Comparative of state-of-the-art GPU sorting algorithms

Abstract

In contemporary computing, parallel sorting methods are gaining considerable attention. With the widespread popularity of parallel processors such as Graphics Processing Units (GPUs) and accessible programming languages like CUDA and OpenCL, we have seen a steady increase in research exploring efficient sorting techniques. This paper serves as a survey of popular sorting algorithms optimized for GPU utilization and recent improvements in the field. Multiple algorithms are examined: Radix sort, Bitonic sort, Quick Sort, TimSort and H-P Sort. Explanations on their methodologies as well as performance evaluations based on reported improvements from recently published papers are provided. Upon inspecting the findings, Radix Sort and Bitonic Sort continue to stand out as particularly promising options for GPU sorting.

Keywords: GPU Sorting, Parallel Sorting, Survey

1 Introduction

Sorting is an elemental operation and one of the most important procedures in computer science. For example, in the field of computer graphics certain rendering techniques require efficient sorting algorithms to work. Sorting is also of great importance in a wide variety of tasks such as scheduling tasks or simply organizing data for analysis or visualization. A recent example are the contributions made by Kerbl et al. in their paper about Gaussian Splatting [Kerbl et al. 2023]. Here, sorting is needed for front-to-back rendering and, in relation to the CUDA implementation, also for sorting the splats into tiles of 16x16 pixels. The authors used a GPU Radix Sort implementation for these workloads. Efficient sorting enables us to enhance the performance of algorithms that are dependent on organized data. Thus, it is not surprising that sorting algorithms have been an integral part of computer science research and education for decades. The need for more efficient sorting algorithms has led to the development of a wide range of techniques and approaches. From traditional sorting algorithms like Bitonic Sort and Radix Sort to more recent advancement concerning Quick Sort and TimSort the field continues to evolve as researchers try to optimize the performance of their sorting algorithms.

Recently, powerful Graphics processing units (GPUs) are being used to sort data simultaneously using multiple processing cores at the same time. This is called parallel sorting and shows multiple differences when compared to the classic method of serial sorting. Due to their different architectures from CPUs, GPUs are more suited for executing many tasks simultaneously. Thanks to programming languages like CUDA, it is even possible to run sorting algorithms on multiple GPUs at the same time. This is of interest because GPUs have generally become more powerful than CPUs due to their vast number of cores. Wanting to use this computing power to its full potential is only natural.

When looking at very large datasets, parallel sorting algorithms have a significant advantage over serial sorting techniques. In serial sorting the elements are usually compared and rearranged one at a time leading to long runtimes when dealing with millions of elements. On the other hand, parallel sorting often makes use of the concepts of divide-and-conquer and parallelism together to improve the efficiency of the sorting process by processing the divided work independently and simultaneously, thus benefiting from the large number of threads. A recent demonstration of this was published in 2022 [Raghunandan et al. 2022]. It compares sequential and parallel Radix Sort implementations and shows a performance advantage of 900% in favor of the GPU implementation. However, the task of utilizing multiple processors efficiently to run parallel algorithms can be challenging, since more processors also means a larger overhead of scheduling and synchronization. This reduces the gained speedup [Rajput et al. 2012]. Additionally, modern GPUs are evolving rapidly, often requiring further optimizations for future hardware generations.

Relatively recent surveys by Singh et al. [2018] and Arkhipov et al. [2017] give an excellent overview. However, in the last months and years there have been several improvements like *Onesweep* Radix Sort [Adinets and Merrill 2022], RMG Sort [Ilic et al. 2023] and enhanced CUDA Quicksort [Ćatić et al. 2023]. This state-of-the-art report aims to give an overview over popular sorting algorithms used on GPUs and recent improvements including the ones mentioned. The specific improvements will be presented as claimed by the respective authors.

2 Background

When looking at sorting algorithms we can differentiate between comparison-based and non-comparison-based techniques. Some of the most popular GPU sorting algorithms include Radix Sort, Bitonic Sort and Quick Sort. The following sections give an overview of these techniques and their performance as well as some of their advantages and disadvantages.

2.1 Comparison-based algorithms

Comparison-based techniques are the more common type and work by comparing elements and swapping them if needed. They have the advantage of being efficient in a predictable way and often easier to implement. However, for specific datatypes these types of sorting algorithms can be less than optimal as they are generally bound by their bound of $n \log n$. Quick Sort and TimSort are comparison based and have an average performance of $O(n \log n)$ where n is the number of elements in the input list. Bitonic Sort is also comparison based and performs in $O(\log^2 n)$.

2.2 Non-comparison-based algorithms

In such cases non-comparison-based algorithms can be more efficient as they do not require direct comparison of the numbers in the input list. Instead, they are dependent on specific characteristics of the data in order to sort them. These advantages usually come at the cost of increased memory usage, more complicated implementations and the fact that they are not applicable to all types of input data. Radix Sort being a non-comparison-based technique behaves differently when compared to comparison-based techniques.

It has a time complexity of $O(n\ \mathrm{d})$ where n is the number of input-elements and d is the number of passes (the *radix digit*), dependent on the bit-size of the numbers to be sorted as well as the number of bits that are sorted in each pass.

2.3 Performance considerations

The performance of sorting algorithms not only depends on the size of the input list. Other factors include the quality of the implementation, especially with parallel implementations. Looking at Radix Sort in particular, the performance depends on the size of a radix digit, and the amount of digits. The number of passes needed to sort the input depends on the bit-size of the numbers that will be sorted in the following way: when sorting, for example, 32-bit integers by sorting 8 bits per pass, 4 passes would be needed, since $4 * 8 = 32$. The larger the radix digit the fewer last-level memory accesses [Adinets and Merrill 2022].

In order to be efficient, GPU algorithms have to be designed to exploit the hierarchical structure of the GPU architecture. Computations are performed in synchronized steps and progress as sequences of short programs represented by *kernel invocations* that are executed by several threads in parallel. This allows consistent dataflow and synchronization across the GPU. Threads are grouped into blocks and within these blocks multiple threads can communicate with each other using fast shared memory and synchronization mechanisms [Adinets and Merrill 2022].

