182.731 GPU Architectures and Computing - Group Project

Group 1:
Hiti Mario, 01327428
Gollmann Christian, 01435044
Hinterseer Simon, 09925802
Loch Adam, 12044788

June 20, 2022

Contents

1 Task description

This report describes implementation and benchmarking of the Kruskal and the Filter Kruskal algorithm, two algorithms used to generate minimum spanning trees of a graph. Both serial and parallel versions are implemented for both algorithms.

2 Compilation and usage

The project is uploaded to TUWEL and github [1]. However due to file size limitations we also provide our solution with pregenerated graphs directly on the VM (server: gpu3mv1; directory: *$HOME/handin/GPU_ArchComp/Ex2*)

We use *CMake* to generate build artifacts and and *make* to invoke the compiler. To compile a release build using all optimizations use

```
cd Ex2
mkdir build && cd build
cmake -D CMAKE_BUILD_TYPE=Release -D BUILD_TESTS=OFF ..
make -j8    // Compiles with 8 Cores
```

To run the code use:

```
// Run with default arguments
cd Ex2/build
make run
// Run with specific arguments:
cd build/src
./ex2 [ARGS]
// Example:
./ex2 --pinned-memory -s 2 -f 1
```

For a complete list of possible parameters please refer to Tab. 1 or use "–help". Note that the parameter "–inputfile [XYZ.csv] requires a second file that contains the solution to the MST problem in the same file format. The filename must be "XYZ_gt.csv" and located in the same folder. The calculated solution is written to *out/MST_calculated.csv*.

3 Software and data structure

The program features several different kernels (CPU and GPU) for each step of the algorithm which are chosen at the start of the program.

The graph is stored as a list of edges which is equivalent to the compressed COO-Format of the adjecency matrix. Data is handled by the EdgeList-class which stores values for source-node (coo1), target-node (coo2) and weights in 3 arrays.

Furthermore it holds 3 pointers to arrays on the GPU which hold a copy of the graph. The data on both devices is not updated simultaneously but only synced when necessary. At the start of every kernel a sync function is invoked that updates the data if and only if the previous kernel ran on a different device.

After each run the result is compared to a precalculated ground truth and the runtimes for each kernel are printed to the console.

[1] `https://github.com/hitimr/GPU_ArchComp/tree/main/Ex2`; Note that commit authors and statistics are falsfied since everyone who pushed from the server used the same account

4 Standard Kruskal

Kruskal's algorithm is used to generate a minimum spanning tree, given a undirected graph where edges between nodes can have different weight values. A graph can have several MSTs. Kruskal's algorithm will find one possible MST by growing several sub-trees inside the graph which over the course of time grow together to one final tree.

4.1 Implementation

Starting point is an unsorted edgelist of integer triples - {weight, start, end} - which means the node *start* is connected to node *end* via an edge with a weight of *weight*. The algorithm then takes the following steps, also refer to figure 1(b):

- The edgelist gets sorted according to the weights.
- Triples of the edgelist get evaluated in order. If the triple's start and end node do not belong to the same sub-tree already, the triple gets added to the result edgelist. start and end node now belong to the same sub-tree. If they had belonged to two different sub-trees before, both trees would have been merged in this step. We keep track of which tree a node belongs to inside an Union-find structure.
- When looking for a node's tree, it is possible to employ Path compression. This helps keeping those look-ups computationally short. More on that topic can be found in the Partition and Filter section, see section 6.
- The algorithm is finished when *(number of nodes - 1)* edges have been added to the final tree.

It is obvious that the computationally most expensive step here is the sorting. It is also the only part that can be parallelized as growing the MST is an inherently sequential task.

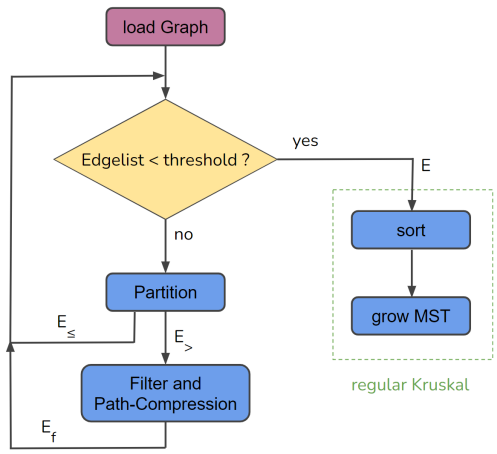
5 Filter Kruskal

As it has been pointed out, that sorting makes up the computationally most intensive part of Kruskal. We therefore also implemented the so called Filter Kruskal algorithm [1]. This algorithm combines the divide-and-conquer principle and filtering to reduce the input for the sort function.

5.1 Implementation

Filter Kruskal's idea is to recursively partition the edgelist into several parts which get sorted individually, see figure 1(a).

- If the edgelist is smaller than a given threshold, it gets passed to the regular Kruskal which will grow the first part of the MST.
- Otherwise, a random pivot element out of the weight values gets chosen and the whole edgelist gets partitioned into two groups: eges, whose weigths are less-equal or greater than the pivot.
- The list of smaller values is given back to the threshold check until it is small enough to be processed by the regular Kruskal.
- When arriving at the point where the algorithm deals with the list of larger values, some parts of the MST have already been grown. Edges that therefore can no longer be part of the MST (that's when start and end node of an edge already belong to the same tree), get filtered out. This step reduces the sorting input. It is here, where the Filter algorithm can become faster than the regular Kruskal.
- The termination criterion is the same as for the regular algorithm.



(a) Flowchart Filter Kruskal

Procedure kruskal(E, T : Sequence **of** Edge, P : UnionFind)
  sort E by increasing edge weight
  foreach $\{u, v\} \in E$ **do**
    if u and v are in different components of P **then**
      add edge $\{u, v\}$ to T
      join the partitions of u and v in P

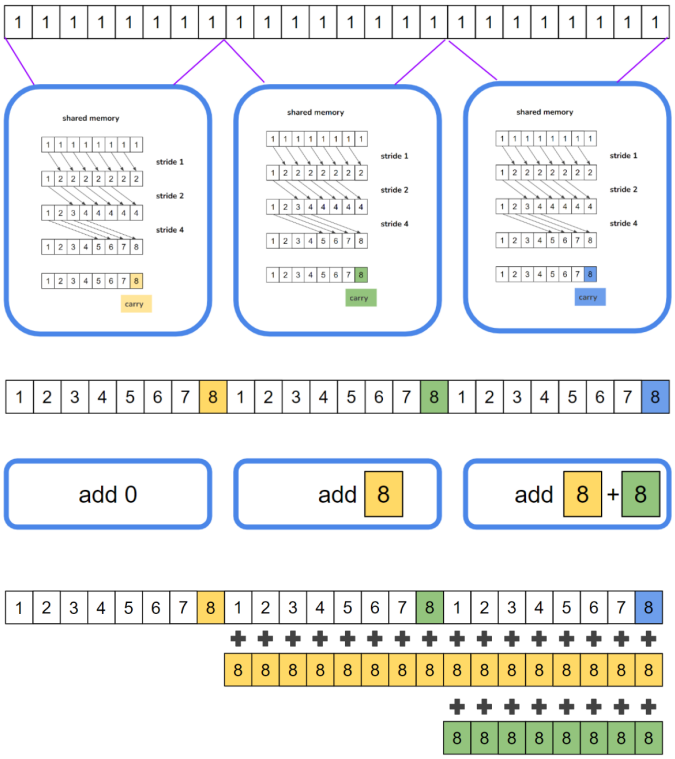
Procedure filterKruskal(E, T : Sequence **of** Edge, P : UnionFind)
  if $m \leq$ kruskalThreshold$(n, |E|, |T|)$ **then** kruskal(E, T, P)
  else
    pick a pivot $p \in E$
    $E_{\leq} := \langle e \in E : e \leq p \rangle$
    $E_{>} := \langle e \in E : e > p \rangle$
    filterKruskal$(E_{\leq}, T, P)$
    $E_{>} :=$ filter$(E_{>}, P)$
    filterKruskal$(E_{>}, T, P)$

Function filter(E)
  return $\langle \{u, v\} \in E : u, v$ are in different components of $P \rangle$

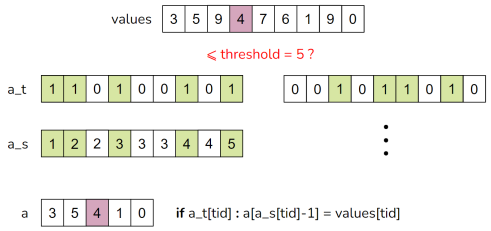
(b) Pseudocode of Kruskal and Filter Kruskal

