, a parallel implementation is done, the CUDA implementation and experimentation are given in section five and six and in section seven presents some conclusions.

2. Related Work

Sorting algorithms are the most widely studied in the computer science and there is too much work done in the sorting problems. Hence, we focus on the parallel sorting algorithms that exploit the modern GPU architectures and CUDA paradigm. In this section, we briefly survey related work in GPU sorting algorithms with use of advantages of CUDA paradigm.

A design of parallel routines for multicore GPU which use advantages of the full programmability offered by CUDA is presented in [14] (Nadathur, Mark, & Michael). They have designed a version of parallel radix sort algorithm as a non-comparison based and merge sort algorithm as a comparison based and where have exploited substantial fine-grained parallelism and decompose of the computation into independent tasks that perform minimal global communication. In their experiments they have achieved higher performances.

Shifu, Jing, Yongming, Junping, & Pheng-Ann, in their paper [20] proposed a sorting algorithm which is a combination of the bucket sort and internal bitonic sort and they achieved many times acceleration over the STL Quicksort implementation. Also they show that their implementation has higher performance than the GPU Quicksort and GPU RadixSort.

Daniel & Philippas in [5] have proposed a parallel Quicksort algorithm designed to take advantage of the high bandwidth of GPUs by minimizing the amount of bookkeeping and inter-thread synchronization needed. They show that their GPU-Quicksort implementation performs better than the fastest known sorting implementations for GPU, such as radix and bitonic sort.

A version of comparison based parallel algorithm for GPU which consists a combination of the bitonic sort algorithm followed by the merge sort is presented in [21] (Xiaochun, Dongrui, Wei, Nan, & Ienne). They have paid more attention on the mapping of the sorting tasks to the GPU, the synchronous execution of threads in a warp (in order to eliminate the barriers in bitonic sorting network) and providing sufficient homogeneous parallel operations for all the threads within a warp (in order to avoid branch divergence). They called their algorithm as GPU - Warpsort and in their experimentation have achieved high performances.

Hagen, Ole, & Norbert, in [9] have proposed an in-place implementation of Batcher's bitonic sorting networks for CUDA-enabled GPUs. They adapted bitonic sort for arbitrary input length and assigned compare/exchange-operations to threads in a way that decreases low-performance global-memory access and thereby greatly increases the performance of the implementation.

Chun-Yuan, Wei Sheng, & Chuan Yi, in [4], they proposed an efficient implementation of a parallel shellsort algorithm, CUDA shellsort, for many-core GPUs with CUDA. And under the uniform distribution of the elements their implementation show high performances and moreover the performance, based on the showed results, is the same for big samples of elements.

3. Odd-Even Sort Algorithm

Odd-even sort algorithm a version of well-known bubble sort algorithm which can be effectively implemented in parallel. In the following section we describe two variants of this algorithm. The first is the simple form given as below.

Let we have a sequence of numbers a_0, a_1, …, a_{n-1}, sorting algorithms starts with first position, element a_0, and for each even position does the exchange of the neighbors, so the element a_{2i} is compared with his neighbor a_{2i+1}. On the next step, algorithm starts from the second position a_1 and for each odd position does the exchange of the neighbors, a_{2i+1} is compared with a_{2i+2}. Those two steps are repeated until there is no changes on the exchange operation. Let *k* be the number of repetitions of the above steps and in one step there are $n/2$ exchange operation so in total the number of comparisons is $k \cdot (n/2)$ and the best case is if we obtain the sorting for $k=1$ and the worst case for $k=n-1$. The complexity of the odd-even sorting algorithm lays between $O(n)$ and $O(n^2)$.

Now, let we try to modify this idea by divide the sequence into subsequences. Let *k* be the number of sub sequences, than algorithm can be divided into two steps. First step, sorts the elements into each sub sequence and the second step does the merge of the sub sequences.

For the first step, we can use any sorting algorithm and as well can be used the odd-even algorithm. Now because the number of elements which is $\frac{n}{k}$, the complexity is between $O\left(\frac{n}{k}\right)$ and $O\left(\left(\frac{n}{k}\right)^2\right)$. If we use quicksort (which is known as better sorting algorithm) then the complexity is $O\left(\frac{n}{k}\log\left(\frac{n}{k}\right)\right)$. The total complexity for this step lays between $O(n)$ and $O\left(\frac{n^2}{k}\right)$.

