Efficient String Sorting on Multi- and Many-Core Architectures

Aleksandr Drozd, Miquel Pericàs*
Tokyo Institute of Technology
Graduate School of
Information Science and Engineering
Meguro-ku, Tokyo 152-8550, Japan
Email: alex@smg.is.titech.ac.jp,
pericas.m.aa@m.titech.ac.jp

Satoshi Matsuoka
Tokyo Institute of Technology
Global Scientific Information
and Computing Center
Meguro-ku, Tokyo 152-8550, Japan
Email: matsu@is.titech.ac.jp

***Abstract*—This paper addresses the issue of efficient sorting of strings on multi- and many-core processors. We propose CPU and GPU implementations of the most-significant digit radix sort algorithm using different parallelization strategies on various stages of the execution to achieve good workload balance and optimal use of system resources. We evaluate the performance of our solution on both architectures and compare efficiency of the sorting algorithm on various key lengths. For the GPU implementation we introduce a communication-reducing strategy to overcome the limitations of the PCIe bus bandwidth. Both implementations achieve sorting rates up to 70 million keys per second sorting throughput with good scalability.**

***Keywords*-sorting, GPU, multi-core**

I. 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*, the efficiency of sorting algorithms becomes of increasing importance. Furthermore, many modern HPC applications do not rely primarily on floating-point computation. As *Big Data* converges with high performance computing, the share of time spent on integer and string processing is growing steadily.

This paper addresses the issue of efficient sorting of strings on multi-core and many-core processors, an important applicatoin which has not received as much attention as sorting of numeric data.

We describe our approach to parallelization of MSD radix sort and discuss its applicability to GPU and traditional multicore CPU architectures. Furthermore, we compare CPU and GPU implementations with regards to the efficiency of the sorting algorithm on varying key length.

A. Approaches to Sorting and Related Work

The classical comparison-based sorting algorithms such as merge- [1] or quick-sort [2] have asymptotic complexity estimation 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 $AverageLCP = \frac{1}{n-1}\sum_{i=0}^{n}(LCP(S_i, S_{i+1}))$ 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 [3]. 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,

* Current Address: Chalmers University of Technology, SE- 412 96, Gothenburg, Sweden. mpericas@acm.org

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 efficient parallel implementation is challenging to implement.

Recently one work dedicated to the GPU string sorting has been published by Deshpande et al. [4]. The authors also use MSD radix sort, emulating it by creating a table with key fragments and bucket IDs, sorting all the buckets simultaneously and then reloading the next key fragments. This allows them to achieve good workload balance, but the solution still suffers significant performance degradation on long keys due to the data transfer overheads. Also, when the input set contains small subsets of identical keys, for example in a set of strings where every key is repeated exactly twice, performance will also suffer due to the low warp utilization.

Another algorithm that is good for sorting strings is 3-way radix quicksort [5], [6]. This algorithm is in a way a combination of three-way quicksort and MSD radix sort. Classical quicksort works by recursively partitioning elements into two groups - less or equal to the pivot and greater or equal to the pivot. 3-way radix quicksort performs such partitioning based on a i_{th} symbol of the string and then for the "equal" partition proceeds recursively to the next symbol.

Burst-sort algorithm [7] also exploits data locality and can deliver slightly better compared to other sequential algorithms. However, because of its use of global data structures is not well-fit to parallel execution.

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 [8] and merge sort [9], [10], among others. There is also bitonic mergesort sort [11] 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, network-bound.

SIMD architectures are putting even more limitations on what can be implemented and, like the recently popular GPUs, provide totally different performance trade-offs. Distribution sorts which are inherently efficient for certain type of data have been especially successfully implemented on GPU. Radix sort which utilizes thread parallelism and high memory throughput was reported to be highly efficient on GPU [12]. They have also presented a quicksort implementation for GPU with inferior performance. To the best of our knowledge, the most efficient GPU radix sort is currently the one from Thrust library presented by Merril and Grimshaw [3].

