Institut für Formale Methoden der Informatik

Abteilung Theoretische Informatik

Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart

Bachelorarbeit

Experimental Study of the AKS Sorting Network

Alexander Fischer

Studiengang:	Informatik, Bachelor of Science
Prüfer:	Prof. Dr. Volker Diekert
Betreuer:	Armin Weiß
begonnen am:	28. Februar 2020
beendet am:	28. August 2020

Contents

Abstract

Sorting networks are usually bound at a depth of $O(log^2\ n)$, since a perfect halver is of at least depth $O(log\ n)$. However, the AKS Sorting Network, by Ajtai, Komlós and Szemerédi, can sort data with depth $O(log\ n)$ by using so-called ε-halvers, which are of constant depth. Such ε-halvers are allowed to have some errors and will eventually be corrected by sending elements to a level above. In this thesis, a CPU and CUDA version are implemented following a paper by Vašek Chvátal [5] and the original paper by Ajtai et al. [1]. Experiments are run on these versions to observe and improve parameters.

1 Introduction

When trying to sort data in parallel with sorting networks, the factor bounding the run-time is the depth of the networks. For most algorithms, the depth is at best of $O(log^2\ n)$, like seen in Bitonic sort or Odd-even mergesort, but in 1983 M. Ajtai, J. Komlós and E. Szemerédi [1] came up with a sorting network of depth $O(log\ n)$ called the AKS Sorting Network. The reduced depth is achieved by the usage of special halvers which are allowed to sort a specfied percentage of their elements wrong. Those wrong sorted elements are then ensured to be sorted into their right positions again in later steps. Although the network is of logarithmic depth, it has a big hidden constant, which makes it not usable for practical applications.

This thesis covers the implementation and the during this process arisen modifications and improvements of the AKS Sorting Network on CPU using C++ and on GPU using CUDA.

2 Sorting Networks

Sorting networks are a special category of sorting algorithms of fixed size where the elements are distributed on so-called wires. These wires go through several comparisons with other wires to achieve a sorted sequence of the elements at the end. Sorting networks can be visualized with lines, symbolizing wires, and connections between them, acting as comparisons, as seen in figure 2.1. In the example from the figure each comparison ensures that elements get switched if the upper of the two wires is holding the greater element. This results in the greatest elements to move towards the bottom wires, while the smallest elements move upwards, to create an ascendingly sorted order at the end.

To test if such network of size n sorts correctly in every case it would be necessary to check all $n!$ input variations. To reduce the amount of checks, one can utilize the zero-one principle [7, p. 223]. This theorem states, that "*if a network with n input lines sorts all* 2^n *sequences of* 0*s and* 1*s into nondecreasing order, it will sort any arbitrary sequence of n numbers into nondecreasing order*".

This arrangement of comparisons can now be divided into parts where every wire is involved in a maximum of one comparison. Since now there are no dependencies inside a single part, they can be run in parallel. Using this parallelization, the amount of time taken to run a sorting network is bound to the maximum comparisons any element has to go through. This bound is also known as depth, and since this thesis concentrates on parallel sorting with sorting networks it is desirable to keep it as small as possible.

There are also other variants of such networks [2]. Balancing networks for example

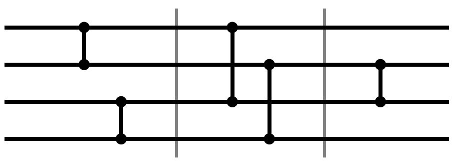

Figure 2.1: Partitioned sorting network for 4 elements

are a different class of sorting networks where instead of comparators balancers are used. The resulting networks can for instance balance the input onto the output wires, assuming that it is possible to have multiple elements on a single wire.

2.1 AKS Sorting Network

The goal of the AKS Sorting Network, created by M. Ajtai, J. Komlós and E. Szemerédi [1] in 1983, is to reduce the typical depth of $O(log^2\ n)$ for sorting networks to $O(log\ n)$.

An example of a sorting network of depth $O(log^2\ n)$ is a perfect halver of size n which routes its output elements to two perfect halvers of size $\frac{n}{2}$ each. These perfect halvers route their output elements to four further perfect halvers of size $\frac{n}{4}$ and this repeats until after perfect halvers of size two. Since the minimum depth of a perfect halver is of $O(log\ n)$ the total depth will result in $O(log^2\ n)$ [5].

The AKS Sorting Network "eliminates" the depth of the halvers by using so-called ε-halvers, which are allowed to have wrong sorted elements of factor ε. To account for wrong sorted elements in ε-halvers, a fraction of the elements get occasionally sent to an upper level where they get sorted into their right partition. These ε-halvers are of constant depth, which results in a total depth of $O(log\ n)$, however, this constant is of a few thousands.

2.1.1 Construction

First off, the size of elements to be sorted has to be of 2^d, where $d \geq 8$ and d is a multiple of four. The body of the AKS Sorting Network consists of so-called separators which are distributed over multiple steps on several levels. As an example, the whole sorting construction for 4096 elements can be seen in table 2.1. For this size it consists of 17 steps and the result will be at step $t = 16$ on level $i = 6$.

Over the course of the first $d - 4$ steps the upper bound α for the levels, meaning the smallest i, is alternating between 0 and 1. After that, beginning with step $d - 4$, the upper levels only send elements upwards every fourth step which results in a steadily decreasing upper bound. On the other side, the lower bound, and therefore highest i, is denoted with ω and keeps sending all elements upwards every third step throughout the whole sorting process (except in step $t = 0$). These functions are realized in equations 2.1, 2.2 and 2.3 from Chvátals paper [5].

On each level i there are 2^i separators sorting a elements. After they are done, the π

presumably wrong sorted elements that are located on the left and right borders are sent a level upwards while the remaining χ elements get sent downwards. These a, π and χ values can be derived from table 2.2 and are also displayed in the example construction in figure 2.1.

$$\alpha(t) \begin{cases} 0, & \text{if } 0 \leq t \leq d-5 \text{ and } t \text{ is even} \\ 1, & \text{if } 0 \leq t \leq d-5 \text{ and } t \text{ is odd} \end{cases} \tag{2.1}$$

$$\alpha(t) \begin{cases} (t-d+5)/2, & \text{if } t \geq d-6 \text{ and } t \equiv 1 \bmod 4 \\ (t-d+6)/2, & \text{if } t \geq d-6 \text{ and } t \equiv 2 \bmod 4 \\ (t-d+7)/2, & \text{if } t \geq d-6 \text{ and } t \equiv 3 \bmod 4 \\ (t-d+8)/2, & \text{if } t \geq d-6 \text{ and } t \equiv 0 \bmod 4 \end{cases} \tag{2.2}$$

$$\omega(t) \begin{cases} (t+2)/3, & \text{if } t \geq 1 \text{ and } t \equiv 1 \bmod 3 \\ (t+4)/3, & \text{if } t \geq 1 \text{ and } t \equiv 2 \bmod 3 \\ (t+6)/3, & \text{if } t \geq 1 \text{ and } t \equiv 0 \bmod 3 \end{cases} \tag{2.3}$$

2.1.2 Expander Graphs

At the lowest level of the AKS Sorting Network the elements that are compared with each other are chosen using (n, d, μ)-expander graphs. Such expander graphs are basically normal graphs with certain conditions on the amount of connections each vertex has to other vertices. For this environment these conditions depend on the three parameters n, d and μ as follows:

- The expander graph consists of two equally sized partitions with each of them holding n vertices.
- Edges only exist between those two partitions, meaning that there is no edge connecting two vertices of the same partition.
- The total edge set consists of d matchings where each matching consists of pairwise distinct edges between the two partitions.
- Every nonempty set S of vertices in one partition satisfies $|N_G(S)| > \min\{\mu|S|, n - |S|\}$, where $N_G(S)$ are the neighbors of set S defined like $N_G(S) = \{u : u \text{ is adjacent to at least one vertex in S}\}$.