Fig. 1: Instruction flow of Kruskal and Filter Kruskal

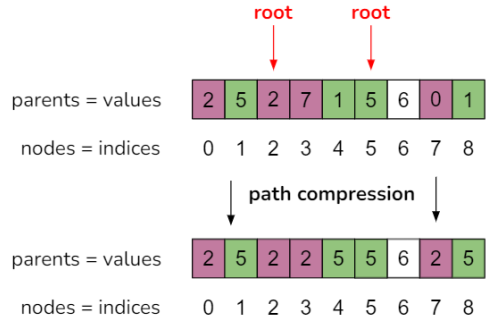
The Filter Kruskal provides more possibility for parallelization as Partition and Filter Kernel can be parallelized in addition to sorting.



(a) Inclusive scan. First an inclusive scan is performed kernelwise. Afterwards each block performs an inclusive scan on the carries array and adds the result up to respective part in the final result



(b) Partition Kernel. The value array gets checked and mask arrays (a_t) are built. Those get scanned (a_s) and on basis of that, values are written to new arrays (a).



(c) Path Compression. Two sub-trees - violet and green. Before path compression, in order to find out which tree node number 3 belongs to, we look up node number 3's parent which is node number 7. Node number 7's parent is node number 0 and so on until we arrive at the root node, node number 2 which is its own parent and therefore can say node number 3 belongs to tree number 2. After path compression, we can read node number 3's tree directly from its parent.

Fig. 2: Workings of the Partition, Path compression and inclusive scan kernel.

6 Partition and Filter Kernel

Filter Kruskal consists of two important parallel sub routines which will be introduced below.

6.1 Partition

Partition splits an array in two based on a condition. In our case, this condition formulates as "is the value less equal than a given threshold?". Figure 2(b) shows the working of the kernel. This kernel makes use of an inclusive scan which we also implemented. A sketch depicting a parallel inclusive scan[2] can be seen in figure 2(a).

During the partitioning, when creating the mask array a_t, memory access patterns are optimal. Neighbouring threads access neighbouring elements inside the values array and write results to neighbouring elements in a_t. We work with a grid- and blocksize of 256 respectively. With a

[2]The general structure and parts of the code of the scan kernel got taken from Dr. Rupp's lecture in Computational Science on Many-Core Architectures. Our implementation is changed to use only two kernels but therefore only works for gridsize = blocksize.

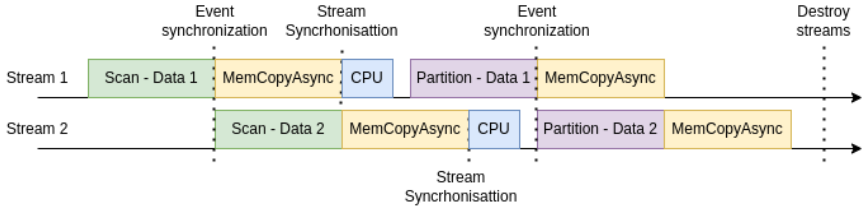


Fig. 3: Visualization of the Partitioning kernel with parallel execution of the kernel and memory copy.

warp-size of 32, that comes to 8 complete warps per block. The elements we are dealing with are integers with a size of 4 bytes each. That means one warp accesses 128 bytes of contiguous memory inside the values and the a_t array. This is an aligned and coalesced memory access pattern that also fits the L1-cache granularity. Irrespective of whether L1-cache is used or not, this also covers the L2-cache granularity of 32 bytes.
However, when writing values from the values array to the final result array a, memory accesses are not guaranteed to be aligned and coalesced anymore. Bus utilization therefore will drop below 100%. In addition to that, in this step we also introduce an if-condition which leads to branching inside the block.

6.1.1 CUDA Streams

To parallelise tasks which work on the separate data chunks or only on the portion of the data, CUDA streams can be used. Each and every CUDA stream can be synchronised independently or together with another streams using events. In our algorithm we perform scan and partition of the data on different data (see Figure 2(b) and 2(a)). This process was parallelised using CUDA streams as presented on Figure 3. To prevent parallel copy of the memory, the streams were synchronised using events.