Comparison-based sorts have also been implemented on GPU. Purcell et al. [13] presented bitonic merge sort on GPUs based on the work by Kapasi et al. [14]. Greß et al. [15] used the sorting technique presented in the Bilardi et al. paper [16] to implement GPU adaptive bitonic sort. Another GPU sort based on bitonic sort was implemented by Govindaraju et al. [17]. Later they presented a hybrid CPU and GPU solution using bitonic-radix sort in the Tera-Sort challenge [18]. 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 have been more successful attempts to implement quick sort on GPU [20]. There are ongoing efforts to optimize comparison-based algorithms for new architectures, e.g. by using vector instructions of modern processors [21].

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; and 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 [22]. There is also 3-way radix quicksort presented by Bentley and Sedgewick [5], [6], which is even more efficient due to more optimal use of caching. 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 buckets are relatively big). On the later stages, as buckets get smaller, we are switching to the 3-way radix quicksort.

II. Parallelizing MSD Radix Sort

The MSD radix sort algorithm is outlined in Alg. 1. Here S is the array of strings and $S[i][j]$ denotes j_{th} symbol of i_{th} string and S_{aux} is an auxiliary array. Even 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 sorting depth, i.e. the position of symbol we use for partitioning strings into buckets.

Input data strings are represented as an array of pointers to the actual stings are stored as a contiguous array of characters (Fig 1)

The naive approach to parallelization of a recursive algorithm would be to use task parallelism for every re-

Algorithm 1 MSD Radix Sort

```
procedure SORT($S, l, r, d$)
    for $i \in (l..r)$ do                                ▷ histogram
        $C[S[i][d]] \leftarrow C[S[i][d]] + 1$
    end for
    for $i \in (0..\sigma)$ do                           ▷ prefix sum
        $O[i] \leftarrow \sum_0^{i-1} C[i]$
    end for
    for $i \in (l..r)$ do                                ▷ moving
        $S_{aux}[O[S[i][d]]] \leftarrow S[i]$
        $O[S[i][d]] \leftarrow O[S[i][d]] + 1$
    end for
    $S \leftarrow S_{aux}$
    for $i \in (0..\sigma)$ do                           ▷ recursion
        if $(C[i] > 1)$ then
            $Sort(S, O[i] - C[i], O[i], d + 1)$
        end if
    end for
end procedure
```

cursion branch. Thus we will be doubling the number of parallel threads on every level of recursion. To partition N strings into the buckets we need to scan N symbols and then partition each of the sub-buckets the same number of symbols in total until the process starts encountering empty buckets, i.e. the amount of workload is the same for every iteration and takes $O(N)$ time in total. If the first iteration is only done by one thread and the second by σ (alphabet size) parallel threads etc. The possible speed-up of such an implementation is obviously limited.

This approach is even less efficient for GPU than for the classical multi-core architecture, since one thread on a GPU is relatively slow and high performance is achieved only when thousands of threads are running in parallel.

Another approach would be to parallelize every iteration of the algorithm, as is typically done for LSD radix sort. To build each bucket we basically count symbols and then move string pointers according to the counters. Counting can be efficiently parallelized within multiple threads when the workload is split into chunks for each thread to process.

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 processing small amounts of data with multiple threads 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 allows us to keep all the processor cores busy the entire time of execution. We suggest starting with parallel partitioning of strings into buckets and then, when buckets are small enough, continue processing each bucket independently in parallel. In the case of many-core CPU we seamlessly switch from one model to another and continue sorting until all the strings are in place. But on GPU, having many small unsorted buckets creates performance issues due to branch divergence rates; so we propose to use a hybrid approach in which the last stages of sorting are always performed on CPU. Moreover, when the recursion branches are independently executed by parallel threads we can seamlessly switch from MSD radix sort to the 3-way radix quicksort algorithm.

The following section describes our implementations for GPU and CPU and discusses different aspects of our model that also influence performance, such as data transmission costs, workload balance, and divergent branching in sorting code.

III. Implementation

For the multi-core CPU implementation we used OpenMP. Our GPU solution is implemented in CUDA, NVIDIA's programming platform for general-purpose computing on GPUs which is the current industry standard.

One of the resources we could use to increase the efficiency of radix sort is to process groups of symbols instead of one symbol at a time. 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 for radix sort with such grouping equals 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: $8 \times S^M$ bytes.

Furthermore, 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 $4^6 \times 8$ 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.