	$t=0$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
i	$a(i,t)$																
0	4096		256		64		16										
1		2048		512		128		32		8							
2			960		1008		252		64		16						
3				384		480		504		126		32		8			
4							192		240		252		64		16		
5										96		120		126		32	
6													48		60		64
7																24	
i	$\pi(i,t)$																
0	0		0		0		0										
1		128		32		8		0		0							
2			192		48		12		4		0						
3				384		96		24		6		0		0			
4							192		48		12		4		0		
5										96		24		6		0	
6													48		12		0
7																24	
i	$\chi(i,t)$																
0	2048		128		32		8										
1		960		240		60		16		4							
2			384		480		120		30		8						
3				0		192		240		60		16		4			
4							0		96		120		30		8		
5										0		48		60		16	
6													0		24		0
7																0	

Table 2.1: Sorting construction for 4096 elements [5].

Conditions on i	$a(i,t)$	$\pi(i,t)$	$\chi(i,t)$
$i=\alpha(t)$ and $\alpha(t+1)=i+1$	$c(i,t)$	0	$\frac{1}{2}\alpha(i,t)$
$i=\alpha(t)$ and $\alpha(t+1)=i-1$	$c(i,t)$	$\frac{1}{16}\alpha(i,t)$	$\frac{15}{32}\alpha(i,t)$
$\alpha(t)<i<\omega(t)$ and $i\equiv t \bmod 2$	$\frac{63}{64}c(i,t)$	$\frac{1}{21}\alpha(i,t)$	$\frac{10}{21}\alpha(i,t)$
$i=\omega(t)$ and $t\equiv 1 \bmod 3$	$\frac{63}{64}c(i,t)$	$\frac{1}{21}\alpha(i,t)$	$\frac{10}{21}\alpha(i,t)$
$i=\omega(t)$ and $t\equiv 2 \bmod 3$	$\frac{15}{64}c(i,t)$	$\frac{1}{5}\alpha(i,t)$	$\frac{2}{5}\alpha(i,t)$
$i=\omega(t)$ and $t\equiv 0 \bmod 3$	$\frac{3}{64}c(i,t)$	$\alpha(i,t)$	0

Table 2.2: a, π and χ values for steps $2\le t\le 3d-21$ [5].

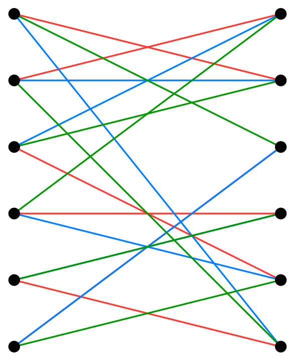
Figure 2.2: Example of an expander graph with n = 6 and d = 3, each matching is of a different color

Chvátal states that there exists an expander graph with above conditions for every positive integer n, if d and μ are positive integers satisfying equation 2.4.

$$(\mu + 1)e^{\mu+2}(\frac{\mu}{\mu + 1})^d < \frac{1}{3} \tag{2.4}$$

When creating a expander graph it can be very time consuming to check for its validity since every possible combination for a set in one partition has to be checked for their neighbors. For n vertices in a single partition there are 2^n sets. Even when assuming that it is possible to check $1 * 10^{12}$ sets per second (a typical personal computer has at most about $4 * 10^9$ cycles per second), it would take about 213 days to check a graph with $n = 64$ and $4.2 * 10^{134}$ years for $n = 512$, assuming that it takes the same time for every set to check all of its neighbors.

Therefore, it is better to generate a random expander graph or use expander graph constructions, which have not been used or tested in this thesis. The above mentioned equation 2.4 already calculates a greater depth d to account for bad random graphs.

2.1.3 Halver

One level higher, there are $(2n, \varepsilon)$-halvers which use an expander graph to actually compare and swap elements with each other. This is done by mapping the input wires of the halver to its underlying expander graph vertices. Then, for every matching of the expander graph, each vertex of the left partition is compared with the vertex it is connected to in the right partition. If the value from the left-hand vertex is greater than the value from the right-hand vertex they will be swapped, similar to a sorting network. Since all edges inside a matching are pairwise distinct, they can be performed simultaneously.

The μ value used in the underlying expander graph for a halver is calculated using $1/(\mu + 1) < \varepsilon$ [5] and the output wires are split into two blocks B_L and B_R consisting of n wires each. The B_L and B_R block will hold for every $k = 1, 2, ..., n$ at most εk of the k largest and smallest elements respectively.

2.1.4 Separator

Another level above are the $(a, f, \varepsilon_B, \varepsilon_F)$-separators, which are used in the body of the AKS Sorting Network. A separator consists of at least one halver or more depending if elements will be sent upwards. The parameter values a and f are for the amount of wires that are input and sent upwards respectively. The output, like the halver, consists out of two equally sized blocks B_L and B_R. On the left and right edge of the output lie the blocks F_L and F_R respectively. These two blocks are inside the B_L and B_R blocks and hold the smallest or rather greatest elements which will be sent upwards.

The ε_B parameter ensures the same criteria on wrong sorted elements of the blocks B_L and B_R like ε does for the halvers. Furthermore, the ε_F value ensures, that for every $k = 1, 2, ..., \frac{f}{2}$ at most $\varepsilon_F k$ of the k smallest and largest elements are placed outside of block F_L and F_R respectively.

All halvers in a separator use the same ε value, which is calculated as $\varepsilon = \min\{\varepsilon_B, \frac{\varepsilon_F}{r+1}\}$ using a r value that is the smallest nonnegative integer such that $2^r\delta \geq 1$. The δ value used for that, on the other hand, needs to satisfy inequation 2.5. Additionally to the ε value, all halvers use the same depth since it is dependent on the ε value.

$$\delta a \leq f \leq a \tag{2.5}$$

The separator consists of $t + 1$ layers, with t being the smallest nonnegative integer such that $2^t f \geq a$. On the first layer ($i = 0$) a single (a, ε)-halver is ran which can be seen in figure 2.3. The layers below the first layer only exist if there are wires to be sent up, meaning $f > 0$. The second layer ($i = 1$) consists out of two $(a - 2^{t-1}f, \varepsilon)$-halvers taking the outer wires of the previous halver as input. The next and all following layers i consist of two $(2^{t-i}f, \varepsilon)$-halvers that take the left or rather right output wires of the previous halver on their side. If there are not enough wires in one of the blocks of the previous halver the missing wires are taken from the output of the first halver that has not been used in the second layer as seen in figure 2.3 with the long arrows going from layer zero to layer two.

