GPU Architectures and Computing
182.731 (VU 4,0) Semester: 2022S

Project: Find connected components in an undirected graph

June 19, 2022

Student data

Matrikelnummer	Firstname	Lastname
01525189	Christoph	Lehr
11810276	Manuel	Reinsperger
11809637	Lukas	Rysavy

1 Problem statement

The goal of this project is, given an undirected graph, to find a list of connected components in said graph, ie. maximal sets of nodes connected to each other. Input graphs are weighted, but only the fact whether two nodes are connected is relevant for the task, and path costs can be ignored.

2 Input data format

Graphs are supplied in input files with the following format: The first line has the format `H <n> <e> <u>`, with the number of nodes `<n>`, number of edges `<e>` and whether the graph is undirected (`<u>`, always 1). This line is followed by `<e>` lines, each describing an edge `E <s> <d> <w>` with source node `<s>`, destination node `<d>` and weight `<w>` (irrelevant), where nodes are numbered from 0 to `<n>`-1.

3 Building and Running

In each project directory a `Makefile` is located which builds each implementation when executing `make` as a `connected_components` binary. Each implementation has 4 modes of operation:

- `generate`: Generates a graph
  - `filename`: Path where the generated graph shall be stored
  - `nodes`: The number of nodes that shall be in the generated graph
  - `density`: How densely connected the graph shall be, with 0 for no connections and 1 for all nodes connected to all other nodes
  - `min_weight`: Minimum weight of an edge
  - `max_weight`: Maximum weight of an edge
- `bench`: Runs the implementations internal benchmarks
  - `rounds`: How many round shall be executed
  - `nodes`: How many nodes shall be generated for the bench-marking graph
  - `do-checking`: Set to `1` if the results shall be checked to be equal.
- `calculate`: Calculates the connected components with a given algorithm
  - `impl`: Which implementation to run
  - `filename`: Which input file shall be used
  - `-generate`: Indicates a graph shall be generated instead of using an input file
    - `nodes`: The number of nodes that shall be in the generated graph
    - `density`: How densely connected the graph shall be, with 0 for no connections and 1 for all nodes connected to all other nodes
- `evaluate`: Runs the evaluation test suite
  - `folder`: The folder which shall be used as an input for the evaluation

3.1 Implementation Selection

Each algorithm has various implementations to select, with the ids listed below:

- Sparse Representation - one thread per Connected component

  0 CPU implementation

  1 GPU implementation

  2 GPU implementation with vector as output

  3 GPU implementation using pinned memory

  4 GPU implementation using pinned memory with vector as output

  5 GPU implementation using zero-copy memory

  6 GPU implementation using zero-copy memory with vector as output

- Dense Matrix Representation
  - 0 CPU implementation
  - 1 GPU implementation
  - 2 GPU implementation using pinned memory
  - 3 GPU implementation using zero-copy memory
- Thrust: consists only of the thrust implementation

3.2 Evaluation

The files generated to evaluate the algorithms are to large to add them to repository/handin. Therefore, the files are stored inside the `evaluate` folder insider our user home directory.

4 Implementations

Throughout the project, we have developed a few different implementations, all fulfilling the common goal of finding connected components in a graph. These implementations differ in the algorithm used, input and output data formats, and, on a more fine-grained level, the way of transferring memory between the host system and the GPU.

Each subsection of this section will focus on one of these implementations, comparing different variants within their scopes. Section 5 will compare the implementations to each other.

4.1 Sparse graph representation - thread per CC

4.1.1 Input format

In this implementation, input is represented in a format especially suitable for sparse graphs: A list of nodes is saved, each entry containing a pointer to a list of node indices it is directly connected to. These edge lists can be located directly after one another in memory, resulting in one large list and pointers to somewhere in the middle of this list to allow for a very compact structure.

4.1.2 Algorithm

The idea behind this version's algorithm is that every connected component is "owned" by a single thread - the thread with the id of the highest node index in the component. Each thread starts at its own node (the node with index = thread id) and grows its own component by traversing the graph. If at some point a thread realizes the component is owned by another thread (because it encounters a node with an index greater than the start index), it terminates. After a finite amount of steps (each thread traverses its component or exits prematurely), a list of connected components is returned.