A. CPU Implementation

For the data-parallel part (*Histogram* and *MoveKeys*) (Alg. 1) we use OpenMP loop parallelization to split the workload between threads. Each thread builds a histogram of symbols (or groups of symbols) for a contiguous partition of the input set. The next step is to perform reduction on the resulting counters. Since OpenMP does not support reduction in arrays we perform it manually in parallel fashion, each thread having access to local counters of all the other threads. For *Histogram* and *MoveKeys* phases every thread works on exactly same subset of keys and maintains "local" pointers to the region where keys belonging to this thread are

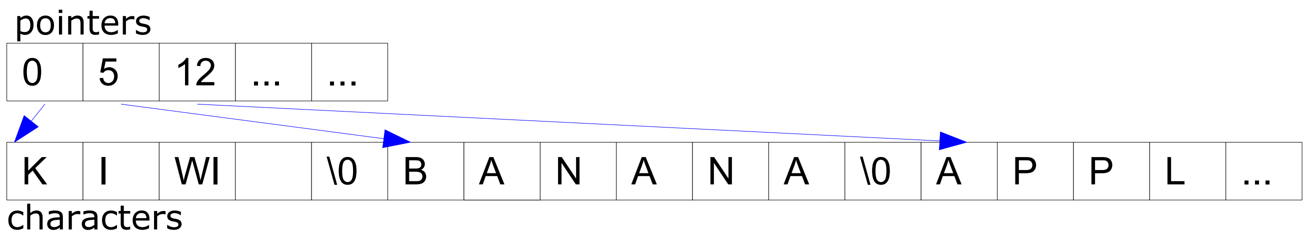


Figure 1: String Array in Memory

to be placed. Obviously, doing so multiplies the amount of memory we need for the counters by the number of threads, which further limits the size of the group of symbols that can be processed in one pass. Also with more threads we have to reduce more data (although this is partly compensated by the fact that we have more threads). Being memory-bound, the reduction stage exhibits considerable work time inflation. On the other hand, the overall contribution of this phase to the execution time is very small, and the performance degradation is insignificant.

In the second stage of the algorithm we use OpenMP task parallelism. In this phase, the implementation switches to the 3-way radix quicksort instead of MSD radix sort.

B. GPU Implementation

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 thousands of 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 at the same time).

Recursive algorithms are challenging to implement on GPUs because, as of now, only the recent Kepler architecture supports 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 data-parallel fashion; (2) processing the resulting (smaller) buckets in combination of data-parallel and task-parallel paradigms; (3) sorting the remaining small buckets with CPU threads.

To implement the first stage of the algorithm we make use of atomic operation to eliminate the reduction phase. Implementing reduction on GPU would be problematic due to the large memory requirements of multiple counters per thread. This approach does not slow down performance due to highly efficient implementation of atomic operations in the newest GPU architectures. On the other CPUs do not benefit from such approach as the number of threads in CPU implementation is much smaller.

As was mentioned above, the efficiency of radix sort could be increased by sorting groups of symbols instead of one symbol at a time. With the use of atomic operations, the amount of symbols that can 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: $8 \times S^M$ bytes.

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 imbalance 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.

IV. Performance Analysis and Optimization

A. CPU implementation performance

We begin by evaluating the overall performance of the implementation in terms of sorting throughput. Figure 2 shows the sorting throughput on different number of keys. We observed stable performance when all the threads are placed on cores belonging to a single socket and less regular performance when threads are scattered across the sockets. Analysis of time spent in each phase of the algorithm shows that the histogram computation on multiple sockets suffers from load imbalance which results in the observed performance variations.