Due to the inherent nature of the architecture of modern GPUs it is generally desirable to have a lot of running threads while reducing the number of registers needed per thread. Memory-bank conflicts should also be avoided as well as thread divergence [Gopi 2017]. These are important factors when optimizing the performance of GPU sorting algorithms. However, when trying to measure differences in performance of implementations of similar sorting algorithms asymptotic time complexity is not the only option. Another option is to run the algorithms that need to be compared on the exact same hardware and to measure how many keys are being sorted with each technique. This results in comparable numbers (e.g. 15 GKey/s is better than 12 GKey/s). Usually, additional information regarding the length of the keys and their distribution as well as the hardware used for testing are provided. Another simple way is to measure the time the algorithm takes to complete.

The following sections will **(a)** provide a more detailed look at the aforementioned algorithms and **(b)** examine recent improvements concerning these techniques. Starting with Radix Sort the contributions made by recent publications *Regions Sort* [Obeya et al. 2019] and *Onesweep* [Adinets and Merrill 2022].

3 Radix Sort

Radix Sort is a non-comparison-based sorting algorithm. As such it can be broken down into the following steps:

1. Finding the largest number in the list to determine the number of passes needed.
2. Sorting the numbers based on the current digit, starting with the most significant digit (MSD) or least significant digit (LSD).
3. Sorting the numbers based on the next digit until all digits have been considered and the list is fully sorted.

For example, when looking at the following array of numbers $[101, 26, 64]$, a simple LSD Radix Sort would:

1. Determine that three passes would be needed since 101 has three base-10 digits (exactly one digit will be sorted per pass for this simple example).
2. Sort the numbers based on their least significant digits $[1, 6, 4]$ resulting in $[101, 64, 26]$.
3. Sort by the next digits resulting in $[101, 26, 64]$ first and finally in $[26, 64, 101]$.

Sorting each individual digit follows the concept of Counting Sort, which counts the amount of numbers that are smaller than each number by counting how often each number occurs in the list and then computing the prefix sums, resulting in an offset list. This subroutine can be broken down into three steps:

1. Counting the numbers.
2. Computing the offset list.
3. Reordering the list.

In real-world implementations the numbers to be sorted are typically 32 or 64-bit integers. Instead of sorting all 32 bits at once like Counting Sort would, Radix Sort sorts e.g. 8 bits per pass, thus needing 4 smaller Counting Sort passes. Because of this, when compared to simple single-pass Counting Sort, it is not needed to keep an array as large as each possible value (e.g. 2^{32} when sorting 32-bit integers), which leads to decreased memory cost (e.g., arrays with a size of just 2^8 per pass).

A major disadvantage of Radix Sort is that the elements of the input list must have a fixed size, typically 32 or 64-bit integers. However, when applied to such data, parallel Radix Sort can be multiple times faster than its sequential counterpart, as demonstrated recently by the parallel implementation of FastBit Radix Sort using MPI and CUDA by Raghundan et al. [2022]. Another disadvantage is the amount of memory needed during sorting. Most parallel Radix Sort implementations require auxiliary memory proportional to the size of the input data. This was recently addressed by a new variant of Radix Sort called Regions Sort which utilizes graphs to reduce these memory requirements [Obeya et al. 2019].

Radix Sort has a worst-case runtime of $\mathrm{O}(n\ \mathrm{d})$ where d is the radix digit. For example, when sorting numbers with 32 bits, d will be equal to 32 divided by the number of passes. This results in an increase in memory requirements as well as in an exponential increase in dynamic instruction counts. However, as mentioned before, increasing d also has the advantage of decreasing the amount of binning iterations. The possibility of this trade-off is one of the advantages of Radix based sorting algorithms [Adinets and Merrill 2022].

When implemented to run on a graphics processing unit, this process is parallelized by splitting the input list into sub-lists - one for each running thread. Each thread applies the first step from above to its respective sub-list. Then the offset lists are computed while considering the counters of the sub-lists in the other threads as shown in Figure 1 [Gopi 2017].

The offset counter o of key k in thread t is equal to

$$o_t^k = \sum_{j<0}^{j=k-1} \sum_{i=0}^{i=m-1} f_i^j + \sum_{i=0}^{i=t-1} f_i^k \tag{1}$$

[Gopi 2017]

When sorting numbers with more than one radix digits there are two options:

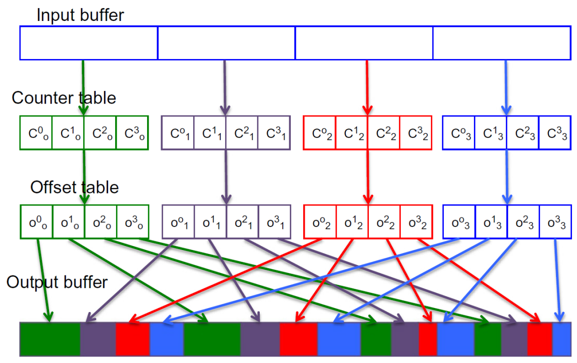


Figure 1: Parallel Radix Sort. The input is divided into sub-lists before computing the final output. Image taken from [Gopi 2017].

1. Starting with the least significant digit (LSD) and moving to the left. This approach typically guarantees stable sorting because it is an essential requirement of the process. Numbers with identical higher-significant-digit must remain in order, or else we would lose the results of previous passes that sorted lower-significant-digits.

2. Starting with the most significant digit (MSD) and moving to the right. Here, the numbers are sorted coarsely during the first pass and consecutive passes refine the sort.

In each pass the list is partially sorted based on the current digit by examining it and placing it in a bin. This is called *binning*. When all elements have been binned the bins are concatenated to create the partially sorted list.

3.1 Onesweep Sort

Based on this, Adinets and Merrill presented a significant performance improvement in their paper *Onesweep: A Faster Least Significant Digit Radix Sort for GPUs* [Adinets and Merrill 2022]. By combining the prefix sum computation with the digit binning instead of performing separate passed for each digit they managed to reduce the number of memory accesses from $\sim 3pn$ to $\sim(2p+1)n$, where p is the number of digit-binning iterations. In order to understand how this is possible we need to look at the three phases of the Onesweep algorithm:

1. Histogram computation.

2. Prefix sum calculation.

3. Digit binning.

The technique used for binning is only possible due to the information gained in the earlier steps. The second step uses the histogram to compute the prefix sum using the aforementioned single-pass method. This in turn enables *Onesweep* to determine the starting index of each digit group during its third phase. Here the parallel architecture of the GPU is utilized to sort the list using parallel threads starting at the LSD of each element.