Since the number of components is unknown in the beginning (between 1 (all nodes connected) and n (no edges)), the number of threads must be n (the number of nodes) in the beginning, and might shrink to any number ≥ 1. In this extreme case, the algorithm is completely sequential (performed by the one surviving thread).

4.1.3 Variants

Using the same algorithm, we implemented two different versions: In one, each thread gets space for their own component of at most n nodes, where it can write results to. Threads that don't own a component simply set the number of nodes of their component to zero. Of course, this means that $O(n^2)$ bytes are required to store the output ($1 - n$ components with $1 - n$ nodes each). However, this block of memory can be reused by the threads as a local worklist.

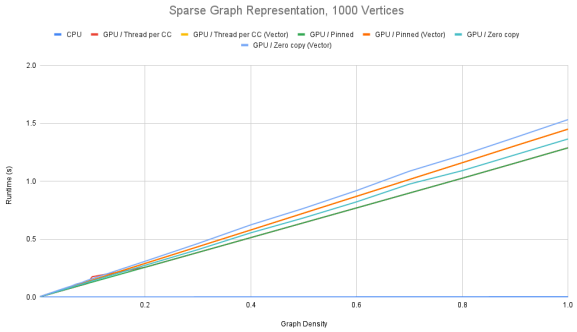
The alternative version stores a single vector with one entry for each node containing the id of the component it belongs to (two indices contain the same id $\Leftrightarrow$ the two nodes belong to the same component). This only needs $O(n)$ bytes of storage for the result, but a worklist of length n is still needed for each thread, negating this advantage somewhat (however, a list of booleans can be used, decreasing storage needed at least by a constant factor). In addition, this approach requires the use of atomic operations (updating the vector with an atomic 'max' operation to ensure the component id is the maximum index of its nodes), which can slow down the process if a lot of conflicting accesses have to be made.

In addition, both of these approaches have been implemented using standard memory transfer, pinned memory, and zero-copy memory.

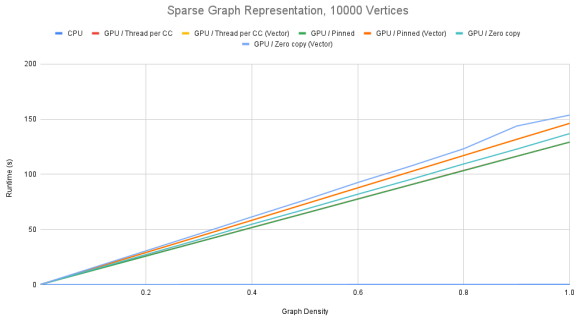
4.1.4 Results

In general, the various implementations running on the GPU are faster than the CPU for low graph densities. As the density increases, however, the CPU version quickly overtakes all of them. Furthermore, the implementation storing its outputs is even faster than the one giving each thread space to store a component for low densities, but even slower for dense graphs. This can be seen in figure 1 for 1000 (1a) and 10,000 (1b) nodes.

When using zero-copy memory, the algorithm takes more time than when device memory is used by a factor proportional to the graph size (mainly number of edges, which is also capped by the number of nodes); this can be



(a) 1000 Nodes



(b) 10000 Nodes

Figure 1: Comparison of algorithms

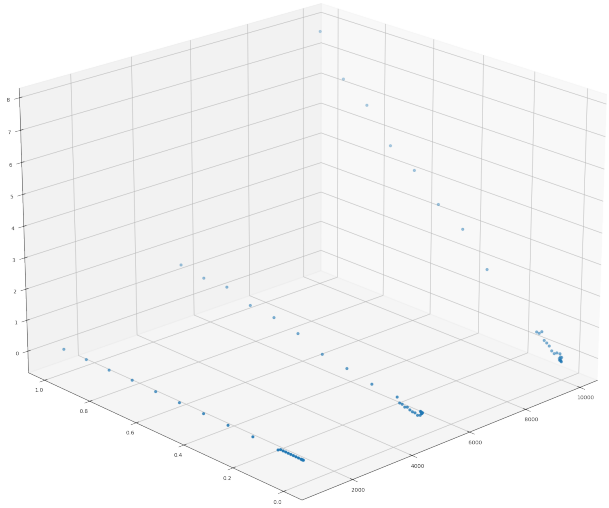


Figure 2: Overhead of zero-copy memory over device memory

seen in figure 2, with the number of nodes and graph density on the horizontal axes (right/left respectively) and the overhead in seconds on the vertical axis.

Using pinned host memory naturally doesn't have any effect on the runtime of the algorithm itself, but it does decrease the time needed for memcpy operations to the device memory. This difference shows no clear pattern related to the graph size as can be seen in figure 3, but is generally higher for larger graphs.

4.1.5 Evaluation

The difference between the performance of the GPU algorithms on sparse and on dense graphs can easily be explained: Dense graphs contain a low number of connected components, while in sparse graphs less nodes are connected and they thus contain more independent components (up to n for a density of 0). If now each component is handled by exactly one thread, the number of effective threads shrinks relatively quickly as the density increases. In the worst case, the graph only contains a single component - here, all of the work is done by a single thread, and the algorithm is therefore completely sequential. Since a CPU thread greatly outperforms a single GPU thread, results are therefore bad in such cases.

The difference between the two output formats (vector and component list) can be explained with atomic operations: While the vector representation is much more compact and therefore fits into caches more easily (making this version faster for low densities), many threads access the same vector elements (nodes) if the graph is dense, leading to collisions of atomic operations and thus slowdowns.

The results on zero-copy memory also match expectations: Memory accesses take longer by a constant factor,

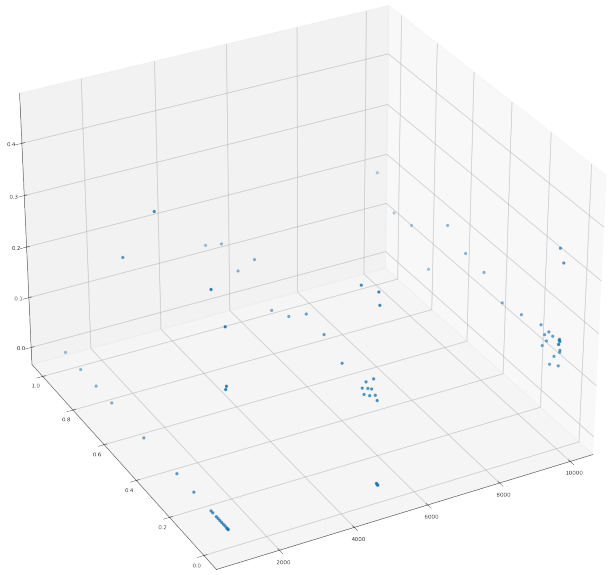


Figure 3: Speedup when using pinned memory

and the amount of memory that needs to be accessed depends on the size of the graph. Therefore, the extra time needed grows proportionally with the graph size, which is exactly what can be observed.

Speedups when pinned memory is used are less consistent, but still match what is expected: The memcpy operation is slower when paging occurs, which happens more frequently for bigger quantities of data. Therefore, speedups are generally higher for larger graphs, because paging can be prevented more often, but there is still a random component involved.

4.2 Dense graph representation

4.2.1 Input format

This implementation uses a dense format for data processing. This means that a full $n * n$ matrix is used, where n is the number of nodes of the graph. It uses a lot more memory than the sparse format for graphs that are not too dense, but allows the algorithm to do column and row based matrix calculations.

4.2.2 Algorithm

This approach takes the given adjacency matrix and iterates over the single rows. The matrix is traversed row by row starting with index 0 or vertex 0 and skips entries which have already been assigned a connected component. When an entry is found, which does not belong to a connected component yet, all vertices which are connected to it are added to an array which holds all found nodes. Next, the algorithm starts iterating over this array and adds all vertices it finds to it. This is repeated until no further vertices are added to the array. Last but not least, each vertex gets assigned a connected component, where the id is the lowest found index.

