GPU Architectures and Computing VU

Computing the Transitive Closure of a Connected Directed Graph

Fenz Thomas (00702096)

Fuchssteiner David (00904294)

Nichitov Tudor (12045663)

Spitzer Julian (01325893)

Vienna, 19th June 2022

1 Introduction

For many graph problems, it is necessary to verify if there is a path from vertex i to vertex j for any given vertex pair in the graph. The transitive closure of a graph represents a solution to this problem. It is an augmentation of the original graph, where each vertex has a direct edge to every other vertex that it can reach.

Formally speaking, for a directed graph $G = (V, E)$, the transitive closure graph $G' = (V, E')$ extends the set of edges with an edge between all vertices that are not directly connected but reachable via other vertices. An example graph together with its transitive closure is shown in Figure 1.

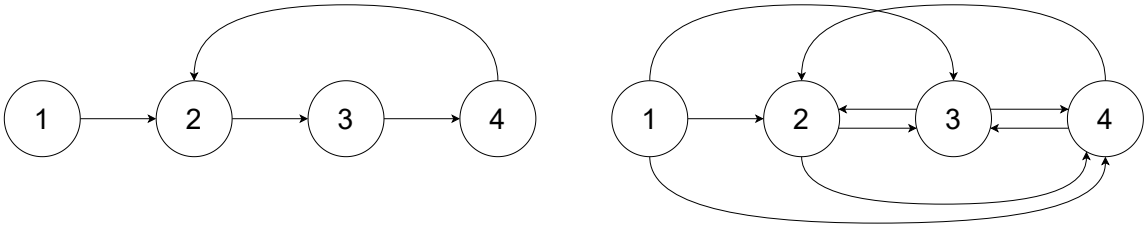


Figure 1: The right graph depicts the transitive closure of the left graph. While the vertices $\{2, 3, 4\}$ are completely connected, vertex 1 is not reachable from the others so it has no incoming edges.

1.1 Graph representation

The input and output representation used for the implementation of this project is based on a simple line-based file format. The first line contains information about the base structure of the graph (number of vertices and edges and whether the graph is directed or not). Then, each consecutive line in the file represents an edge of the graph, specifying the source and the destination vertex and its associated weight.

For this problem, only directed graphs are used and the weight can always be assumed to be 1 as it is only important whether there is a path and not what the shortest path is.

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

Figure 2: The adjacency matrices of the graphs in Figure 1

.

Internally, the graph is represented as an adjacency matrix, which is an $|V| \times |V|$ matrix where each row i contains a 1 in column j if there is an edge from i to j. The adjacency matrices for the graphs in Figure 1 are presented in Figure 2.

2 Approaches

2.1 Matrix Multiplication

A rather simple albeit inefficient approach for solving the problem comes by multiplying the adjacency matrix A with itself up to $|V|$ times. As one can easily see, A^2 gives a matrix where every non-zero $(A^2)_{ij}$ marks a path from vertex i to j in two steps.

When doing a proper matrix multiplication where the original edges are all entered with a weight of 1, the values in these fields would of course be 2 for A^2. As we do not care about the length of the path, the matrix multiplication can be simplified by using logical AND instead of multiplication and logical OR instead of addition. This way, the resulting matrix will only contain $1s$ for reachable nodes and $0s$ for unreachable nodes.

To compute the connectivity matrix A^* it is now necessary to calculate all powers up to $A^{|V|-1}$ as it might take up to $|V|-1$ steps to get from one vertex to another while visiting every other vertex first (e.g. a graph that is a straight path). As $A^{|V|-1}$ only gives the reachable vertices in exactly $|V|-1$ steps, all of the matrices $A, A^2, \dots A^{|V|}$ have to be connected, resulting in (with $|$ being the element-wise OR operation):

$$A^* = A|A^2|A^3|\dots|A^{|V|-1}$$

As each matrix multiplication has a runtime of $\mathcal{O}(|V|^3)$ and it needs to be multiplied up to $|V|-1$ times, the total runtime of this solution is $\mathcal{O}(|V|^4)$, making it rather inefficient. Additionally, as the full adjacency matrix needs to be stored, the required memory is in $\mathcal{O}(|V|^2)$.

2.2 Floyd-Warshall algorithm

The Floyd-Warshall algorithm in its original form is used to compute the shortest path for each vertex pair of a graph. It does so by initializing a $|V| \times |V|$ matrix D where every entry is initialized to ∞. In a 3 time nested loop, these distances are then updated iteratively until the shortest path is found. Each of these iterations performs a simple comparison: If $D_{ik} + D_{kj} < D_{ij}$ (i.e. the shortest known distances from i to k and k to j together are smaller than the shortest known distance from i to j) set $D_{ij} = D_{ik} + D_{kj}$.