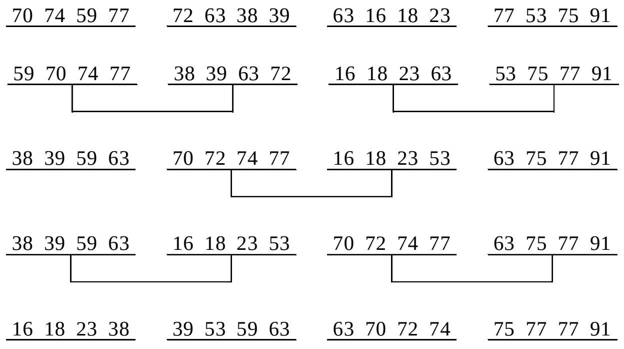
In the next step, the sorted subsequences have to be merged. For merge operation the idea of odd-even algorithm is used and instead of exchange between elements we do merge of the subsequences. The merge operation is done for the two neighbor subsequences and for the left subsequence we chose the first *k* smaller (depend on the sorting type, does it ascending or descending) elements and the remaining we store on the right subsequence. Merge operation is done in way of merging two sorted sequence and the complexity is linear to the number of elements. This operation will be done alternatively and as it explained for the case where all elements are contained into one sequence. So instead of an element now we work with a sequence. The whole sort will be achieved after $k-1$ sub steps where into one step will be done $\frac{k}{2}$ alternatively merge operations. Taken into account that one merge operation has complexity $O\left(\frac{n}{k}\right)$, we obtain the complexity $(k-1)\cdot\frac{k}{2}\cdot O\left(\frac{n}{k}\right)$ and for the overall complexity is

$$T(n,k)=O\left(\frac{n^2}{k}\right)+(k-1)\cdot\frac{k}{2}\cdot O\left(\frac{n}{k}\right)$$

$$T(n,k)=O\left(\frac{n^2}{k}\right)+O(kn)=O\left(\frac{n^2}{k}+kn\right)$$

and $n>k^2$ (which is very realistic condition), thus we have $T(n,k)=O\left(\frac{n^2}{k}\right)$.

The described idea is illustrated with a following simple example,



```
70 74 59 77    72 63 38 39    63 16 18 23    77 53 75 91
59 70 74 77    38 39 63 72    16 18 23 63    53 75 77 91
38 39 59 63    70 72 74 77    16 18 23 53    63 75 77 91
38 39 59 63    16 18 23 53    70 72 74 77    63 75 77 91
16 18 23 38    39 53 59 63    63 70 72 74    75 77 77 91
```

4. Parallel Odd-Even Sort Algorithm

The idea of sorting by divide into subsequences is a good starting point to design a parallel algorithm. Let us start with a simple case.

Let *n* be the number of elements of the sequence and also let us supposed to have *n* processing elements (where processing element can be a process or thread). As it is known odd-even algorithm does alternatively exchange of the neighboring elements and after *n-1* repeats the sequence is sorted. Let us take the sequence a_0, a_1, …, in the step *i*, where $1\le i<n$ then the processing element *j*, $0\le j<n$ will operate as follow

$$a_j=\begin{cases}\min(a_{2k},a_{2k+1}), & j=2k\\ \max(a_{2k},a_{2k+1}), & j=2k+1\end{cases},\ i=2k$$

and

$$a_j=\begin{cases}\min(a_{2k+1},a_{2k+2}), & j=2k+1\\ \max(a_{2k+1},a_{2k+2}), & j=2k\end{cases},\ i=2k+1$$

Algorithms starts with first position, element a_0, and for each even position does the exchange of the neighbors, so the element a_{2k} is compared with his neighbor a_{2k+1}. In the next step, algorithm starts from the second position a_1 and for each odd position does the exchange of the neighbors, a_{2k+1} is compared with a_{2k+2}. Those two steps are repeated until there is no changes on the exchange operation.

