IPSJ SIG Technical Report

Vol.2013-ARC-207 No.21
Vol.2013-HPC-142 No.21
2013/12/17

MSD Radix String Sort on GPU: Longer Keys, Shorter Alphabets

Aleksandr Drozd[1,a)] Satoshi Matsuoka[1,b)]

Abstract: This study proposes a GPU implementation of radix sort, which has so far been most successfully presented in the Thrust library from Cuda SDK. While Thrust performs sorting starting from least significant digit (LSD) to benefit from GPU architecture features, it can deal only with fixed-length keys. Our solution features MSD radix sort or bucket sort which allows for variable length keys, but is recursive and therefore difficult to implement on GPU. However, we achieved comparable performance by performing the sorting in several stages which use different parallelization schemes depending on the size and number of buckets. This solution also benefits from shorter alphabets and would be particularly efficient for, e.g., genomic data.

1. Introduction

Sorting is one of fundamental and most widely studied algorithmic problems in computer science. Sorting routines are used standalone for storage and manipulation of data and as a basis for more complex algorithms in various areas from graph and spatial data processing to molecular biology.

With data becoming Big Data, efficiency of sorting algorithms becomes of increasing importance. There are two ways to improve it: optimization of the algorithms themselves and their adaptation to massively parallel hybrid architectures. An efficient algorithm should combine workload balancing, economic data transfers and overall computation model that is adequate to strength and limitations of underlying hardware platform.

In this paper we address the issue of efficient sorting of strings on GPU which has not received as much attention as sorting of numeric data.

1.1 Approaches to sorting

The classical comparison-based sorting algorithms such as merge-[8] or quick-sort[12] have asymptotic complexity estimation as as $N \log N$. However this allows us to estimate only the amount of comparison operations needed to complete the sort and not the actual performance time. So, for these sorts execution time depends on the cost of comparison operation, and that depends on the actual data. For example, lexicographic sorting of strings requires comparison of many symbols, and that makes the complexity of the sorting dependent on what is called the longest common prefix (LCP) = $\frac{1}{n-1}\sum_{i=0}^{n}(LCP(S_i, S_{i+1}))$ where $LCP(S_i, S_{i+1})$ is the number of symbols two adjacent strings in have in common.

Therefore for sorting of strings it is preferable to use an algorithm that does not depend on comparison, such as one of distribution-based sorts. These algorithms are less generic, but can deliver faster performance for certain data types, up to linear performance in the ideal case. In this paper we focus on radix sort which is a well-known example of this class of algorithms.

Radix sort comes in two flavors: sort that starts from the most significant digit (MSD radix sort) or from the least significant digit (LSD radix sort). The term "digit" is used because radix sort is generally applied to integers; it actually refers to any amount of bits in the binary representation of the number. However, since this paper describes application of radix sort to strings, we shall hereafter use the term "symbol" rather than "digit".

LSD radix sort is perhaps the most commonly used one; it performs well on short-length keys such as integer numbers. The most efficient implementation of this algorithm is now part of CUDA SDK[15]. However, LSD sort is bound to short keys of fixed length, which does not cover many types of data.

The reason why LSD radix sort is bound to short keys is that it starts from the rightmost symbol and proceeds to the previous one while maintaining stability of the sort, and then the algorithm is repeated until the first symbol is reached. On relatively long keys this approach would not be efficient because comparing the first several symbols should be enough to determine the order of strings. Moreover, with a long key the number of iterations goes up, and the performance decreases accordingly.

MSD radix sort does not have this problem in that it starts from the leftmost symbol and then moves up to the next symbol only for the strings the order of which is not yet determined. It can be viewed as bucket sort because this process basically consists of recursive distribution of strings into buckets: at first all strings are placed into different buckets depending on their first symbol, and then the strings inside each bucket are partitioned again by the next symbol. This process is fairly intuitive, but its recursive

[1] Tokyo Institute of Technology, Meguro-ku, Tokyo 152-8550, Japan
[a)] alex@smg.is.titech.ac.jp
[b)] matsu@is.titech.ac.jp

nature makes it challenging to implement on GPU.

