- Parallel GPU based Sorting Algorithms - Optimization and Performance

John Martinez (JJM4562) Pankaja A. (PA6979)

Abstract

This paper presents a comparative performance analysis of Parallel GPU based sorting algorithms. We implemented different versions of **`BrickSort`** and **`RadixSort`**, and compared them in different NVIDIA GPU architectures by using CUDA. Moreover, we describe optimization techniques used to make these algorithms more efficient in each newer version, including tools, libraries and methods.

1. Introduction

An important operation in any database system, sorting using an efficient algorithm can drastically improve the overall performance of the system. Data-driven algorithms especially use sorting to gain efficient access to data. Many sorting algorithms on sequential and parallel architecture have been developed, all looking to minimize the time to perform the function. Moreover, with the progress of general–purpose computing on GPUs, lot of efforts have been put in developing high performance GPU-based sorting algorithms. CUDA is a parallel computing framework developed by NVIDIA, providing access to several features in their GPUs, such as shared memory, scattered reads, faster downloads, readbacks to and from GPU, and full support for integer and bitwise operations. These features make CUDA a highly desired and efficient parallel computing framework, which can easily extract the computing capacity of modern GPUs.

2. Description of the project

We implemented two known and basic algorithms and utilized various optimization techniques in our implementations. **Brick sort** runs in two phases, one odd and the other even. In the odd-phase, odd-labeled processes exchange values with right neighbors. In the even-phase, even-labeled processes exchange values with right neighbors. The algorithm for parallel **`brickSort`** has two stages. The first is a parallel **`sort`** performed at the block level using shared memory as follows:

```
1.  procedure ODD-EVEN_PAR(n)
2.  begin
3.      id := process's label
4.      for i := 1 to n do
5.      begin
6.          if i is odd then
7.              if id is odd then
8.                  compare-exchange_min(id + 1);
9.              else
10.                 compare-exchange_max(id − 1);
11.         if i is even then
12.             if id is even then
13.                 compare-exchange_min(id + 1);
14.             else
15.                 compare-exchange_max(id − 1);
16.     end for
17. end ODD-EVEN_PAR
```

After each block has performed its **sort**, their contents are recursively merged using the same compare-exchange methods in global memory. In the other hand, **Radix sort** starts by partitioning the array into bits starting from least (LSD) to most significant bit (MSD). After every pass on the array, once each element's correct position has been determined, the algorithm is done and repeats for the next MSD. To find the exact position, an **exclusive scan** (Blelloch's) of the array is performed along with other operations as explained below:

1. **Partition (transform)** by bit - starting from LSD to MSD filter the array.
2. Resulting array will be of **0s** for *True* elements and **1s** for *False* elements.
3. Perform **exclusive scan** to calculate new indices for *True* elements.
4. Perform **exclusive scan** to calculate new indices for *False* elements.
5. **Scatter** all elements into a new array using new indices. Repeat for next bit.

We first implemented a non optimized, parallel version of these two algorithms and tested them using varied sized arrays of random numbers. Although our implementation produced correct results, we were not happy with their performance as we were aiming for our code to produce quicker results, accurately and consistently. Hence, we explored various optimization techniques which we thought would cater to the latter. In the next section, we describe the optimization techniques we took into consideration.

2.1 CUDA Optimization techniques

We started by optimizing data transfers. We minimized any data transfers between host and device as much as possible. Moreover, we started creating more intermediate data structures on the device, which operated

without being copied or mapped back to the host. Also, any small transfers were batched into one large transfer. We then followed by making more use of shared memory, mostly by using dynamic allocations thus bringing data closer to the ALUs, and by minimizing bank conflicts by tuning data access alignment (i.e. stride) to a warp size as we describe below. To improve data access time, we started using structs/unions of arrays hoping to take advantage of memory read/write coalescing and exposed more vectorization opportunities by checking the results using NVIDIA's profiler **`nvprof`**. Among other CUDA tools we used, we inspected the assembly code generated by the compiler using **`nvdisasm`** and tried defining our own inline PTX single instructions. Likewise, we made use of more atomic instructions and tuned the code to the running architecture on each target platform by using specific compiler flags. Additionally, we tried minimizing divergent branching by restructuring a lot of the `if-statements` in our code and avoided global sync whenever possible. This optimization was specific for flow execution by trying to create enough work for all threads.