Exchange operations can be done in parallel. If for each element we map a processing element then during one step all exchange operations will be done in the same time and the overall time complexity of the step is $O(1)$. As we mention before the sort is done after *n-1* steps and the time complexity of the parallel sorting of sequence with *n* elements by use of *n* processing elements is

$$T(n)=n\cdot O(1)=O(n).$$

On the general, the number of processing elements is less than the number of elements. Let *n* be the number of elements of the sequence and *k* the number of processing elements. We use the above elaborated idea where we divide the sequence into subsequences. We divide the sequence into *k* subsequences and each subsequence will have $\frac{n}{k}$ elements. Now, first step is sorting of the subsequences and this can be done in parallel, so, each sorting element will sort his part and we obtain *k* sorted sequence. This operation we call local sort. The time complexity of the local sort is between $O\left(\frac{n}{k}\log\left(\frac{n}{k}\right)\right)$ and $O\left(\left(\frac{n}{k}\right)^2\right)$ (depend on the sorting algorithm used).

For the next step, we define the operation merge which take two sorted subsequences and as a result gives one sorted subsequence with same length as input sequences. The merge operation is defined as follow: each processing element works with owned subsequence and the subsequence of the first his right neighbor and by use of the merge operation combines those two subsequences into one sorted sequence with doubled number of elements. Let *i* be the identification number of the processing element. In case of even step, if *i* is even then the processing element as a result takes the first $\frac{n}{k}$ elements otherwise takes the second part (second $\frac{n}{k}$ elements). For the odd step it does similar and if *i* is odd then procesing element takes the first part otherwise second part.

The complexity of one merge is $O\left(\frac{n}{k}\right)$ and the same is for whole step, according to above explanation the time complexity of this part of the algorithm is

$$T(n,k) = k \cdot O\left(\frac{n}{k}\right) = O(n)$$

and for both parts is

$$T(n,k) = O\left(\left(\frac{n}{k}\right)^2\right) + O(n) = O\left(\left(\frac{n}{k}\right)^2 + n\right)$$

$$T(n,k) = O\left(\left(\frac{n}{k}\right)^2\right),\ for\ n > k^2.$$

5. Implementation of Odd-Even Sort Algorithm in CUDA Technology

5.1. Introduction to CUDA

The Compute Unified Device Architecture (CUDA) is a parallel programming paradigm released in 2007 by NVIDIA. It was originally intended as a platform for programming graphics applications, but later it was found that could be used for to include the GPU in solving general purpose problems and to enable parallel solutions by use of the kernels of the GPU as a processing elements. CUDA use the C/C++ programming language with some extensions to allow use of the GPU specific features. CUDA has specific functions, called kernels. Kernel is a function or a program which is invoked from CPU and is executed many times of the same function in parallel in GPU. CUDA programming paradigm is a combination of a serial and parallel execution and serial part is executed in the host (CPU) and parallel part in the device (GPU). Host is responsible for transfer data to the device and as well as to invoke kernels which will be execute to device. Figure 1 illustrate the basic model of CUDA working

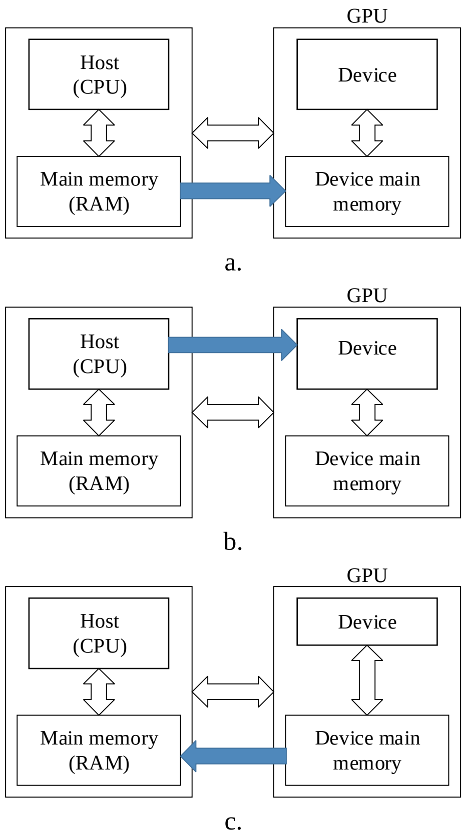


Fig. 1. a. Transfer data from host main memory to device main memory; b. Invoke kernel function; c. Return the results from device to host.