1.2 Related work

Sorting algorithms have been studied extensively, and there have also been numerous attempts to develop parallel approaches to sorting. Comparison-based sorts are the most commonly used and applicable to various kinds of data. The most efficient algorithms are based on divide-and-conquer approach and are tricky to parallelize efficiently. We now have parallel versions of quick sort [17] and merge sort[1], [7] , among others. There is also bitonic mergesort sort [2] which was developed to be more paralellization-friendly.

The efficiency of certain algorithms and their implementation is also relative to the type of data being sorted and underlying hardware architecture. Most of these sorting algorithms are memory-bound, and, for distributed memory systems, communication-bound.

SIMD architectures are putting even more limitations on what can be implemented but, like the recently popular GPUs, provide totally different performance tradeoffs. Distribution sorts which are inherently efficient for certain type of data have been especially successfully implemented on GPU. Radix sort which utilizes thread parallelizm and hight memory throughput was reported to be highly efficient on GPU [18]. They have also presented a quicksort implementation for GPU with inferior performance. To the best of our knowledge, the the most efficient GPUu radix sort is currently the one from Thrust library presented by Merril and Grimshaw [15].

Comparison-based sorts were also implemented on GPU. Purcell et al. [16] presented bitonic merge sort on GPUs based on the work by Kapasi et al. [13]. Greß et al. [11] used the sorting technique presented in the Bilardi et al. paper [5] to implement GPU adaptive bitonic sort. Another GPU sort based on bitonic was implemented by Govindaraju et al. [10]. Later they presented a hybrid CPU and GPU solution using bitonic-radix sort in Tera-Sort challange[9]. An approach that combines several algorithms was presented by Sintorn et al. [19]; their solution splits the data with a bucket sort and then uses merge sort on the resulting blocks. Finally, there were more successful attempts to implement quick sort on GPU [6]. There are ongoing efforts to optimize comparison-based algorithms for new architectures, i.e. by using vector instructions of modern processors [20] However, all the above-mentioned radix sorts perform better on numerical data, since they are LSD radix sorts and can not work with long keys.

None of comparison-based sorts are efficient for string data. The one algorithm that is known for high performance on strings is MSD radix sort [14]. There is also 3-way radix quicksort presented by Bentley and Sedgewick [3], [4], which is even more efficient due to more optimal use of caching. However, at present, to the best of our knowledge, there are no parallel implementations of radix sort that could handle long string keys. This paper addresses this gap. Our solution is based on MSD radix sort which is less complex and more GPU-friendly, and equally efficient on the initial stages of the algorithm (while the bucket are relatively big). On the later stages, as buckets get smaller, we are switching to the 3-way radix quicksort.

This paper proceeds as follows. Section 2 describes our approach to parallelization of MSD radix sort. Section 3 presents performance analyses for different architectures. Section 4 discusses the results and outlines directions for future work.

2. Parallelizing MSD Radix Sort

The naive approach to parallelization of a recursive algorithm would be to use task parallelism for every recursion branch. Thus we will be doubling the number of parallel threads on every level of recursion, but in the beginning the number of threads is low, which significantly limits the performance of the algorithm. First of all, the amount of workload is the biggest for the initial iterations. Secondly, this approach is even less efficient for GPU than the the classical multi-core architecture, since one thread on a GPU is relatively slower and high performance is achieved only when we have thousands of threads running concurrently.

Another approach would be to parallelize every iteration of the algorithm. To build each bucket we are basically counting symbols and then filling array of pointers according to the counters. Counting can be efficiently parallelized in data-parallel fashion when the workload is split between parallel threads processing their parts independently. The next step is reduction of results.

The problem with this approach is that it performs well in the beginning of the recursive execution when the buckets are relatively big, but as they get smaller the parallel execution with small amounts of data becomes a waste of resources. However, by this time we already have enough buckets to make use of the model in which one thread or a small group of threads are processing one bucket.