3. Design alternatives

Still not content with our results, we turned our attention towards GPU accelerated libraries and decided to use *Thrust* and *CUB* mainly, and *ModernGPU* to a lesser extent. *Thrust* is a parallel algorithms library similar to C++'s STL, used to abstract away many of CUDA's lower level details, like hiding the complexity of memory management. Indeed, its strength does rely on such abstractions and rich library of memory management, iterators and algorithm functions that simplify a lot of the GPU coding and make prototyping very fast. However, it was not suited for our needs since it was too high-level. In contrast, *CUB* (CUDA UnBound) is a configurable C++ template library of parallel primitives and other utilities for constructing CUDA kernel software. *CUB's* strength lies in the granularity of said primitives; it provides abstractions for complex block-level, warp-level, and thread-level operations. As a result, this makes it both highly adaptable to fit the needs of any enclosing kernel computation, and tunable to different grain sizes. This suited our needs and, with *CUB*, we were able to first get an idea of the target's potential to sort data by employing different byte alignments using its **`DeviceRadixSort`** class methods on a 32-bit (`int`) cache page. We tested 1 to 4 byte alignments in all three targets on all sizes of arrays. Furthermore, we fine-tuned our

code according to the the target architecture by employing **`BlockRadixSort`** methods with different combinations of threads-per-block sizes vs. elements-per-threads . This was done keeping in mind the warp size of 32 that is predominantly suggested by the literature referenced.

4. Performance Tests and Results

We decided to sort single value (known as key-only) **`int32`** arrays of different sizes ranging from 32K (2^{15}) up to 32M (2^{25}) in log steps, each with non-uniform randomly generated values using time as a seed. As mentioned before, we targeted 3 different platforms and tuned our code accordingly to extract the best results possible. On all of them, we used the newest CUDA 8.0 toolkit and optimized the nvcc compiler flags to accommodate for each. The platforms are:

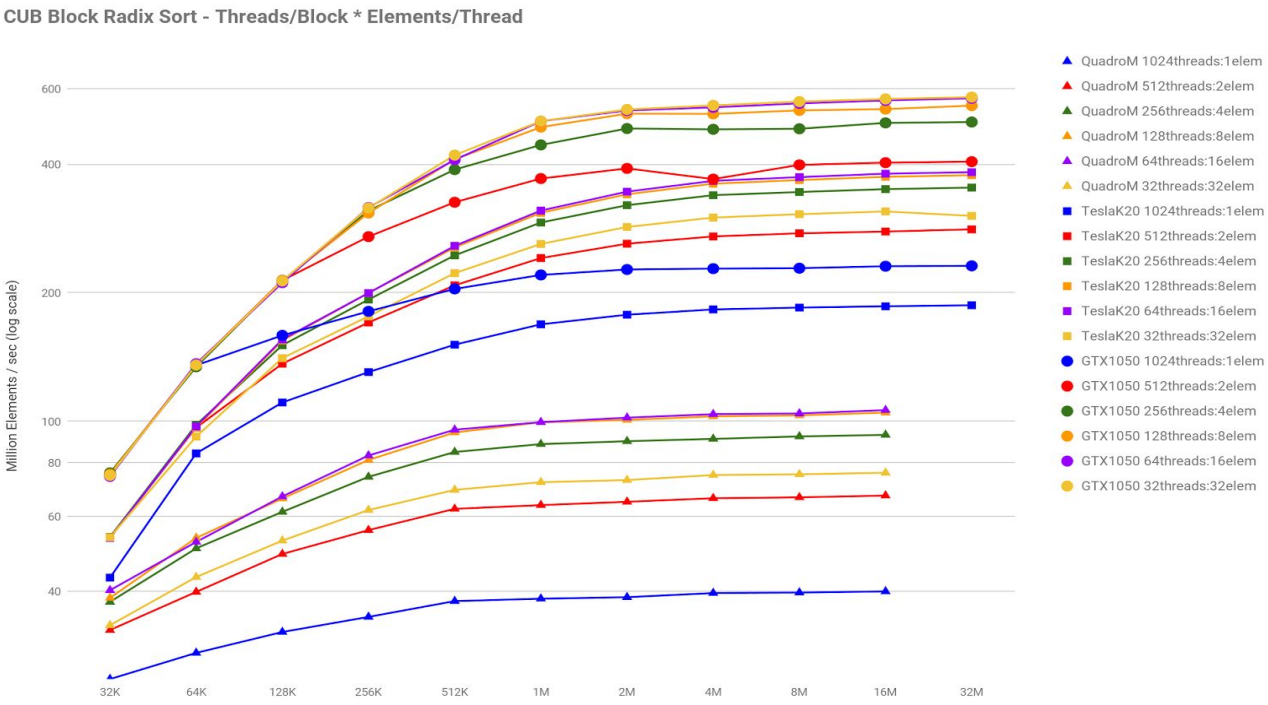
- **TACC's Stampede Computer,** which has accelerators using Tesla Kepler 20m GPUs. These have a 3.5 compute capability (CC), 4.6GB of VRAM with a 320-bit bus running at 2.6GHz. With 13 Stream Multiprocessors (SM), each with a grid of 192 CUDA cores (ALUs) running at 706 MHz, the Tesla K20m has a maximum theoretical throughput of 3.52 TFLOPS.
- A **gaming desktop** with a new generation, mid/low-range 6.0 CC GeForce GTX 1050 Ti GPU card, which has 3.9 GB of VRAM with a 128-bit bus running at 3.5 GHz. It has a Pascal architecture with 6 SMs, 128 ALUs each, running at 1.45 MHz giving it a maximum theoretical throughput of 2.23 TFLOPS.
- A **workstation laptop** with a 3.0 CC Quadro K1100 Mobile GPU, which has 2.0 GB of VRAM with a 128-bit bus running at 1.4 GHz. It has a Kepler architecture with 2 SMs, 192 ALUs each, running at 706 MHz giving it a maximum theoretical throughput of 0.54 TFLOPS.