In general, CUDA provides three main types of the function qualifiers which are device, global and host. Functions which have to be executed in device (GPU) have to be declared with qualifier *__device__*, these function are callable from the device. Functions which have to be invoked from the host (CPU) but the execution will be in the device with qualifier *__global__* and those which execution will be in host with *__host__* and these are callable only from host.
CUDA execution model is based on a hierarchy of abstraction layers: grids, blocks, warps and threads (Fig. 2). The thread is the basic execution unit and it represents the processing element. A block is a batch of threads cooperating together and therefore all threads in a block

share the same memory (each thread has its local memory and together have the common memory). Threads in a block can be waiting one to each other, so can be synchronized (the global synchronization isn't provided). A grid is composed by several blocks. In turn, a warp is a group of threads of the same block and which can be executed in parallel, in a SIMD way and threads of a same block are scheduled warp by warp. The CPU is responsible of transferring data between host and device memories as well as invoking the kernel code, setting the grid and block dimensions. The kernel invoke is as follow

kernel_function<<<gridDim, blockDim>>>(…arguments…);

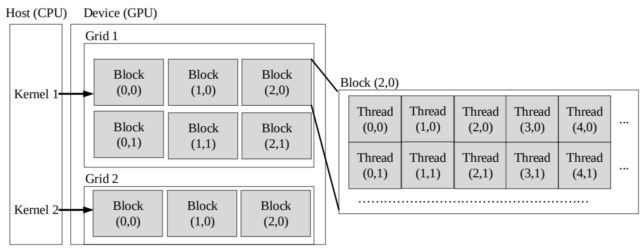


Fig. 2. Hierarchy of abstraction layers: grids, blocks and threads.

The CUDA paradigm provides some built-in variables to use this structure efficiently. To access the id of a thread block the *blockIdx* variable (values from 0 to *gridDim-1*) is used and to access its dimension the *blockDim* variable is used while *gridDim* gives the dimensions of the grid. Each individual thread is identified by *threadIdx* variable, can have values from 0 to *blockDim-1*. *WarpSize* specifies warp size in the threads. All these variables are built-in in kernel.

Different memory spaces are available. The GPU has its memory which is named as global memory and this can be used from all threads of all blocks. This memory also is used and from CPU (CPU transfer the data which have to be processed from its main memory to this memory). Global memory can be with large capacity. The next is shared memory. Shared memory is faster compared to global memory and it is divided into many parts. Each block has its own part of shared memory and this part of shared memory is accessible only from the threads within a block and all threads within a block, share this part of memory for both read and write operations To declare variables in shared memory *__shared__* qualifier is used and to declare in global memory *__device__* qualifier is used. Each thread also contains its own local memory. Normally local variables of the kernel functions are allocated here. Sometimes they are allocated on global memory [8, 16, 22].

5.2. CUDA Implementation

Using the CUDA possibilities, for the above developed algorithm we have analyzed two implementations. First one implementation is when the number of elements of the sequence is less than the maximum number of the threads in a block. In this case the whole sequence is copied to the block's shared memory and for each element of the sequence is mapped one thread. Each thread takes two elements (the element with the same index as thread identification number and the right/left neighbor element) and in accordance with the algorithm's rule as a result gives the minimum or maximum of the input elements. This operation is repeated many times (as many time as is the number of elements of the sequence).
Second implementation is limited by the maximum number of blocks. So the sequence is divided in many subsequences (up to the maximum number of the blocks) where a sub sequence is mapped into a block and implementation goes in two steps. The first step is local sort. According to the number of elements of the subsequence and maximum number of thread available for the block we have analyzed two different situations. If the number of elements in a subsequence (in a block) is less or equal to the number of threads than is used the same idea as in first implementation. Otherwise, if the number of elements in a block is greater than the number of threads than is applied the sorting in two phases. The subsequence in a block is divided in subsequences, up to the number of threads and each thread does the sort in his part. After the finishing of the first phase, starts the merge phase which continue to whole sort of the subsequence of the block. Now, if the number of sorted subsequences is smaller than the number of threads in a block, for the second step we define one block and each subsequence we map to a thread of this block. Threads deals similar as in the first implementation but instead of elements it takes two subsequences and does the merge of those. If the number of sub sequences is greater than the maximum number of available threads on the block than we have defined for each subsequence a block with one thread. On the shared memory of each block two neighbor subsequences are copied, as in the follow figure

Block no.	0	1	2	3	4	5	…
Seq. no.	0-1	0-1	2-3	2-3	4-5	4-5	…

a.