Combining the two approaches would allow us to keep the GPU busy the entire time of execution. We suggest starting with partitioning strings into buckets and then, when buckets are small enough, continuing to process each bucket independently in parallel. And on the third stage of sorting we have only small unsorted buckets at different data points which could be more efficiently dealt with on CPU rather than on GPU.

The following section describes an actual implementation of this approach on GPU and discusses different aspects our model that also influence performance, such as data transmission costs, workload balance, and divergent branching in sorting code.

3. Implementation

We chose GPU because of its unprecedented performance achieved through high parallelism, which is why it is widely used in world top supercomputers. Our solution is implemented in CUDA, NVIDIA's programming platform for general-purpose computing on GPUs which is the current industry standard. Figure 1 shows the outlined of the algorithm.

S is the array of strings and $S[i][j]$ denotes j_{th} symbol of i-th string and S_{aux} is auxiliary array. Though we use double-buffering technique, S and S_{aux} are storing only pointers to strings, so the increase in memory consumption is not significant. C is the array of counters for each letter of the alphabet and O is the array of pointers to the beginning of each bucket. N is the number of strings being sorted and σ is alphabet size. d denotes

```
procedure SORT($S, l, r, d$)
    for $i \in (0..N)$ do                                   ▷ counting
        $C[S[i][d]] \leftarrow C[S[i][d]] + 1$
    end for
    for $i \in (0..\sigma)$ do                              ▷ offsets
        $O[i] \leftarrow \sum_0^{i-1} C[i]$
    end for
    for $i \in (0..N)$ do                                   ▷ moving
        $S_{aux}[O[S[i][d]]] = S[i]$
        $O[S[i][d]] \leftarrow O[S[i][d]] + 1$
    end for$S = S_{aux}$
    for $i \in (0..\sigma)$ do                              ▷ recursion
        $Sort(S, O[\sigma] - C[\sigma], O[\sigma], d + 1)$
    end for
end procedure
```

Fig. 1: Sort Algorithm

sorting depth, i.e. the position of symbol we use for partitioning strings into buckets.

The execution model on GPU is very different from that of traditional multi-core systems. While GPUs provide a much higher level of parallelism (new cards boast as many as 512 SM cores per die), programs are executed in the so-called Single Instruction Multiple Threads (SIMT) paradigm. This means that threads on the same multiprocessor are performing the same instructions at the same time, and this does not allow us to run independent tasks on different cores (although it is possible to launch several parallel kernels (part of a program running on GPU) at the same time).

Recursive algorithms are challenging to implement on GPUs because, as of now, only the newest Kepler architecture currently support recursion, and that support is rather limited. Recursive launch of kernels is used to bring control of their execution entirely to GPU, but it is not meant to be used in highly recursive algorithms. Another hardware characteristic to be considered is that programs on GPU can address only on-board GPU memory which has relatively high bandwidth, but the data has to be transferred from host memory and back, which is relatively slow.

As described in Section 2, our algorithm is executed in three stages:

(1) processing the first (biggest) buckets in thread-parallel fashion; (2) processing the resulting (smaller) buckets in combination of thread-parallel and task-parallel paradigms; (3) sorting the remaining small buckets with CPU threads.

To implement the first the stage of the algorithm we made heavy use of atomic operations which allowed us to avoid reduction and to thus decrease memory usage (which would be significant with many counters per thread). This approach does not slow down our performance due to highly efficient implementation of atomic operations in the newest GPU architectures.

Another resource that we could use to increase the efficiency of radix sort is sorting groups of symbols instead of one symbol at a time. With the use of atomic operations, the amount of symbols that could be processed simultaneously is only limited by the available memory and the length of the alphabet. Generally speaking, the amount of buckets in radix sort with N symbols in the alphabet is equal to: S^M where S is the alphabet size and M is the length of the group. Assuming that we use 64-bit integers as counters it is easy to estimate how much memory the program will require for storing buckets: $8 * S^M$ bytes.