The algorithm is implemented on the CPU by iterating over each row of the adjacency matrix, but on the GPU this process can be executed in parallel. For each node a thread shall be spawned, therefore a row of the matrix can be evaluated in constant time instead of iterating over n iterations.

4.2.3 Results

In Figure 4 and Figure 5 the run-time measurements of the algorithms using dense matrices are depicted. On the x-axis, the density of the graphs are depicted, and on the y-axis the run-time in seconds. All graphs used for the measurements have 10,000 vertices with various density.

What we can see in Figure 4 is that run-time of the CPU algorithm grows linearly with the density of the graph, in contrast the GPU algorithm with parallel workflow which is mainly dependent on the size of the graph.

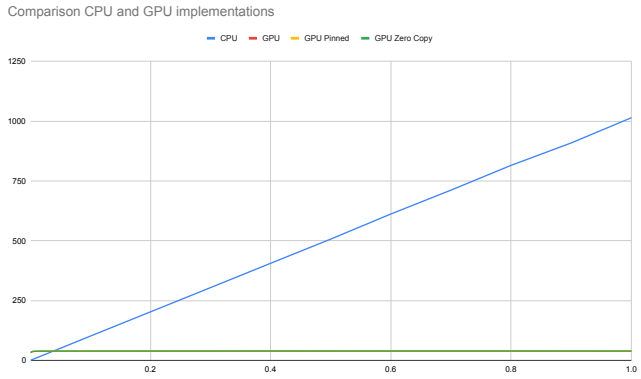


Figure 4: Comparison of dense matrix CPU and GPU implementations

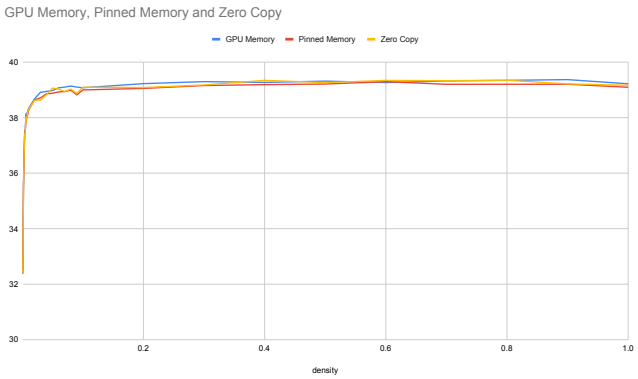


Figure 5: Comparison of memory types

in Figure 5 a comparison of the same algorithm using normal GPU memory, pinned memory and zero-copy memory is shown. For the comparison the adjacency matrix, which is the basis on which the algorithm operates on, was stored in the different memory types. When looking at the diagram of Figure 5, execution times are quite similar. The access to the memory optimised to access one row at a time in parallel by all threads, this is optimizes the memory latency. As the host memory is significantly slower than the device memory, this points out that the limiting factor is the algorithm itself and not the memory interface.

4.2.4 Evaluation

In general this algorithm performs quite bad in sparse graphs, where the CPU profits from less comparisons required to be executed. When the graphs get denser, the GPU implementation takes over the lead, as it has the benefit of being able to execute many comparisons in parallel.

4.3 Thrust

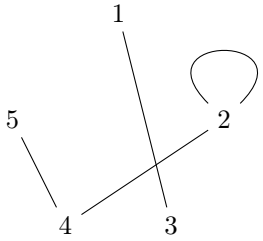
4.3.1 Input format

This implimentation uses the same input format already presented in Section 4.2.1.

4.3.2 Algorithm

In this algorithm the idea was to treat the calculation of connected components as iterative matrix-like fix-point computation. As the weights can be disregarded for this algorithm they are overwritten in the process.

First off a pre-processing step is taken, which sets all non-zero values of the matrix to their respective column number.



1	**2**	**3**	**4**	**5**	↓
0	0	**3**	0	0	
0	**2**	0	**4**	0	
1	0	0	0	0	
0	**2**	0	0	**5**	
0	0	0	**4**	0	

Then the first step of the iterative process is taken, by calculating the maximum value per row. This vector is then pivoted and again applied to the non-zero values of the columns, whereby the value is only overridden if it was smaller.

					→
0	0	3	0	0	**3**
0	2	0	4	0	**4**
1	0	0	0	0	**1**
0	2	0	0	5	**5**
0	0	0	4	0	**4**

This is repeated, until the vector of maximum row values is the same as for the previous iteration, in which case the components have been found. Each position in the resulting vector represents the component id that the node belongs to.

3	**4**	**1**	**5**	**4**	↓
0	0	3	0	0	
0	**4**	0	**5**	0	
3	0	0	0	0	
0	**4**	0	0	5	
0	0	0	**5**	0	

					→
0	0	3	0	0	3
0	4	0	5	0	**5**
3	0	0	0	0	**3**
0	4	0	0	5	5
0	0	0	5	0	**5**

3	**5**	**3**	5	**5**	↓
0	0	3	0	0	
0	**5**	0	5	0	
3	0	0	0	0	
0	**5**	0	0	5	
0	0	0	5	0	

					→
0	0	3	0	0	3
0	5	0	5	0	5
3	0	0	0	0	3
0	5	0	0	5	5
0	0	0	5	0	5

This algorithm has been implemented using thrust transform iterators, gather and transform, which automatically manage the computations as well as the memory accesses. Therefore there is only one variant of this algorithm.

4.3.3 Evaluation

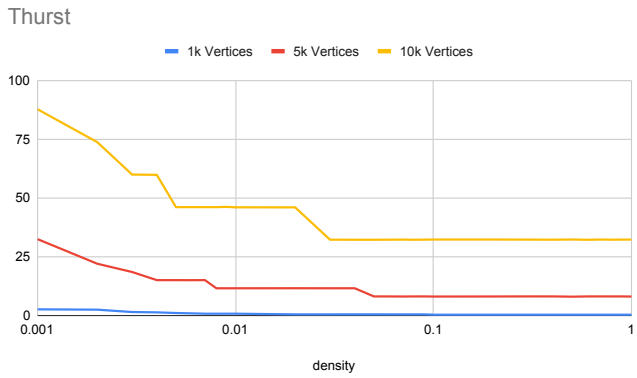


Figure 6: Comparison different node counts

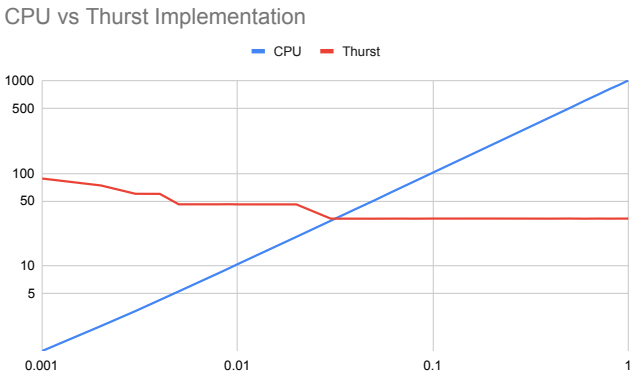


Figure 7: Comparison to CPU algorithm

As can be seen in 6, the performance of the algorithm increases as the density rises. This is due to the fact, that the algorithm needs fewer iterations to find the fix-point, as all connections to a node are checked at once. This is especially noticeable in 7, where it can be seen that the algorithm strongly outperforms the CPU implementation at higher densities.