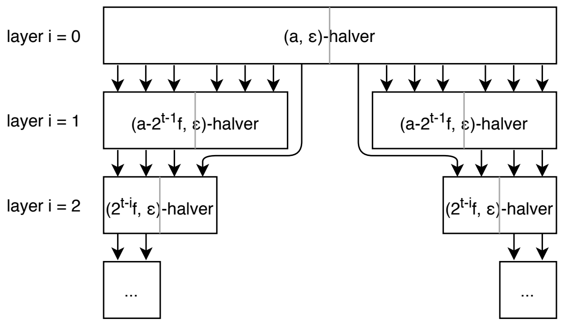

Figure 2.3: Overview of the separator layers

The only missing requirements are the ε_B and ε_F values. These two values are calculated using inequations 2.6 and 2.7 and used for all separators in the sorting construction. These values are preferably as big as possible to reduce the depth of halvers in cost of a little higher but acceptable error rate.

$$\varepsilon_B \leq \frac{1}{128} - \frac{d}{2}(1 + \frac{1}{1 - 64\delta^2}) \tag{2.6}$$

$$\varepsilon_F \leq 2\delta(1 - 16\delta) \tag{2.7}$$

3 AKS Sorting Network Implementation

The AKS Sorting Network has been implemented on the CPU and GPU. For the CPU version C++ was used along with classes and functions from its standard library, and the version for the GPU is written in CUDA with C/C++. In the CPU implementation it is also possible to replace the expander graph used in halvers by implementing the abstract `ISorter` class while the graph in the GPU version is not that easily replaceable since more adjustments need to be made. The CPU version can be compiled with `make`, its benchmark version with `make benchmark` and the GPU version with `make cuda`.

3.1 CPU Implementation

The data used for testing the CPU implementation is generated with a C++ random number generator and the input data is padded with the maximum value of its datatype. The padding ensures a input length of 2^d with d being at least 8 and a multiple of four.

Additionally, a list of lists for a mapping from a separator to its wires and from the wires to the corresponding data element is created (`wireMapping`). This way it is possible to easily know which wires are in which separator and move wires to other separators.

When sorting there are several levels which can be run in parallel. This parallelization is implemented using C++ threads and the amount of threads used on each level can be adjusted with the `threadSplit` constant. This is not only needed because a too high thread count could affect performance due to their overhead but it also could reach the thread limit on linux systems which is mostly above fifty thousand but can still be reached quite quickly in nested thread calls.

The underlying expander graph can be switched out with other ones by implementing the `ISorter` interface and passing its type to the templated `sort` function.

3.1.1 Construction

Besides the data and its size there are also a ε_B and ε_F value needed when sorting using the AKS Sorting Network. These values are set by default to well tested constants but can also be passed into the `sort` function. If only one of those two values is passed, the best possible value for the other one has to be calculated. This is done in the `testAndCalculateEpsilonValues` function by finding the best δ value and then the best possible other ε value using the inequations 2.6 and 2.7.

Since it is desirable for the other ε value to be as big as possible, only the cases where those inequation are equal need to be looked at. This results in the rearranged equations 3.1 and 3.2, however the second equation is an approximation because the calculation would be too time consuming and the function is in the needed interval almost linear. Now it is possible to calculate the missing ε value with the obtained δ value by using it in above inequations 2.6 or 2.7 with equal signs instead of less or equal.

$$\delta = \frac{1}{32}(1 \pm \sqrt{1 - 32\varepsilon_F}) \tag{3.1}$$

$$\delta = 0.007797 - 0.007797 * 128\varepsilon_B \tag{3.2}$$

Each separator in the body gets its own index, calculated with $2^i +$ *its index inside this level*, since there are 2^i separators on level i. The main part of this implementation is a `for` loop over each step $1 < t < 3 * d - 20$. Each iteration the separators for every second level are run with their currently assigned wires from the `wireMapping`. After all computations for separators of a step have been ran, the wires of every separator need to be moved upwards or downwards. This is implemented in the `moveWires` function where `upwards/2` wires are taken from each border and sent to the separator a level above while the rest will distributed on the two corresponding separators in the level below.

When all steps have finished, the now mixed distribution of wires onto separators needs to be reordered by using the mapping. Then each 64 sized wire block that has been assigned to a separator needs to be sorted itself using a fixed sorting network of size 64.

Now there is still a possibility of wrong sorted elements since we rely on randomly generated expander graphs. Therefore, a loop over each border between those 64 sized blocks is ran and it is checked that elements left to the border are smaller than elements to the right of it. If there are wrong sorted elements, they will be swapped with elements of the other block.

For debugging purposes the data can be written to files before and after the AKS Sorting Network and also before and after the fixed sorting network near the end of the program.

3.1.2 Separator

First the variables needed for the arrangement and the calls of the separators are calculated. In the inequation 2.5 it is desired to get the greatest δ value, since it is used in the derivation of r, which needs to be as small as possible. This is, because it is used in $\varepsilon = \min\{\varepsilon_B, \frac{\varepsilon_F}{r+1}\}$, which should be a greater value to be able to use more inaccurate halvers. Looking closer at the calculation of the t value, one can see that it is basically the same calculation as the r value when striving for the smallest value. The depth value d_0 for the halvers can also already be calculated, since the same value is used in every halver and can be retrieved using the implemented `calculateDepth` function from the used expander graph class.

After calculating all needed values, the first layer is ran in form of a single (a, d_0)-halver. Then the t additional layers below are executed like mentioned in section 2.1.4.

3.1.3 Halver

Inside the halver it is checked if a generated expander graph has been passed. When none was passed, a local `std::vector` of size n named `vertices` is created that holds the index of the vertex it would be connected to in each position. For each depth of the halver this list is shuffled to simulate a random expander graph.

Then the `sortingOperation` function is ran for each vertex for each depth. Inside this function, the corresponding data elements are resolved using the wireMapping with the provided value from either the `vertices` or the `getCompareVertex` function if a sorter has been passed.

In order to take care of the correct wires, like shown in section 2.1.4 and figure 2.3 a `direction` is passed to show from which side to take wires from and a `outerSize` that is half the size of the halver on layer $i = 1$. When a halver needs more than `outerSize` vertices it takes the missing one after index 2∗`outerSize`. This index will be correct since this case can only occur in layer $i = 2$.

3.1.4 Expander Graphs

For expander graphs, there is a `ISorter` class with an unimplemented method `getCompareVertex` to get the connected vertex index for a given index and depth. To save storage when there exist multiple graphs of the same size, there is another unimplemented method `extendDepth` to extend the depth to its parameter. This way the same graph can be used for various depths by just accessing the depths that are needed.

The `ISorter` objects used in the `sort` function are generated before the first step. First all needed graphs are counted by basically simulating a whole sorting procedure and then sorting them by their usage amount. After that, they will be generated by calling their constructor, beginning with the most used graph. While doing that the needed memory to generate them is also calculated and checked if it would exceed the specified memory threshold (`maxGenerationKB`). If a graph would overstep this memory threshold, or if it is only needed once, its generation is skipped and its vertices will be generated locally later on, like described in section 3.1.3.

For the expander graphs, there is already a `RandomExpanderGraph` class implemented, which extends `ISorter`. This class holds an adjacency matrix `adjMatrix` with a list of connected vertices for each depth in form of two nested `std::vector`, where the outer one indexes the vertex and the inner one the depth with the corresponding connected vertex index as value. Therefore, `getCompareVertex` just returns the value in `adjMatrix` for the vertex index at the specified depth and `extendDepth` resizes the `adjMatrix` and adds missing depth values.

3.1.5 Improvements

One obvious big speed up for this sorting network is the usage of threads. In this implementation, the standard C++ threads `std::thread` have been used in several places. Looking at the construction of the network it is clear that all separators inside a step can be run in parallel, since they all operate on their own wires and do not depend on any other separator inside the current step. This means, that it is possible to create threads that take care of single or multiple separators simultaneously, while the main thread waits for all to finish in order to continue with the movement of wires and precede to the next step.

When looking a level lower at a separator, it is visible that each layer can also be run in parallel, since the left and right side use their own set of wires. To prevent unnecessary thread overhead for small halvers, layers only run in parallel for layer sizes of at least 100. Going another level deeper to the halvers, it is apparent that even more threads can be used for the comparison operations. All comparisons for a matching of the expander graph could be run in parallel, but since one comparison per thread would create too much threads, resulting in a too big overhead, each thread does multiple comparisons. To prevent errors with the random number generator when multiple threads want to change it at the same time, each thread holds its own generator.