This algorithm has a total runtime of $\mathcal{O}(|V|^3)$ while requiring $\mathcal{O}(|V|^2)$ in memory.

2.3 APSP using SSSP

The third approach that we used was presented in [1] which solves the All-Pairs Shortest Path (APSP) problem by applying a Single-Source Shortest Path (SSSP) algorithm $|V|$

times. The SSSP algorithm used for finding the shortest path to all nodes from a single source is Dijkstra's algorithm.

As with the other approaches, a simplified version is used for our purposes because there was no interest in the shortest path but only in the binary information of reachable/unreachable.

Contrary to the other solutions, an adjacency matrix is not needed. Instead, the vector E (size $|E|$) is used for storing all the edges and a vector C (size $|V|$) to store the costs to reach the other vertices. In our case of course we don't store costs but only 0s and 1s to specify whether a vertex is reachable. Additionally there is a mask vector M (size $|V|$) which stores a marking for each vertex if it needs to be checked.

Initially, only the source vertex is set to 1 in C and M. The algorithm then runs until M is empty. In each iteration, all vertices that are marked in M are processed. For each vertex, it is removed from M and all of its outgoing edges are checked and the neighbors entries in C are set to 1. If that was previously 0, it is the first time this vertex is discovered and it gets added to M.

The SSSP algorithm has a time complexity of $\mathcal{O}(|V| \log |V| + |E|)$. As it needs to be run for each vertex, the overall time complexity is $\mathcal{O}(|V|^2 \log |V| + |V||E|))$. Even though the final output after running for all vertices is of size $\mathcal{O}(|V|^2)$, the algorithm itself only uses vectors of size $|V|$ and $|E|$, the space complexity on the GPU is $\mathcal{O}(|V| + |E|)$.

3 Implementations

In total we compare 9 implementations, where 1 implementation targets the CPU and the other 8 implementations target the GPU. Following, we briefly outline all 9 implementations.

3.1 CPU: Sequential Floyd-Warshall

The CPU implementation is a straightforward sequential implementation of the Floyd-Warshall algorithm. It is primarily intended as a baseline to compare with the GPU implementations.

3.2 GPU: Matrix Multiplication

This is our only implementation of the matrix multiplication approach, as it turned out to be quite ineffective and not worth it to pursue further.

In total it uses 4 $|V| \times |V|$ matrices on the device for storing the original adjacency matrix A, a current step matrix A^n, the currently being calculated matrix A^{n+1} and the actual result matrix $A + A^2 + A^3 \ldots$.

The GPU kernel performs the matrix multiplication of a single element and then also adds it to the result matrix. The kernel is launched for each entry in the matrix, so up to $|V|^2$ threads are running in parallel. So in the best case ($|V|^2$ threads running in parallel) the running time can be reduced to $|V|^2$.

Between the individual calls of the kernel it is necessary to copy back A^{n+1} to A^n for the next step to be performed. Additionally, there is a flag which is set if A^{n+1} is different from A^n. If this flag is found cleared after a run, the algorithm can finish early. This can drastically reduce the running time of the algorithm on dense graphs but is quite useless on sparse graphs.

3.3 GPU: Naïve Floyd-Warshall

The first GPU implementation of Floyd-Warshall is the most simple and direct way possible. We created a kernel containing a single iteration of the innermost loop, and repeatedly assign as many threads running this kernel as possible.

Limitations. Since the GPU is designed to efficiently schedule a large number of threads, this approach works reasonably well and typically offers some speedup over the same algorithm on a CPU, but it suffers from global memory bus saturation. It is easy to see that the core algorithm requires 3 words to be sent from global memory to the multiprocessors for the calculation and 1 word to be sent back to global for storage. This is 16 bytes sent across the global memory bus for each task performed.

3.4 GPU: Naïve Floyd-Warshall Pinned Memory

For the pinned memory approach, we simply reuse the naïve Floyd-Warshall implementation. Instead of using pageable memory (as it is the default) we allocate pinned memory on the host side before transferring the data to the GPU. The effect is negligible.

3.5 GPU: Naïve Floyd-Warshall Zero-Copy Memory

For the zero copy approach we again reuse the naïve Floyed-Warshall implementation. Instead of copying the data to the device memory, the kernel directly accesses the host memory. As expected, this results in a drastic performance reduction. The efficiency is clearly limited by the relatively low bandwidth to access host memory. We found using zero-copy memory is absolutely not suited to target our problem.

