Vienna University of Technology

Institute of Computer Engineering

Project Report

CUDA - Single-Source-Shortest Paths

Group2_GPU_CA_SS2020

June 21, 2020

Contents

1 Problem Definition

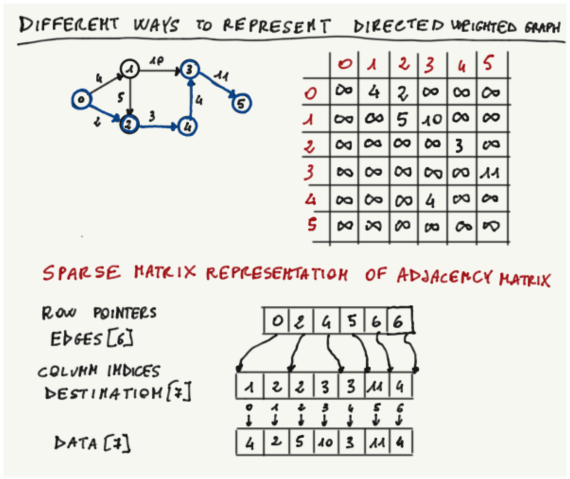
The problem at hand is the implementation of a CUDA-based parallel algorithm for finding the single-source shortest paths in a weighted graph and its comparison to its sequential counterpart. A well-known algorithm to solve this problem was presented by E.W. Dijkstra [2]. Further, a random graph generator has to be implemented, which generates a weakly connected graph with a specific desired density. We explored different memory transfer methods, namely standard memory transfer, pinned memory, zero-copy and also an implementation employing the thrust-library [1].

2 Graph Representation

A commonly used data structure to represent graphs is the adjacency matrix. The adjacency matrix has a fixed size of n^2 nodes, which is inefficient on sparse graphs. Which is why the sparse matrix representation was used for our project. The sparse matrix representation stores the graph using three arrays, the edges, destinations and weights (or data) array. The edges array has the size of n where n is the number of nodes, while destinations and weights have the size of e, where e is the number of edges in the graph. An entry at index i in the edges array references to a location in the destinations array, where the neighbours of the node i are stored. Consequently, a graph with a low density has a smaller memory usage than a graph with the same size but a higher density.

The density of a graph is specific in the range of $[0, 1]$ and is multiplied by the maximum amount of edges possible in the graph with nodes n. The amount of edges is therefore calculated by Equation 1.

$$num\_edges = density * num\_nodes * (num\_nodes - 1) \tag{1}$$


DIFFERENT WAYS TO REPRESENT DIRECTED WEIGHTED GRAPH
0
1
2
3
4
5
4
2
5
10
3
4
11
SPARSE MATRIX REPRESENTATION OF ADJACENCY MATRIX
ROW POINTERS
EDGES [6]
COLUMN INDICES
DESTINATION [7]
DATA [7]
0 1 2 3 4 5 6

	0	1	2	3	4	5
0	∞	4	2	∞	∞	∞
1	∞	∞	5	10	∞	∞
2	∞	∞	∞	∞	3	∞
3	∞	∞	∞	∞	∞	11
4	∞	∞	∞	4	∞	∞
5	∞	∞	∞	∞	∞	∞

EDGES [6]	0	2	4	5	6	6	
DESTINATION [7]	1	2	2	3	3	11	4
DATA [7]	4	2	5	10	3	11	4

Figure 1: *Adjacency matrix and sparse matrix representations*

3 Graph Input/Output

It is desirable to be able to read graphs from files and also output graphs or paths as files. For verification purposes, especially for the resulting paths, a human readable file format was needed. The file format is specified as follows:

```
H <NUMNODES> <NUMEDGES> <UNDIRECTED_BOOLEAN>
E <source_id> <destination_id> <weight>
```

Where H (header) describes how many nodes and edges the graph consists of and E (edge) simply describes the source and destination of an edge and its weight. Hence, the output file of a path is always a subset of the edges of the corresponding path.