For every amount of input there will be eventually separators towards the end that are quite small. These separators, however, still execute halvers with expander graphs having depths of several thousands when using the proposed depth calculations in [5] or [1]. Therefore, tests have been made to find better depths that will be used instead. During this process, which is described more precisely in section 4.3 it was discovered that a depth of 1000 is enough for all separator sizes to be within the error tolerance. More accurately, the depth will be set to 250 if the size is below 100 and to 800 up to a size of μ. Otherwise, a depth of 500 will be used.

Another possibility for improvement found during the depth tests is the usage of an alternative sorting method for separators that allow no errors. Hence, a perfect halver has been implemented, which basically just compares each element with every element from the other halve. When the perfect halver is used instead of ε-halvers that are not allowed to have errors, it is possible to reduce the depths of the other regularly computed separators, because the perfect halver does not introduce any errors. This leads to a very small depth of 30 for the other separators when using perfect halvers. Therefore, it will reduce overall run-time and also sometimes the run-time of separators since those cases where no errors are allowed will only appear at small sizes, where the quadratic run-time of a perfect halver is less than the constant one of an ε-halver. More information to the perfect halver usage can be found in section 4.4.

3.1.6 Benchmark

For the CPU implementation there is also a benchmark variant, which runs the AKS Sorting Network multiple times for different sizes based on the passed parameters. It is required to pass a starting (`-f <amount>`) and ending (`-t <amount>`) exponent, which results in the program running every fourth exponent in this interval. For each of those exponents the sorting network is run several times based on the passed value (`-r <amount>`, default = 1) with new randomly generated data each time. Information and statistics for the batch of runs for each size is written to console and into the `dbg_benchmark.txt` file. By enabling the `createStatistics` constant, a csv file `dbg_statistics.txt` is created with data of the `SortOutput` struct for each run.

3.2 GPU Implementation

In comparison to the CPU implementation, the GPU implementation is less configurable. It uses a `RandomExpanderGraph` from above as underlying expander graph for sorting. A simplified overview of the program flow can be seen in figure 3.1. There it is noticeable, that in each step of the sorting construction, all levels are executed in parallel while also preparations for the wire movements are done. For each level, a kernel is launched that takes care of all separators on its level. Since all separators on the same level are of the same size, and therefore are also equally constructed, they are perfectly suitable to run together in a single kernel and benefit from the CUDA core speed improvement when executing similar operations.

In the code for this implementation all variables allocated on GPU (also known as device) side carry the prefix `d_`, while all variables on CPU (also known as host) side use the prefix `h_` . Data on device side is currently never stored in shared memory, since it is never used enough to be of sufficient advantage to account for transfer times to and from shared memory. Additionally, it is limited at 48 KB per block and also 48 KB per streaming multiprocessor, which is reached quickly with greater input quantities.

To run the kernels and memory transfers in parallel, several CUDA streams are created. The single stream `generalStream` is used for data transfers and the array of streams `levelStreams` is reserved for separator kernel launches, where each level gets its own CUDA stream. This allows the memory transfer and all separator kernels to be run simultaneously. Operations, like the movement of wires, are executed in the default stream, which is inherently synchronous to the other streams.

3.2.1 Wires

One main area, that is handled differently in this version, is the management of wires. There are two arrays `d_separatorMapping` and `d_dataMapping` that belong together.

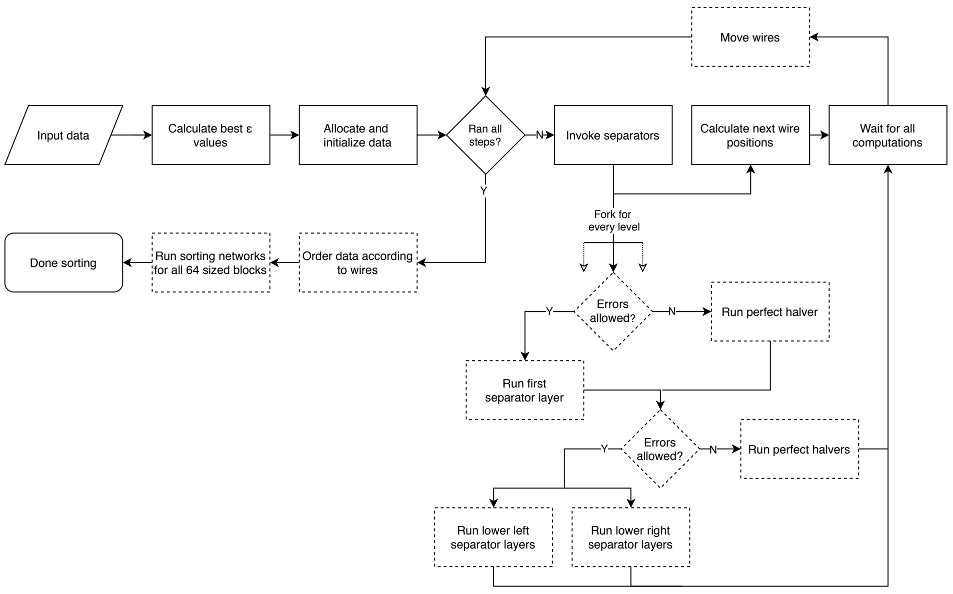
Move wires
Input data
Calculate best ε values
Allocate and initialize data
Ran all steps?
N
Invoke separators
Calculate next wire positions
Wait for all computations
Y
Fork for every level
Done sorting
Run sorting networks for all 64 sized blocks
Order data according to wires
Errors allowed?
Y
N
Run perfect halver
Run first separator layer
Errors allowed?
Y
N
Run perfect halvers
Run lower left separator layers
Run lower right separator layers

Figure 3.1: Simplified flowchart of the CUDA implementation (dashed processes are run on the GPU)

The `d_separatorMapping` array maps separator indices from the sorting construction to an array index i in `d_dataMapping`. Beginning at i in `d_dataMapping` are the wires used in this separator, with the data at index $i + k$ being the element behind wire k.

Movement of wires upwards or downwards is done in a separate kernel on the GPU. Since these moving operations all happen in parallel, it is not possible for threads to know where to place wires by themselves without atomic operations or locks. Therefore, it was decided to precalculate the space all wires will need on the CPU in the corresponding arrays `h_separatorMapping` and `h_dataMapping`. Afterwards, those offsets are copied over so the threads only need to read out their offset.

Since data needs to be read from the mapping arrays, while also new wire mappings are written, it is necessary to have two arrays for each mapping. These mappings are referenced by two pointers pointing to the currently used array and the array used in the next step. During a step, the moved wires are put into `d_newSeparatorMapping` and `d_newDataMapping`, and before the next step, their pointers will be swapped with the other two device arrays to make the new mappings the current mappings.

3.2.2 Construction

The general construction of the whole sorting body is similar to the CPU implementation with the invocations of separators, but now the movement of wires needs to be prepared by calculating the distribution of all wires onto the separators. After that, these offsets need to be copied to the GPU, and all this happens parallel to the execution of separators. To achieve this parallel execution of several separators, CPU calculations and data movement to the GPU, an array of CUDA streams is used. In this array each level has its own stream which is used in the invocation of the `separatorKernel`. Seeing that these kernels take care of all separators on a single level, it is clear that all levels will run in parallel, while the CPU continues its execution and prepares the wire movement.