The LSD Onesweep Radix Sort algorithm is 1.4x-1.6x times faster than the previous Radix Sort implementation that had been provided in CUB version 1.10, which was the state-of-the-art at the time as can be seen in Figure 2. Furthermore, Onesweep's MSD variant is more consistent performance wise than HRS (the former state-of-the-art MSD Radix Sort implementation). [Adinets and Merrill 2022]

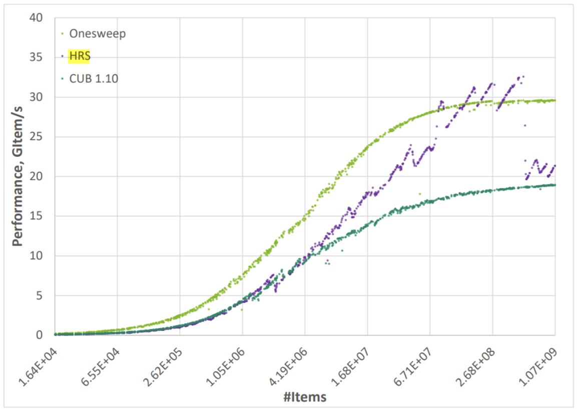


Figure 2: Performance of Onesweep when sorting uniformly distributed elements compared to HRS and the Radix Sort implementation from CUB version 1.10. Image taken from [Adinets and Merrill 2022].

3.2 Regions Sort

As mentioned before, requirements concerning auxiliary memory are also important to consider when evaluating the efficiency of sorting algorithms. MSD Regions Sort tries to optimize memory consumption for parallel Radix Sort by utilizing graph data-structures. The algorithm can be broken down into four phases:

1. Local sorting - splits input list into sub-lists and sorts them individually.

2. Graph construction - model dependencies of elements that will need to be swapped using a *regions graph*.

3. Global sorting - identifies independently executable tasks using the region graph and runs them in parallel in order to move elements to their correct locations.

4. Recursion - sorts the sub-lists that share the same bins in parallel recursively until all elements have been sorted.

Regions Sort achieves parallelism through its third phase. This depends on the graph that is constructed in the second phase. The graph models dependencies by representing elements as nodes and connecting them with edges. When two nodes are connected in this way it means that these two elements will need to be swapped in order to achieve a sorted list. Due to the properties of the region graph, Region Sort only requires $O(K \log r \log n)$ auxiliary memory while different approaches often require additional memory. K stands for the number of sub-lists in which the input list is divided, and r depicts the range of the input elements (e.g. if the input contains integers between 0 and 100 then r is equal to 100).

Two different variants of Regions Sort are proposed by the authors, **(a)** one that processes *cycles* in the regions graph and **(b)** one that is based on processing *2-paths*. Both have their own specific use cases based on the input data. In practice, the 2-paths variant often performs better as shown in Figure 3 [Obeya et al. 2019].

A major limitation of the Regions Sort algorithm when compared to other Radix Sort implementations is that it is not stable. Thus, it may change the relative order of equal elements. However, when stability is not a requirement it is worth a consideration. The total number of operations performed when using Regions Sort to sort an input list of length n is $O(n \log R)$.

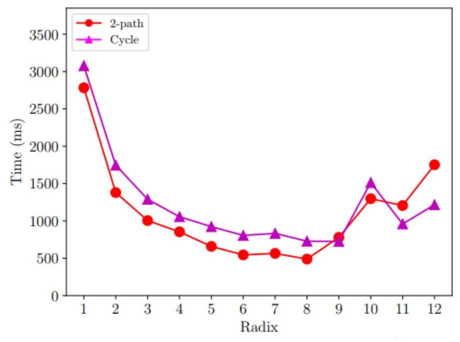


Figure 3: Performance of Regions Sort variants (*2-path* and *cycle*) when sorting uniformly distributed elements. The x-axis depicts the radix size of the input list and the y-axis the runtime of the sorting algorithms. Image taken from [Obeya et al. 2019].

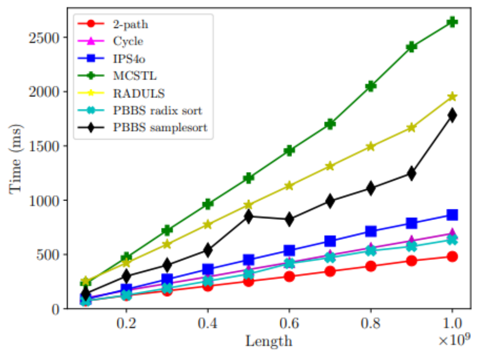


Figure 4: Performance of Regions Sort (*2-path* and *cycle* variants) when sorting uniformly distributed elements compared to other parallel Radix Sort implementations as well as PBBS Sample Sort. The x-axis depicts the length of the input list and the y-axis the runtime of the sorting algorithms. Image taken from [Obeya et al. 2019].

3.3 RMG Sort

Another advantage of parallel GPU algorithms today is that it is possible for multiple physical GPUs to directly interconnect using so-called peer-to-peer (P2P) connections. Algorithms that use systems like this (e.g. NVLink and NVLSwitch) are called *multi-GPU sorting algorithms*. RMG Sort is an example of this concept. It uses Radix key partitioning on each GPU in a way so that each bucket is small enough for a P2P key swap. The algorithm also utilizes P2P Key swaps so that each GPU contains keys of a distinct value range so that no further key swaps are needed. This is done to reduce inter-GPU communication and improving load balancing across GPUs. [Ilic et al. 2023]

RMG Sort employs Most Significant Bit (MSB) radix partitioning. This allows for a single all-to-all P2P key swap, which means this is independent of the number of GPUs used. This is why RMG Sort scales linearly with the input size and can significantly reduce data transfer between GPUs. The authors demonstrate the algorithm's excellent efficiency in handling large data volumes. When compared to parallel Merge Sort they report an increase in performance by up to 60% and 20% compared to Peer-to-Peer Merge Sort. Though is it important to mention that the relative performance between named algorithms strongly depends on the number of GPUs used and the amount of input data. This is because there exists a bottleneck on the specific hardware used, namely the CPU and its data transfer rate to the GPU. Due to this bottleneck the P2P key swap takes three times longer when using four GPUs instead of two as shown in Figure 5 [Ilic et al. 2023].