6.2 Filtering

The Filter Kernel's job is to reduce the workload for the sorting by removing edges that - based on previous building of the MST - cannot occur in the final MST anymore. The kernel works very similar to the partition kernel but instead of creating two final results, only one array of values, that meet a certain condition, is kept.
Another part of the filtering is path compression, see figure 2(c). The filter Kruskal algorithm works by spawning many isolated sub-trees which by time grow together to one final tree. Records are kept of which node belongs to which sub-tree. This is done by giving each node information of their parent node. The first node spawning a sub-tree, the root, gets assigned itself as parent - the sub-tree's ID is the node's ID. So when querying which tree a node belongs to, we walk along the chain of parents until we find a node that is its own parent. This process can be sped up by assigning each node already the root's ID.
As shown, during Filter we have to follow the chain of parent nodes in order to find the tree a node belongs to. Since any node can have any parent, memory accesses will be all over the place. It is not unlikely, that bus utilization will drop to 1/32. What makes this even worse, is that one node could find their root after one iteration, and another node might need 10 iterations to find their root. This introduces a serious warp divergence since finding a parent is a recursive device function. However, we did not find a way to circumvent this issue.

7 Sorting

As described in the Section 5, performance of the sorting algorithm also highly influences the overall performance of the Kruskal algorithm. While CPU based implementation of the algorithm can perform well on small arrays, with the increased number of elements, the performance significantly drops. Considering the optimal, average computational complexity - $O(n \log n)$, such behaviour is expected. However, taking advantage of the highly parallel architecture of the GPU, this process can be speed up. During the development of the GPU-based Kruskal algorithm, following methods were implemented:

- Odd-even sort;
- Merge sort;
- Radix sort.

Those algorithms were compared with the reference solutions from `std` (CPU) and `thrust` (GPU) libraries as a stable baselines. Due to the nature of the Kruskal algorithm, we will sort three vectors based on the values of the weights.

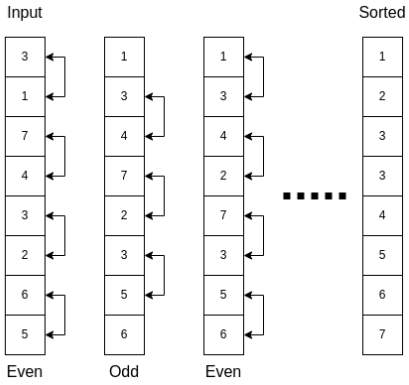
7.1 Odd-even sort

One of the most naive, parallel sorting algorithm is the Odd-even sort. In this approach, on every iteration of the algorithm we compare only two closest elements. On every even iteration two elements with indexes $2k$ and $2k + 1$ (and if it is required we swap) and on every odd iteration with indexes $2k + 1$ and $2k + 2$. This requires in the worst case n iterations, where n is the size of the vector. Visualization of the odd-even sort is presented on Figure 4(a). Every single pair is handled by the separate thread to avoid concurrency.

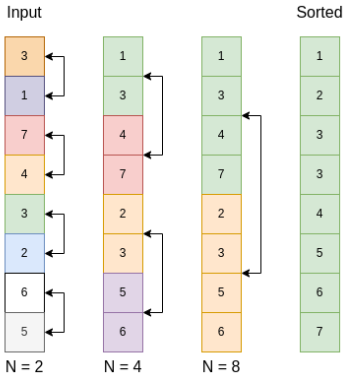
7.2 Merge sort

Merge sort is one of the most popular GPU based sorting algorithms - `Thrust::sort` implementation is based also on it. In this method, sorted sub-arrays are created starting from the length $N = 2$ and with each iteration the length of the sorted sub-arrays is increased by the merge with the closest neighbors. This process is visualized on Figure 4(b). We implemented two different variations of this algorithm:

- When two sub-arrays are merged, every element is handled by the single thread. To determine the position in the resulting sub-array, we add position in the current sub-array and the hypothetical position in the other sub-array. That results in the concurrent memory access pattern.
- When two sub-arrays are merged, both of them are handled by the single thread. Starting from the smallest element in the resulting array, smaller element from the front of the sub-arrays is chosen and placed in the resulting. This takes place until both sub-arrays are empty. That results in much more efficient memory access pattern. However, due to a limited speed of the single thread, last stages of the algorithm where a single thread handles large chunk of data, it is significantly slower than same procedure executed on the CPU.



(a) Odd-even sort. On every pass two closest elements are compared and swapped if required.



(b) Merge sort. On every iteration, sorted sub-array is created by merging two smaller sub-arrays.