4 Algorithm

Harish and Narayanan [3] propose a parallelised version of Dijkstra's algorithm. While we stay close to the original algorithm we can avoid having two kernels by using *atomic* operations. Algorithm 1 shows the pseudo-code of our implementation. Note that *tid* denotes the thread id as well as the node id, while *nid* denotes the id of neighbouring nodes.

Algorithm 1: CUDA kernel for the SSSP algorithm.

```
Data: $edges$, $destinations$, $weights$, $previous\_node$, $mask$, $cost$,
      $nodes\_amount$, $edges\_amount$
Result: $previous\_node$
1  $tid$ = current thread id;
2  if $tid \geq nodes\_amount$ or NOT $mask[tid]$ then
3      return;
4  get edges of the node $tid$;
5  set $mask[tid] = false$;
6  while edge $e(tid, nid)$ in edges of $tid$ do
7      $new\_cost = cost(nid) + weight[e]$;
8      $atomicMin(cost(nid), new\_cost)$;
9      if $cost(nid) == new\_cost$ then
10         $previous\_node(nid) = tid$;
11         $mask[nid] = true$;
```

5 Implementation

We implemented the code using C++11, CUDA 9.0.176 (10.1) and build the program with CMake >3.8 and tested to code on the systems described in Section 6. The program was compiled in *release* mode, as the sequential code is significantly slower in *debug* mode.

In this chapter we describe the implementations using different memory transfer methods. All variations use the same CUDA kernel to perform the SSSP algorithm. The sequential SSSP algorithm works similar to the CUDA kernel, with the difference being the breadth-first search and utilizing a mask array. But before the description of the different SSSP implementations we briefly describe how the graph generation is performed.

Graph Generation We have implemented two different graph generations. Our first implementation runs completely on the CPU and starts by building a tree to achieve weakly connectedness. Afterwards, it adds edges randomly until the desired density is fulfilled. This implementation has one big flaw though, it is very slow due to having to reallocate a lot of memory for temporary vectors. Which is why we decided to implement a second version, which does not guarantee to generate a connected graph, but pre-allocates all needed memory beforehand by randomly assigning how many edges each node will have and then assigns random destinations on the GPU.

Standard Memory Transfer In the Standard Memory Transfer approach, we use the standard *cudaMemcpy* in all instances of memory transfer. Note that there are three instances during which memory transfer is used in this mode:

- Setting up of GPU arrays (copying of nodes, edges and costs to the GPU)
- Copying the mask from the GPU to the CPU (to search for any *true* value)
- Copying the resulting paths from the GPU to the CPU

Pinned Memory Pinned Memory allows the data to be transferred much faster to and from the GPU. That is achieved by allocating the host memory for an array with *cudaMallocHost*. The cost for the faster transfer is a more costly allocation than the standard method (with *malloc*). Therefore, the array that most benefits from our implementation with Pinned Memory is the *mask* array, since it is transferred from the GPU to the CPU in every iteration. This results in a great performance boost.

Zero Copy The Zero Copy approach eliminates the need to use *cudaMemcpy* altogether. With *cudaHostGetDevicePointer*, we map the host-side *mask* address to the device-side *d_mask* address, which guarantees that *mask* is up to date when the kernel is finished. With this modification, the *cudaMemcpy* in every iteration can be replaced by a *cudaDeviceSynchronize*, resulting in much faster execution times than the Standard Memory Transfer.

Thrust The thrust library simplifies the interface to CUDA, i.e. by providing types for vectors on the host *host_vector* and vectors on the device *device_vector* which handle the memory allocation and transfer in the background. The implementation is similar to the standard memory transfer, but we utilised the interface provided by thrust wherever possible.