3.6 GPU: Naïve Floyd-Warshall Pitched

Another variation of the naïve Floyd-Warshall is using pitched memory. With this approach we attempt to avoid non-aligned memory accesses. The key idea here is to shift each row in the adjacency matrix to ensure that each new row starts with a new memory word. This results in a slightly increased memory footprint since for each row we allocate additional unused memory.

Limitations. In the first approach using pitched memory we need additional computations to determine the necessary indices, which results in more computational effort. The advantages of this approach are negated by the negative effects of computing these indices.

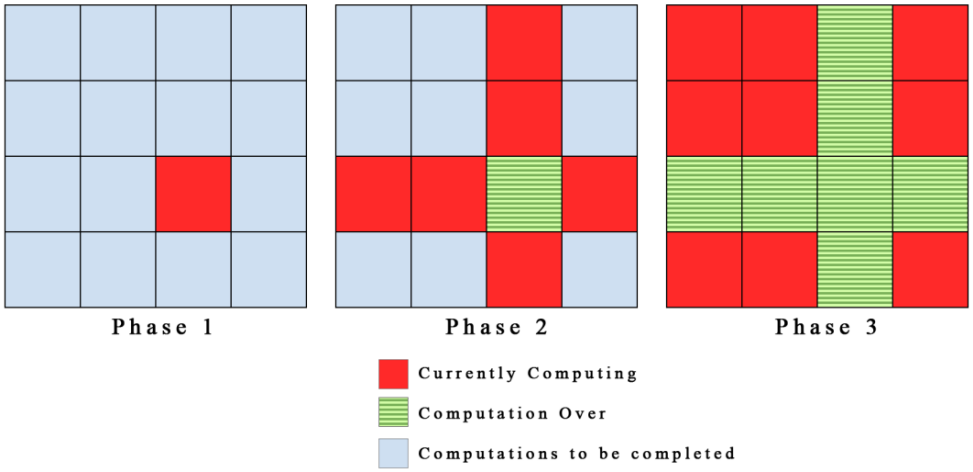
3.7 GPU: Naïve Floyd-Warshall Pitched 2D-Block

This version is a slight adaption of the previous one, where we change the view we have on the adjacency matrix. The main difference is that we define a 2D block instead of a single dimensional one. So, viewing the same problem from a slightly shifted angle, we can reduce the overhead index computations and achieve a slight improvement over the 1d pitched memory approach. The effect compared to the first naïve Floyd-Warshall approach is negligible.

3.8 GPU: Tiled Floyd-Warshall Shared-Memory

In this approach we follow the findings described by Katz et.al. in [3] and slightly refined by Lund et.al. in [2]. To target the limitations of the naïve Floyd-Warshall algorithm regarding global memory bus saturation, we split the base algorithm in 3 phases of computations as

shown in Fig. 3a. For this we split the adjacency matrix into a 2D grid where each block inside has a dimension of 16x16. The number of blocks in each grid dimension results from the size of the adjacency matrix. The computation is done in an iterative process. The size of one grid dimension defines the number of iterations. Each iteration is further divided in three phases described as follows:



(a) **Illustration of Phases in Tiled Floyd-Warshall.** (Fig. originally published in [2])

Phase 1. In the first phase we compute a single primary block. Primary Blocks are those along the main axis of the adjacency matrix. In the first iteration we compute the primary block in the left upper corner. Since we carefully chose the size of the block, the complete primary block of the adjacency matrix fits in the shared memory. Global memory accesses are strictly aligned and coalesced to fully utilize the global bus. The sub-problem can then be solved using the shared memory and is written back to the global memory once it is finished. **Note**: all other blocks are idle in this phase.

Phase 2. In the second phase we compute all secondary blocks. Secondary blocks are column- (respectively row-) aligned blocks. So, blocks which are orthogonal to the primary block. In this phase we update the missing edges dependent on the primary block.

To speedup this process, as in phase 1, we transfer the secondary and the primary block into the shared memory and once the computation is done, we write back the result to the global memory.

Phase 3. In phase 3 we update all other blocks. Each block is updated dependent on the row and column aligned secondary block.

With this shared memory cache efficient GPU implementation we can increase the efficiency compared to the naïve approach by a factor of 5 to 10, which we demonstrate in chapter 4.

Edge Counting. We found out that tracing the edge insertion during the above described computations would have an non-negligible effect on the run time. We target this problem by counting the edges within a separate kernel. To maximize its efficiency we applied a pattern to sum up the complete adjacency matrix. The effect of this improvement is marginal compared to a naïve implementation. Since the sub task of counting edges takes only a tiny fraction of the total process time, a simple naïve approach for summing up the matrix is hardly outperformed.