3.2.3 Separator

Separators are executed by running the `invokeSeparator` function, which takes amongst other things, the CUDA stream for the current level, the vertex array, both mapping arrays and the current level index, to calculate how many separator need to be run. Inside the function, first the needed values are calculated, just like in the CPU implementation. Then a `separatorKernel` is invoked, in which each thread block takes care of one separator. Inside the kernel, pointers of needed data to the correct offset for its separator index are created and elements in the passed vertices array are initialized. The vertices array also lies in global memory, because shared memory is limited and would only be sufficient for small separators. After these operations are done, the device function `halverSorting` is called which simulates a halver for the first layer of the separator. The lower layers are also run in the same kernel, however, the first halve of threads inside a block takes care of the left side halvers, while the second halve computes the right side halvers.

3.2.4 Halver

The `halverSorting` and `halverSortingOperation` functions are almost identical to its CPU variants. The two major differences are, that graphs are always generated locally and that the used shuffle function is the device function `shuffleParallel`, which shuffles the vertices array using the whole thread block. The parallel algorithm used for this is the MergeShuffle by Axel Bacher et al. [3], unfortunately, it was not possible to make it work in time and therefore, a Fisher-Yates shuffle is currently run on a single thread. Since the vertices are stored in global memory, each thread needs to know at which offset the vertices for its current separator are located. Seeing that a separator needs as much vertices as wires, it is beneficial to use the same mapping that is used for wires for vertices and conveniently this mapping is already available inside a halver.

3.2.5 Expander Graphs

The graphs used in the CUDA implementation are always locally created and behave like a `RandomExpanderGraph`. For shuffling of vertices, random numbers are required, which are provided by the cuRAND library included in the CUDA Toolkit [9]. For the generation, each thread needs a `curandState_t`, which is initialized before the sorting procedure and holds the current state of the underlying XORWOW generator used by default. It was chosen to not generate graphs beforehand, because GPUs typically have less memory capacities than the CPU. Additionally, part of the storage is already needed for sorting data along with other helper arrays, like wire mappings, so the graphs would consume the remaining memory very quickly.

3.2.6 Improvements

A small and barely noticeable improvement is the usage of pinned host memory allocated with `cudaMallocHost`. This type of memory allows the CUDA driver to directly access the pinned memory instead of having to transfer memory from the default pageable memory to pinned memory when using it for `cudaMemcpy` operations. Furthermore, pinned memory also allows memory transfers to be executed in parallel to kernel executions. This is useful to already transfer data needed for the `moveWiresKernel` while the separators kernels are sorting data.

Moreover it is possible to enable the usage of perfect halvers, instead of regular ε-halvers, by changing the `usePerfectHalver` constant. These halvers are implemented as device functions just like in the CPU version and are also called the same way.

4 Analysis

The following sections cover different optimizations and changes for the implementations, along with tests showing their impact. Testing for the CPU implementation has been done on a Ubuntu 20.04 system with an Intel i5-9500 (6 cores @ 3.00 GHz) and 60 GB random access memory. The default constants chosen for the tests are as follows:

```
constexpr bool usePerfectHalver = false;
constexpr uint32_t maxGenerationKB = 10000000;
constexpr bool regenerateGraphs = false;
constexpr bool useThreads = true;
constexpr uint32_t threadSplit = 4;
constexpr uint32_t generationThreads = 50;
constexpr uint32_t halverThreshold = 10000;
constexpr uint32_t separatorThreshold = 1000;
constexpr uint32_t mergeShuffleThreshold = 10000000;
```

Additionally, tests have only been run on integers, using the `std::minstd_rand` random number generator, compiled with the `-O2` optimization flag and up to a size of 2^{24} (16777216), since sizes greater than that take too much time to finish.

A first good way to verify that the sorting construction is correct and each wrong sorted element gets eventually placed into its right partition, is to intentionally introduce errors. This can be done in the CPU version by enabling the disabled block inside the separator function. This block uses a simple, but not optimized, perfect halver and places elements outside their intended partition afterwards. Without changing any values, this code already introduces as many errors possible and when executing the program it is observable that still all elements are correctly sorted at the end.

4.1 Sorting Time

In figure 4.1 the sorting times for different data types can be seen. For structs the `TestStruct` in `benchmark.cpp` is used, which holds a integer (`key`) for comparisons between structs and two additional integers without use. There is barely a difference

size	256	4096	65536	1048576
integer -O3	1.017 ms	57.022 ms	1.284 s	51.063 s
integer	1.004 ms	55.044 ms	1.277 s	51.386 s
long	1.039 ms	57.813 ms	1.331 s	56.679 s
struct	1.811 ms	63.989 ms	1.450 s	62.776 s

Table 4.1: Average sorting times

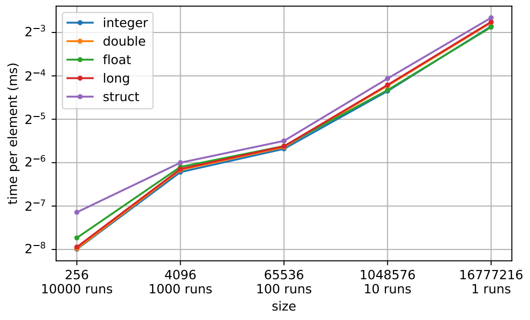

Figure 4.1: Sorting times for different data types

between the numerical data, besides towards the right side, where the 32 bit data types are slightly faster. Structs are, as expected, slower in general and then there is also a graph for integer sorting with the `-O3` optimization flag. This flag improves the sorting speed a little and does not seem to introduce errors.

Overall, the sorting times in this area of sizes are a lot higher than usual sorting algorithms like `std::sort` or Bitonic sort, since the AKS sorting network relies on a huge constant. Eventually, its sorting time will fall below, but this will not happen until sizes of more than 2^{11000} elements which is physically impossible to test. However, it is possible to do simulations and calculate how many operations would be needed, which is done in section 4.7.

4.2 Expander Graph Generation

In these two implementations of the AKS sorting network a lot of time is spent on generating expander graphs. This is, because a permutation of n elements needs to be computed for each matching in a graph with $2n$ vertices. Since it is desirable to perform this shuffle in-place, the Fisher-Yates shuffle with run-time $O(n)$, which is also used in `std::shuffle`, seems to be the best option.

It was also tried to use linear congruential generators in combination with the Hull-Dobell theorem [6]. This theorem sets requirements on the parameters of the generator, resulting in the output being a permutation of a gapless sequence of numbers. However, it would be hard to find hundreds of unique parameter sets that would match the requirements, and the quality of the permutations is also questionable.

For a parallel execution, the MergeShuffle by Axel Bacher et al. [3] was chosen. This algorithm basically shuffles partitions of the data in parallel and then merges them together. The CPU implementation uses the normal Fisher-Yates shuffle inside ε-halvers up to a threshold of ten million. This threshold was decided as best after doing a few test, but could not be fully tested because it is in a number range where testing takes a lot of time.

Additionally to the shuffle algorithm, the underlying random number generator also plays a huge role. Here, three generators have been compared with each other: `std::mt19937`, `std::minstd_rand` and `xoroshiro64**` from [4].

Regarding the influence on sorting time, `std::mt19937` takes the most time, while `std::minstd_rand` and `xoroshiro64**` are tied as seen in figure 4.3. This is, because the period of the Marsenne Twister is way higher than the other two, with 2^{19937} against 2^{32} (`std::minstd_rand`) and 2^{64} (`xoroshiro64**`). Looking at the quality of their random numbers by running separators and counting the amount of valid (satisfying the ε values) runs, the results in figure 4.4 can be seen. Compared to a default run from figure 4.5 using `std::minstd_rand`, there is almost no difference between the generators. There are small deviations between them, but these vary between tests and are rather the same overall.