It is obvious that shorter alphabets with the same amount of memory will allow for processing of more symbols per iteration. A good example of an area that uses such data is genomics, where the alphabet consists of four nucleotides coded A, C, G, and T (some databases also use N for inconclusive read results). In such a case sorting six symbols at a time would require only $4^6 * 8$ bytes which would take up only 15 Kb, which is insignificant compared to the gigabyte-sized strings to be sorted. Moreover, this amount of buckets is already sufficient to start the parallel sorting by buckets on the next stage.

The second stage of the algorithm repeats the same logic, but with smaller groups of threads independently processing different buckets. This is more efficient, since the buckets get smaller at this stage. On the other hand, since the groups of threads now have their own counters this limits the amount of groups that can be executed in parallel. We balance these parameters to keep GPU cores saturated.

At the point when the distribution of buckets exhibits high workload balance which we cannot cope with on GPU, and the sort is continued on CPU. There we can use 3-way radix quicksort which is a more advanced version of the same algorithm that cannot be performed on GPU due to architecture limitations.

4. Performance Analysis and Optimization

We used the following hardware configuration for performance benchmarks.

- Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz, 6 cores with hyper-threading
- OS: Scientific Linux release 6.1 (Carbon)
- Memory: 126 Gb
- GPU: Tesla K20c
- GPU compute capability: 3.5
- GPU memory: 4.6 GB

First of all we evaluated overall performance of the implementation in terms of sorting throughput. Figure 2 shows sorting throughput on different number of keys for small and for big alphabets. We a observed that on longer alphabets it takes some time to saturate the performance. The reason is that counters for all possible combinations of symbols of a given length are allocated even if not all of them are used and this time is constant for any workload size.

Figure 4 shows exact time spent in each phase of the algorithm. ”copy” if moving string from host to device, ”count” is counting the number of occurrences for each prefix, offsets” is prefix sum of the counters and ”move_keys” denotes scanning strings for second time and placing corresponding pointers according to offsets.

Though the focus of this work was the GPU implementation of MSD radix sort algorithms we tried to run the same algorithm on traditional CPU. We tried to keep the code as similar as possible, but as CPU architecture does not provide equivalently efficient atomic operations as GPU we had to perform additional step for reduction. Figure also 2 shows performance of sorting from standard C library, performance of sequential version of 3-way radix quick sort on CPU and multi-core version of MSD radix quicksort. For CPU implementation we also measured time spent for different phases and how it changes with the when increase of the number of threads. Figure 6 shows this dynamics.

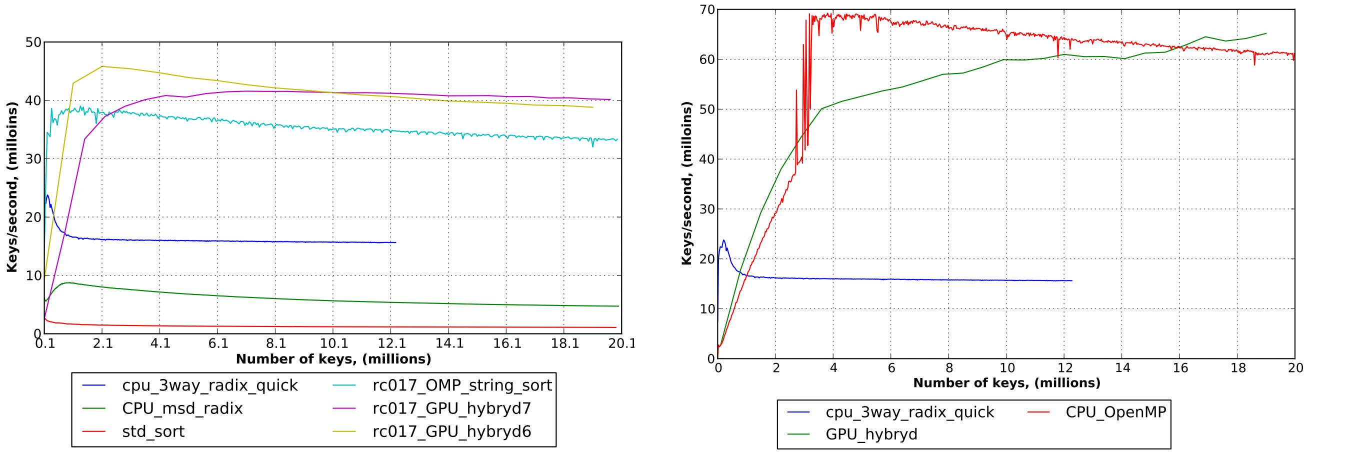

Fig. 2: Sorting throughputh on short and long alphabets

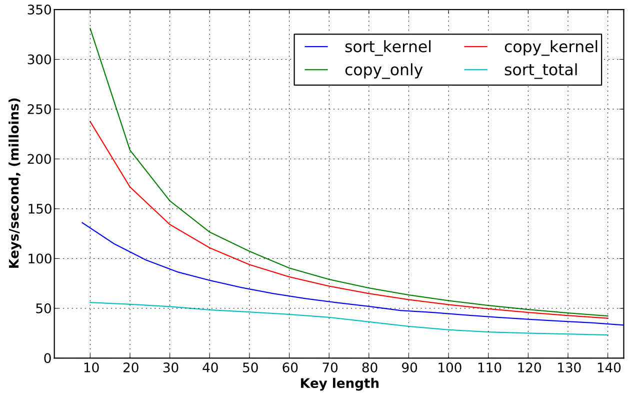

Fig. 3: Correlation of performance and key length

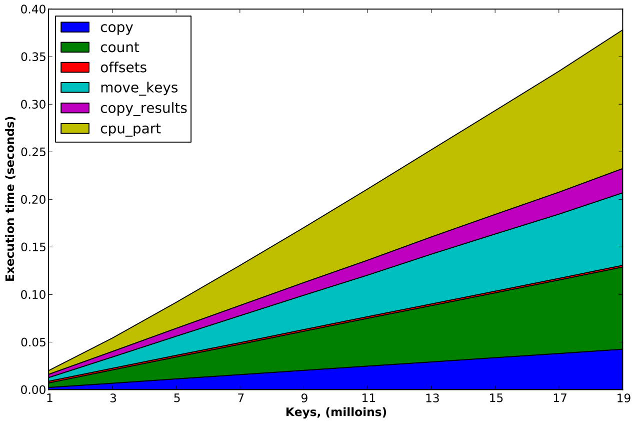

Fig. 4: Execution time breakdown for GPU implementation

For the GPU implementation analysis of hardware counters done with Compute Visual profiler shows that sort kernel is memory bound and uses only few percent of available memory bus bandwidth. Organizing memory access patterns in a way that writes and loads are or at least localized is a one of the fundamental optimizations for CUDA kernels. In our algorithms though, we examining i_{th} symbol of every variable-length string and strings are occupying continuous span in global memory. It is very difficult to organize efficient memory access in this context. Few things we can to do are two disable L2 cache with compiler options and prioritize L1 cache size over available shared memory with CUDA API call. Then instead of loading consequent symbols one by one for reading the prefix of the string we do one 32 bytes memory load into local buffer and then iterate over its. This gave us 15-20 performance improvement.

If we continue recursive sorting past the level when buckets are getting small enough - high branch divergence starts causing performance degradation. We found that optimal cut off level is about 6-8 symbols for small alphabets (like 4 symbols of genomic data) and 3-5 symbols for longer alphabets.

But the definite bottleneck is moving data to and from the device -it takes considerable share of time and it grows proportionally to the length of the key. Figure 3 shows maximal performance we can get depending on the size of the key. Top green shows maximal troughput if we only have to copy data to GPU.