The following are the results for our preliminary sorting capacity tests using different byte alignments per **`int`** type cache page, without any validation of the resulting data nor optimization for the buffer size on the operations.

	TeslaK20m		GTX1050Ti		QK110M	
	16M	32M	16M	32M	16M	32M
	16777216	33554432	16777216	33554432	16777216	33554432
1 byte	3741.387	3794.751	2860.851	2897.852	648.761	644.658
2 bytes	1965.466	1993.259	2020.045	2037.466	345.318	343.169
3 bytes	1374.813	1395.499	1540.866	1558.836	242.115	235.601
4 bytes	998.904	1014.764	1092.795	1108.731	174.242	173.613

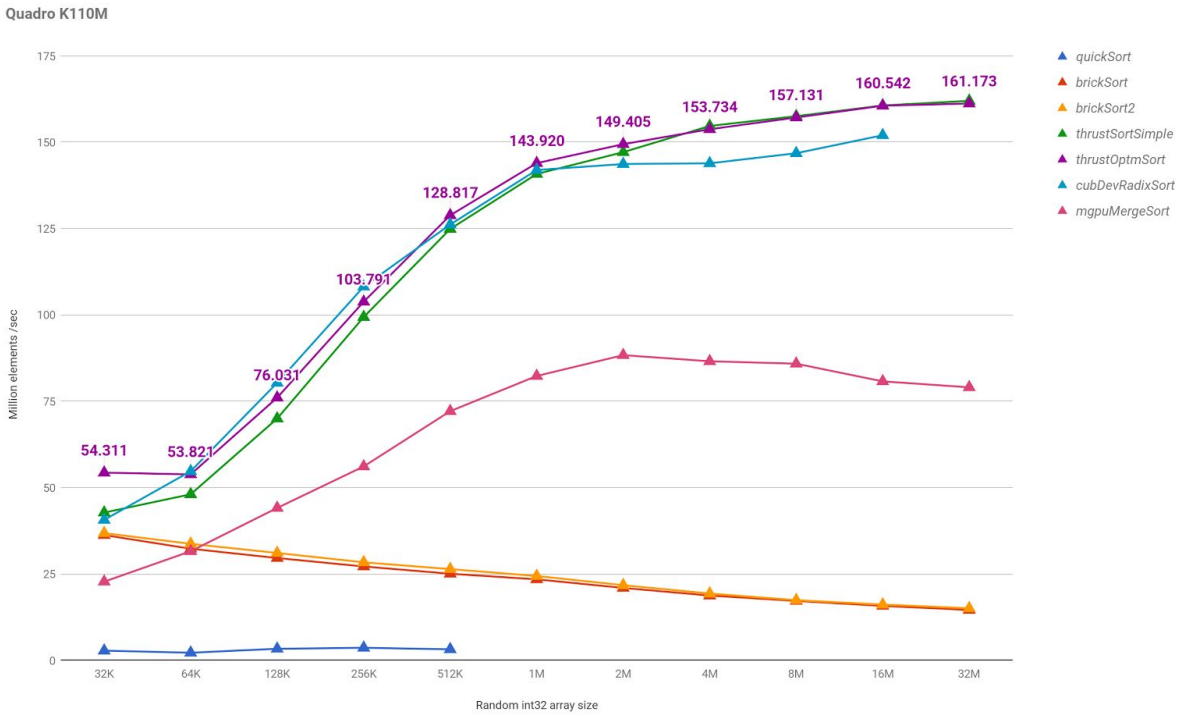
Potential sorting capacity using different alignments with CUB's DeviceRadixSort
Results in Million Elements per Second