When expander graphs are needed multiple times it is desirable to generate them once and reuse them numerous times. To see how impactful this can be, the used memory created by expander graphs, along with its proportion that is reused, has been pictured in figure 4.2. There it is visible that about an eighth is reused at least ten times, and about every 32nd graph is used at least one hundred times. Therefore, graphs are generated before the start of the sorting network, by filling the amount of memory defined in `maxGenerationKB` with graphs, starting with the most used ones. Single use graphs are never generated beforehand, since they do not benefit from reusage and, additionally, would suffer from the higher access times of random access memory.

Another feature that has been implemented is the regeneration of graphs as soon as memory is available again, which can be enabled with `regenerateGraphs`. Since the

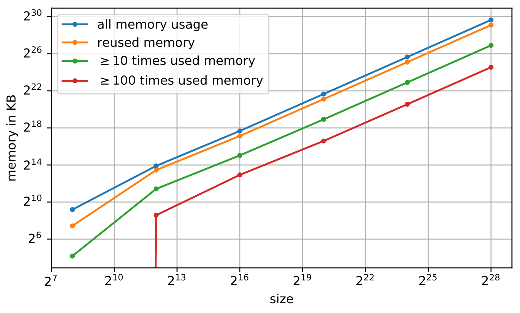

Figure 4.2: Memory usage for different sizes

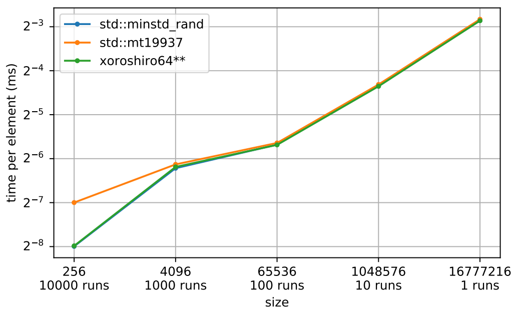

Figure 4.3: Time comparison of different random number generators

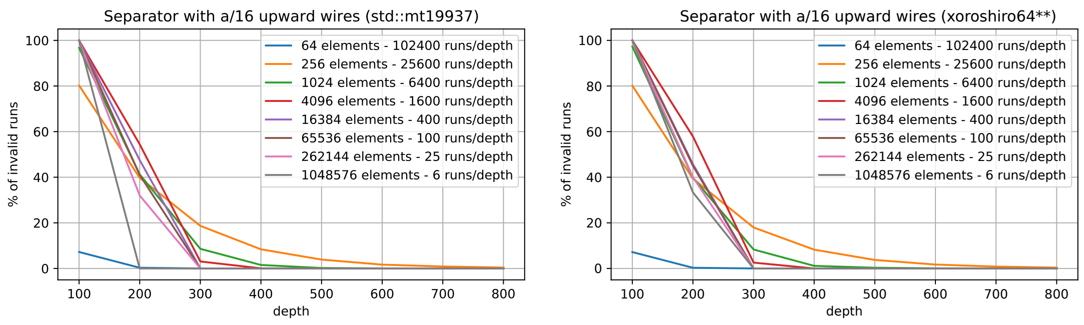

Figure 4.4: Proportion of invalid runs with std::mt19937 and xoroshiro64**

usage amount is attached to the graphs, it can be decremented each time it is accessed and if it hits zero the memory is deleted to make space for new graphs or other things, because it has to be deleted eventually anyway. This regeneration can improve sorting speed a little. A run with, for example, 1048576 elements and a memory limit of 10MB takes normally 74 seconds while it takes only 68 seconds with graph regeneration enabled.

4.3 Expander Graph Depth

The proposed formula for the depth of random expander graphs in Chvátals paper [5] results in relatively high values. Typical μ values inside separators are around 400 to 600, which will be calculated to depths of 150000 to 350000, even for separator sizes of 24. Using the formula of the original AKS paper [1] depths will be a bit smaller, but still high, at around 5000 to 10000.

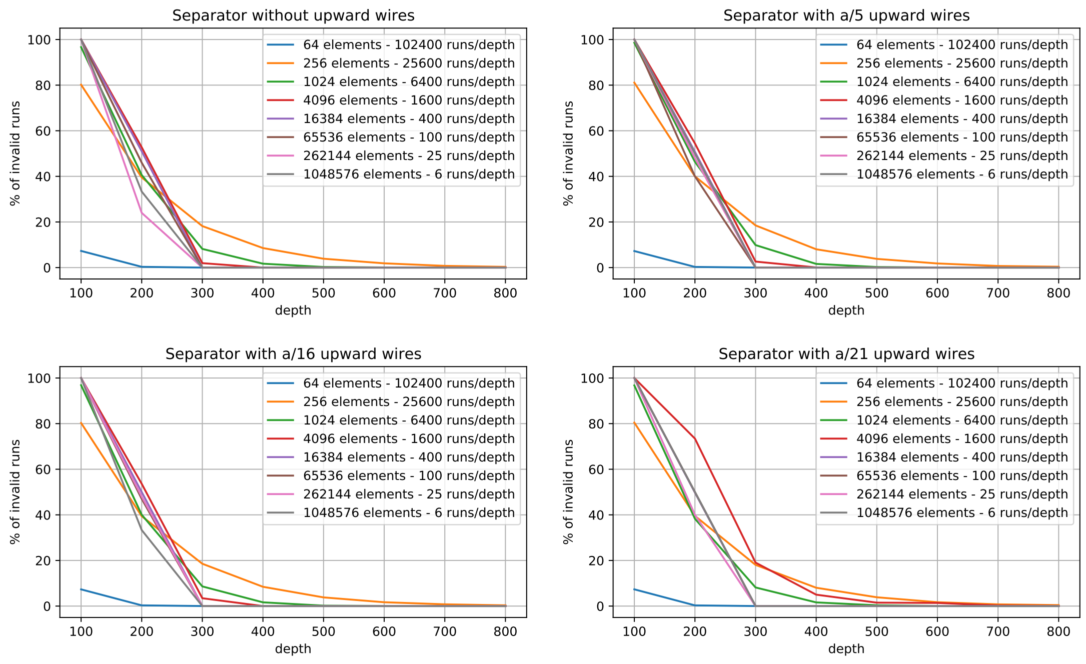

Figure 4.5: Percentages of invalid separator runs for several depths

Therefore, tests for separators have been made in which invalid runs are counted for different depths. A run is valid if it satisfies the constraints of the ε_B and ε_F values on separators. The corresponding testing code is in `aks.cpp` and a test, for example, with a size of 4096 elements and 10 runs is started using `./cpuAks 4096 -t -r 10`. The results of those tests with $\varepsilon_B = 0.00243$ for the four different situations of upward wire proportions can be seen in figure 4.5.

In the four diagrams it is observable that most runs are valid with a depth of at least 500. A exception to that are separators of size 256, which need a depth of at least around 800 to be mostly valid. This is, probably, because until around $\frac{1}{\varepsilon_B}$ elements, which becomes to 412, a separator is not allowed to have a single error, and therefore, is more likely to lead to invalid runs the closer the size gets to this value. Additionally, separators of sizes below 100 tend to be good enough with a depth of around 250. With this information, the depth calculation of random expander graphs can be adjusted to 250 if the size lies below 100, to 800 if with sizes up to and including μ and 500 else. The μ value is chosen here, because it will be at least $\frac{1}{\varepsilon_B}$ or greater depending on ε_F and the amount of upward wires.