3.9 GPU: Floyd-Warshall (Thrust)

This implementation makes use of the thrust library for allocating and copying the memory to/from the device. We found thrust very convenient for memory management but unfortunately could not come up with a way of implementing the actual algorithm with it that would not be overly complicated. Because of this, the actual algorithm uses the exact same kernel described in Chapter 3.3.

3.10 GPU: APSP with SSSP (Thrust)

Here the approach presented in Chapter 2.3 is implemented. Again it uses thrust for memory management because it is much more convenient. The Edge vector E is sorted by the source vertices and is implemented as a simple integer list where each entry only declares the destination vertex. There is an additional E_{idx} vector of size $|V|+1$ which for each vertex contains the index of the first entry in E that belongs to it. So given a vertex i, the neighbors of it are stored in E from the indices $E_{idx}[i]$ to $E_{idx}[i+1]-1$.

Contrary to the original implementation presented in [1] only one kernel is required as there are no consistency issues when only storing reachability instead of actual costs, which means that C can be directly updated instead of using an intermediate U vector.

4 Evaluation

4.1 Methodology

The presented results are computed on the provided VM which comprises 8 CPU cores and about 12 GB of main memory. The provided CUDA device is a NVidia Tesla T4 with the Turing Architecture, a compute capability of 7.5, 2560 CUDA Cores and 16GB GPU Memory.

Before we present our findings regarding improvement strategies, we introduce briefly our graph generator. Dependent on the implementation, the properties of the input graph influence the execution time. To evaluate the influence of graph properties like density but also variants in structural patterns, connectivity, etc. we decided to benchmark our implementations on 8 types of graphs. For the extensive evaluations, as shown in the following sections, we conducted over 14k single tests. Each problem instance is tested with 10 samples. In section 4.4 and 4.5 we discuss our results.

4.2 Input Graph Generation

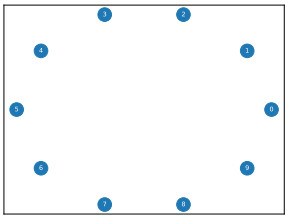
In Fig. 4, we show different graph types used as input for our algorithms. The Python graph generator creates 8 different types of graphs with a variety of type-specific parameters. The simple graph types as shown in Fig. 4a-4d, are used to debug our transitive closure implementations and to test edge cases. The graph types comprising random edge insertion as shown in Fig. 4e-4h are used to benchmark our results on larger examples which is demonstrated in the following sections.

4.3 Correctness of results

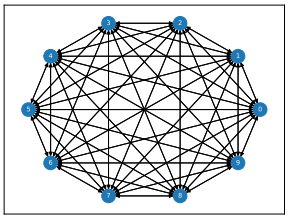
To evaluate the correctness of the computed transitive closure achieved with all implementations, we first manually visualized the output graphs for small examples. For larger instances we compare the output files of the CUDA implementations with the output file using the sequential CPU implementation. To output graphs we implemented a writer, which outputs the graph in the required format. Additionally, the edges are output in a strict order to be able to easily detect differences in the output graph files. Since the transitive closure of certain graphs often results in a complete graph, we skip the file output for large test runs to avoid the bottleneck of persisting data to disk.

4.4 Random Graph with $|V| \in \{100, 200, ..., 1000\}$

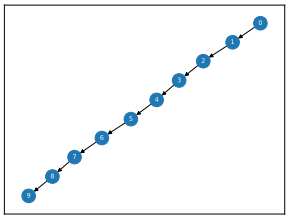
On smaller input graphs, we evaluated all implementations.