Out of curiosity, we did check the validity of the results a few times and realized that to properly use *CUB* libraries, you must allocate buffers up to 8x (!!) the size of the data being inputted and use 4 bytes alignment, the slowest, to sort correctly. This caused a drastic drop on the targets' capacity to sort. We're still not sure if this is was mistake in our implementation or a weakness of the library, but following tests using other libraries seems to point to the latter. After that, we decided to tune our **`brickSort`** code by replacing our shared memory **`sort`** kernel with *CUB's* optimized **`BlockRadixSort`** class methods, while keeping our own **`merge`** kernel implementation. We also ran tests using both **`DeviceRadixSort`** and **`BlockRadixSort`** methods with different combinations of threads/block (tpb) versus elements/threads (ept), while maintaining grid sizes of multiples of 1024 to find the best suited allocations.

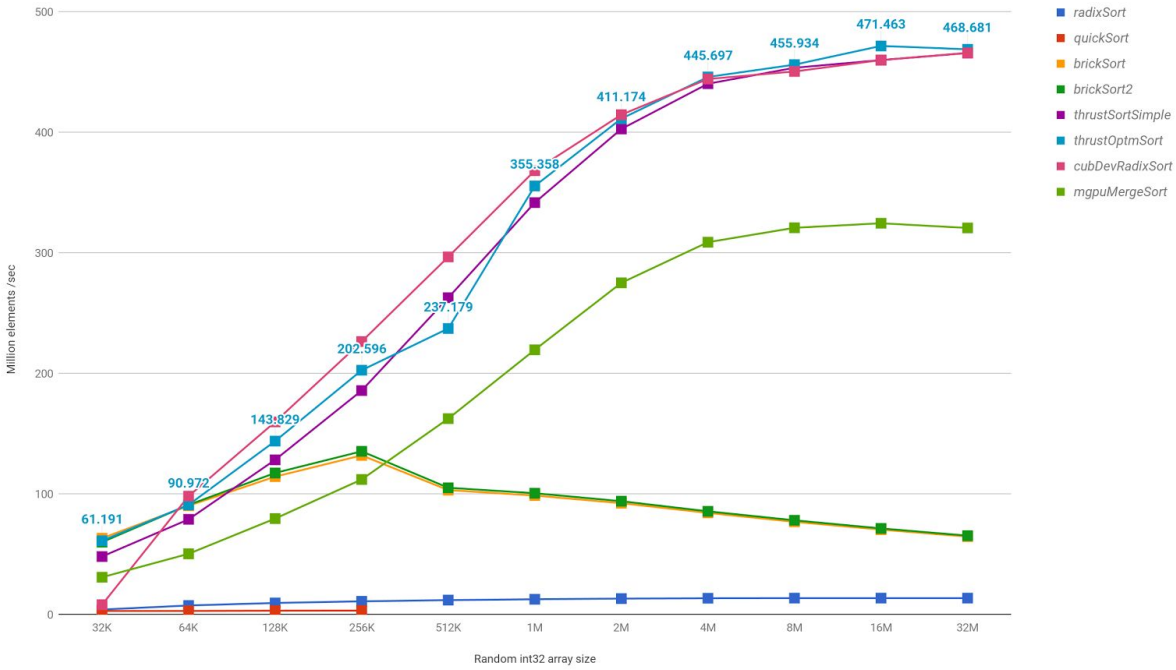


The Kepler architecture based GPUs seemed to perform better with allocations of either `64 tpb * 16 ept` or `128 tpb * 8 ept`, while the newer Pascal based GPU beat the others when using a grid allocation of `32 tpb * 32 ept`, probably due to the newer, optimized set of assembly instructions in it and its higher ALU clock frequency.

