GPU Architectures & Computing

Floyd-Warshall CUDA Implementation

Alexander Sing (01619928)
Patrick Fürst (0927543)
Panagiotis Lantavos-Stratigakis

Algorithm Abstract

The Floyd-Warshall algorithm is an algorithm for the evaluation of the shortest paths between nodes in a graph with positive or negative edges, but not negative cycles. The algorithm iteratively expands the number of vertices checked by one on each iteration, eventually converging on the optimal path for each pair of nodes.

Graph Generation

For graph generation, we implemented two approaches. The first approach creates a random graph with exactly **N** vertices and **N*(N-1) * p** edges, whereas **p** is the defined density. This simple implementation randomly chooses two vertices for a possible edge and adds the edge to the graph if it's not already included. This is repeated until the required density is reached. Since the finite set of possible edges gets smaller the higher the density gets, this implementation can run very long to find a new possible edge for dense graphs.

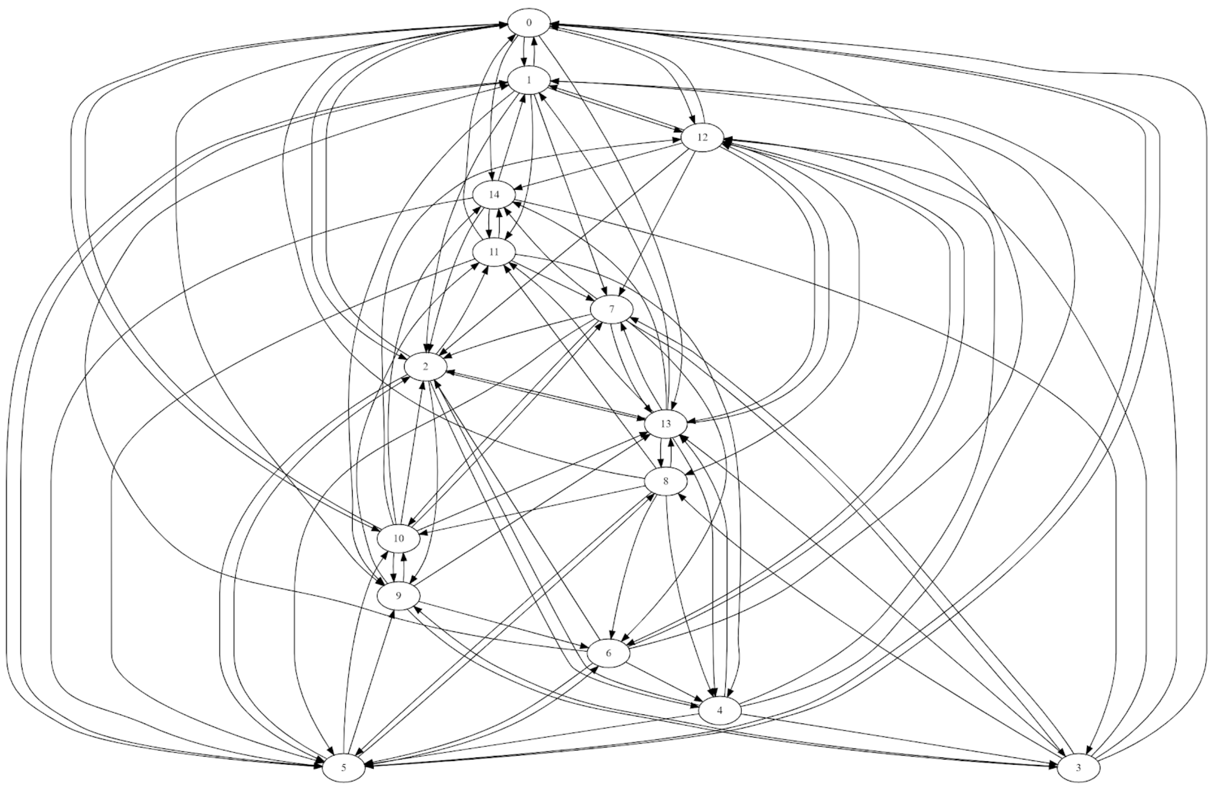
The second approach creates a random graph with exactly **N** vertices and approximately **N*(N-1) * p** edges. This algorithm just iterates over each possible edge and adds it to the graph with a probability of **p**. This way the exact number of edges is not known beforehand, but it can be shown that by the law of large numbers the expected value for the number of edges is **N*(N-1) * p**. This approach is much faster in generating large dense graphs and is the reason we choose to use it. The very small variation in the density can be neglected for performance analysis of the algorithms.