Block no.	0	1	2	3	4	5	…
Seq. no.		1-2	1-2	3-4	3-4	5-6	…

b.

Fig. 3. a. For the phase which starts from zero, even phase, b. odd phase

This operation repeats *k - 1* times, where k is the number of blocks used.

6. Practical Experiments

In order to show the practical effects of the above explained algorithm, we implemented the above idea and it is executed in a machine with NVidia GPU. Technical features of the machine used for testing are Intel i5 processor, 8 GB RAM and GeForce GT 645M with 2GB VRAM, set up under Windows 8.1 operating system. CUDA 7.0 paradigm and C++ is used for CUDA implementation.

Methodology of experimentation - the same algorithm is executed several times (typically 10 times) and as an execution time the average time is taken. Three different algorithms are executed, the standard Odd-Even algorithm (where the basic operation is comparison of neighbor elements), modified Odd-Even algorithm (called block Odd-Even, which consists of two phases, the first phase where a local sorting is performed and second phase where a merge is performed) and block Odd-Even algorithm implemented with CUDA paradigm. It is measured only the calculations (sorting) time and data transfers from RAM to VRAM and vice versa are not taken into account. Average time results are given in tables and graphs.

Because of the technical features of the system, executions are done for integer numbers and for the number of elements *n = 1K, 2K, 4K, 8K, 16K, 32k, 64K, 128K, 256K, 512K* and *1024K* (power of number 2 and K - kilo). The block sizes *BS = 16, 32, 64, 128, 256, 512* and *1024* for executions are used. Due to GPU technical limitations (memory restrictions and limited number of threads per block), we could not use block sizes greater than 1024. Results are expressed into tables and graphs and the first table's row shows the number of elements (expressed in kilo), while the first column the block size.

Table 1. Run time of sequential implementation of Block Odd-Even algorithm

N(K) / Bs	1	2	4	8	16	32	64	128	256	512	1024
1	0.0029	0.0128	0.0424	0.1622	0.683	2.7843	11.048	43.482	170.56		
16	0.0009	0.003	0.0106	0.0419	0.1621	0.6581	2.5212	9.7517	38.343	153.17	
32	0.0001	0.0014	0.0058	0.0215	0.0834	0.3235	1.2836	4.9074	19.163	76.127	
64	0.0003	0.0009	0.004	0.0122	0.0435	0.1631	0.6286	2.466	9.7439	38.533	
128	0.0004	0.0008	0.0029	0.0088	0.0258	0.0876	0.3214	1.2516	4.8447	19.119	76.354
256	0.0005	0.0015	0.004	0.0088	0.0212	0.0601	0.191	0.6712	2.4961	9.7048	38.594
512	0.0013	0.003	0.0061	0.0126	0.0267	0.0598	0.1584	0.4657	1.5187	5.375	20.234
1024	0.0029	0.0054	0.0114	0.0219	0.0447	0.0943	0.2029	0.475	1.2237	3.6393	11.953

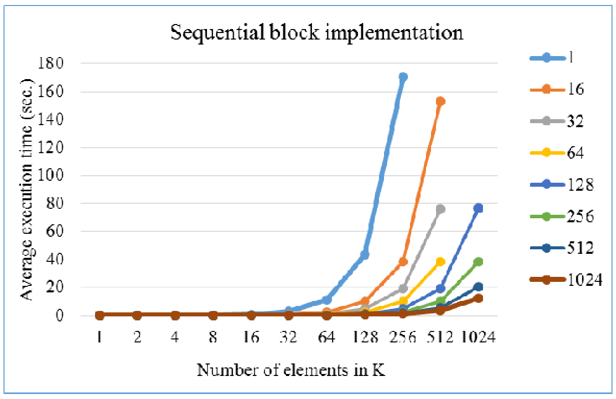


Fig. 4. Run time of sequential implementation of Block Odd-Even algorithm

Table 2. Run time of CUDA implementation

N(K) / Bs	1	2	4	8	16	32	64	128	256	512	1024
16	0.0008	0.0007	0.001	0.0008	0.0013	0.0012	0.0019	0.0028	0.005		
32	0.0004	0.0008	0.001	0.0009	0.001	0.0011	0.0015	0.0025	0.0044		
64	0.0009	0.0007	0.0008	0.0009	0.0011	0.0012	0.0019	0.0024	0.004	0.008	
128	0.0008	0.0008	0.0009	0.0007	0.001	0.0013	0.0017	0.0028	0.005	0.0089	0.017
256	0.001	0.001	0.0008	0.0012	0.001	0.002	0.003	0.0052	0.0095	0.0173	0.034
512	0.001	0.0009	0.001	0.0015	0.002	0.0034	0.0058	0.0103	0.0194	0.0381	0.075
1024	0.0013	0.0011	0.0017	0.0022	0.0036	0.0063	0.0113	0.0216	0.0417	0.082	0.165