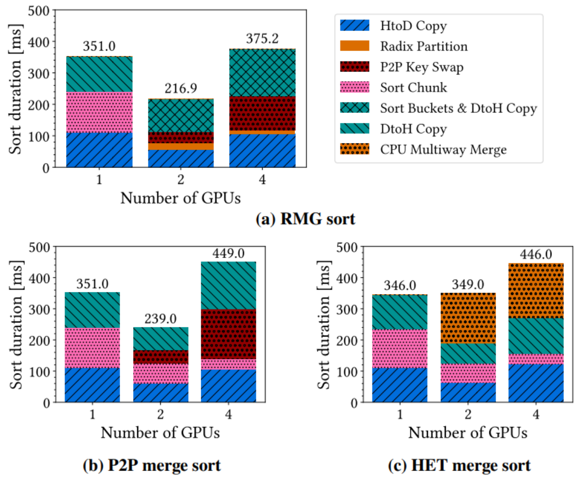


Figure 5: Performance of RMG Sort compared to parallel Merge Sort and P2P Merge Sort when sorting 2 billion integers on the IBM AC922. Image taken from [Ilic et al. 2023].

4 Bitonic Sort

Bitonic Sort is an in-place comparison-based sorting algorithm suited for parallelism. In-place refers to the amount of memory needed during the sorting process. An in-place algorithm does not need extra memory, but it works using the memory required to hold the input (additional constant amounts of memory for variables etc. is not relevant for this specification).

The main concept on which Bitonic Sort is based on are so-called *bitonic sequences*. These are sequences of elements that are made-up of a part that is monotonically sorted in ascending order and one part that is monotonically sorted in descending order. Such a list has the following structure: $x_0 \leq ... \leq x_k \geq ... \geq x_{n-1}$ [Gopi 2017]. This also means that traditionally Bitonic Sort was only able to work with lists with a length that is equal to a power of 2.

The general composition of Bitonic Sort encompasses the following steps:

1. Recursively split the list into two smaller sequences which are sorted in a different order.
2. Merge the bitonic sequences together to receive the final sorted output list.

The process of breaking the list down into bitonic sequences is referred to as *Bitonic Split* while the part that combines the created sequences into larger bitonic sequences is called *Bitonic Merge*. This combination of steps leads to a runtime of O(n log^2 n), which is not ideal considering other more traditional comparison-based algorithms like Quick Sort and TimSort are faster. However, it is possible to divide the workload between multiple threads to vastly decrease the actual runtime of Bitonic Sort. This is the main advantage of Bitonic Sort. Parallel Bitonic Sort is scalable, so generally, the more threads are used the sooner the algorithm will be able to finish its workload.

In 2010 Peters et al. [2010] managed to develop an in-place implementation for CUDA that made it possible to be used for any input size. By using a technique called *Virtual Padding* it is possible for Bitonic Sort to work on lists with sizes that do not need to have a length that is a power of 2. To achieve this the sequence is padded with max-values and the algorithm is modified in such a way that these values are never moved during the sorting process. This enables these elements to not have to exist in memory which results in no increased memory usage. The same optimization also reduced the number of global memory accesses and thus improved the overall performance of the algorithm. This was done by making sure multiple operations on the same data are completed before additional communication through global memory is required as well as reducing the number of launched kernels. By partitioning the planned operations into subsets such that each subset can be processed by a single thread in one kernel launch it is possible to minimize the amount of global memory accesses needed to read or write an element from the list to ≤ 1. This way operations in one subset are unaffected by operations in the other subset [Peters et al. 2010] and [Peters et al. 2011].

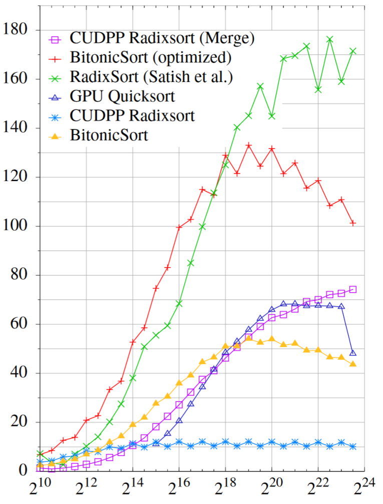


Figure 6: Comparison of the runtime of the optimized Bitonic Sort by Peters et al. versus a variety of other GPU sorting algorithms. The x-axis shows the size of the input list. The y-axis depicts the sorting rate. Image taken from [Peters et al. 2011].

4.1 Adaptive Bitonic Sort

This variation is also a parallel sorting algorithm that uses bitonic sequences. Recursion is used in the same way in order to execute certain parts of the algorithm simultaneously. Nevertheless, adaptive Bitonic Sort is different from traditional Bitonic Sort in the way that it is dependent on the respective input data. It can be faster than traditional Bitonic Sort by a factor of log n. This means its runtime is $O(\frac{nlogn}{p})$ where p is the number of processors used to run the algorithm [Zachmann 2011] and $p \leq \frac{n}{logn}$. However, due to the data-dependence the system that is used to run the algorithm needs certain specifications that are not required when running traditional Bitonic Sort. Specifically, the architecture must provide a type of flow control. With the given specifications it is possible to run Adaptive Bitonic Sort on a GPU and make use of its scalability to reduce the number of passes needed to O($log^2 n$) with the maximum number of processors $p = \frac{n}{logn}$. For this to work on GPUs, which typically do not support random-access writes, modifications were made to the kernel. Nodes of the Bitonic Tree (the data-structure used to represent the data internally) are not written back to their original locations after being processed. Instead, a stream is used to update the pointers of parent nodes. Due to these optimizations and techniques implemented Adaptive Bitonic Sort is able to out-speed traditional Bitonic Sort by 30% [Zachmann 2011].

4.2 IBR Bitonic Sort

Based on the work done on Adaptive Bitonic Sort a new sorting algorithm based on Adaptive Bitonic Sort was published in [Peters et al. 2012]. This so-called Interval Based Rearrangement (IBR) Bitonic Sort does not use a special data-structure but instead just uses the given array representation of the input data with some additional information. Sequences are represented as intervals. This means IBR Bitonic Sort has the advantage of one being able to easily switch to a different algorithm. By combining IBR Bitonic Sort with Bitonic Sort from [Peters et al. 2011] the authors managed to achieve another significant improvement to the speed of the algorithm. This hybrid approach achieves a complexity of O(n log n). With the already mentioned advantage of not being reliant upon a complex data structure it outperforms traditional Bitonic Sort [2011], especially with large datasets as shown in Figure 7.