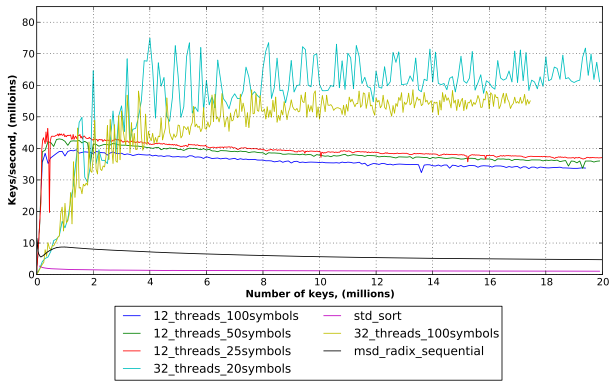


Figure 2: Sorting throughput

While measuring how the implementation responds to the increase of threads number we used the `numactl` tool to control thread allocation, forcing threads to first utilize all cores of the first socket and then start using the cores of the second socket. We also configure the system to interleave all memory across the NUMA nodes. The code was instrumented with our own low overhead instrumentation library called LoI* to measure the performance of phases and work performed within each phase. LoI collects timings for kernels and phases using only timestamp counters, and it reports information on average kernel times, variation between kernel times of different threads, phase execution times, and idleness/overheads. In this context, *idleness/overheads* refers to the amount of time in a phase in which threads are not performing useful work. It is a measure of both runtime overheads (such as OpenMP API calls) as well as load imbalance. Figure 3 shows the time spent in each phase for a workload consisting of 10 million strings on the 2-socket test system with 32 cores. The figure indicates that all phases scale similarly as the number of cores is increased. At small number of cores, the recursion phase improves quickly with number of threads. After approximately 10 cores, the gains become smaller and all phases improve at a similar rate. The figure also shows how the histogram phase suffers from performance variations when the number of threads spans more than one socket (>8 OpenMP threads). Figure 7 shows overall performance scaling for the 1-socket and 2-sockets system. The speed-up saturates at about 16× for the 2-socket system.

To further understand the performance of the parallel CPU implementation, we analyze the work performed within each phase. By evaluating average kernel times for single-threaded and multithreaded executions it is possible to measure the work time inflation suffered by the kernels due to resource sharing and the lack of useful work due to runtime overheads and parallel idleness. Both factors can be encoded as two factors OVR_N and WTI_N which represent the time stretch compared to the *ideal* parallel execution time at N threads:

$$T_{\text{parallel}} = \frac{T_{\text{serial}}}{N} \times \text{OVR}_N \times \text{WTI}_N$$

Measuring these factors provides a quick qualitative analysis of the bottlenecks present in each phase of the code.

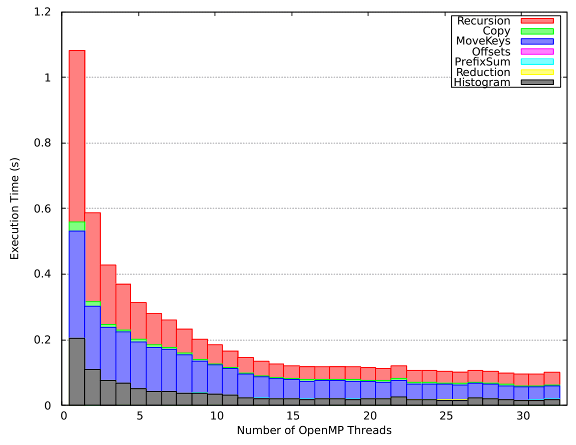
Figure 4 shows the results obtained when instrumenting the three main phases and kernels of the code: *histogram* -accumulating counters for different keys, *move keys* and *recursion*. We begin by analyzing the work time inflation in Figure 4 (a). The plot shows that *movekeys* suffers considerable WTI, increasing its total work by about 3× when all the cores of the first socket are populated with one thread. This phase is memory intensive and has bad locality, and it quickly saturates the memory subsystem. Starting to populate the second socket provides some relief by better exploiting the memory controllers available in the system, but at 32 threads the WTI peaks at more than 3.5×. As part of our future work we will consider locality optimizations to improve the work time of this phase. The *histogram* and *recursion* phases also suffer some WTI at large scale, reaching about 2× at 32 threads. The execution stretch due to idleness and runtime overheads is shown in Figure 4 (b). The low stretch of the *recursion* and *movekeys* phases indicate that these phases have a very well load balanced parallel execution. However, the *histogram* phase is problematic, increasing its execution time by 50% due to imbalance. The instrumentation indicates that all OpenMP threads execute the same number of iterations. The imbalance stems from the fact that different threads observe different work time inflations, with some threads occasionally performing up to 60% more work than others. This figure indicates that there is some room to improve performance by tweaking the data locality of the access patterns.