4.4 Separator Optimizations

Since there are problems in the section above with separators that are not allowed to have errors, an alternative perfect halver has been implemented to replace the ε-halver in exactly these cases. This perfect halver works by simply comparing each element of one halve with every element of the other halve. This results in $(\frac{n}{2})^2$ operations and a depth of $\frac{n}{2}$. The perfect halver will be used for the first layer, if the separator is not allowed to have errors regarding ε_B, and for all layers below the first layer, if there cannot be errors regarding ε_F. This results in the improved times seen in table 4.2. As already mentioned in the CPU implementation, this allows the graph depths of the normal ε-halvers to be of 30. Looking at this, it seems plausible that the reduced depth comes from the perfect halver doing more than just halving and partly sorting the data. However, this is does not affect the amount of wrong sorted elements, which can be seen by enabling the randomization of the halves after a perfect halver execution in `perfectHalver.hpp`.

4.5 Epsilon Values

Two parameters that can have a big influence on run-time are the ε values. These values affect how many errors inside a separator are allowed, and therefore, also the depth of ε-halvers which can, when chosen badly, result in a multiple of the usual sorting time.

To test which ε values are the best, a simulation of the AKS sorting network has been written in python. This program calls the same separators the algorithm would, but does not actually sort data and just calculates the operations executed when a halver would have been ran. Additionally, the depth, meaning the most operations needed to execute a halver in parallel, are accumulated. The resulting graphs for ε_B at sizes 2^{20} and 2^{80} can be seen in figure 4.6.

The lowest points for the graphs are almost similar in both diagrams with 0.00243 being the best ε_B value regarding operation count and 0.00181 the depth count. While the best value for depth is especially useful for executions with many threads like on a graphics card, the value for minimum operations is preferred for the CPU implementation. This can be seen when comparing CPU runs with different ε values: 2^{24} elements are sorted in 2334 seconds with $\varepsilon_B = 0.00243$, 2344 seconds with $\varepsilon_B = 0.00207$ and 2343 seconds with $\varepsilon_B = 0.00181$. Therefore a ε_B value of 0.00243 is chosen as de-

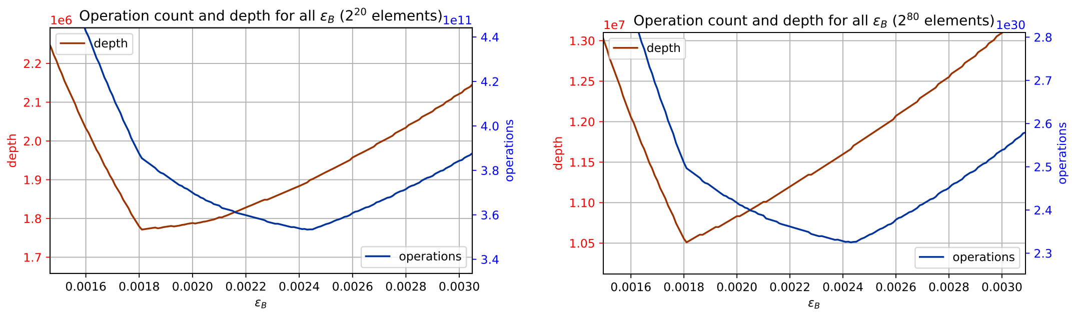

Figure 4.6: Tests for best ε values with 2^{20} and 2^{80} elements

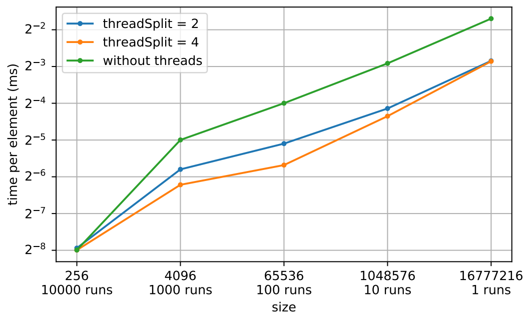

Figure 4.7: Tests for different thread configurations

fault for the CPU implementation. For the GPU it seems that this value is also the best for executable sizes.

4.6 Parallelization

The `threadSplit` constant decides onto how many threads work is distributed in the body and halvers. This results in maximum $2 + \frac{d-6}{6} * 2 * \texttt{threadSplit} * \texttt{threadSplit}$ threads running simultaneously, because in a construction can be at most $2 + \frac{d-6}{6}$ active levels per step, since until $d - 6$ every third step the construction is expanded by one level downwards, and each separator calls at most two halvers concurrently. In figure 4.7, different thread configurations have been tested and it is observable, that in general more threads are better. In this case, the tests have been run on six cores, and therefore, it is best to use a `threadSplit` of six or more, if the user is allowed to run the resulting maximum simultaneous threads, which was not the case for the testing system.

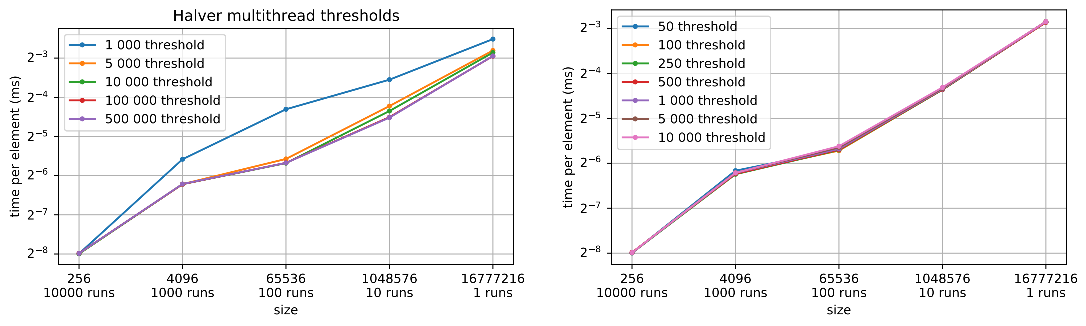

Figure 4.8: Tests for halver and separator multithread thresholds

In the ε-halvers implementation for the CPU it is possible to distribute comparisons onto threads. Since this would be slower at small sizes, a threshold (`halverThreshold`) needs to be set at which threads are used. Possible values for it have been tested in figure 4.8 and the originally chosen 100 thousand are together with 500 thousand the fastest.

For separators a similar threshold exists (`separatorThreshold`), which is also tested in figure 4.8. This threshold does not result in a great variety of sorting times, but looking closely, a value of 100 is the fastest of them and will therefore be used in the final version.

4.7 Complexity

To examine the sorting network further than 2^{24} elements, a simulation in python has been written which calls the same separators the usual algorithm would do, but accumulates the operation count needed instead of actually performing them. This simulation was also used for the ε values in section 4.5, and has been added to the provided files (`simulation.py`) for those interested. The results of this for sizes up to 2^{800} can be seen in figure 4.9. The left diagram shows the operations per element, where *normal* uses the depth calculation from the original paper [1], *custom depths* the default depths for this implementations defined in section 4.3 and *perfect halver* the alternative halvers, if no errors are allowed. Two reference graphs have been added to show approximations for *custom depths* and *perfect halver*, which show that the default depths follow roughly $2^{10.75} * log\ n$ and the perfect halver usage $2^{6.8} * log\ n$ operations per element. The sorting network with perfect halver usage already falls below the Bitonic sorts operations at about 2^{110}, however, these are only sequential results.

