TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Computing treewidth on the GPU

Citation for published version (APA):
van der Zanden, T. C., & Bodlaender, H. L. (2018). Computing treewidth on the GPU. In *12th International Symposium on Parameterized and Exact Computation, IPEC 2017* (pp. 1-13). Article 29 (Leibniz International Proceedings in Informatics (LIPIcs); Vol. 89). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.IPEC.2017.29

DOI:
10.4230/LIPIcs.IPEC.2017.29

Document status and date:
Published: 01/02/2018

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 06. Aug. 2024

Computing Treewidth on the GPU*

Tom C. van der Zanden[1] and Hans L. Bodlaender[2]

1 **Department of Computer Science, Utrecht University, Utrecht, The Netherlands**
T.C.vanderZanden@uu.nl

2 **Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven and Department of Computer Science, Utrecht University, Utrecht, The Netherlands**
H.L.Bodlaender@uu.nl

Abstract

We present a parallel algorithm for computing the treewidth of a graph on a GPU. We implement this algorithm in OpenCL, and experimentally evaluate its performance. Our algorithm is based on an $O^*(2^n)$-time algorithm that explores the elimination orderings of the graph using a Held-Karp like dynamic programming approach. We use Bloom filters to detect duplicate solutions.

GPU programming presents unique challenges and constraints, such as constraints on the use of memory and the need to limit branch divergence. We experiment with various optimizations to see if it is possible to work around these issues. We achieve a very large speed up (up to 77×) compared to running the same algorithm on the CPU.

1998 ACM Subject Classification G.2.2 Graph algorithms

Keywords and phrases treewidth, GPU, GPGPU, exact algorithms, graph algorithms, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.IPEC.2017.29

1 Introduction

Treewidth is a well known graph parameter that measures how 'tree-like' a graph is. The fact that many otherwise hard graph problems are linear time solvable on graphs of bounded treewidth [6] has been exploited in many theoretical and practical applications. For such applications, it is important to have efficient algorithms, that given a graph, determine the treewidth and find tree decompositions with optimal (or near-optimal) width.

The interest in practical algorithms to compute treewidth and tree decompositions is also illustrated by the fact that both the PACE 2016 and PACE 2017 challenges [12] included treewidth as one of the two challenge topics. Remarkably, while most tracks in the PACE 2016 challenge attracted several submissions [13], there were no submissions for the call for GPU-based programs for computing treewidth. Current sequential exact algorithms for treewidth are only practical when the treewidth is small (up to 4, see [17]), or when the graph is small (see [16, 4, 26, 14, 25]). As computing treewidth is NP-hard, an exponential growth of the running time is to be expected; unfortunately, the exact FPT algorithms that are known for treewidth are assumed to be impractical; e.g., the algorithm of [3] has a running

* Due to space constraints, several tables in this paper have been abridged or omitted. The complete set of results is presented in the full version of this paper, available on arXiv [23], https://arxiv.org/abs/1709.09990.


© Tom C. van der Zanden and Hans L. Bodlaender;
licensed under Creative Commons License CC-BY

time of $2^{O(k^3)}n$. This creates the need for good parallel algorithms, as parallelism can help to significantly speed up the algorithms, and thus deal with larger graph sizes.

In this paper, we consider a practical parallel exact algorithm to compute the treewidth of a graph and a corresponding tree decomposition. The starting point of our algorithm is a sequential algorithm by Bodlaender et al. [4]. This algorithm exploits a characterization of treewidth in terms of the width of an *elimination ordering*, and gives a dynamic programming algorithm with a structure that is similar to the textbook Held-Karp algorithm for TSP [18].

Prior work on parallel algorithms for treewidth is limited to one paper, by Yuan [25], who implements a branch and bound algorithm for treewidth on a CPU with a (relatively) small number of cores. With the advent of relatively inexpensive consumer GPUs that offer more than an order of magnitude more computational power than their CPU counterparts, it is very interesting to explore how exact and fixed-parameter algorithms can take advantage of the unique capabilities of GPUs. We take a first step in this direction, by exploring how treewidth can be computed on the GPU.