For both approaches, we also make sure to always generate a connected graph, by tracking the remaining vertices to connect.

Internally the graph is represented as an edge map with a weight property but using helper methods it can be converted to an adjacency matrix or a boost graph representation.

All our implementations use the adjacency matrix representation, which can be easily handled by the Floyd-Warshall algorithm and is a good choice for dense graphs, where the Floyd-Warshall algorithm shows its strength.

To inspect the created graphs we added the possibility to write out the graph as a dot file to be visualized with the program GraphViz. An example graph with 15 vertices and a density of 0.5 can be seen below.



Result Verifications

In order to verify the final results of each implementation, we calculate Fletcher's checksum of each results distance matrix. Specifically, we used Fletcher-64 checksum in order to reduce the collisions of the different result matrices. It is also the natural solution when working with distance matrices that are 32 bit integers. The case of collision should be neglectable small in our case, especially with the large graphs. Using the checksum approach allows us to just compare the checksum values in the end and not compare a result with a reference result, which can get very large and consume a lot of memory.

CPU Implementation

We implemented a simple serial version of the algorithm running on a single core on the CPU according to the original paper[1]. Our implementation changes the adjacency matrix in place to contain all the shortest paths in the end. This saves memory and allocation time. The serial implementation serves more as a validator for the correctness of our GPU implementation and a baseline performance result rather than an actually performant CPU implementation, which would include parallelization.

[1] https://dl.acm.org/doi/pdf/10.1145/367766.368168

Additionally, besides our own CPU implementation, we use the ready implementation of the Floyd-Warshall algorithm included in the Boost C++ library for additional result validation. We observed that although Boost is a very well known and widely used library, its implementation of the algorithm was not performant, having worse results than our own embarrassingly non-parallel implementation.

GPU Implementation

Thrust

When parallelizing the algorithm, we can not write the result back into the input matrix, since we don't know the order of how threads access the input data and thus requires to allocate a seconds matrix to save the result, in order to not overwrite the input.

This allows us to look at each cell in the adjacency matrix in parallel, compare the distance of the current cell (i,j) with the intermediate distance [(i,k),(k,j)] and write out the better result into the results matrix. For k, this is done exactly **N** times to cover all neighbours.

This implementation is realized with Thrust **transform** using **counting_iterator**. After each iteration, we swap the **result** and **input** vector to reuse the previous result as input and the other vector to save the next result. This Thrust code can be found at "*apps/floydWarshallThrust/floydWarshallThrust.cu*".

CUDA

For the CUDA implementation, we used Thrust vectors for ease of use in the memory allocation and management and implemented kernels working with both the Thrust vectors as well as raw pointers.

Initially, the adjacency matrix of the graph is copied to the device, and then an iterative kernel is run **k** times, each time checking additional vertices to find the shortest paths. There is no host/device memory interaction in between the kernel runs. Essentially each run represents an iteration of the outer loop of the Floyd-Warshall algorithm, while the two inner loops are parallelized in GPU.

After the algorithm completes we have our second host/device memory interaction, copying the shortest path matrix from the device to the host.