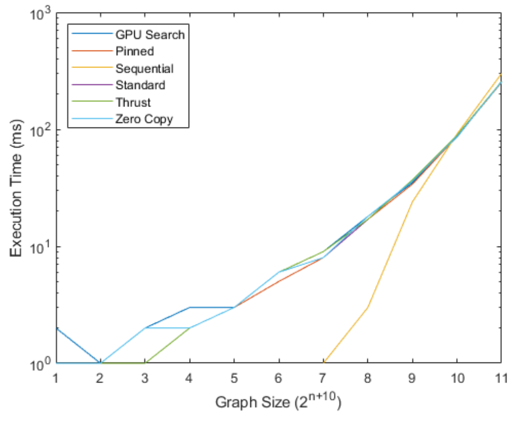
GPU Search The only reason there is any practical need to copy the *mask* array on every iteration is so that a search can be performed on the host (using $std :: find$). To avoid the need to copy the mask altogether, we implemented an additional method that performs a search on the device side.

Finding *true* on the host side is much faster when a *true* value is present in the beginning of the array, as the search terminates early in that case. To avoid searching the whole array on the device side, we start by searching only the first $numBlocks_0 = 1$ block of size $M\_BLOCKSIZE$.

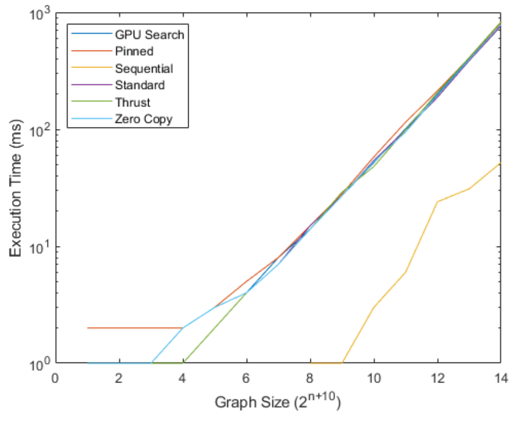
If no *true* value is found, we search the next $numBlocks_i \leftarrow 2 * numBlocks_{i-1}$ blocks and so on until the whole array is traversed. This rule is based on the assumption that, if no *true* values are found in n blocks, the probability that a *true* value is present in the next n blocks decreases exponentially; and to counter for that, we increase n exponentially as well.

System	CPU	RAM	GPU	CUDA Version
Home	i7-6700 4GHz	16GB	GTX1060 6GB	10.1
Institute	i7-5820K 3.3GHz	32GB	GTX TITAN X 12GB	9.0.176

Table 1: *System specifications.*



(a) *Results plotted for the Home system.*



(b) *Results plotted for the Institute system.*

Figure 2: *Results plotted for graphs with* $|N| * 50 = |E|$.

Home, $\lvert E\rvert = \lvert N\rvert * 50$											
#Nodes	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}
GPU Search:	2	1	2	3	3	6	9	18	35	86	257
Pinned:	1	1	2	2	3	5	8	17	34	87	252
Sequential:	0	0	0	0	0	0	1	3	24	91	300
Standard:	1	1	1	2	3	6	8	17	37	88	255
Thrust:	1	1	1	2	3	6	9	17	37	89	256
Zero Copy:	1	1	2	2	3	6	8	18	36	87	254

Institute, $\lvert E\rvert = \lvert N\rvert * 50$														
#Nodes	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}
GPU Search:	1	1	1	2	3	4	8	14	27	52	101	200	399	827
Pinned:	2	2	2	2	3	5	8	15	28	58	115	213	409	817
Sequential:	0	0	0	0	0	0	0	1	1	3	6	24	31	52
Standard:	1	1	1	1	2	4	7	15	27	54	99	188	387	766
Thrust:	1	1	1	1	2	4	7	14	29	48	99	205	412	831
Zero Copy:	1	1	1	2	3	4	7	14	27	53	95	195	395	789