Our algorithm is based on the elimination ordering characterization of treewidth. Given a graph $G = (V, E)$, we may *eliminate* a vertex $v \in V$ from G by removing v and turning its neighborhood into a clique, thus obtaining a new graph. One way to compute treewidth is to find an order in which to eliminate all the vertices of G, such that the maximum degree of each vertex (at the time it is eliminated) is minimized. This formulation is used by e.g. [16] to obtain a (worst-case) $O^*(n!)$-time algorithm. However, it is easy to obtain an $O^*(2^n)$-time algorithm by applying Held-Karp style dynamic programming as first observed by Bodlaender et al. [4]: given a set $S \subseteq V$, eliminating the vertices in S from G will always result in the same intermediate graph, regardless of the order in which the vertices are eliminated (and thus, the order in which we eliminate S only affects the degrees encountered during its elimination). This optimization is used in the algorithms of for instance [15] and [25].

We explore the elimination ordering space in a breadth-first manner. This enables efficient parallelization of the algorithm: during each iteration, a wavefront of states (consisting of the sets of vertices S of size k for which there is a feasible elimination order) is expanded to the wavefront of the next level, with each thread of the GPU taking a set S and considering which candidate vertices of the graph can be added to S. Since multiple threads may end up generating the same state, we then use a bloom filter to detect and remove these duplicates.

To reduce the number of states explored, we experiment with using the minor-min-width heuristic [16], for which we also provide a GPU implementation. Whereas normally this heuristic would be computed by operating on a copy of the graph, we instead compute it using only the original graph and a smaller auxiliary data structure, which may be more suitable for the GPU. We also experiment with several techniques unique to GPU programming, such as using shared/local memory (which can best be likened to the cache of a CPU) and rewriting nested loops into a single loop to attempt to improve parallelism.

We provide an experimental evaluation of our techniques, on a platform equipped with a Intel Core i7-6700 CPU (3.40GHz) with 32GB of RAM (4x8GB DDR4), and an NVIDIA GeForce GTX 1060 with 6GB GDDR5 memory (Manufactured by Gigabyte, Part Number `GV-N1060WF2OC-6GD`). Our algorithm is implemented in OpenCL (and thus highly portable). We achieve a very large speedup compared to running the same algorithm on the CPU.

2 Preliminaries

Treewidth

For a detailed description of treewidth and its characterization, we refer to [11]. Our algorithm is based on the $O(2^n nm)$-time algorithm of Bodlaender et al. [4]. Though the characterization in terms of tree decomposition is more common, we recall only the characterization in terms of elimination orderings that is used by this algorithm:

Let $G = (V, E)$ be a graph with vertices $v_1, \ldots v_n$. An *elimination ordering* is a permutation $\pi : V \to \{1, \ldots, n\}$ of the vertices of G. The *treewidth* of G is defined as $\min_\pi \max_v |Q(\{u \in V \mid \pi(u) < \pi(v)\}, v)|$, where $Q(S, v)$ is the set of vertices $\{u \in V \setminus S \mid$ there is a path $v, p_1, \ldots, p_m, u$ such that $p_1, \ldots, p_m \in S\}$, i.e., $Q(S, v)$ is the subset of vertices of $V \setminus S$ reachable from v by paths whose internal vertices are in S.

An alternative view of this definition is that given a graph G, we can *eliminate* a vertex v by removing it from the graph, and turning its neighborhood into a clique. The treewidth of a graph is at most k, if there exists an elimination order such that all vertices have degree at most k at the time they are eliminated.

GPU Terminology

Parallelism on a GPU is achieved by executing many *threads* in parallel. These threads are grouped into *warps* of 32 threads. The 32 threads that make up a warp do not execute independently: they share the same program counter, and thus must always execute the same "line" of code (thus, if different threads need to execute different branches in the code, this execution is serialized - this phenomenon, called *branch divergence*, should be avoided). The unit that executes a single thread is called a *CUDA core*.

We used a GTX1060 GPU, which is based on the Pascal architecture [20]. The GTX1060 has 1280 CUDA cores, which are distributed over 10 Streaming Multiprocessors (SMs). Each SM thus has 128 CUDA cores, which can execute up to 4 warps of 32 threads simultaneously. However, a larger number of warps may be assigned to an SM, enabling the SM to switch between executing different warps, for instance to hide memory latency.

Each SM has 256KiB[1] of register memory (which is the fastest, but which registers are addressed must be known at compile time, and thus for example dynamically indexing an array stored in register memory is not possible), 96KiB of shared memory (which can be accessed by all threads executing within the thread block) and 48KiB of L1 cache.

Furthermore, we have approximately 6GB of global memory available which can be written to and read from by all threads, but is very slow (though this is partially alleviated by caching and latency hiding). Shared memory can, in the right circumstances, be read and written much faster, but is still significantly slower than register memory. Finally, there is also texture memory (which we do not use) and constant memory (which is a cached section of the global memory) that can be used to store constants that do not change over the kernel's execution (we use constant memory to store the adjacency lists of the graph).