A straight-forward task-parallel implementation of the algorithm based on recursively spawning OpenMP tasks for each symbol showed inferior performance and scalability. The performance of this version is shown in Figure 7 compared to the multikey implementation analyzed here.

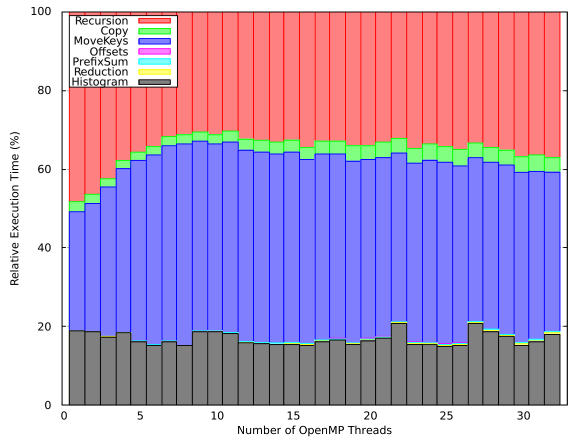
B. GPU implementation

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 are 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 to disable L2 cache with compiler options

* https://bitbucket.org/mpericas/loi-pub

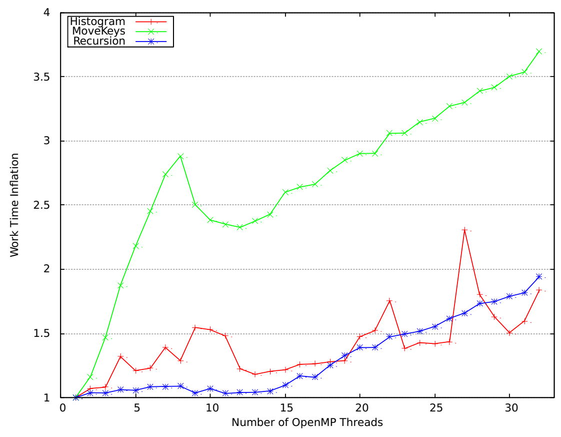


(a) Absolute time on 10^7 keys

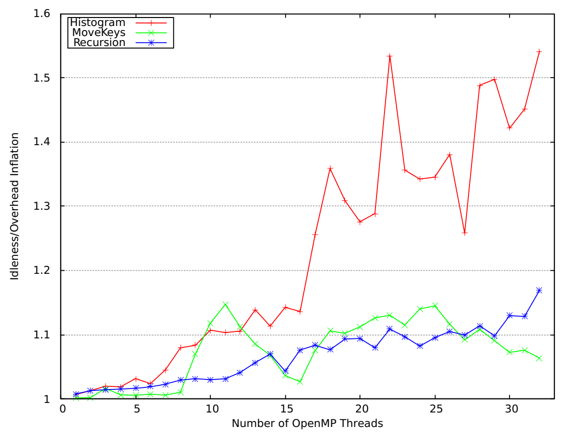


(b) Relative time on 10^7

Figure 3: Time spent in different phases



(a) Work time inflation



(b) Execution stretch due to imbalance and runtime overheads

Figure 4: Scalability analysis for 10^7 keys

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 for GPU implementation 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 5 shows maximal performance we can get depending on the size of the key. Top green line shows maximal throughput if we only have to copy data to GPU. Line

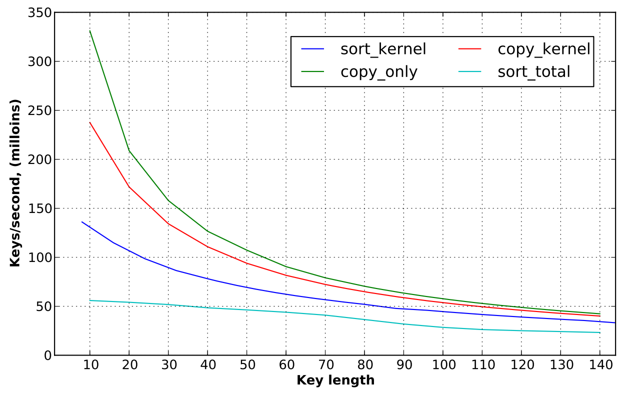


Figure 5: Correlation of performance and key length

labelled *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. Figure 6a shows time spent on each phase of sorting for hybrid implementation, particularly time spent on moving data to and from the device. Short keys (up to 20 symbols) were used for this experiment.

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. 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 least 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 strings 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 6b.

V. Conclusion

To the best of our knowledge, this is the first attempt to parallelize a sorting algorithm efficient for the processing string data. We presented our implementations of MSD radix sort for two parallel architectures (CPU and GPU). Our solution features a two-stage algorithm that balances different parallelization strategies to achieve good scalability.

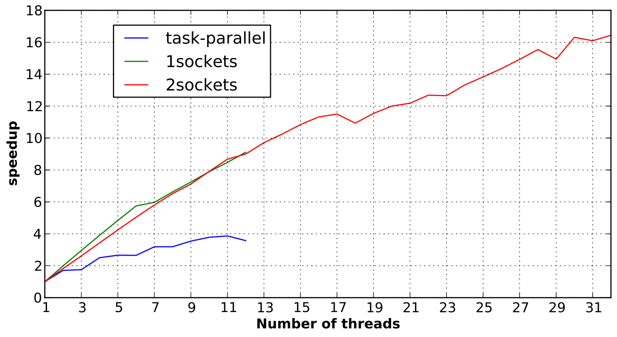


Figure 7: Scaling

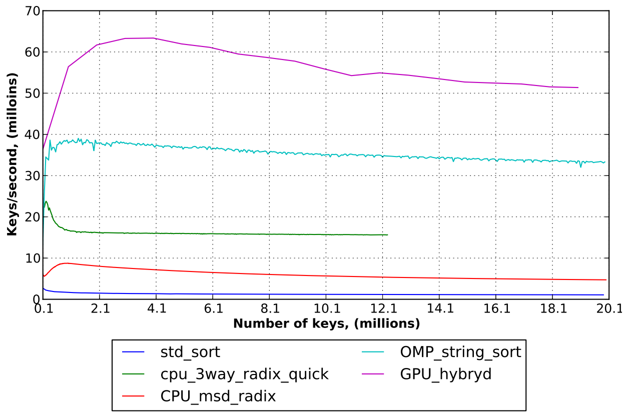


Figure 8: Sorting throughput of improved implementation

Performance analysis confirmed that MSD radix sort can be efficiently parallelized and achieve high performance on string data, especially on data with shorter alphabets. Figure 8 shoes sorting throughput for final hybrid implementation and for the CPU-Only implementation running on 1-socket system.

Our implementation also showed that when the keys are longer, MSD radix sort outperforms competing algorithms such as merge sort. We analysed performance to validate our approach and used locality optimizations. For GPU implementation we identified host-to-device communication overhead as a bottleneck and introduced a communication-reducing strategy to overcome this issue.

More research needs to be done on testing MSD radix sort on other parallel architectures, such as MIC. Another possible direction for further work is testing the applicability of this algorithm to distributed memory systems.

References

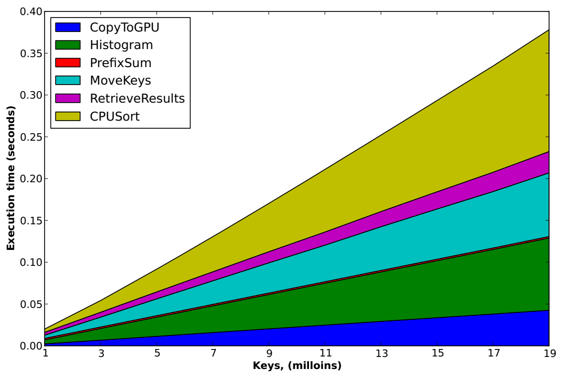
[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3rd ed.)*. MIT Press and McGraw-Hill, 2009 [1990].

[2] C. A. R. Hoare, "Algorithm 64: Quicksort," *Communications of the ACM*, vol. 4, no. 7, p. 321, july 1961.