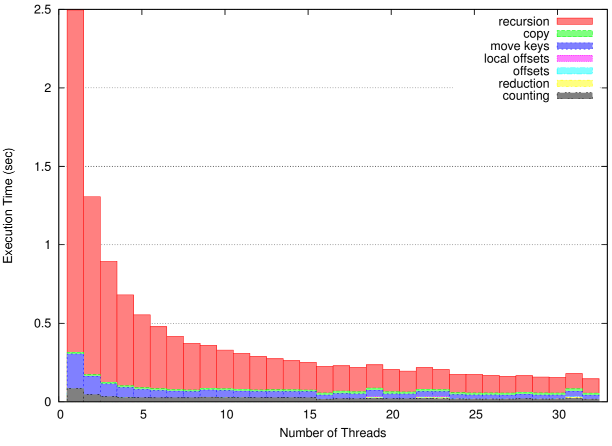

Fig. 5: Execution time breakdown for CPU

Line labeled *copy_kernel* accounts for moving sorted pointers back and also one kernel launch. Kernel launch overhead is about 20 microseconds even if it does not perform any work and we obviously need to launch at least one kernel. So this is the best possible performance for such an implementation.

CPU implementation, on the other hand has fixed performance irrelevant to the length of the keys - only to the overall number and statistical properties like the size of the alphabet.

To overcome this impediment we used the following technique.

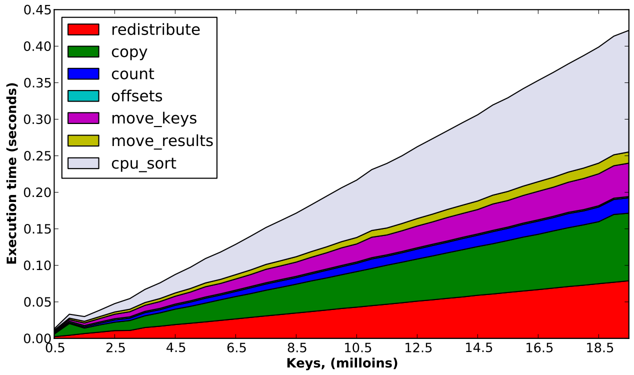

Fig. 7: Execution time distribution for second implementation

We chopped of first symbols of every string and repartitioned them into new memory block along with the pointers to original location. Then we transferred only this part to GPU and performed MSD radix sort there. Though the keys are seemingly fixed-length now, at leas for the GPU part - we can not use LSD radix sort, as its every iteration is oblivious of previous iterations and those information about partitioning is not preserved. After N iterations of MSD radix sort on the other hand we have strings sorted by N first symbols and also start and end position of every bucket as a by-product of an algorithm.

This re-partitioning of string of course is bringing additional overhead, but it is justified by overall performance improvement except for extremely short keys. We also parallelized re-partitioning process on CPU using OpenMP, although this process is memory-bound and does not scale much. New distribution of execution time is shown on Figure 7.

5. Conclusion

We have presented a hybrid MSD radix sort algorithm for GPU and showed that GPUs can cope with the problem of sorting string data. While MSD Radix Sort is less efficient for short/fixed sizes keys, it is suitable for variable-size and long keys. Our solution achieved 60 million strings per second sorting throughput. It is twice as fast as the same algorithm run on 12-core CPU and 20 times faster than string sorting done with standard sorting routine.

We shows that the same parallelization technique is applicable to traditional multicore processors too and yields high sorting throughput. It seem interesting to try the same approach on Intel MIC systems, this possibility will be investigated in future work.

References

[1] B., M. K.: Article: Analysis of Parallel Merge Sort Algorithm, *International Journal of Computer Applications*, Vol. 1, No. 1, pp. 66–69 (2010).

[2] Batcher, K. E.: Sorting networks and their applications, *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), Vol. 32, New York, NY, USA, ACM, pp. 307–314 (online), DOI: 10.1145/1468075.1468121 (1968).