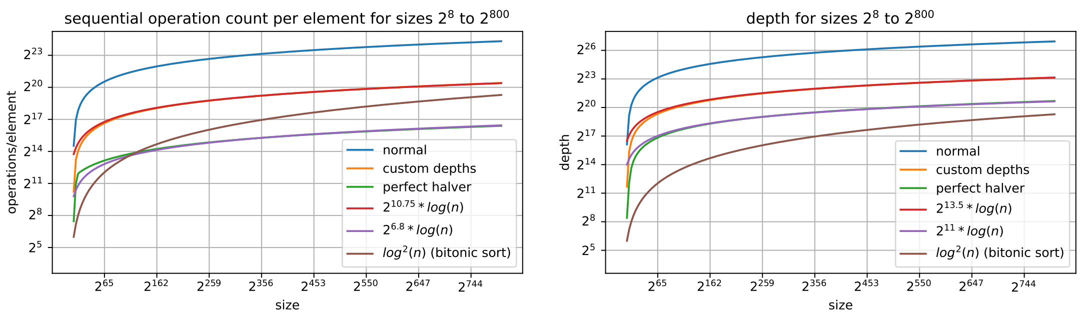

Figure 4.9: Complexity simulations of operations and depth

size	256	4096	65536
CUDA	62 ms	1.179 s	24.271 s
CPU	1.0038 ms	55.044 ms	1.277 s
CUDA with perfect halver	<1 ms	92 ms	1.768 s
CPU with perfect halver	0.0014 ms	8.235 ms	0.166 s

Table 4.2: Comparison between CPU and CUDA for integer data

The right-hand diagram shows the respective depths, which would be the run-time for a parallel execution. Here the Bitonic sort will be beaten at 2^{2048} elements by the sorting network with perfect halver usage and at 2^{11585} with the custom depths.

4.8 CUDA Optimizations

There have also been added perfect halvers to the CUDA version, and the results can be seen in table 4.2. Unfortunately, these do not go beyond 65536, because the implementation has bugs and therefore the other test results could also include errors.

4.9 Small Optimizations and Changes

For the CPU version, there has also been added a small part towards the end, which fixes wrong sorted elements after the 64-sized sorting networks are done. It basically looks at each border between those 64-sized blocks for elements that should be on the other side of the border. However, this only resolves wrong sorted elements as long as they were placed into a adjacent block, which is mostly the case unless there have been a lot of errors during sorting.

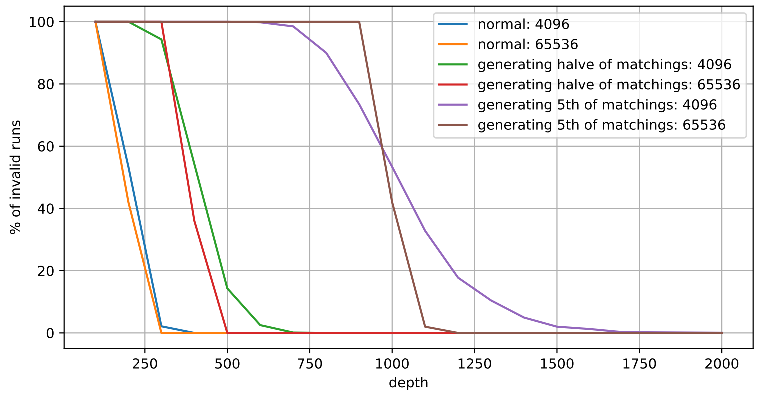

Figure 4.10: Errors of halvers with and without reused graph matchings

4.10 Doubtful or Unsuccessful Optimizations

One strategy that was tried without success is the reusage of graph matchings inside the same graph. This means, that there are for example only 100 matchings generated for a graph that actually needs 500 matching, and thus, just keeps reusing the generated matchings to save space and generation time. When testing halvers where only a halve or a fifth of the matchings have been generated, the results in figure 4.10 can be observed. It seems, that there are double the amount of errors when generating only a half of the matchings, resulting in twice the depth required to sort inside the allowed error tolerance, and about five times the depth for a fifth of generated matchings. This is therefore not an improvement in terms of run-time, but could be an option, if generating random data is very expensive and storage limited.

It was also tried to replace the perfect halver, that can be enabled to speed up separators, with a bitonic sorting network. The plan was to run the Bitonic sort until the data is halved and thus not execute the whole sorting process, which would result in a better run-time than the currently implemented perfect halver. However, the standard Bitonic sort is only applicable on sizes of a power of two and in the chosen alternative from [8], which is a modified version that sorts any size of elements, there is no point during sorting at which the data is halved. This means, that the Bitonic sort had to be run completely which would lead to the AKS sorting network being mostly only a composition of Bitonic sorters. The almost working, adapted implementation of this sorter has been put into `bitonicSort.hpp` for interested readers.

5 Potential Improvements and Conclusion

This chapter is about improvements that could be made to the implementations if better algorithms or methods are found. One of those is the usage of different expander graphs. The currently used expander graph relies on many operations to a random number generator and also is of a relatively great depth to account for bad random graphs. Expander graph constructions, for example, could be used to generate better graphs that ideally also have a smaller depth to reduce the amount of comparisons during the sorting.

The `RandomExpanderGraph` could also be improved by using better shuffle algorithms. It would be, for example, ideal if every element in a permutation could be determined deterministically, so there are no extra arrays needed to store vertices, and therefore, they would also not needed to be generated beforehand.

Another approach to expander graph generation is to precalculate some of them and store them within the code. These graphs can be well tested for errors, and therefore, also have a smaller depth. A further method is to use fixed seeds for the random number generators. These fixed seeds, ideally, create good expander graphs and are found by testing different seeds until a graph created with it is good enough. However, these approaches are not really useful for small graphs if there are already alternative halvers used for separators that are faster, like the perfect halver.

It could also be worth to look closer at the proofs regarding the whole sorting construction. If the sorting construction is designed to hold a higher amount of elements below the nodes in the last step, it would be feasible to use higher ε values. This allows the halvers to make more errors, and therefore, also be of a lesser depth, which could decrease the total sorting time.

To conclude, the AKS Sorting Network is not useful for practical applications, nonetheless, run-time can be improved a bit. These enhancements can be achieved by adjusting the used depths for random expander graphs, changing random number generators or using alternative methods for certain cases. Additionally, ε values and thresholds, at which threads are used, can be tweaked.

Bibliography

[1] Miklós Ajtai, János Komlós, and Endre Szemerédi. An o(n log n) sorting network. https://dl.acm.org/doi/10.1145/800061.808726.

[2] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *J. ACM*, 41(5):1020–1048, 1994.

[3] Axel Bacher, Olivier Bodini, Alexandros Hollender, and Jérémie O. Lumbroso. Mergeshuffle: A very fast, parallel random permutation algorithm. *CoRR*, abs/1508.03167, 2015.

[4] David Blackman and Sebastiano Vigna. Scrambled linear pseudorandom number generators. http://vigna.di.unimi.it/ftp/papers/ScrambledLinear.pdf.

[5] Vašek Chvátal. Lecture notes on the aks sorting network. https://users.encs.concordia.ca/~chvatal/notes/aks.pdf.

[6] T. E. Hull and A. R. Dobell. Random number generators. https://chagall.med.cornell.edu/BioinfoCourse/PDFs/Lecture4/random_number_generator.pdf.

[7] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[8] Hans Werner Lang. Bitonic sorting network for n not a power of 2. https://www.inf.hs-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm.

[9] NVIDIA. curand library programming guide. https://docs.nvidia.com/pdf/CURAND_Library.pdf.

All links were checked on August 25th, 2020.

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Date and Signature: