Sorting Networks

NAMRATHA SANJAY, MARIA AGUILAR

ACM Reference Format:
Namratha Sanjay, Maria Aguilar. 2020. Sorting Networks. 1, 1 (March 2020), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Sorting can be termed as the re arrangement of data on arrays or lists according to a comparison operator on their elements, these comparison operators are used to decide the new order of elements in the respective data structure. While performing sorting in practice, the numbers to be sorted are rarely isolated values. Each number is generally part of a data collection called a record. Each record contains a key, which is the value to be sorted. Remainder of the record consists of satellite data, which are usually carried around with the key. Usually when a sorting algorithm rearranges the keys, it must rearrange the satellite data as well.

By definition, a sorting network is a combination of comparison and swap operations where the number of elements to be sorted is required to perform the operations and fix the sequence of comparisons [**Christophides2008**]. Sorting is one of the most widely studied algorithms as it is computational building block of fundamental importance. The importance of sorting has led to efficient sorting algorithms for variety of parallel architectures e.g. Batcher odd-even merge sort, bitonic sort etc. Database systems make extensive use of sorting operations. Sorting routines are also used in computer graphics where construction of spatial data structures is essential as well as geographic information systems. Efficient sort routines are also useful building blocks in implementing algorithms like sparse matrix multiplication and parallel programming patterns such as MapReduce. It is therefore important to provide efficient sorting routines on practically any programming platform, and as computing evolve there is a continuing need to explore efficient architectures to improve sorting algorithm performance.

Sorting networks have a wide range of applications on a variety of hardware architectures, however in this paper we limit to multi core CPU and GPU implementation and how performance is comparatively better on a GPU architecture than CPU. This is further discussed in the paper.

Author's address: Namratha Sanjay, Maria Aguilar.

XXXX-XXXX/2020/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 SORTING ALGORITHMS

There are several algorithms that have been developed for sorting networks, where the main difference is the number and type of functions they use for sorting, such as the Bitonic sort and Odd-Even merge sort algorithms by Kenneth E. Batcher.

Bitonic sort is a comparison-based sorting algorithm that can be run in parallel and can be used as a type of sorting network. In bitonic sorting a random sequence of numbers is split into two and is then converted into a bitonic sequence, one that increases monotonically then decreases. Initially unsorted sequence of numbers enters through input pipes, where a series of comparators switch two entries to be in either increasing or decreasing order eventually bitonic sequences are created recursively. It works by breaking the data set down to smaller bitonic sequences that are then merged, forming longer bitonic sequences. This eventually results in a bitonic sequence that is monotonically increasing then decreasing. The final step is to create one long sequence that is monotonically increasing, which is equivalent to a sorted data set.

Serial implementation of the bitonic sorting network completes its work in $O(nlog^2n)$ comparisons, which fail the ideal comparison-based sort efficiency of $O(nlogn)$. Parallel implementation of the sort, however, can lead to dramatic speedups, depending on the implementation. Although, bitonic sorting is fastest and provides good speed-ups for a small input size but fails for large inputs the execution time increased proportionally with the n inputs according to [1].
One of the keys to success in sorting algorithms development is taking advantage of parallelism by threads and SIMD (Single Instruction Multiple Data) instructions in multi-core processors to achieve high performance. Furthermore, most of the works suggest the shared-memory model and the use of processors over which a min and a max instruction could run[3]. Note that these operations are supported by x86-64 architectures.
Much of the work found that improvements might be focus on reducing key size. Furthermore, better results can be achieved by the combination of algorithms.

3 SURVEY - ARCHITECTURE ANALYSIS

Available research exists on a sorting-network implementation over a multi-core CPU and many core GPU, taking advantage of full programmability offered by CUDA (Compute Unified Device Architecture). The specifications are as follows, a Quad − Core i7 CPU, where each core has 4-wide SSE (SIMD) and two SMT (Simultaneous multithreading) threads. The threads share 32KB L1 cache and 256KB L2 cache. Within this model, about 17 SSE instructions are needed to perform 4-wide steps of bitonic merge. Overall, a total of 4.25 instructions/element are produced by 4 elements, obtaining a performance of 2.5 cycles/element/iteration per core. On the other hand, a GPU implementation was performed over the multicore NVIDIA GPU GTX 280, where each core can have up to 32 threads. This configuration led to 2 instructions/element needed by 16-wide bitonic network. Considering 4 cycles needed by instruction, it results on a performance of 9 cycles/element/iteration per core [5].
According to the results reported above, there is a considerable performance improvement running the merging-sort algorithm over a GPU than CPU. Moreover, Table 1 shows the best reported running times of a comparison, ~~played~~ by Satish[5], for sorting networks algorithm implemented over the CPU and GPU platforms described.

	CPU	GPU
BW [Gbps]	30.0	141.7
GFlops	103.0	1933.3
256 K	1.2 ms	1.3 ms
1 M	15.3 ms	5.0 ms
4 M	23.3 ms	21.6 ms
16 M	101.5 ms	94.5 ms
64 M	439.7 ms	1381.8 ms

Table 1. Performance comparison of merge sort algorithm over two different architectures

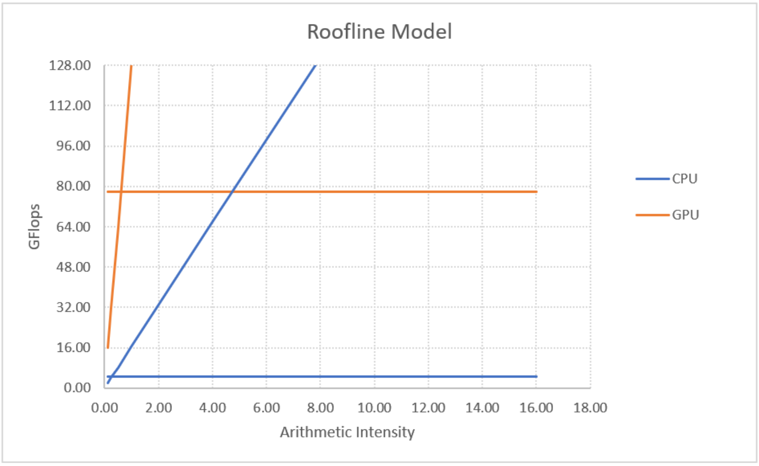

Fig. 1. Roofline models for CPU and GPU

The roofline model depicts the differences between CPU and GPU architectures for sorting algorithm. Some assumptions have been considered such as cache blocking to help sorting. One of the most important facts from the roofline model is the memory bandwidth, which is noticeable higher in GPU than CPU. This advantage leads to better computation rates [4].A total of $(n + nlog_2(n/4))$ comparisons are performed by the algorithm to sort a sequence of length n. According to the number of instructions per model reported before, the merge-sort algorithm is considered compute bound for both CPU and GPU architectures[2][4].

Finally, ongoing research on sorting optimization refers to hardware improvements, especially parallelism of memory access via GPU. Another trend is considering hybrid architecture allowing multiple CPU cores to simultaneously send data to a single GPU.

4 DISCUSSION

In this section we discuss about the performance, energy efficiency of algorithms and its impact due to various hardware features and also merits of the architecture.

As the ~~advent of~~ advancement in hardware architectures is inevitable the performance of algorithms are decided on the basis of how they utilize the hardware features. We now discuss on some of the architectural advancements and their impact on sorting performance in context of database operations.

4.1 Thread-Level and Data-Level Parallelism

Thread-Level Parallelism (TLP) is easier to exploit for most of the algorithms as current processors have increased compute power by adding more cores to exploit TLP and adding SIMD units to exploit DLP. Many algorithms that sort in parallel have been proposed for multi-core systems such as merge sort and quick sort naturally involve combining or splitting different blocks of data in parallel
Data-Level Parallelism is harder to exploit in certain sorting algorithms. To effectively utilize the SIMD units in a multi-core system the data to be sorted needs to be contiguously laid out in memory. In the absence of contiguous accesses, gather/scatter to memory are required, which are slow on most architectures. However as SIMD widths have been increasing, sorts implemented using SIMD are becoming more efficient.

4.2 Memory Bandwidth

Sorting involves rearrangement of data residing in memory and thus it is typically memory intensive. The bandwidth of memory has not much improvement as much as the computational capacity of modern day processors. Therefore algorithms that require high memory bandwidth are likely to become bandwidth bound and hence stop scaling as they can't match up to the increasing number of cores and SIMD width. In order to overcome this bandwidth dependency of certain algorithm or application, architectures have introduced on-chip local storage in the form of cache hierarchies on CPUs and shared memory on GPUs.

4.3 Latency Effects/ILP (Instruction Level Parallelism)

Inefficient or low utilization of functional units due to instructions with high latency are because they block the execution of dependent instructions. This is observed due to last level cache misses that have long latency memory accesses. In addition to cache misses, misses to auxiliary structure called Translation Lookaside Buffer (TLB) which is used to perform a conversion from virtual to physical memory addresses, can also result in significant performance degradation. In order to overcome this drawback caches and TLBs are organized such that they have minimal misses when physically contigious regions of memory are accesssed such accesses are called streaming accesses. Algorithms that have streaming access pattern therefore have minimal impact from cache and TLB misses. Among sorting algorithms, merge sort has a streaming access pattern, resulting in low misses.

Since sorting algorithms are working on large data sets and have memory bandwidth constraints it is easier to fall on the memory bound side, in order to avoid becoming memory bound as it might dominate the compute requirements of the merge network, merge sort can perform the first few iterations in cache when array size is small, the passes only read and write data to cache with a single read of data from/to main memory. In case of large size of array which is too large to reside in cache then use multi-way scheme data only needs to be read and written once more from main memory this is the case for a CPU multi-core system. However this is not necessary for a GPU platform, because the number of compute instructions is higher on GPU due to overhead of shuffle operation and also the bandwidth-to-compute ratio of the GPU platforms is higher than CPUs. Even a simple implementation is about 4X away from bandwidth bound.

5 POTENTIAL DESIGN PROPOSAL

Our proposal for design of architectural feature is based on the trend with continuous increase in number of cores in the processors and the GPU. Although the core count is not projected to grow along with the memory bandwidth, it is one of the ways some recent works point out for performance improvement, considering power constraints. Furthermore, some distinguished architectural features, such as arbitrary gather/scatter, general interleave instructions and unaligned memory access support should be considered for better performance in future architectures which are intended to increase SIMD width up to 64-wide and beyond[6].

In order to reach an effective GPU sort, the improvements must be centered on either increasing the effective use of on-chip memory and registers, or decreasing memory contention on the algorithm [2]

6 CONCLUSION

Sorting networks capable of sorting thousands of items in the order of microseconds can be constructed with present-day hardware. Such fast sorting capability can be used to manipulate large sets of data quickly and solve some of the communications problems associated with large scale computing systems.
Much of the research related to sorting algorithms is focused on hardware improvements instead of software optimizations to take advantage of parallelism. Architectural features may be exploited for better performance. In modern processors, some authors recommend take advantage of cache blocking, SIMD vectoring, work partitioning, load balancing, and multi-way merging. Some obtained improvements have been latency decreasing, compute density increasing, and bandwidth bound stages elimination (for large input sizes).
Even though the gap narrows between CPUs and GPUs, and synchronization and coalesced global memories represent barriers on GPUs, they are still the most suitable option to perform sorting algorithms since most of the works show they got better performance with many-core GPUs against multi-core CPUs.

REFERENCES

[1] Nancy M. Amato et al. “A Comparison of Parallel Sorting Algorithms on Different Technical Report 98-029 Department of Computer Science”. In: November (1996).

[2] Dmitri I. Arkhipov et al. “Sorting with GPUs: A Survey”. In: (2017). arXiv: 1709.02520. URL: http://arxiv.org/abs/1709.02520.

[3] Timothy Furtak, José Nelson Amaral, and Robert Niewiadomski. “Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms”. In: *Annual ACM Symposium on Parallelism in Algorithms and Architectures* (2007), pp. 348–357. DOI: 10.1145/1248377.1248436.

[4] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Amsterdam: Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.

[5] Nadathur Satish et al. “Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* October 2014 (2010), pp. 351–362. ISSN: 07308078. DOI: 10.1145/1807167.1807207.