[3] Bentley, J. and Sedgewick, R.: Fast algoprithms for sorting and searching string, *Proc. Annual ACM-SIAM Symp. on Discrete Algorithms*, New Orleans, Luisiana, ACM/SIAM, pp. 360–369 (1997).

[4] Bentley, J. and Sedgewick, R.: Sorting Strings with Three-Way Radix Quicksort, *Dr. Dobbs Journal* (1998).

[5] Bilardi, G. and Nicolau, A.: Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared Memory Machines, Technical report, Cornell University, Ithaca, NY, USA (1986).

[6] Cederman, D. and Tsigas, P.: GPU-Quicksort: A practical Quicksort algorithm for graphics processors, *J. Exp. Algorithmics*, Vol. 14, pp. 4:1.4–4:1.24 (online), DOI: 10.1145/1498698.1564500 (2010).

[7] Cole, R.: Parallel merge sort, *SIAM J. Comput.*, Vol. 17, No. 4, pp. 770–785 (online), DOI: 10.1137/0217049 (1988).

[8] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C.: *Introduction to Algorithms (3rd ed.)*, MIT Press and McGraw-Hill (2009 [1990]).

[9] Govindaraju, N., Gray, J., Kumar, R. and Manocha, D.: GPUTeraSort: high performance graphics co-processor sorting for large database management, *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, New York, NY, USA, ACM, pp. 325–336 (online), DOI: 10.1145/1142473.1142511 (2006).

[10] Govindaraju, N. K., Raghuvanshi, N., Henson, M., Tuft, D. and Manocha, D.: A cache-efficient sorting algorithm for database and data mining computations using graphics processors, Technical report, UNC (2005).

[11] Greß, A. and Zachmann, G.: GPU-ABiSort: Optimal parallel sorting on stream architectures, *IN PROCEEDINGS OF THE 20TH IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS 06) (APR*, p. 45 (2006).

[12] Hoare, C. A. R.: Algorithm 64: Quicksort, *Communications of the ACM*, Vol. 4, No. 7, p. 321 (1961).

[13] Kapasi, U. J., Dally, W. J., Rixner, S., Mattson, P. R., Owens, J. D. and Khailany, B.: Efficient conditional operations for data-parallel architectures, *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, New York, NY, USA, ACM, pp. 159–170 (online), DOI: 10.1145/360128.360145 (2000).

[14] Knuth, D. E.: *The Art of Computer Programming, Volume 3: Sorting and Searching*, Vol. 3, Addison-Wesley Professional (2011).

[15] Merrill, D. and Grimshaw, A.: Revisiting Sorting for GPGPU Stream Architectures, Technical Report CS2010-03, University of Virginia, Department of Computer Science, Charlottesville, VA, USA (2010).

[16] Purcell, T. J., Donner, C., Cammarano, M., Jensen, H. W. and Hanrahan, P.: Photon mapping on programmable graphics hardware, *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, Aire-la-Ville, Switzerland, Switzerland, Eurographics Association, pp. 41–50 (online), available from ⟨http://dl.acm.org/citation.cfm?id=844174.844181⟩ (2003).

[17] Sanders, P. and Hansch, T.: Efficient massively parallel quicksort, *Proceedings 4th International Symposium, IRREGULAR'97 Paderborn, Germany*, Springer-Verlag, pp.13–24 (online), DOI: 10.1007/3-540-63138-0_2 (1997).

[18] Sengupta, S., Harris, M., Zhang, Y. and Owens, J. D.: Scan Primitives for GPU Computing, *Graphics Hardware 2007*, San Diego, CA, ACM, pp. 97–106 (2007).

[19] Sintorn, E. and Assarsson, U.: Fast parallel GPU-sorting using a hybrid algorithm, *Journal of Parallel and Distributed Computing*, Vol. 68, No. 10, pp. 1381–1388 (online), DOI: 10.1016/j.jpdc.2008.05.012 (2008).