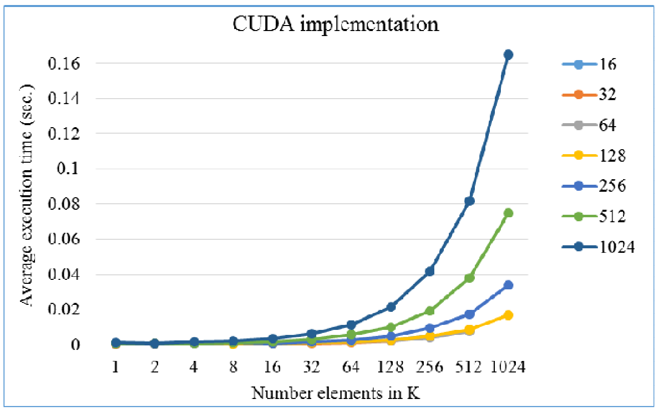


Fig. 5. Run time of CUDA implementation

7. Conclusion

The analysis and obtained results conclude that modification in blocks of the Odd-Even algorithm, compared to the standard algorithm shows dozen times higher performances. It was expected since the algorithm operates as follow: first the sequence is divided into subsequences (in blocks), which is sorted quickly compared with the sequence of whole and to achieve the whole sequence sorted it continues with the merger of sorted subsequences which is with a complexity $O(n)$. This was noticed even in theory, where the complexity is calculated and it is $O\left(n^2/k\right)$ and k is the number of elements of the subsequence. The practical results confirm the theoretical complexity, so when k increase execution time is in decrease.

The idea of separation of sequence in blocks and modification of Odd-Even is used for NVidia CUDA paradigm implementation. In CUDA implementation, the logical CUDA block for the subsequence (block) is used. Compared to CPU execution, GPU execution shows very high performance (and the same is confirmed by mathematical complexity calculation). Analyzing the executions for different block size we conclude that in case of BS = 128 the results are better. This result is justified because of the GPU technical parameters, warp size (which is 32) and the way of algorithm implementation (as shown above).

As a general conclusion is that despite the technical limitations of the GPU, the achieved results show that the use of GPU and CUDA paradigm shows high performance in solving of the problem of sorting.