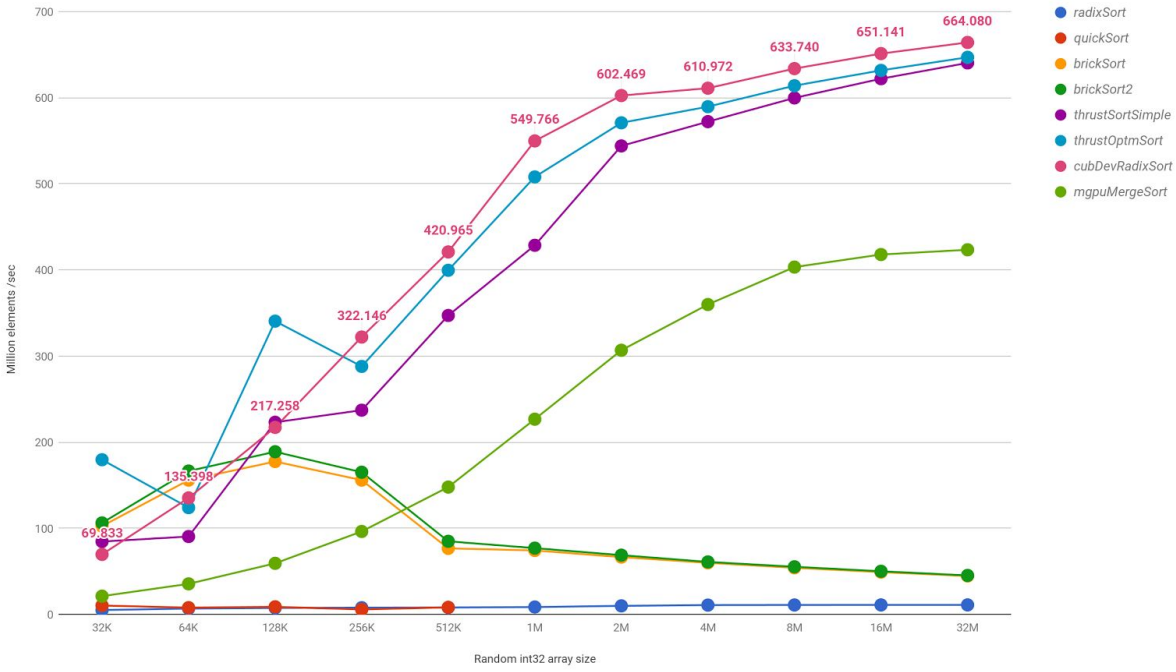
For our main comparison test we specifically measured the time to load the array onto the device and sort it. We added the memory copy time to maintain it as a constant while averaging the sort time in the different targets. We ran **`thrust::sort`** (supposedly the fastest GPU sorting currently) in a naive way, and another fully utilizing all the possible tuning afforded by the *Thrust* library. To compare against the latter, we ran our own code using *CUB's* device-wide **`DeviceRadixSort`** class members, since the *Thrust* library uses them under the covers for its sort implementation. We also ran our original, non-library, **`brickSort`** and its second variant as previously described. In addition, we ran our very naive, non optimized implementation of **`radixSort`** using our own **`scan`**, **`transform`** and **`scatter`** kernel code. Lastly, we found the time to ran a very simple execution *Modern GPU (mgpu)* library's sort too. The results are in million elements per second, overall best sorting method has values printed out for each run.



TeslaK20m



GTX1050Ti



5. Conclusions

No matter what we did, the libraries methods outperformed our own kernel methods. What's more, replacing a few of the latter methods with CUB's improved the performance of the whole sort exercise by reducing that specific operation by more than half the time, as shown below:

```
==23750== Profiling application: ./brickSort2_ 262144
==23750== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 82,23%  49,615ms      3712  13,366us  8,9910us  149,09us  oddEvenMergeGlobal(int*, int*, int, int)
 12,96%  7,8163ms        32  244,26us  243,16us  248,89us  oddEvenSortShared(int*, int*, int)
...
==23828== Profiling application: ./brickSort3_ 262144
==23828== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 88,83%  50,196ms      3712  13,522us  8,9270us  158,53us  oddEvenMergeGlobal(int*, int*, int, int)
  6,01%  3,3916ms        32  105,99us  103,33us  107,23us  void cubBlkSort<int>(int*, int*)
...
```