Shared memory resides physically closer to the SM than global memory, and it would thus make sense to call it "local" memory (in contrast to the more remote global memory). Indeed, OpenCL uses this terminology. However, NVIDIA/CUDA confusingly use "local memory" to indicate a portion of the global memory dedicated to a single thread.

[1] A *kibibyte* is 2^{10} bytes.

3 The Algorithm

3.1 Computing Treewidth

Our algorithm works with an iterative deepening approach: for increasing values of k, it repeatedly runs an algorithm that tests whether the graph has treewidth at most k. This means that our algorithm is in practice much more efficient than the worst-case $O^*(2^n)$ behavior shown by [4], since only a small portion of the 2^n possible subsets may be feasible for the target treewidth k. A similar approach (of solving the decision version of the problem for increasing values of k) was also used by Tamaki [22], who refers to it as *positive-instance driven dynamic programming*.

This algorithm lends itself very well to paralellization, since the subsets can be evaluated (mostly) independently in parallel. This comes at the cost of slightly reduced efficiency (in terms of the number of states expanded) compared to a branch and bound approach (e.g. [14, 25, 26]) since the states with treewidth $< k-1$ are expanded more than once. However, even a branch and bound algorithm needs to expand all of the states with treewidth $k-1$ before it can conclude that treewidth k is optimal, so the main advantage of branch and bound is that it can settle on a solution with treewidth k without expanding all such solutions (of width k).

Listing 1 Algorithm for computing treewidth. Note that lines 7–19 compute the degree of v in the graph that remains after eliminating the vertices in S.

```
1  for k:=0 to n-1 do
2    inp:={∅};
3    for i:= 0 to n-k-2 do
4      outp = {};
5      foreach set S in inp do
6        foreach vertex v ∉ S do
7          stack := {};
8          degree := 0;
9          push v to stack;
10         while stack ≠ ∅ do
11           pop vertex u from stack;
12           foreach unvisited neighbor w of u do
13             mark w as visited;
14             if w ∈ S
15               push w to stack;
16             else
17               degree := degree+1;
18           endforeach
19         endwhile
20         if degree ≤ k
21           outp := outp ∪ {S∪{v}};
22       endforeach
23     endforeach
24     inp := outp
25   endfor
26   if inp ≠ ∅
27     report the treewidth of G is k;
28 endfor
```

To test whether the graph has treewidth at most k, we consider subsets $S \subseteq V$ of increasing size, such that the vertices of S can be eliminated in some order without eliminating a vertex of degree $> k$. For each k, the algorithm starts with an input list (that initially contains just

the empty set) and then forms an output list by for each set S in the input list, attempting to add every vertex $v \notin S$ to S, which is feasible only if the degree of v in the graph that remains after eliminating the vertices in S is not too large. This is tested using a depth first search. Then, the input and output lists are swapped and the process is repeated. If after n iterations the output list is not empty, we can conclude that the graph has treewidth at most k. Otherwise, we proceed to test for treewidth $k+1$. Pseudocode for this algorithm is given in Listing 1.

We include three optimizations: first, if $C \subseteq V$ induces a clique, there is an elimination order that ends with the vertices in C [4]. We can thus precompute a maximum clique C, and on line 7 of Lisiting 1, skip any vertices in C. Next, if G has treewidth at most k and there are at least $k+1$ vertex-disjoint paths between vertices u and v, we may add the edge uv to G without increasing its treewidth [10]. Thus, we precompute for each pair of vertices u, v the number of vertex-disjoint paths between them, and when testing whether the graph has treewidth at most k we add edges between all vertices which have at least $k+1$ disjoint paths (note that this has diminishing returns, since in each iteration we can add fewer and fewer edges). Finally, if the graph has treewidth at least k, then the last $k+1$ vertices can be eliminated in any order so we can terminate execution of the algorithm earlier.

We note that our algorithm does not actually compute a tree decomposition or elimination order, but could easily be modified to do so. Currently, the algorithm stores with each (partial) solution one additional integer, which indicates which four vertices were the last to be eliminated. To reconstruct the solution, one could either store a copy of (one in every four of) the output lists on the disk, or repeatedly add the last four vertices to C and rerun the algorithm to obtain the next four vertices (with each iteration taking less time than the previous, since the size of C has increased).