Fig. 4: Visualization of the sorting algorithms

7.3 Radix sort

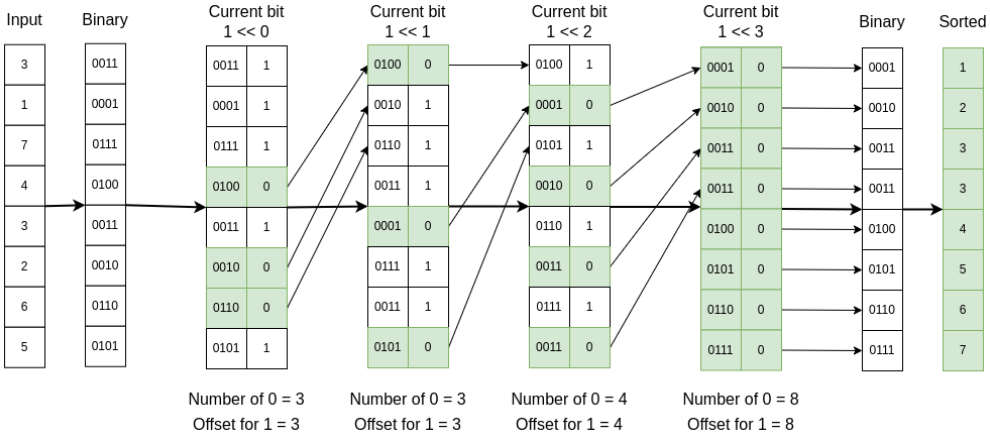


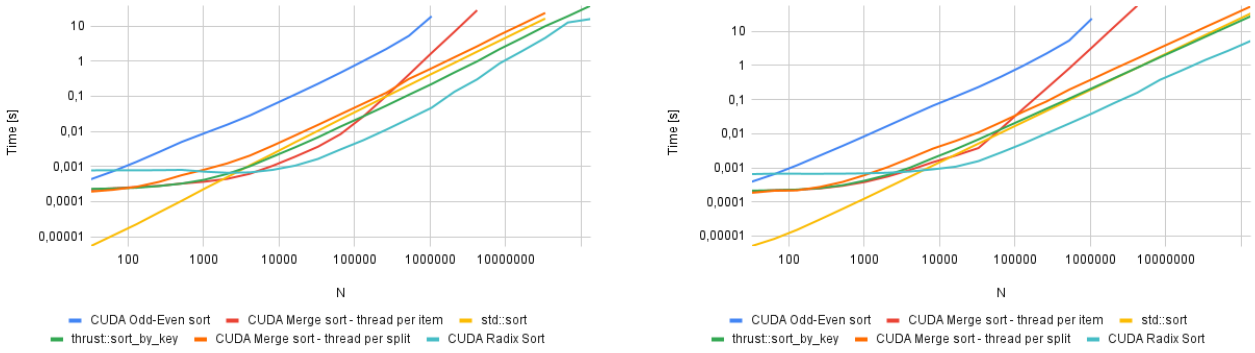
Fig. 5: Visualization of Radix sort. On every iteration, currently analyzed bit is compared (starting from Least Significant Digit) and the one with smallest value are moved in front of the array. Than the rest is moved behind in the same order that was before this iteration

Radix sort is an example of the non-comparative sorting algorithms. Binary representation of the variable is used to determine the final position of the number. Computational complexity of it can be expressed as a $O(\omega n)$, where ω is the length of the binary representation of the number and n is the length of the array. Starting from the Least Significant Digit (LSD), final position of the number is dependent on the case:

- For 0's: Final position is the initial position in the array among all other zeros;
- For 1's: Final position is the initial position in the array among all other ones with an offset equal to a number of zeros in the array.

Implementation of can be simplified to 4 main steps:

1. Calculate position within the data/thread block for number's parity; (partial prefix scan);



(a) Initialized with random values

(b) Initialized with values in a reversed order

Fig. 6: Average sorting time for different algorithms for various length of the array

2. Calculate position within the whole array for number's parity; (full prefix scan);
3. Calculate number of total 0's in the array;
4. Move element to the resulting destination.

In our implementation we used 4-way Radix sort, which takes into account two bits at the time. So instead of (0's and 1's) we add additional two buckets during sorting (3's and 4's) to limit the number of iterations for the kernels.