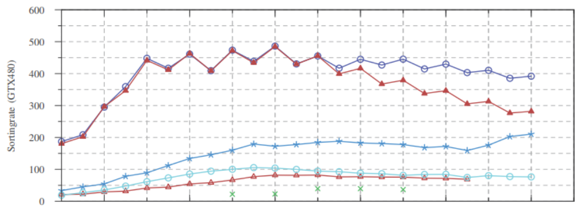
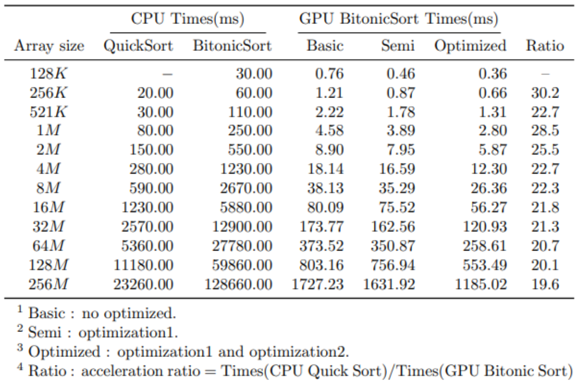


Figure 7: Comparison of the runtime of the hybrid IBR Bitonic Sort by Peters et al. (dark blue) versus traditional Bitonic Sort (dark red), as well as Warp Sort (green), GPU Sample Sort (blue) and GPU Quick Sort (light blue). The x-axis shows the size of the input list. The y-axis depicts the sorting rate. Image taken from [Peters et al. 2012].

4.3 Further Optimizations

Further optimizations for CUDA-based Bitonic Sort were made by Mu et al. [2015]. By using shared memory for sub-sequences that are small enough to fit into a single block's shared memory the synchronization between blocks was improved. This way, less time is used accessing global memory. Mu et al. also propose that threads use their high-performance registers, so that two steps can be combined into one. The faster access times when compared to global memory lead to further decreased runtime. The authors managed to reduce the number of kernel launches as well as the number of global memory accesses and thereby improving the performance of their algorithm by 40% when compared to standard Bitonic Sort as shown in Figure 8 [Mu et al. 2015].

In 2021 Rudrawar et al. [2021] demonstrated the current and future potential of running Bitonic Sort on graphics processing units. The authors compare serial and parallel implementations of Bitonic Sort using CUDA. Specifically, comparisons using two different *block sizes* were performed (128 and 256). Blocks are chunks of elements that are processed together in parallel, so the block size determines how many elements are processed simultaneously by the threads of the GPU. Choosing the correct block size to optimize the performance of the sorting algorithm involves trade-offs. When working with larger block sizes more elements can be processed at each moment in time. However, if the chosen block size is too large it can lead to inefficiencies in memory accesses. For their specific implementation the authors found that a block size of 128 worked best. With this their implementation of Bitonic Sort only took 60ms



	CPU Times(ms)		GPU BitonicSort Times(ms)			
Array size	QuickSort	BitonicSort	Basic	Semi	Optimized	Ratio
$128K$	–	30.00	0.76	0.46	0.36	–
$256K$	20.00	60.00	1.21	0.87	0.66	30.2
$521K$	30.00	110.00	2.22	1.78	1.31	22.7
$1M$	80.00	250.00	4.58	3.89	2.80	28.5
$2M$	150.00	550.00	8.90	7.95	5.87	25.5
$4M$	280.00	1230.00	18.14	16.59	12.30	22.7
$8M$	590.00	2670.00	38.13	35.29	26.36	22.3
$16M$	1230.00	5880.00	80.09	75.52	56.27	21.8
$32M$	2570.00	12900.00	173.77	162.56	120.93	21.3
$64M$	5360.00	27780.00	373.52	350.87	258.61	20.7
$128M$	11180.00	59860.00	803.16	756.94	553.49	20.1
$256M$	23260.00	128660.00	1727.23	1631.92	1185.02	19.6

[1] Basic : no optimized.
[2] Semi : optimization1.
[3] Optimized : optimization1 and optimization2.
[4] Ratio : acceleration ratio = Times(CPU Quick Sort)/Times(GPU Bitonic Sort)

Figure 8: Comparison of the runtime of standard GPU Bitonic Sort versus implementations using one and both described optimizations made by Mu et al., as well as CPU Quick Sort and Bitonic Sort times for reference. The first column shows the size of the input list. Image taken from [Mu et al. 2015].

for sorting 4 million elements when using the parallel GPU variant, compared to ≈ 1250ms when using the serial CPU variant as shown in Figure 9 [Rudrawar et al. 2021]. The hardware used for the experiment was an Intel Haswell E5-2620V3 Six core processor (2.4 GHz/15MB Cache) and NVIDIA'S Tesla K20 GPU. Based on this result the authors conclude that using GPUs for sorting is viable and efficient especially for large datasets because the gains in speed increase proportionally to the size of the data.

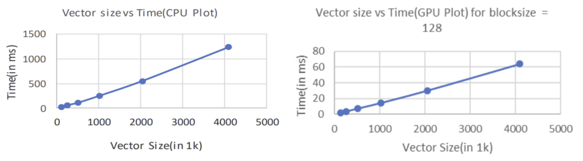


Figure 9: Comparison of the runtime of Bitonic Sort in milliseconds when using a CPU approach versus a parallel GPU-based approach. The x-axis shows the size of the input list divided by 1,000. Image taken from [Rudrawar et al. 2021].

5 Quick Sort

Quick Sort is a popular comparison-based in-place sorting algorithm that has seen use in sequential applications as well as parallel ones. The algorithm builds upon the divide-and-conquer methodology. As mentioned in earlier the runtime complexity of Quick Sort is O(n log n) on average, but in the worst case it can decay into O(n^2). Implementations of this algorithm follow these steps:

1. Choose a pivot element p from the given elements in the input-list.
2. Move every element that is smaller than p to its left side and every element larger than p to its right side. This moves p into its correct position.
3. Recursively repeat this procedure on the left and right parts of the list, selecting new pivot elements for each sub-list as seen in Figure 10.

At the end of the run each element resembles a sequence with a length of one, so no further recursion is needed. When read from left to right together they make up the final sorted list. The worst case happens when in each step the selected pivot element splits the current sequence into two sub-sequences that are as unequal in length as possible. This happens, for example, when the selected pivot element is always the largest element in the respective sub-list. Consequently, the method of selecting the pivot element is of major importance with respect to the performance of the Quick Sort implementation.