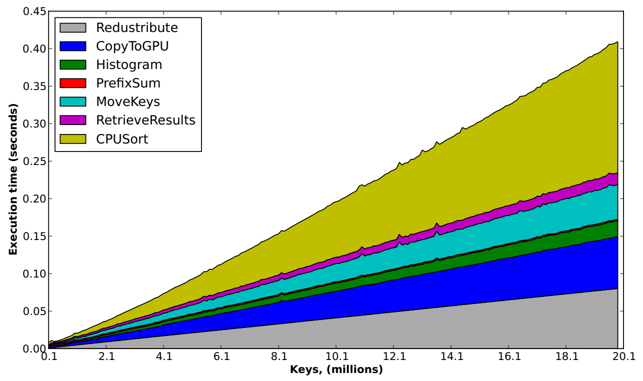
[3] D. Merrill and A. Grimshaw, "Revisiting sorting for gpgpu stream architectures," University of Virginia, Department of Computer Science, Charlottesville, VA, USA, Tech. Rep. CS2010-03, 2010.

[4] A. Deshpande and P. Narayanan, "Can gpus sort strings efficiently?" in *IEEE High Performance Computing (HiPC), 2013*, 2013.

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



(a) first implementation



(b) Implementation with re-partitioned strings

Figure 6: GPU execution time breakdown

[6] ——, "Sorting strings with three-way radix quicksort," *Dr. Dobbs Journal*, 1998.

[7] R. Sinha and J. Zobel, "Cache-conscious sorting of large sets of strings with dynamic tries," *J. Exp. Algorithmics*, vol. 9, Dec. 2004. [Online]. Available: http://doi.acm.org/10.1145/1005813.1041517

[8] P. Sanders and T. Hansch, "Efficient massively parallel quicksort," in *Proceedings 4th International Symposium, IRREGULAR'97 Paderborn, Germany*. Springer-Verlag, June 1997, pp. 13–24. [Online]. Available: http://dx.doi.org/10.1007/3-540-63138-0_2

[9] R. Cole, "Parallel merge sort," *SIAM J. Comput.*, vol. 17, no. 4, pp. 770–785, Aug. 1988. [Online]. Available: http://dx.doi.org/10.1137/0217049

[10] M. K. B., "Article: Analysis of parallel merge sort algorithm," *International Journal of Computer Applications*, vol. 1, no. 1, pp. 66–69, February 2010, published By Foundation of Computer Science.

[11] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, ser. AFIPS '68 (Spring), vol. 32. New York, NY, USA: ACM, 1968, pp. 307–314. [Online]. Available: http://doi.acm.org/10.1145/1468075.1468121

[12] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," in *Graphics Hardware 2007*. San Diego, CA: ACM, August 2007, pp. 97–106.

[13] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan, "Photon mapping on programmable graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ser. HWWS '03. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2003, pp. 41–50.

[14] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany, "Efficient conditional operations for data-parallel architectures," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 33. New York, NY, USA: ACM, 2000, pp. 159–170. [Online]. Available: http://doi.acm.org/10.1145/360128.360145

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

[16] G. Bilardi and A. Nicolau, "Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines," Cornell University, Ithaca, NY, USA, Tech. Rep., 1986.

[17] N. K. Govindaraju, N. Raghuvanshi, M. Henson, D. Tuft, and D. Manocha, "A cache-efficient sorting algorithm for database and data mining computations using graphics processors," UNC, Tech. Rep., 2005.

[18] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "Gputerasort: high performance graphics co-processor sorting for large database management," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '06. New York, NY, USA: ACM, 2006, pp. 325–336. [Online]. Available: http://doi.acm.org/10.1145/1142473.1142511

[19] E. Sintorn and U. Assarsson, "Fast parallel GPU-sorting using a hybrid algorithm," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1381–1388, Oct 2008. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2008.05.012

[20] D. Cederman and P. Tsigas, "Gpu-quicksort: A practical quicksort algorithm for graphics processors," *J. Exp. Algorithmics*, vol. 14, pp. 4:1.4–4:1.24, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1498698.1564500

[21] K. R. Tian Xiaochen and R. Suda, "Register level sort algorithm on multi-core simd processors," in *proceeding of the IA^3 Workshop on Irregular Applications; Architectures & Algorithms*, 2013.

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