3.2 Duplicate Elimination using Bloom Filters

Each set S may be generated in multiple ways by adding different vertices to subsets $S' \subseteq S$; if we do not detect whether a set S is already in the output list when adding it, we risk the algorithm generating $\Omega(n!)$ sets. To detect whether a set S is already in the output, we use a Bloom filter [2]: Bloom filters are a classical data structure in which an array A of m bits can be used to encode the presence of n elements by means of k hash functions. To insert an element S, we compute k independent hash functions $\{H_i | 1 \leq i \leq k\}$ each of which indicates one position in the array, $A[H_i(S)]$, which should be set to 1. If any of these bits was previously zero, then the element was not yet present in the filter, and otherwise, the probability of a false positive is approximately $(1 - e^{-kn/m})^k$.

In our implementation, we compute two 32-bit hashes $h_1(S), h_2(S)$ using Murmur3 [1], which we then combine linearly to obtain hashes $H_i(S) = h_1(S) + i \cdot h_2(S)$ (which is nearly as good as using k independent hash functions [19]).

In our experiments, we have used $\frac{m}{n} \geq 24$ and $k = 17$ to obtain a low (theoretical) false positive probability of around 1 in 100.000. We note that the possibility of false positives results in a Monte Carlo algorithm (the algorithm may inadvertently decide that the treewidth is higher than it really is). Indeed, given that many millions of states are generated during the search we are guaranteed that the Bloom filter will return some false positives, however, this does not immediately lead to incorrect results: it is still quite unlikely that all of the states leading to an optimal solution are pruned, since there are often multiple feasible elimination orders.

The Bloom filter is very suitable for implementation on a GPU, since our target architecture (and indeed, most GPUs) offers a very fast atomic OR operation [21]. We note that addressing a Bloom filter concurrently may also introduce false negatives if multiple threads

attempt to insert the same element simultaneously. To avoid this, we use the initial hash value to pick one of 65.536 mutexes to synchronize access (this allows most operations to happen wait-free, and only a collision on the initial hash value causes one thread to wait for another).

3.3 Minor-Min-Width

Search algorithms for treewidth are often enhanced with various heuristics and pruning rules to speed up the computation. One very popular choice (used by e.g. [16, 25, 26]) is minor-min-width (MMW) [16] (also known as MMD+(min-d)) [7]). MMW is based on the observation that the minimum degree of a vertex is a lower bound on the treewidth, and that contracting edges (i.e. taking minors) does not increase the treewidth. MMW repeatedly selects a minimum degree vertex, and then contracts it with a neighbor of minimum degree, in an attempt to obtain a minor with large minimum degree (if we encounter a minimum degree that exceeds our target treewidth, we know that we can discard the current state). As a slight improvement to this heuristic, the second smallest vertex degree is also a lower bound on the treewidth [7].

Given a subset $S \subseteq G$, we would like to compute the treewidth of the graphs that remains after eliminating S from G. The most straightforward method is to explicitly create a copy of G, eliminate the vertices of S, and then repeatedly perform the contraction as described above. However, storing e.g. an adjacency list representation of these intermediate graphs would exceed the available shared memory and size of the caches. As we would like to avoid transferring large amounts of data to and from global memory, we implemented a method to compute MMW without explicitly storing the intermediate graphs.

Our algorithm tracks the current degrees of the vertices (which, conveniently, we already have computed to determine which vertices can be eliminated). It is thus easy to select a minimum degree vertex v. Since we do not know what vertices it is adjacent to (in the intermediate graph), we must select a minimum degree neighbor by using a depth-first search, similarly to how we compute the vertex degrees in Listing 1. Once we have found a minimum degree neighbor u, we run a second dept-first search to compute the number of neighbors u has in common with v, allowing us to update the degree of v. To keep track of which vertices have been contracted, we use a disjoint set data structure.

The disjoint set structure and list of vertex degrees together use only two bytes per vertex (for a graph of up to 256 vertices), thus, they fit our memory constraints whereas an adjacency matrix or adjacency list (for dense graphs, noting that the graphs in question can quickly become dense as vertices are eliminated) would readily exceed it.

4 Experiments

4.1 Instances

We selected a number of instances from the PACE 2016 dataset [12] and libtw [24].

All instances were preprocessed using the preprocessing rules of our PACE submission [8], which split the graph using *safe* separators: we first split the graph into its connected components, then split on articulation points, then on articulation pairs (making the remaining components 3-connected) and finally - if we can establish that this is safe - on articulation triplets (resulting in the 4-connected components of the graph). We then furthermore try to detect (almost) clique separators in the graph, and split on those. For a more detailed treatment of these preprocessing rules, we refer to [5].