References

[1] Ananth, G., George, K., Vipin, K., & Anshul, G. Introduction to Parallel Computing, Second Edition. Boston: Addison Wesley, 2003.
[2] Bilal, J., Bartolomeo, M., Carlo, R., Fiaz, G. K., & Omar, K. Fast Parallel Sorting Algoritms On GPUs. International Journal of Distributed and Parallel Systems (IJDPS), 3(6), 2012, pp. 107-118
[3] Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J., & Zagha, M. An Experimental Analysis of Parallel Sorting Algorithms, Theory of Computing Systems. Theory of Computing Systems, 31(2), 1998, pp. 135-167.
[4] Chun-Yuan, L., Wei Sheng, L., & Chuan Yi, T. Parallel Shellsort Algorithm for Many-Core GPUs with CUDA. International Journal of Grid and High Performance Computing, 4(2), 2012, pp. 1-16.
[5] Daniel, C., & Philippas, T. GPU-Quicksort: A Practical Quicksort Algorithm. Journal of Experimental Algorithmics (JEA), 14, 2009, pp. 1-22.
[6] David, R. H., Joseph, J., & David, A. B. A New Deterministic Parallel Sorting Algorithm with an Experimental Evaluation. Journal of Experimental Algorithmics (JEA), 3(4), 1996.
[7] Dominik, Z., Marcin, P., Maciej, W., & Kazimierz, W. Comparison of Hybrid Sorting Algorithms Implemented on Different Parallel Hardware Platforms. Computer Science Journal, 14(4), 2013, pp. 679-691.
[8] Ghorpade, J., Parande, J., Kulkarni, M., & Bawaskar, A. GPGPU processing in CUDA architecture. Advanced Computing: An International Journal (ACIJ), 3(1), 2012, pp. 105-120.
[9] Hagen, P., Ole, S.-H., & Norbert, L. Fast in-place, comparison-based sorting with CUDA: a study with bitonic sort. Journal Concurrency and Computation: Practice & Experience, 23(7), 2011, pp. 681-693.
[10] Herruzo, E., Ruíz, G., Benavides, J. I., & Plata, O. G. A New Parallel Sorting Algorithm based on Odd-Even Mergesort. 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2007 , pp. 18-22. Naples, Italy: IEEE Computer Society.
[11] Ionescu, M., & Schauser, K. Optimizing parallel bitonic sort. Proceedings 11th International Parallel Processing Symposium, Geneva, Switzerland: IEEE Computer Society Press. 1999.
[12] Ajdari, J. Performance Estimation of Hypercube Bitonic Sorting Algorithm Implemented With MPI Paradigm In An Experimental Cluster Computer. 3rd Balkan Conference in Informatics (BCI'2007). 1, p. . Sofia, Bulgaria. 2007
[13] Knut, D. E. The Art of Computer Programming, volume 3. Sorting and Searching. Boston: Addison Wesley. 1973
[14] Nadathur, S., Mark, H., & Michael, G. (2009). Designing Efficient Sorting Algorithms for. Proceedings of the 2009 IEEE International Parallel & Distributed Processing Symposium, 2009, pp. 1-10. Rome: IEEE. 2009
[15] Nikolaj, L., Vitaly, O., & Sanders, P. GPU sample sort. Cornell University, Computer Science. Ithaca, New York: Cornell University Library. 2009.
[16] NVidia Corporation. CUDA C Programming Guide. Retrieved from Programming Guide: CUDA Toolkit Documentation: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3X1M5pnlg, (2015, March 5).
[17] Oded, G., Robert, M., & David, A. B. GPU Merge Path - A GPU Merging Algorithm. Proceeding ICS '12 Proceedings of the 26th ACM international conference on Supercomputing 2012, pp. 331-340. San Servolo Island, Venice, Italy, AMC, 2012.
[18] Ranieri, B., Gabriele, C., Franco Maria, N., & Fabrizio, S. Sorting using Bitonic network wIth CUDA. Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR’09), pp. 33-40. Boston, USA, 2009.
[19] Rub, C. On Batcher’s Merge Sort as Parallel Sorting Algorithms. Proceedings 5th Annual Symposium on Theoretical Aspects of Computer Science, pp. 410-420. Paris, France, 1998.
[20] Shifu, C., Jing, Q., Yongming, X., Junping, Z., & Pheng-Ann, H. A Fast and Flexible Sorting Algorithm with CUDA. In S.-L. C. Arrems Hua, Algorithms and Architectures for Parallel Processing, pp. 291-290. Berlin: Springer-Verlag, 2009.
[21] Xiaochun, Y., Dongrui, F., Wei, L., Nan, Y., & Ienne, P. High Performance Comparison-Based Sorting Algorithm on Many-Core GPUs. Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium, pp. 1-10. Atlanta, GA: IEEE, 2010.
[22] Heru S., Arry Y., Ari W. Performance Analysis Cluster and GPU Computing Environment on Molecular Dynamic Simulation of BRV-1 and REM2 with GROMACS, IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 4, No 2, pp. 131-135, July 2011

Jaumin Ajdari, Assist. Prof. at Faculty of Contemporary Sciences and Technologies, South East European University. His current research interest is in parallel processing, data processing and databases.

Bujar Raufi, Assist. Prof. at Faculty of Contemporary Sciences and Technologies, South East European University. His current research interest is in adaptive web, semantic web, computer graphics and data analytics.

Xhemal Zenuniu, Assist. Prof. at Faculty of Contemporary Sciences and Technologies, South East European University. His current research interest is in semantic web, contemporary distributed systems, intelligent agent and data analytics.

Florije Ismaili Assist. Prof. at Faculty of Contemporary Sciences and Technologies, South East European University. His current research interest is in web services, cloud computing and information retrieval.