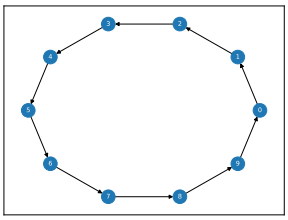
(a) **Empty Graph.**



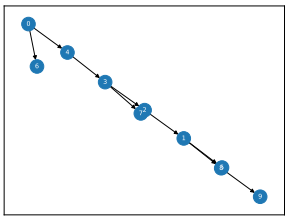
(b) **Complete Graph.**



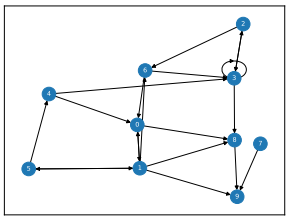
(c) **Path Graph.**



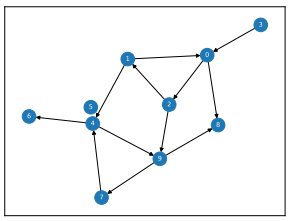
(d) **Ring Graph.**



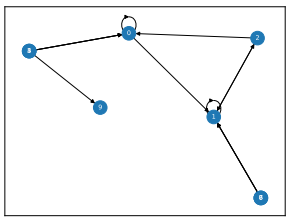
(e) **Random Tree Graph.**



(f) **Custom Random Graph.**



(g) **Erdös Renyí Graph.**



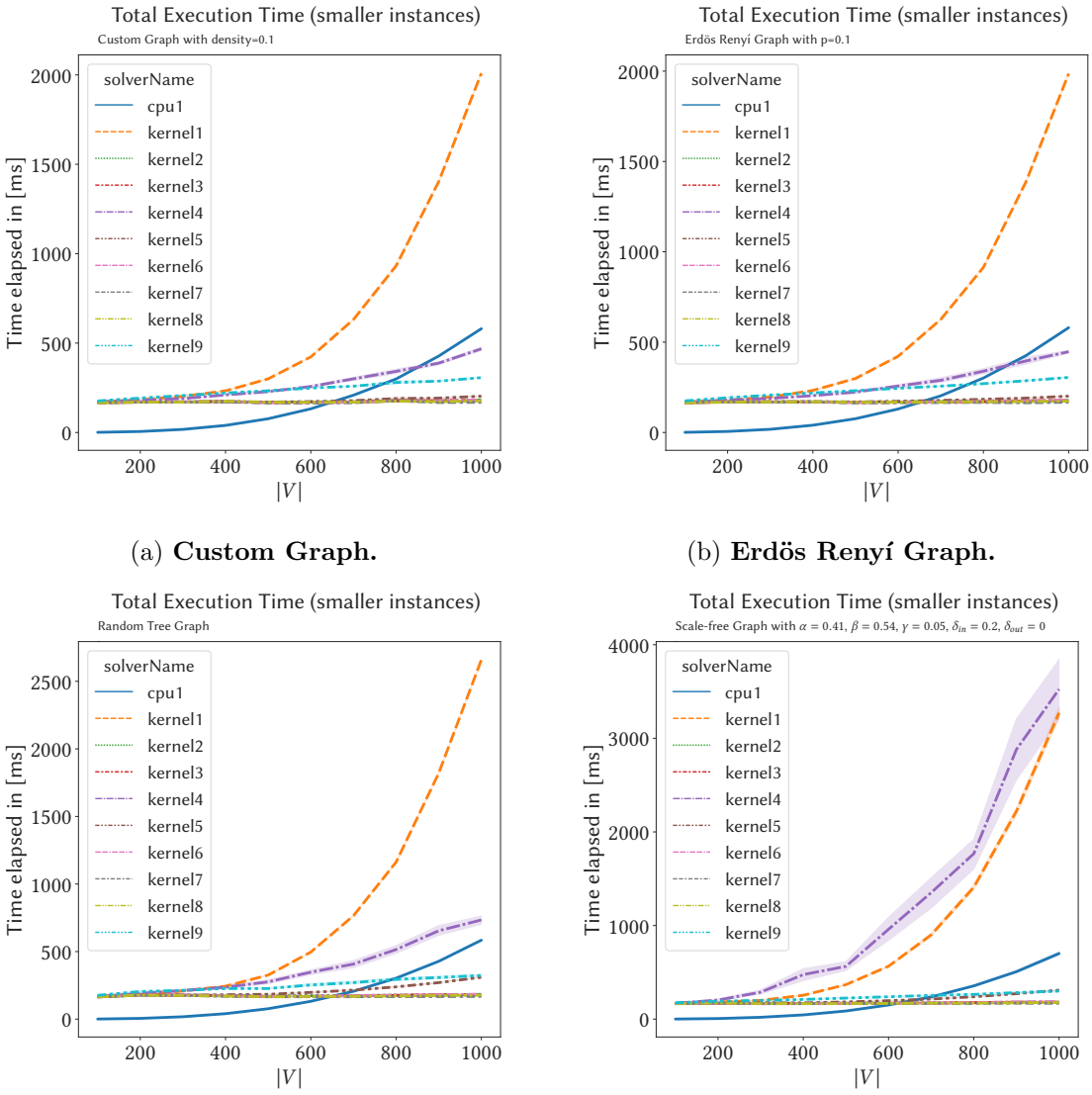
(h) **Scale-free Graph.**

Figure 4: Small instances of each evaluated graph type.

In Fig. 5, we compare all implementations on 4 different random graphs with a size in the range of $[100, 1000]$. Kernel 1, the matrix multiplication approach, performs worst due to its massive memory consumption. Whereas the sequential CPU implementation outperforms all CUDA implementations on very small problem instances. Starting with about 700 nodes the advantage of utilizing GPU compute power can be revealed. Interestingly, using a CUDA device adds a nearly constant amount of execution time independently of problem instance.