Still, we wish we had more time. Due to its constraints, we were unable to try more optimization approaches such as concurrent kernel execution or using `#pragma` calls for unrolling `for-loops` when threads were targeting shared memory, along with many other techniques like those mentioned by *Merrill and Grimshaw* in their excellent paper in our references. We also wished we had more time to optimize our implementation of our **Radix Sort**, or even try the shared memory `smem_merge` algorithm described by *Hou, et al*, also in our references.

On another matter, given our results, we see now why the CUDA platform is so popular. Although it's not as user-friendly as many would want you to believe, the fact that a new generation, inexpensive, low-end GPU was able to outperform a very costly, top-of-the-line GPU accelerator from a previous architecture, proves its appeal and accessibility to anyone wanting to optimize any computing task.

6. References

Bell, Nathan, and Jared Hoberock. "Thrust: A productivity-oriented library for CUDA." GPU computing gems Jade edition 2 (2011): 359-371.

Bozidar, Darko, and Tomaz Dobravec. "Comparison of parallel sorting algorithms." arXiv preprint arXiv:1511.03404 (2015).

Capannini, Gabriele, Fabrizio Silvestri, and Ranieri Baraglia. "Sorting on GPUs for large scale datasets: A thorough comparison." Information Processing & Management 48.5 (2012): 903-917.

Dehne, Frank, and Hamidreza Zaboli. "Parallel Sorting for GPUs." Emergent Computation. Springer International Publishing, 2017. 293-302.

Faujdar, Neetu, and S. P. Ghrera. "Performance Analysis of Parallel Sorting Algorithms using GPU Computing."

Harris, Mark. "How to Access Global Memory Efficiently in CUDA C/C++ Kernels." [Online] NVidia Corporation (2013).

Harris, Mark. "How to Optimize Data Transfers in CUDA C/C++." [Online] NVidia Corporation (2012).

Hou, Kaixi, et al. "Fast segmented sort on GPUs." ACM Int'l Conf. on Supercomputing (ICS). 2017.

Jan, Bilal, et al. "Fast parallel sorting algorithms on GPUs." International Journal of Distributed and Parallel Systems 3.6 (2012): 107.

Kipfer, R. Westermann P. "Chapter 46. Improved GPU sorting." GPU Gems 2 (2005).

Luitjens, Justin. "Faster parallel reductions on Kepler." [Online] NVidia Corporation (2014).

Ma, Lin, Roger D. Chamberlain, and Kunal Agrawal. "Analysis of classic algorithms on GPUs." High Performance Computing & Simulation (HPCS), 2014 International Conference on. IEEE, 2014.

Merrill, Duane, and Andrew Grimshaw. "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing." Parallel Processing Letters 21.02 (2011): 245-272.

NVidia, CUDA. "Binary Utilities: Application Note" NVIDIA, Santa Clara, CA (2017).

NVidia, CUDA. "C best practices guide." NVIDIA, Santa Clara, CA (2017).

NVidia, CUDA. "Programming guide." NVIDIA, Santa Clara, CA (2017).

Polychroniou, Orestis, and Kenneth A. Ross. "A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort." Proceedings of the 2014 ACM SIGMOD international conference on Management of data. ACM, 2014.

Rajput, Ishwari Singh, Bhawnesh Kumar, and Tinku Singh. "Performance comparison of sequential quick sort and parallel quick sort algorithms." International Journal of Computer Applications 57.9 (2012).

Satish, Nadathur, Mark Harris, and Michael Garland. "Designing efficient sorting algorithms for manycore GPUs." Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on. IEEE, 2009.

Singh, Dhirendra Pratap, Ishan Joshi, and Jaytrilok Choudhary. "Survey of GPU Based Sorting Algorithms." International Journal of Parallel Programming (2017): 1-18.

Yildiz, Zehra, Musa Aydin, and Guray Yilmaz. "Parallelization of bitonic sort and radix sort algorithms on many core GPUs." Electronics, Computer and Computation (ICECCO), 2013 International Conference on. IEEE, 2013.

Presentation charts: https://goo.gl/wk9mfm