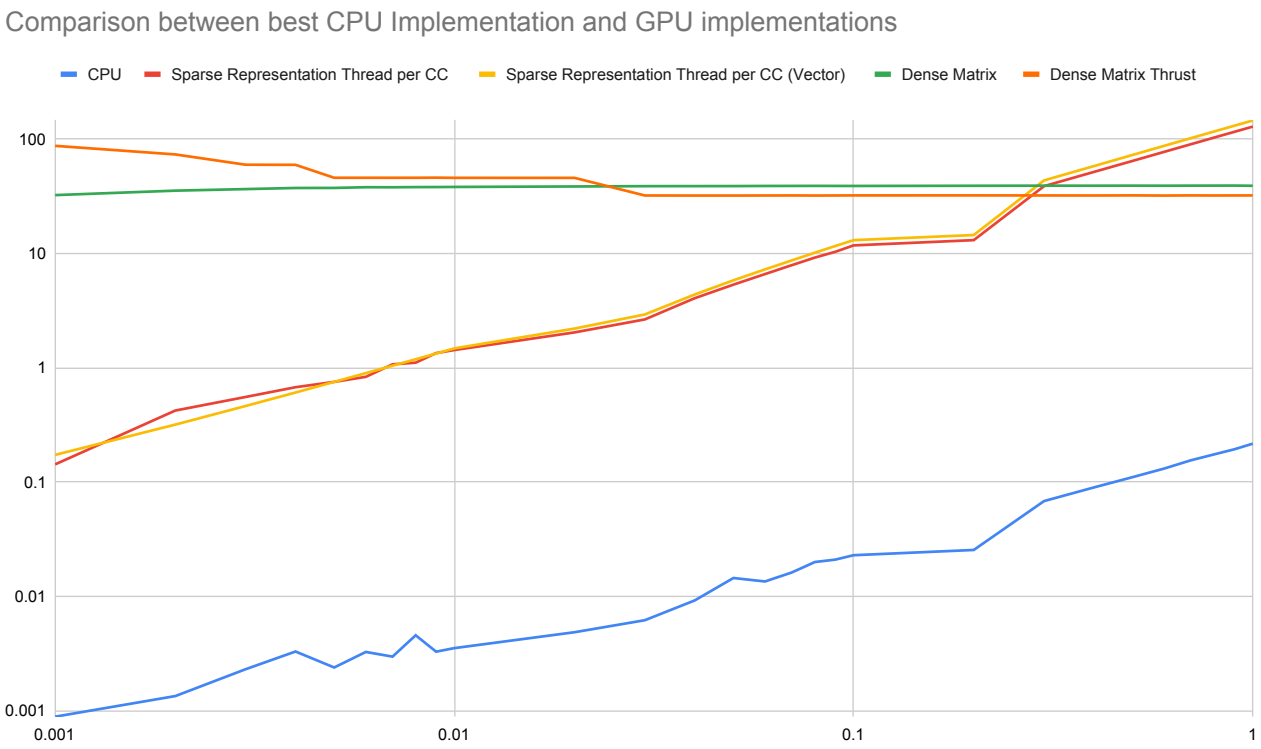


Figure 8: Comparison of the best CPU implementation and the GPU implementations

5 Comparison

In Figure 8 the run-time measurements of the single implementations are depicted. For the GPU implementations, we used the devices memory and each tested graph consists of 10,000. On the x-axis the graph density is depicted in a logarithmic scale. The test points were scattered in a logarithmic fashion to reduce the run-time required for our tests. The y-axis the shows the run-time in seconds.

The graph problem of finding connected components is commonly known to be hard to parallelize and doesn't benefit greatly from adding more threads to solve such a problem. Finding connected components requires a lot of communication between the single threads and optimized data structures are required. The sparse graph representation uses pointers and lists which are beneficial an play out the strengths of the CPU by having a higher clock and being faster when accessing random memory addresses. For less dense graphs this approach can be used on the GPU, but the memory architecture profits from aligned memory access, which hurts the algorithm when considering denser graphs. This can be seen in Figure 8 as the lines for the sparse representation significantly grow at a graph density of 0.2. In contrast when switching to the dense matrix representation, the required memory is aligned and the GPU has optimized access, but it requires a lot more iterations over the data structure. In Figure 8 the dense matrix representation has a fairly constant execution time, mainly dependent on the graph size, but this run-time is way higher in contrast to the CPU. Computing the dense matrix representation via the Thrust algorithm even exhibits performance uplifts with higher densities, as explained in 4.3.3.