7.4 Conclusion

On the Figure 6(a) and 6(b) comparison of the average (over 10 runs) run-time for 4 methods implemented by the authors and 2 reference solution are presented. All the global parameters (Number of Threads per Block and Number of Blocks) were optimized using grid search. The results presents performance for the CPU to CPU data for sorting 3 arrays by the values from one of them.

As we can clearly observe, Odd-Even sort performance is by far the worst among all the compared algorithms. The most efficient, for large vectors ($N > 10^4$) was Radix sort implemented by the Authors. However, due to the number of kernels executed in our implementation, the computational overhead makes it not usable for smaller vectors. It outperformed the optimized `Thrust` implementation. What is worth to point out, Merge sort implementation where thread is utilized per item, is more efficient than `thrust`for the vector with size smaller than number of threads - later, it becomes unefficient due to concurrent memory access pattern.

8 Results

8.1 Benchmarks

For benchmarks, very sparse graphs, where each node connects to only 10 other nodes and graphs with certain densities (10%, 50% and 90%) were used. The graph sizes vary in a way so that the largest graph of each group has about $3 \cdot 10^7$ edges. The program was run in different configurations with 10 repetitions and the median run times are being presented here. For very large graphs, the repetitions had to be reduced to 3, because of program crashes, that are likely due to memory leaks.

When comparing a pure CPU configuration with configurations that make use of the GPU, it should be noted, that for small graph sizes, the pure CPU configurations are faster both for the regular and filter Kruskal algorithms. For larger graphs however, making use of the GPU allows for nearly 10-fold speedup (see figure 7).

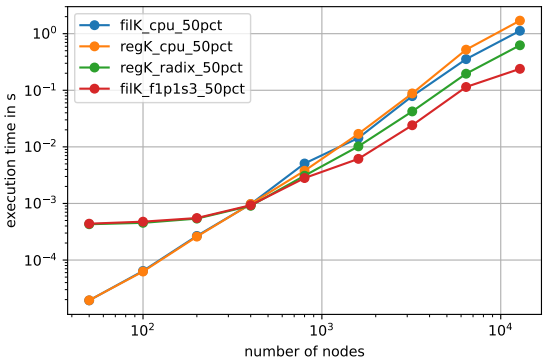
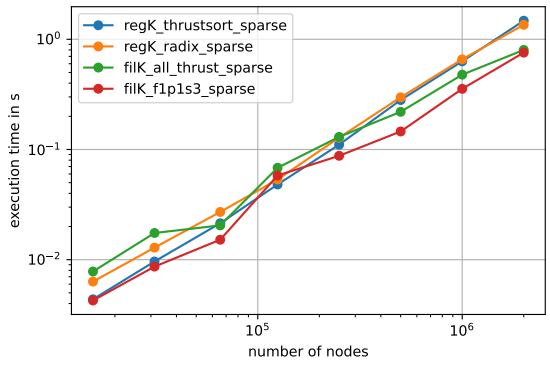
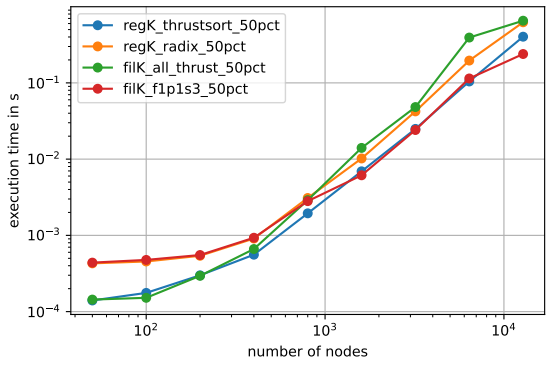


Fig. 7: For large graphs, making use of the GPU results in nearly 10-fold speedup. regK ... regular Kruskal, filK ... filter Kruskal, f1p1s3 ... kernel configuration using custom GPU kernels including radix sort. For complete list of launch arguments refer to Tab. 1 in the Appendix.

Figures 8(a) and 8(b) compare both filter Kruskal with regular Kruskal as well as thrust kernels with full custom kernels on different graph densities. For sparse graphs, while the run times are generally similar, varying at most by a factor of 2, filter Kruskal with custom kernels performed best, while the two configurations using the regular kruskal algorithm were slowest. For the graph with 50% density, the custom kernel filter Kruskal configuration is the fastest once the graph exceeds a certain size.



(a) For sparse graphs, filter Kruskal is faster for all graph sizes.