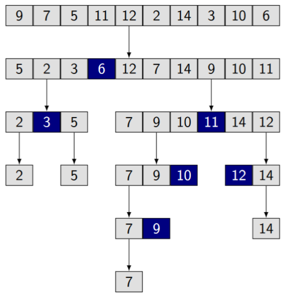


Figure 10: An example of a Quick Sort run. The pivot element is marked blue in each recursive call. Image taken from Algorithms and Complexity Group at Vienna University of Technology.

When looking at parallel implementations of Quick Sort the importance of well-balanced partitioning of the data is emphasized. Such an implementation would generally look like this: [Rajput et al. 2012]

1. Partition the data into k sub-lists, where each sub-list has a length of $\lceil n/k \rceil$.
2. Select a pivot element p on one processor.
3. Share p with the other processors.
4. In each processor split the data according to p.
5. Split the processors into two groups. In each group two processors at a time swap elements so that one group of processors ends up with all of the numbers smaller than p and vice versa.
6. Repeat steps 4 and 5 recursively.
7. Run Quick Sort locally on each processor.

In the following section modern implementations of parallel Quick Sort running on GPUs using the CUDA architecture and recent improvements on this front will be examined.

5.1 CUDA Quick Sort

In their paper [2016] the authors Manca et al. propose an implementation of Quick Sort that is based on GPU Quick Sort. It is optimized to work on the CUDA architecture while trying to optimize memory accesses and inter-block communication. Due to the newly added support for atomic primitives Manca et al. were able to utilize this to their advantage. They designed their block-oriented version of Quick Sort to use atomic primitives to perform the synchronization between blocks in the GPU. Their specific variant of Quick Sort also used a version of Bitonic Sort to finalize the sorting process. Though, according to the authors, the main factor of improvement was due to the optimized accesses to global memory. With these improvements their algorithm managed a four-fold improvement in speed compared to the previous state-of-the-art GPU Quick Sort. For large, structured datasets CUDA Quick Sort even out-performed Thrust Radix Sort [Manca et al. 2016].

With the goal of fine-tuning existing CUDA Quick Sort [Ćatić et al. 2023] proposes multiple changes to the algorithm. The authors identified the following areas that could be improved. They are elements of the part of the algorithm that takes up most of the time:

1. Selecting an optimal pivot element p.
2. Flow control and resulting thread divergence (branch efficiency).

Splitting the input-list into sub-lists as equally as possible would result in an optimal workload distribution. Theoretically, choosing the median of the entire input-array would result in such a distribution. In practice, however, finding the median of large lists is too time-consuming of a task. Thus, approximations are commonly used instead. Often the middle element of the list is chosen as the pivot element. If the list was already (partially) sorted, this can avoid reaching the worst-case runtime O(n^2). However, in some cases it could also lead to choosing the largest or smallest element and thus also result in a polynomial runtime. Instead, Ćatić et al. chose an approximation using the *Median-Of-Three* [1984] method. Here the pivot element is chosen as the median of the first, the middle and the last elements. The goal of this approach is to avoid the worst-case performance of Quick Sort.

Since when working with the CUDA architecture the smallest unit of execution is a warp, all threads belonging to the same warp must execute the same piece of code. If some threads diverge due to a branch in the code the, the execution will happen sequentially instead of simultaneously. This results in unnecessary usage of resources. Ćatić et al. propose two strategies to mitigate this problem:

1. Using *loop unrolling* to remove loops in the code.
2. Replacing if/else statements with singular expressions using truth tables.

Even though the branch efficiency suffers by removing if/else statements, the overall performance can still be improved. This is because the probability of the code entering into a divergent branch is significantly lower. Thus, less excessive workloads are generated [Ćatić et al. 2023]. After implementing all these optimization techniques and testing the improved CUDA Quick Sort on an NVIDIA GTX 1160 Super graphics card the authors were able to report a significant reduction in the number of iterations performed during the sorting process as shown in Figure 11.

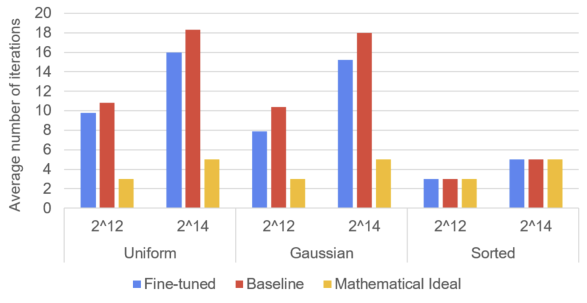


Figure 11: The average number of iterations performed by fine-tuned CUDA Quick Sort (blue) compared to the state-of-the-art CUDA Quick Sort implementation (orange) with different data-sizes. Image taken from [Ćatić et al. 2023].

5.2 Sorted Matrix

In 2023 Gupta et al. published their paper about an implementation of a new parallel sorting algorithm for GPUs that makes use of Quick Sort in a preparation step. Their algorithm uses a matrix to represent the input-list of elements that need to be sorted before outputting the sorted list. The matrix constructed has m rows and $\lceil n/m \rceil$ columns where m is a factor of n. The two core operations of the algorithm use Quick Sort and are described as follows:

1. Sorting the matrix row-wise in ascending order.
2. After the first step is done the matrix is sorted column-wise in ascending order.

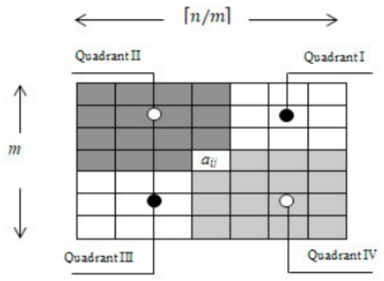


Figure 12: The element positioned in the middle of a matrix with dimensions $\lceil n/m \rceil$ times m splits the matrix into four quadrants. Image taken from [Gupta et al. 2023].

The middle element splits the matrix intro four quadrants as shown in Figure 12. The properties of the partially sorted matrix are then utilized in further steps: after sorting the elements first row-wise and then column-wise the algorithm searches for elements that are smaller than the element in the middle of matrix. This search is performed in Quadrant 1 and Quadrant 3 by comparing each element from the Quadrant with the middle element. However, because of the two sorting steps performed earlier the algorithm knows whether to move along the current row or the current column. Using this methodology the number of elements that are smaller than the middle element is computed which makes it possible to simply move it to its correct position in the matrix because the matrix' indices are known. This technique leads to a runtime complexity of O($\lceil n/m \rceil log \lceil n/m \rceil$). This also means that this algorithm's best case happens when $m = \sqrt{n}$ since then the runtime is O($\sqrt{n} log \sqrt{n}$) [Gupta et al. 2023].