[20] Tian Xiaochen, K. R. and Suda, R.: Register Level Sort Algorithm on Multi-Core SIMD Processors, Technical report, Tokyo University (2013).

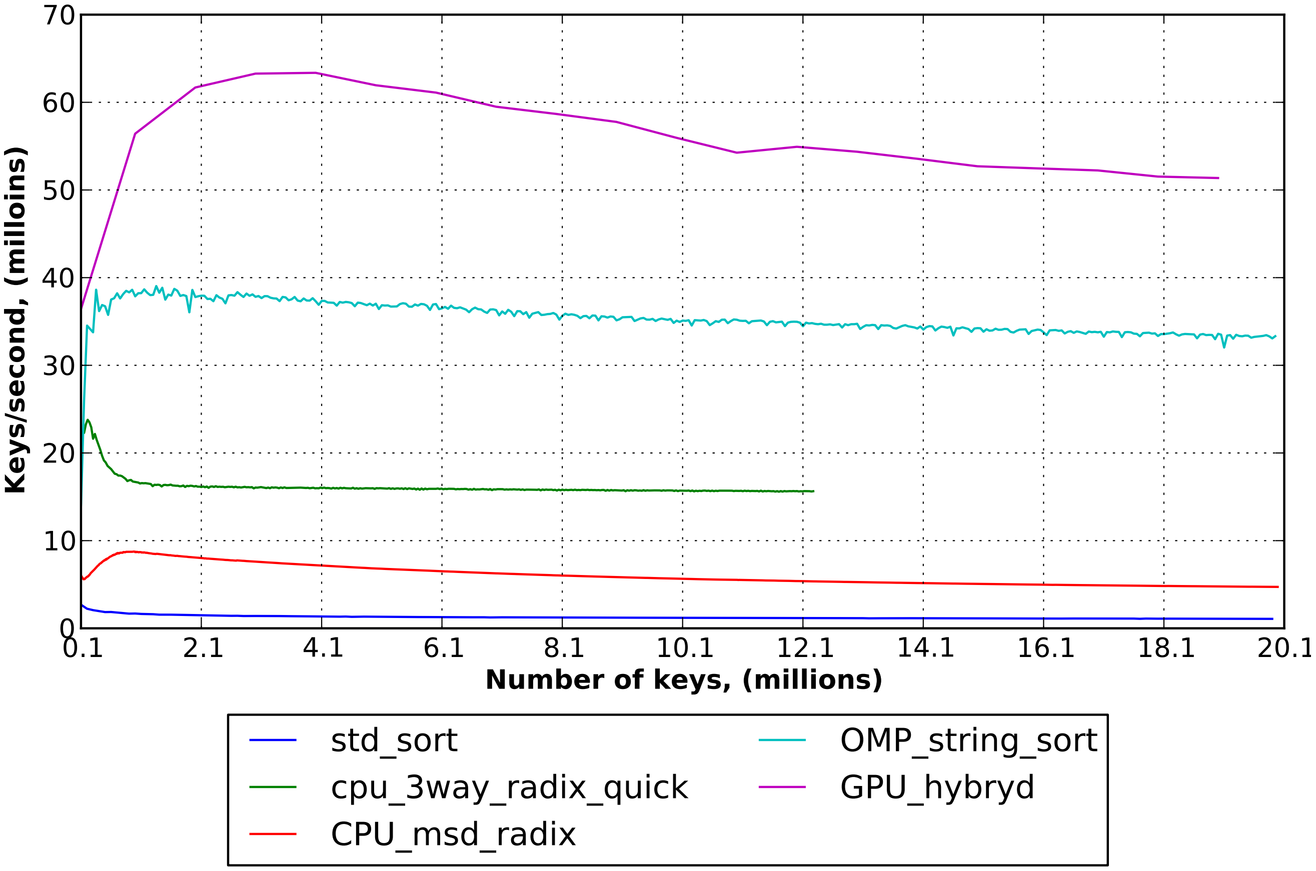

Fig. 6: Sorting throughput of improved implementation