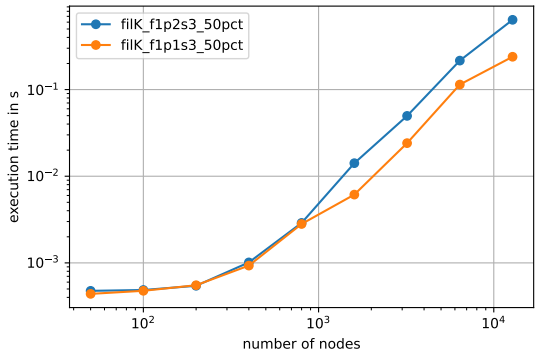


(b) For graphs of density 50%, custom kernel filter Kruskal is preferable once the graph size exceeds a certain size.

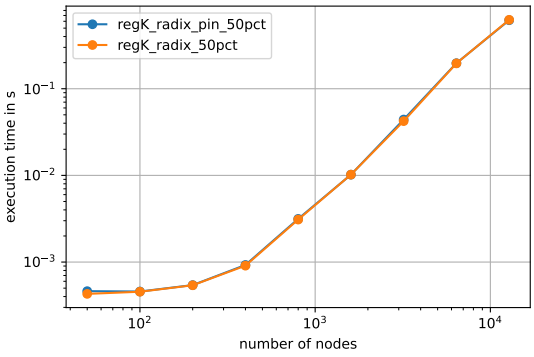
Fig. 8: Comparison of configurations of the Kruskal algorithm on sparse and dense graphs

When comparing the custom kernels, the usage of CUDA streams (See section 6.1.1) was tried for the partition kernel. This did however not result in a speedup. Neither did the use of pinned memory since times for data transfer was not the limiting factor (see figures 9(a) and 9(b)).

Zero-copy memory was not considered in this project, because it simply hides the need to synchronize between host and device. Instead, we came up with our own system to make synchronization between host and device more convenient. For the edgelist class, we implemented two synchronize functions that synchronize all data members in just one call and that also only if the most recent data isn't already on the device / host.



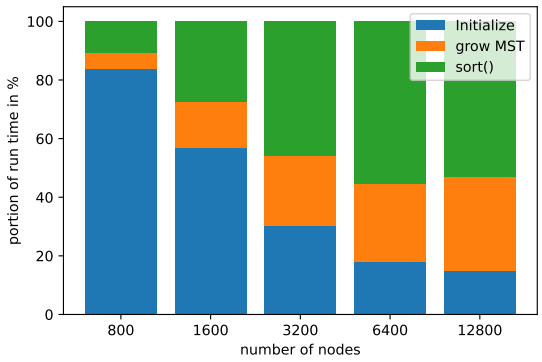
(a) Using CUDA streams (flag -p 2) did not result in a speedup.



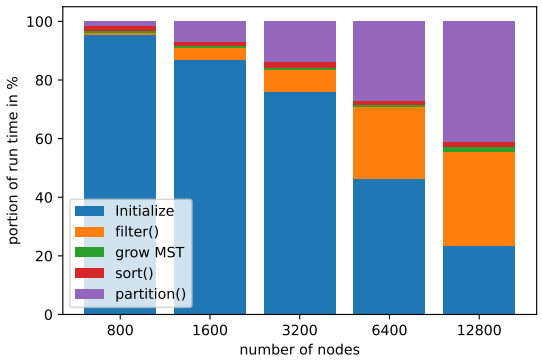
(b) Using pinned memory did not result in a speedup. Showing regular Kruskal with custom kernels.

Fig. 9: Testing CUDA memory transfer optimization features. Graph density: 50%

When comparing the amount of run time spent by the portions of the program, it should be noted, that the regular Kruskal algorithm is dominated by sorting with increasing graph size (see figure 10(a)). The time for sorting is remarkably low for the filter Kruskal algorithm (see figure 10(a)).



(a) Regular Kruskal is dominated by sorting.



(b) Filter Kruskal is dominated by filter and partition.

Fig. 10: Run time spent in different parts of the program. Using Radix Sort. Graph density: 50%

8.2 Discussion

Overall our implementations of regular Kruskal and filter Kruskal achieve similar results. Only for large, sparse graphs ($> 10^7$ edges) filter Kruskal starts to perform better. Unfortunately due to memory problems we were not able to run benchmarks beyond that.