In Kernel 4, we utilize zeroCopy memory, which leads to a drastic performance reduction compared to the same algorithm copying the data to the device memory. Kernels 2-3 and 5-9 roughly need the same amount of time for computations on smaller problem instances.

In Fig. 6, we measured the time used for copying the data from the host machine to the CUDA device. The shades in the line plot expose the variance of measurements on 10 different samples, where the line itself is the mean over these 10 values. On the left image, we measured the time for copying the data from Host $\rightarrow$ Device. All measurements are roughly in the same order of magnitude except Kernel 3 (pinned memory), which is



(a) **Custom Graph.**

(b) **Erdös Renyí Graph.**

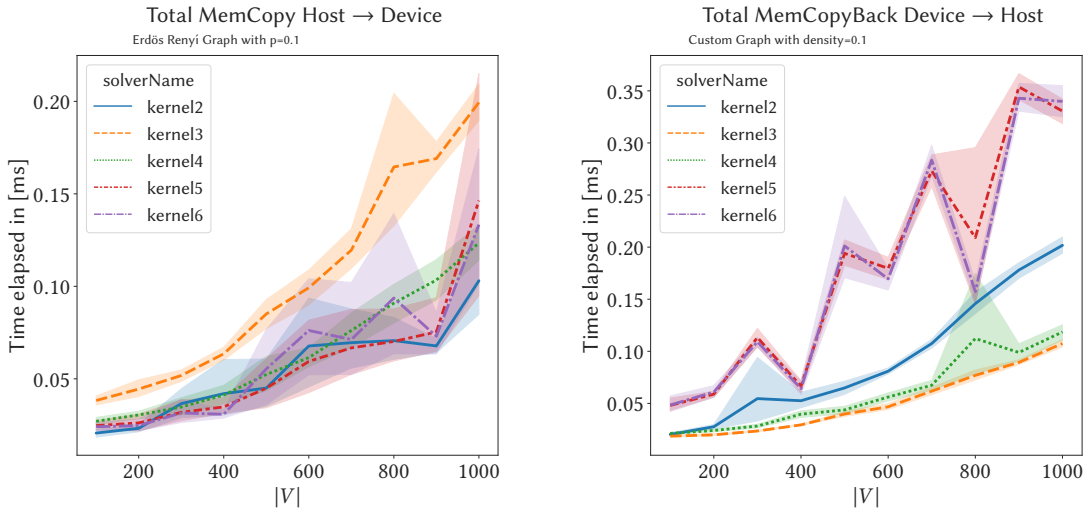
(c) **Random Tree Graph.**

(d) **Scale-free Graph.**

Figure 5: Total execution time on 4 different random graphs (smaller problem instances).

constantly taking more time to transfer the data from the host to the device memory. On the right image we show the cost of copying data back from the device to the host. Here the differences are more clear. Whereas Kernel 3 (pinned memory) and Kernel 4 (zeroCopy memory) need the least amount of time, both implementation using pitched memory (kernel 5 and kernel 6) need the most amount of time. Kernel 2 uses the default strategy of copying memory back and forth.

Note: The differences are in the magnitude of tens to hundreds of μ-*seconds*.

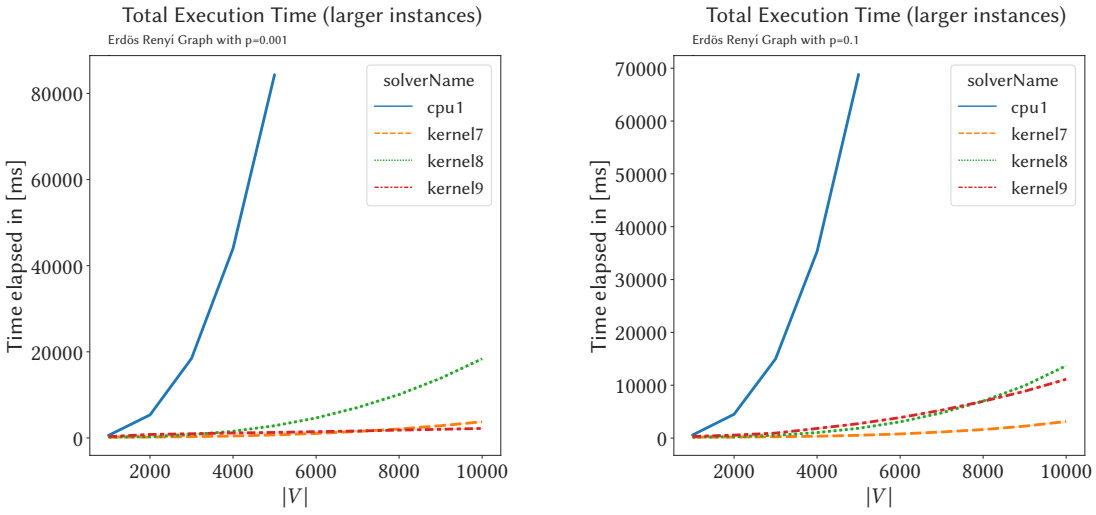


(a) **Host → Device.** (b) **Device → Host.**

Figure 6: Cost of copying memory between Host → Device and vice versa using a variety of copy strategies.

4.5 Random Graph with $|V| \in \{1k, 2k, ..., 10k\}$

On larger input graphs we evaluated only a selection of the more efficient CUDA implementations.



(a) **Erdös Renyí (low density).** (b) **Erdös Renyí (high density).**

Figure 7: CPU vs GPU. Total execution time on low density (left) and high density (right) Erös Renyí Graphs (larger problem instances).

In Fig. 7, we compare the results achieved with the sequential CPU implementation to the faster CUDA kernels. The left image shows the comparison on a Erdös Renyí Graph

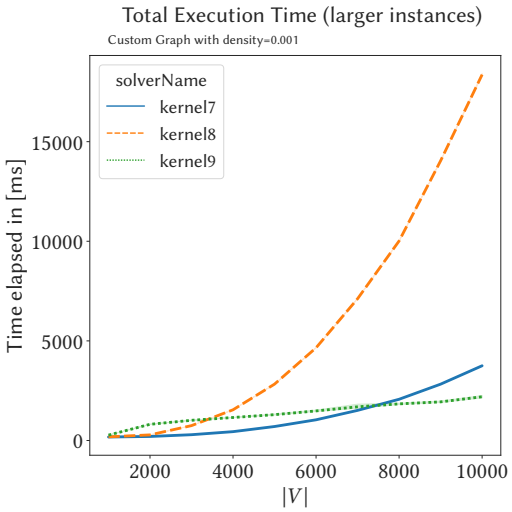
with an edge probability of $p = 0.001$ (Note: edge probability and density of graph is nearly the same). In the right figure, we show the same with an edge probability of $p = 0.1$. In both scenarios we clearly see the potential of GPU power. Our CUDA implementations outperform the sequential implementation by orders of magnitude.

In Fig. 8, we show the results of the fastest CUDA implementations on the larger problem instances (up to $|V| = 10000$). Kernel 7 is the tiled Floyd-Warshall algorithm optimized for massive parallelism. Kernel 8 is the naïve Floyd-Warshall algorithm supported by memory allocations using the thrust library. Kernel 9 is the Dijkstra-based implementation using support of the thrust library. Interestingly, on high density graphs, Kernel 7 is the most efficient implementation as shown in 8a and 8c. Kernel 9 on the other hand clearly outperforms all other implementations on sparse graphs as in 8b, 8d, 8e and 8f. This is due to the fact that contrary to all other algorithms it is the only one that has a time complexity depending on $|E|$. Additionally it has linear space complexity in both $|E|$ and $|V|$, which explains why it outperforms the other algorithms mostly with a high number of vertices and a lower number of edges. Hence, regarding execution times, Kernel 7 but also Kernel 8 are robust against a variation in density of input graphs.

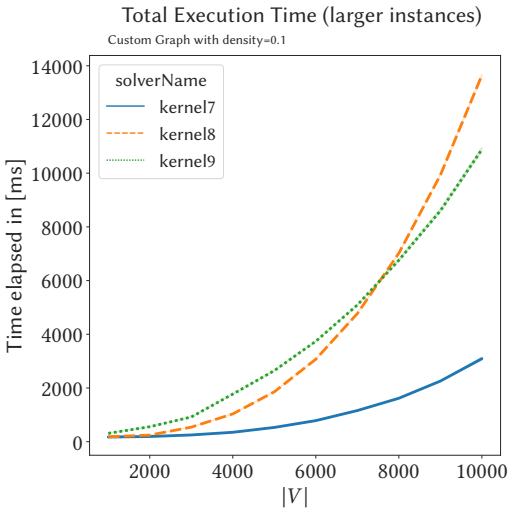
4.6 Reproducibility

All results are reproducible. The graph generation code and the plotting code is implemented using Python 3. We provide a Conda `enironment.yml` file to install all necessary dependencies. The CUDA implementations of the transitive closure can be compiled using make. For convenience reasons, we provide several bash scripts to generate input graphs and bash scripts to run the transitive closure with each implementation/configuration. For details, please review the attached `README.md` file.