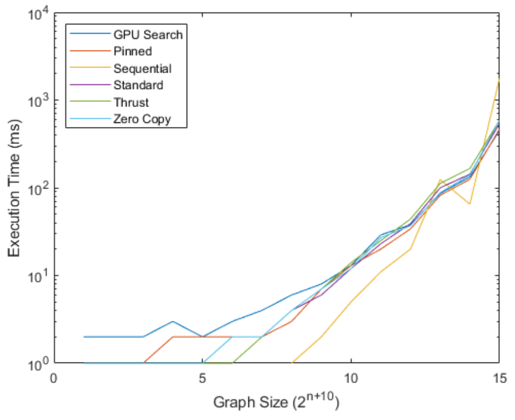
Home, $\lvert E\rvert = \lvert N\rvert * 5$													
#Nodes	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}
GPU Search:	2	2	2	2	2	3	4	6	8	15	28	67	159
Pinned:	1	1	2	1	2	2	3	5	8	14	26	63	157
Sequential:	0	0	0	0	0	0	0	1	2	5	13	104	559
Standard:	1	1	1	1	1	2	3	4	8	15	40	70	193
Thrust:	0	1	1	1	1	1	2	4	8	16	31	79	186
Zero Copy:	1	1	1	1	1	2	3	4	8	14	26	65	199

Institute, $\lvert E\rvert = \lvert N\rvert * 5$															
#Nodes	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
GPU Search:	1	1	1	1	1	2	2	3	6	10	18	35	67	157	298
Pinned:	2	1	2	2	2	2	2	4	6	11	20	37	74	175	342
Sequential:	0	0	0	0	0	0	0	0	1	3	10	17	34	83	162
Standard:	0	0	0	1	1	1	2	3	6	11	21	39	75	163	271
Thrust:	0	0	0	1	1	1	2	4	7	12	24	44	81	177	345
Zero Copy:	1	1	1	1	1	1	2	3	6	11	21	40	77	167	324

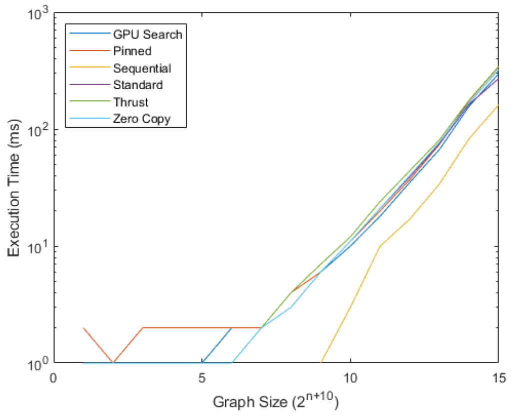
Table 2: *Performance table comparing the results of the different system. The time is measured in milliseconds.*

6 Results & Conclusion

The implementation was tested on two different systems denoted as *Home* and *Institute*, see Table 1 for a short system specification. Table 2 shows the results of our experi-



(a) *Results plotted for the Home system.*



(b) *Results plotted for the Institute system.*

Figure 3: *Results plotted for graphs with* $|N| * 5 = |E|$.

ments. On top of each table, the number of edges relative to the number of nodes is shown. Figure 2 and 3 shows the same results plotted. Lower density also let us use larger graphs for the experiments.

Our experiments suggest that, if the graph is large and sparse, the parallel implementation performs slightly better, while the sequential implementation is considerably faster if the graph is very dense. We conclude that the computation time for the parallel algorithm is slower for dense graphs, as a node with a high degree will slow down the algorithm, as all threads in a block need to wait for that thread for synchronising the mask array. For example if most nodes in a block have n neighbours, but one has m neighbours with $m >>> n$ the computation time is almost the same as if all nodes have m neighbours.

Another problem that arises is that the algorithm can only use the global memory and can not utilise shared memory [3]. Also, we suppose that the frequent random accesses on global memory leads to loss of cache cohesion.

References

[1] Thrust - parallel algorithms library. `https://thrust.github.io/`. Accessed on 06/12/2020.

[2] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[3] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *International conference on high-performance computing*, pages 197–208. Springer, 2007.