The authors then implemented their algorithm using three parallel CUDA kernels, two of which are used to perform non-parallel Quick Sort in a single thread each to sort the matrix. The third kernel performs the searches in Quadrant 1 and Quadrant 3 and stores the elements in their correct positions in the output-list. When comparing their implementation to the Radix Sort available in the CUDA Thrust library they were able to report a performance improvement of up to 500% [Gupta et al. 2023]. Since Quick Sort takes up around 50% of the runtime the authors propose to improve this part of their algorithm in the future by parallelizing it and using shared memory to optimize memory accesses.

6 Other approaches

In this section two approaches for GPU sorting that differ from the already discussed algorithms are briefly presented. First a parallel implementation of TimSort using CUDA is discussed [2023], followed by GPU H-P Sort [2024].

6.1 CUDA TimSort

TimSort is a stable in-place hybrid sorting algorithm that is commonly used as the standard for solving real-world sorting problems.

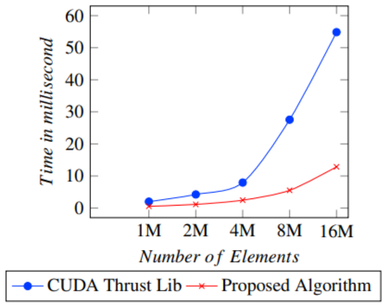


Figure 13: Sorted Matrix approach outperforms CUDA Radix Sort. The graph shows the result when working with uniformly distributed numbers. Image taken from [Gupta et al. 2023].

It uses the strength of both Merge Sort and Insertion Sort. The key concept is utilizing partially sorted sub-sequences (*runs*) that already exist in the data. TimSort works using the following steps:

1. Identify existent runs that are sorted either in ascending or descending order.
2. Either merge ascending runs (invert descending runs) or use Insertion Sort to sort unsorted sub-sequences.

During the merging process a technique called *galloping* is used to utilize possible structure inside the runs. Each time two elements from two sub-sequences are compared a counter keeps track of the number of times each sequence provided the smaller element. When the same sequence provides the smaller elements multiple times the galloping start. This means that the next element to be compared will be chosen by skipping multiple elements in-between. This results in an over-shoot. The algorithm then searches for the fitting element inside the last sub-sequence before the over-shoot happened. This way, multiple elements can be copied into the final array at once.

TimSort has the advantage that it is very fast when used on data that is already partially sorted and can reach a best-case performance of O(n) in such cases. Its average runtime lies in O(n log n). TimSort uses more memory than e.g. optimized Quick Sort. Also, if the comparison of elements can be done very efficiently, Quick Sort is usually the faster algorithm. This is why TimSort is often mainly used for lists of objects and not for primitives like integers.

In 2023 Disha and Mondal [2023] attempted to demonstrate the potential of parallel implementations of TimSort on modern GPUs using CUDA. Their implementation divides the input-data into multiple segments that are distributed to different threads. Each thread performs sorting and merging operations on its designated part of the data. The general methodology does not differ from the basic concept of TimSort explained above.

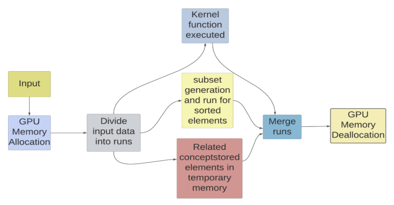

Kernel function executed
Input
GPU Memory Allocation
Divide input data into runs
subset generation and run for sorted elements
Related conceptstored elements in temporary memory
Merge runs
GPU Memory Deallocation

Figure 14: Diagram of the execution steps of the parallel TimSort implementation by Disha and Mondal. Image taken from [Disha and Mondal 2023].

With this, the authors report a significant reduction in runtime when compared to traditional TimSort run on a CPU. Specifically, the GPU-based algorithm a speedup equal to a factor of 25 with a dataset made-up of 10,000 elements [Disha and Mondal 2023]. The authors do not include any test results with larger datasets than this making it difficult to assess the full potential in of their implementation, in terms of scalability.

6.2 Improved H-P Sort

H-P sort is a parallel sorting algorithm that makes use of histogram and prefix computations to sort integers that are within a known range. It works by:

1. Computing the histogram h1 of the input array.
2. Calculating the prefix sum of h1.
3. Computing the histogram h2 of the prefix sum.
4. Calculating the prefix sum of h2.

The first two steps are referred to as Phase 1, and the last two steps as Phase 2. In their paper [Takase et al. 2024] the authors demonstrate performance improvements to this algorithm. Their implementation does not require one to specify the exact range of the integers in the input-list. Instead, only an upper and lower bound must be provided. The bounds can be wider than the range of the actual dataset. The following improvements to Phase 1 were made: **(a)** partitioning the input array into sub-arrays and computing histograms for each reduces duplication of values, **(b)** cyclic partitioning reduces the amount of exclusive control conflicts. Phase 2 was improved by utilizing the advantages gained through the improvements made to Phase 1. Now, it is no longer necessary to compute histograms during Phase 2. By performing the needed operations of Phase 2 directly on the output array the memory requirements were reduced to be only the requirements for Phase 1.

When comparing the performance of this implementation of the H-P Sort algorithm to CUB Sort (CUDA library) and the previous version of H-P Sort on an NVIDIA RTX 3090 GPU shows that the performance of H-P sort not only depends on the size of the data n, but also on the range of the provided integers. When the ratio $\delta = \frac{n}{range}$ is larger Phase 1 takes less time. Phase 2 is not affected by δ [Takase et al. 2024]. A speedup of up to 3.46 times compared to the previous H-P Sort was reported. In certain cases, the algorithm also out-performed CUB Sort as shown in Figure 15. According to the authors, future improvements for H-P Sort may focus on making it stable and improving cases where $\delta = 1$, as it is slower than CUB Sort in such cases.

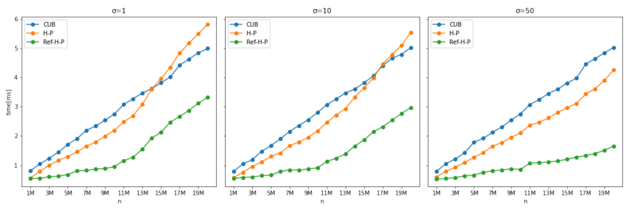


Figure 15: Runtime of base H-P Sort (orange), CUB Sort (blue) and improved H-P Sort (green) on the hardware name above. $\delta = 10$, and σ was set to 1, 10 and 50. $\sigma = \frac{range}{len}$ where *len* is the number distinct values in the data. Image taken from [Takase et al. 2024].