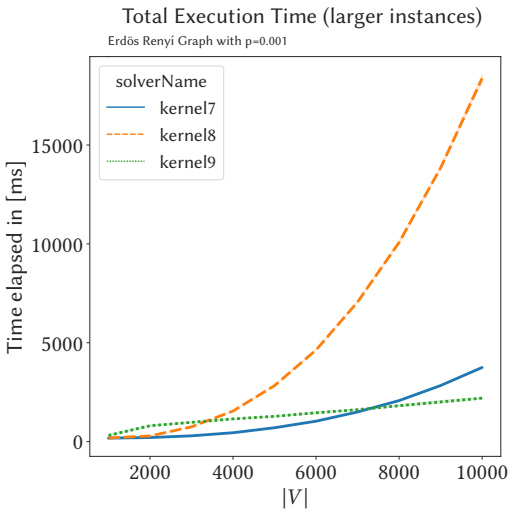
All code including the input graphs and the achieved results stored in a csv file as well as the created evaluation plots are located on the VM in the directory `~/dev/TU-Vienna-GPU-Ass2/`.



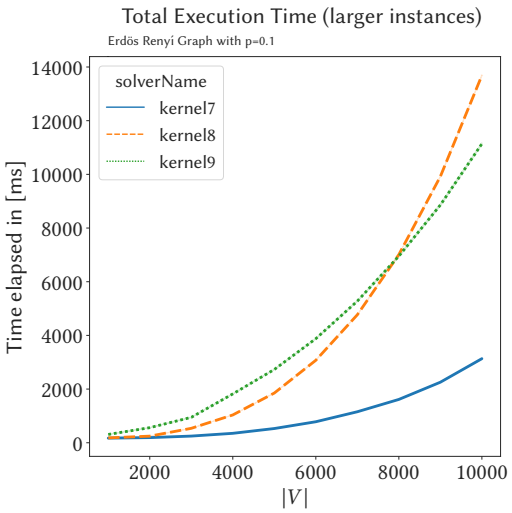
(a) **Custom Graph (den=0.001).**



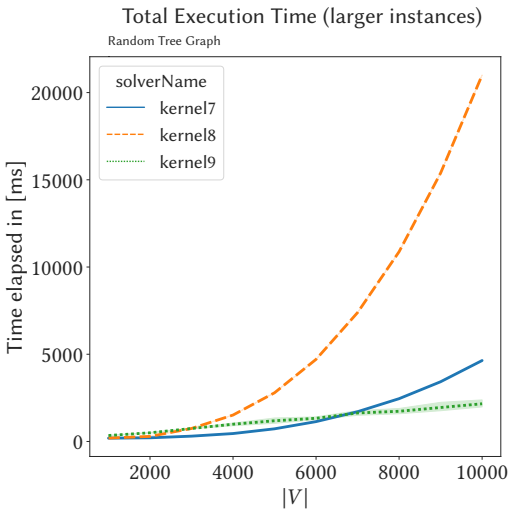
(b) **Custom Graph (den=0.1).**



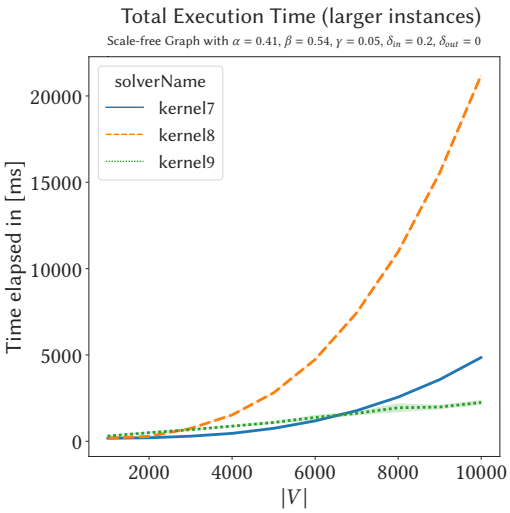
(c) **Erdös Renyí (den=0.001).**



(d) **Erdös Renyí (den=0.1).**



(e) **Random Tree (low density).**



(f) **Scale-free Graph (low density).**

Figure 8: Total execution time of fast CUDA implementations on a variety of graphs with different densities (larger problem instances).

5 Bibliography

[1] Pawan Harish **and** Petter J Narayanan. "Accelerating large graph algorithms on the GPU using CUDA". **in***International conference on high-performance computing*: Springer. 2007, **pages** 197–208.

[2] Gary J. Katz **and** Joseph T. Kider Jr. "All-Pairs Shortest-Paths for Large Graphs on the GPU". **in***Graphics Hardware*: **byeditor**David Luebke **and** John Owens. The Eurographics Association, 2008.

[3] Ben Lund **and** Justin W. Smith. "A Multi-Stage CUDA Kernel for Floyd-Warshall". **in***CoRR*: abs/1001.4108 (2010). arXiv: `1001.4108`.