The main advantage of filter Kruskal is that it drastically reduces the runtime of the sorting algorithm. However this advantage requires partition() and filter() to be at least as scalable. While we managed to achieve very good performance for sorting (even beating thrust for large array sizes) the advantage diminished by our implementation of the other kernels. This can clearly be seen in Fig. 10(b) where sort requires almost no runtime, while sorting is still a significant part of the regular Kruskal algorithm. As mentioned in Section 6.2, memory access patterns for the filter are bad. If we could come up with a better way to manage parent nodes, filtering might be accelerated. In the Partition kernel, we currently call a kernel for creating the smaller edgelist and a kernel for creating the larger edgelist. Those two kernel calls might be merged into one single kernel, that would probably save some time as well.

Neither CUDA-streams nor pinned memory showed any significant change in performance. This is most likely due to the fact that there is comparatively very little data transferred between host and device during the calculation. Pinned memory did however have a consistently positive impact on the runtime of our unit tests (see Screenshot 11)

The fact, that pure CPU configurations performed better than GPU configurations up to a certain graph size, could be due to latency phenomena caused by kernel launches and data transfers. Another possible explanation would be that for small problem sizes a high single thread performance could prove more valuable than a high number of threads.

Regarding sorting algorithms, the radix sort kernel was able to outperform the `Thrust` implementation for large problem sizes (see section 7). However the results in the final benchmarks were less conclusive in this matter. For filter Kruskal configurations, this can be explained by the smaller problem sizes, due to multiple calls caused by the divide-and-conquer nature of this algorithm. Additionally the low percentage, that these configurations actually spend on sorting reduces the benefits of an optimized sorting kernel. For regular Kruskal configurations, it was surprising to see, that `Thrust` configurations were slightly faster than radix configurations for dense graphs, while for large sparse graphs, both performed almost equally.

9 Appendix

Table 1: CLI arguments that can be used to launch the program

CLI Arg	value type	Description	Default
–help, -h	n.a.	Produce help message	
–mst-kernel, -m	int	Algorithm used to calculate the MST 0 = regular kruskal 1 = filter kruskal	1
–sort-kernel, -s	int	Kernel used for sorting 0 = GPU Bubble sort 1 = GPU merge sort 2 = GPU Thrust sort 3 = GPU Radix sort 4 = CPU serial sort	3
–partition-kernel, -p	int	Kernel used for the partition step 0 = CPU serial 1 = GPU 2 = GPU with streams 3 = GPU with thrust	2
–filter-kernel, -f	int	Kernel used for the filtering step 0 = CPU naive 1 = GPU 2 = GPU thrust	2
–pinned-memory	n.a	If specified the EdgeList class uses pinned memory	n.a.
–compress-level, -c	int	Specifies how often the UnionFind-Datastructure is compressed 0 = no compression 1 = compress while growing the MST 2 = compress before filtering on CPU 3 = compress before filtering on GPU	1
–recursion-depth, -r	int	Maximum number of times the EdgeList is split before being passed to the regular Kruskal routine	32
–repetitions, -n	int	Number of times the calculation is repeated. Useful for benchmarking as the first run on a GPU usually takes longer.	1
–input-file, -i	string	file containing graph input data. Must be relative to project root folder (Ex2)	
–outputfile_timings, -t	string	output file contianing detailed timing results. Must be relative to project root folder	

```
    Start 1: test_sort
1/8 Test #1: test_sort .........................   Passed    1.21 sec
    Start 2: test_sort_pinned
2/8 Test #2: test_sort_pinned ..................   Passed    1.01 sec
    Start 3: test_partition
3/8 Test #3: test_partition ....................   Passed    0.29 sec
    Start 4: test_partition_pinned
4/8 Test #4: test_partition_pinned .............   Passed    0.28 sec
    Start 5: test_edgeList
5/8 Test #5: test_edgeList .....................   Passed    0.28 sec
    Start 6: test_edgeList_pinned
6/8 Test #6: test_edgeList_pinned ..............   Passed    0.26 sec
    Start 7: test_filter
7/8 Test #7: test_filter .......................   Passed    4.22 sec
    Start 8: test_filter_pinned
8/8 Test #8: test_filter_pinned ................   Passed    4.19 sec
```

Fig. 11: Our unit tests are the only application where pinned memory consistently outperformed regular memory

References

[1] Vitaly Osipov, Peter Sanders, and Johannes Singler. "The filter-kruskal minimum spanning tree algorithm". In: *2009 Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2009, pp. 52–61.