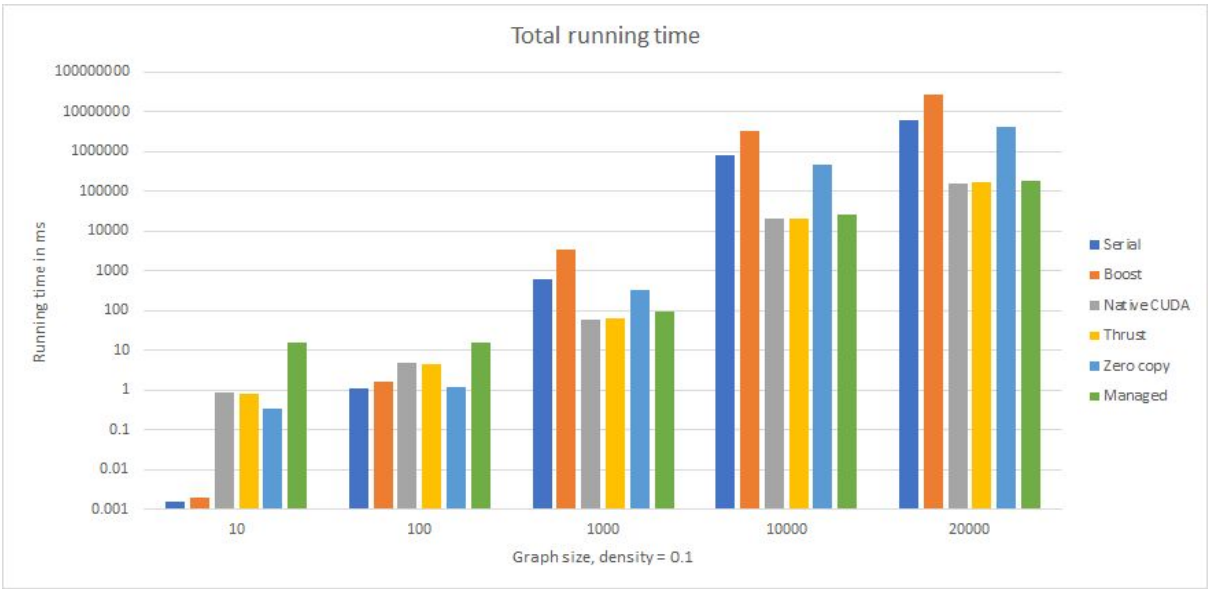
Host-Device Memory Access Patterns

Other than the standard way of simply using the onboard memory of the GPU, we also implemented two versions that make use of zero copy memory and managed memory respectively.

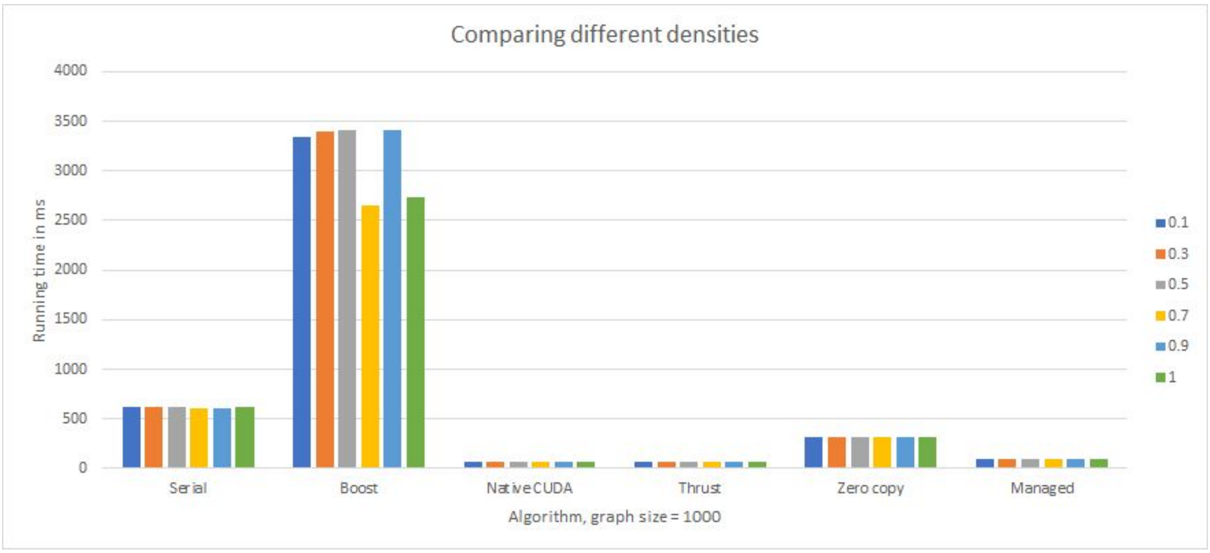
The former does not copy the data from the host memory to the device memory but rather creates a pointer that the device can use as a mean to directly access the data in the system memory. This of course introduces access latency but removes the need to allocate device memory and transfer the data.

The latter leaves the memory management up to the driver which only copies data to the device when it is required there. It provides the convenience of not having to worry about what data actually needs to be copied to the GPU as the driver will take care of it. However, execution is stalled upon data request, thus creating a large overhead for small chunks of data that are only accessed a small amount of times.

Results



The achieved running times reflect what we expected. For small graphs the overhead of allocating, copying to device and afterwards copying back as well as not being able to utilize the parallelization holds all CUDA implementations back. The point of break even is reached at roughly 100 nodes, where, for a short period, the zero copy version is the fastest CUDA implementation and keeps up with our serial version. From then on, the other CUDA versions, our native as well as the thrust version, are by far the best performing ones, being chased by the managed one. For the largest graphs tested, the serial versions are about 100 times slower than the parallel ones.

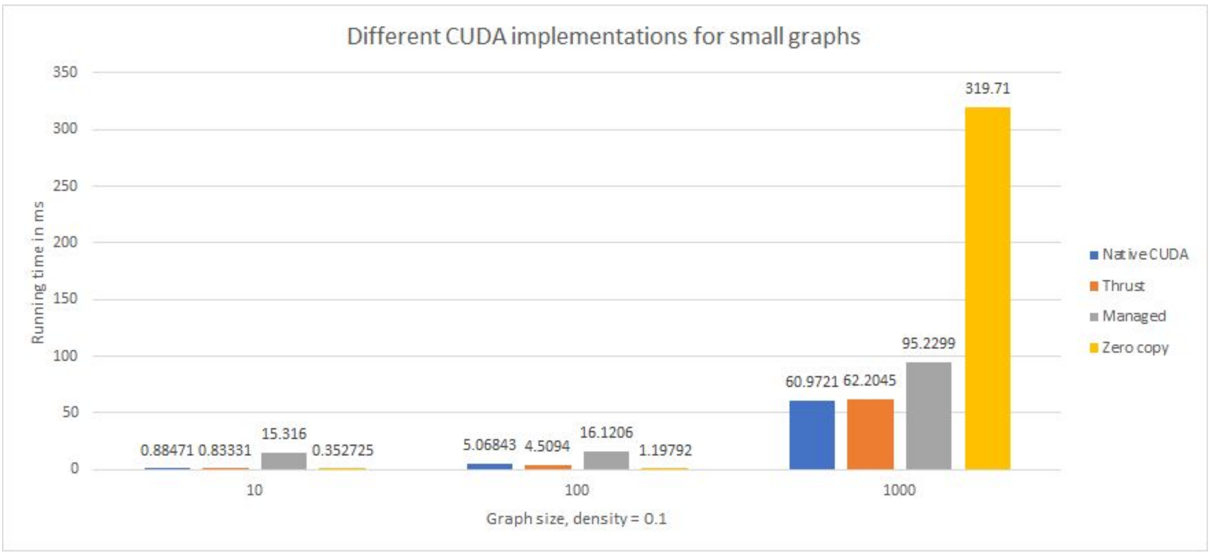


As expected the graph size has a large impact on the running times, whereas the density does not have an impact. This is due to the fact that the algorithm has to check the entire adjacency matrix anyway, no matter how many zeros it contains.

Host-Device Memory Access Patterns

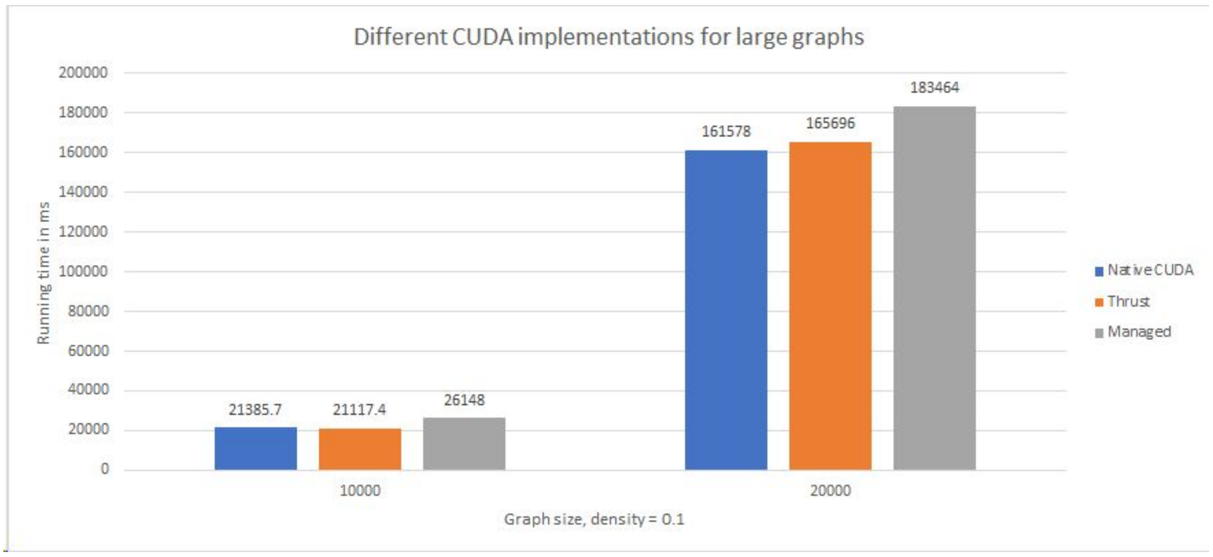
We experimented with different memory patterns, pinned, zero-copy and unified memory for the CUDA implementation, but ultimately due to the nature of the GPU implementation which performs one access at the start and one at the end of the algorithm run, with no intermittent memory operations, they were not particularly beneficial to the total runtime of the algorithm.

For very small graphs (Up to about a thousand nodes), using zero-copy memory is beneficial to the total runtime, as in those cases, the actual computing operations are extremely fast, about 1% of the total time, while most of the runtime is spent on memory copies. In that case, the runtime is improved by using zero-copy memory and allowing the GPU to access host memory directly.

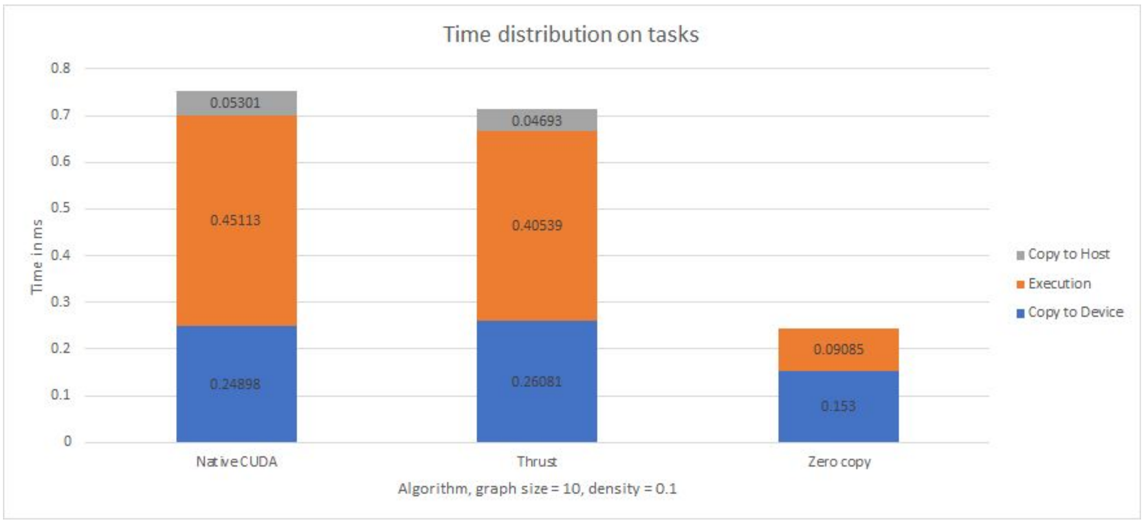


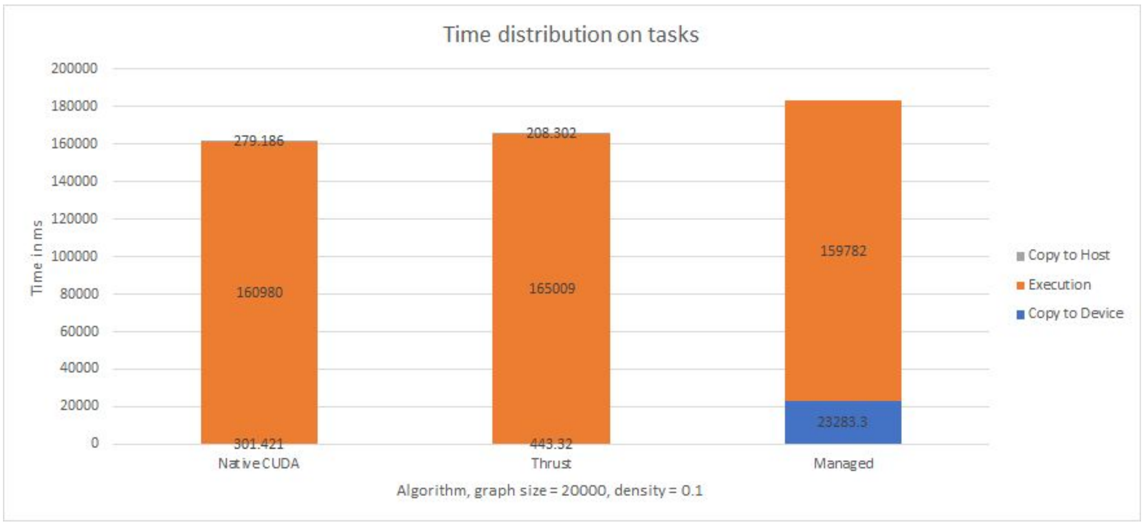
However, for larger graphs, where the cubic complexity of the algorithm overtakes the memory time, this approach lags behind the copy ahead of time approach by a large margin. There, more than 90% of the execution time is spent on operations and during that time, all threads are in full load, so the high latency of accessing the host memory is far more significant.

Unified memory has similar advantages (With the addition of being considerably friendlier to use for the programmer) that scale up to larger graph sizes as the adjacency matrix gets cached in the device memory, ultimately though, for very large graphs, the granted performance improvement is relatively unremarkable.

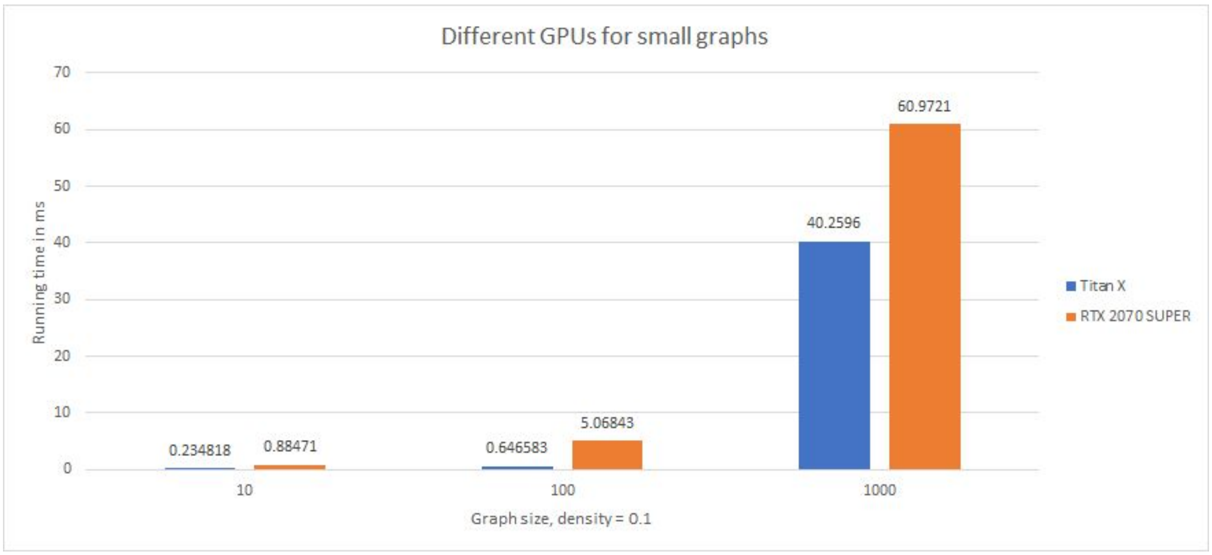


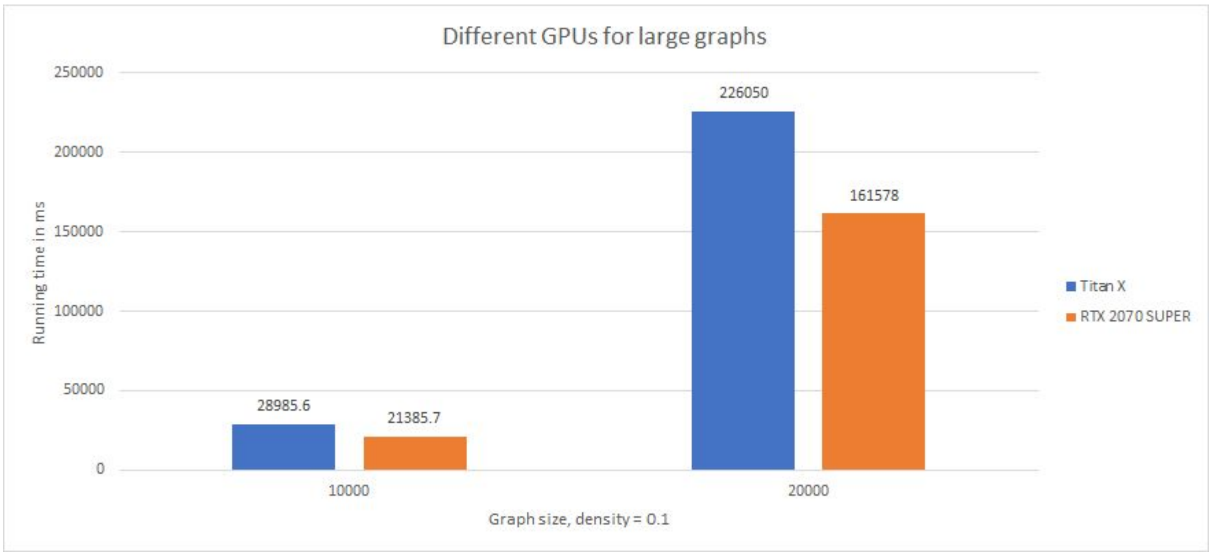
The time spent on copying the data on the one hand and executing the algorithm on the other hand also reflects the expectations and highlights once again the strength of zero copy compared to the other CUDA implementations on small graphs and its weakness on larger graphs.





Finally, we also ran the CUDA versions on different GPUs and the results were rather surprising at first glance. It turns out that for smaller graphs the newer but lower tier RTX 2070 SUPER loses to the older but higher tier Titan X while for the larger two graphs it was the other way around. This could of course be due to more workload on the Titan X, since it is accessed by multiple people, but we don't have the information on that. The graphs show the running times of our native CUDA version on the different GPUs and graph sizes.





Note that all results were obtained by averaging multiple runs. Due to differences in the running times, the number of runs varies. The boost implementation was only run once, because even that was barely feasible time wise. The zero copy version and the Titan runs were averaged over 4 runs, while the other results are averaged over 10 runs.

All other charts besides the last two rely on data collected on the 2070 SUPER.

Running the experiments

How to generate our sample graphs and compile and run the applications is described in detail in the Readme.md file within the repository. Creating graphs with nodes N of up to 20.000 nodes and evaluating the algorithms is taking hours of processing time.