7 Conclusion

This survey presented three popular GPU sorting algorithms and their recent improvements as well as two alternative approaches. The mechanism and inner workings as well as the performance of each algorithm was reported as claimed by the respective authors and discussed. An analysis of the reported algorithms shows that Radix Sort is still being improved upon, even though it is already a hard to compete with GPU sorting algorithm. With its ability to exploit parallelism efficiently and its adaptability to different hardware configurations, it remains a strong contender. Its ongoing improvements suggest that it will continue to be a dominant force in GPU sorting. Bitonic Sort is very viable and greatly benefits from improvements made to GPU hardware and adjusting its parameters accordingly. However, its performance characteristics may limit its applicability in certain scenarios, particularly when dealing with extremely large datasets. Quick Sort, TimSort, and H-P Sort, while not as commonly used in GPU sorting, offer unique advantages in terms of stability, adaptability, and worst-case performance guarantees. Further research and optimization efforts may reveal ways to leverage these algorithms more effectively on GPU architectures, potentially broadening the range of applications where GPU sorting can be applied. Quick Sort continues to be of importance as a stand-alone algorithm as well as for hybrid-techniques and has seen further optimizations to workload distribution and resource management on the GPU architecture. H-P Sort has shown to be competitive in certain cases but suffers more restrictions than other algorithms.

Looking ahead, there are several areas in which GPU sorting algorithms could be improved. Firstly, optimizations aimed at reducing memory bandwidth usage and improving data locality could lead to significant performance gains, especially for algorithms that heavily rely on memory access. Secondly, research into novel parallel algorithms specifically designed for the GPU architecture could further push the boundaries of sorting performance. This includes exploring new sorting paradigms that exploit the unique features of GPU hardware. In conclusion, GPU sorting algorithms have demonstrated impressive performance improvements over the years, thanks to advancements in both hardware and algorithmic optimizations. While challenges such as memory bandwidth limitations and algorithmic complexity remain, ongoing research and development efforts hold promise for further enhancing the efficiency and scalability of GPU sorting in the future.

References

ADINETS, A., AND MERRILL, D., 2022. Onesweep: A faster least significant digit radix sort for gpus.

ARKHIPOV, D. I., WU, D., LI, K., AND REGAN, A. C., 2017. Sorting with gpus: A survey.

DISHA, D. Y., AND MONDAL, M. N. I. 2023. Accelerate implementation of timsort algorithm using cuda. *International Conference on Computer and Information Technology (ICCIT) 26*, 1–5.

DONG, X., DHULIPALA, L., GU, Y., AND SUN, Y. 2024. Parallel integer sort: Theory and practice. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 301–315.

ERKIÖ, H. 1984. The Worst Case Permutation for Median-of-Three Quicksort. *The Computer Journal 27*, 3 (01), 276–277.

GOPI, M. 2017. Chapter 12 - gpu sorting algorithms. In *Advances in GPU Research and Practice*, H. Sarbazi-Azad, Ed., Emerging Trends in Computer Science and Applied Computing. Morgan Kaufmann, Boston, 307–326.

GUPTA, S. K., SINGH, D. D. P., AND CHOUDHARY, D. J. 2023. New gpu sorting algorithm using sorted matrix. *Procedia Computer Science 218*, 1682–1691.

ILIC, I., TOLOVSKI, I., AND RABL, T. 2023. Rmg sort: Radix-partitioning-based multi-gpu sorting.

KERBL, B., KOPANAS, G., LEIMKÜHLER, T., AND DRETTAKIS, G. 2023. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics 42*, 4 (July).

MANCA, E., MANCONI, A., ORRO, A., ARMANO, G., AND MILANESI, L. 2016. Cuda-quicksort: an improved gpu-based implementation of quicksort. *Concurrency and computation: practice and experience 28*, 1, 21–43.

MU, Q., CUI, L., AND SONG, Y. 2015. The implementation and optimization of bitonic sort algorithm based on cuda.

OBEYA, O., KAHSSAY, E., FAN, E., AND SHUN, J. 2019. Theoretically-efficient and practical parallel in-place radix sorting. In *The 31st ACM symposium on parallelism in algorithms and architectures*, 213–224.

PETERS, H., SCHULZ-HILDEBRANDT, O., AND LUTTENBERGER, N. 2010. Fast in-place sorting with cuda based on bitonic sort. 403–410.

PETERS, H., SCHULZ-HILDEBRANDT, O., AND LUTTENBERGER, N. 2011. Fast in-place, comparison-based sorting with cuda: a study with bitonic sort. *Concurrency and Computation: Practice and Experience 23*, 7, 681–693.

PETERS, H., SCHULZ-HILDEBRANDT, O., AND LUTTENBERGER, N. 2012. A novel sorting algorithm for many-core architectures based on adaptive bitonic sort. 227–237.

RAGHUNANDAN, AISHWARYA, B., ASHWATH RAO, B., AITHAL, P. K., AND KINI, G. N. 2022. A parallel implementation of fastbit radix sort using mpi and cuda. In *Electronic Systems and Intelligent Computing: Proceedings of ESIC 2021*. Springer, 1–13.

RAJPUT, I. S., KUMAR, B., AND SINGH, T. 2012. Performance comparison of sequential quick sort and parallel quick sort algorithms. *International Journal of Computer Applications 57*, 9, 14–22.

RUDRAWAR, S., HEBARE, D., POPHALE, A., AND LOKHANDE, M. 2021. A gpu parallel implementation of bitonic sort using cuda.

SINGH, D. P., JOSHI, I., AND CHOUDHARY, J. 2018. Survey of gpu based sorting algorithms. *International Journal of Parallel Programming 46*, 1017–1034.

TAKASE, K., HAGIHARA, T., FUJIMOTO, N., AND WADA, K. 2024. Efficient gpu-implementation of hp sort based on improved histogram computation. 134–144.

ZACHMANN, G. 2011. Adaptive bitonic sorting. *Encyclopedia of Parallel Computing*.

ĆATIĆ, I., MUJIĆ, M., NOSOVIĆ, N., AND HRNJIĆ, T. 2023. Enhancing performance of cuda quicksort through pivot selection and branching avoidance methods. 1–5.