Table 1 Performance of the algorithm on several benchmark graphs, using global memory and a work size of 128.

Name	$\|V\|$	tw	Time (sec.)		Exp
			GPU	CPU	
1e0b_graph	55	24	779	-	1730 $\times 10^6$
1fjl_graph*	57	26	1730	-	3680 $\times 10^6$
1igd_graph	59	25	107	5120	261 $\times 10^6$
1ubq*	47	~~11~~	1130	-	2300 $\times 10^6$
8x6_torusGrid*	48	~~7~~	1110	-	2100 $\times 10^6$
BN_98	47	21	689	-	1590 $\times 10^6$
contiki_dhcpc_handle_dhcp*	39	6	1490	-	2930 $\times 10^6$
DoubleStarSnark	30	6	34,5	873	87,6 $\times 10^6$
KneserGraph_8_3*	56	~~24~~	1710	-	4130 $\times 10^6$
myciel5*	47	19	2000	70.600	4000 $\times 10^6$
NonisotropicUnitaryPolarGraph_3_3	63	53	1,16	60,4	1,56 $\times 10^6$
queen8_8	64	45	26,3	2040	57,9 $\times 10^6$
RandomBarabasiAlbert_100_2*	41	12	1610	-	3280 $\times 10^6$
RandomBoundedToleranceGraph_60	59	30	0,274	0,635	0,0560 $\times 10^6$
SylvesterGraph	36	15	248	-	632 $\times 10^6$
te*	62	~~7~~	1170	-	2160 $\times 10^6$

4.2 General Benchmark

We first present an experimental evaluation of our algorithm (without using MMW) on a set of benchmark graphs. Table 1 shows the number of vertices, computed treewidth, time taken (in seconds) on the GPU and the number of sets S explored. Note that the time does not include the time taken for preprocessing, and that the vertex count is that of the preprocessed graph (and thus, the original graph may have been larger).

The size of the input and output lists were limited by the memory available on our GPU. With the current configuration (limited to graphs of at most 64 vertices - though the code is written to be flexible and can easily be changed to support up to 256 vertices), these lists could hold at most 180 million states (i.e., subsets $S \subseteq V$ that have a feasible partial elimination order) each. If at any iteration this number was exceeded, the excess states were discarded. The algorithm was allowed to continue execution for the current treewidth k, but was terminated when trying the next higher treewidth (since we might have discarded a state that would have lead to a solution with treewidth k, the answer would no longer be exact). The states were the capacity of the lists was exceed are marked with *, if the algorithm was terminated then the treewidth is stricken through (and represents the candidate value for treewidth at which the algorithm was terminated, and *not* the treewidth of the graph, which is likely higher).

For instance, for graph 1ubq the capacity of the lists was first exceeded at treewidth 10, and the algorithm was terminated at treewidth 11 (and thus the actual treewidth is at least 10, but likely higher). For graph myciel5, the capacity of the lists was first exceeded at treewidth 19, but still (despite discarding some states) a solution of treewidth 19 was nevertheless found (which we thus know is the exact treewidth).

For several graphs (those where the GPU version of the algorithm took at most 5 minutes), we also benchmarked a sequential version of the same algorithm on the CPU. In some cases,

Table 2 Running time (sec.) for various work group sizes (W), using shared (S) or global (G) memory. Each cell lists the average result of 4 test runs, where the complete set of runs was executed in a randomized order.

Name	$\|V\|$	tw	Time (sec.)			
			$W = 32$	$W = 64$	$W = 128$	$W = 256$
1igd_graph (G)	59	25	109	107	107	107
1igd_graph (S)	59	25	94,8	95,6	98,2	103
1ku3_graph (G)	60	22	238	235	235	235
1ku3_graph (S)	60	22	214	217	222	230
queen8_8 (G)	64	45	29,5	26,6	26,3	26,0
queen8_8 (S)	64	45	25,1	24,1	24,5	25,0

the algorithm achieves a very large speedup compared to the CPU version (up to 77×, in the case of queen8_8). Additionally, for myciel5, we also ran the CPU-based algorithm, which took more than 19 hours to finish. The GPU version only took 34 minutes.

The GPU algorithm can process a large amount of states in a very short time. For example, for the graph 1fjl, 3680 million states were explored in just 1730 seconds, i.e., over 2 million states were processed each second (and for each state, a $\Theta(|V||E|)$-time algorithm is executed). The highest throughput (2.5 million states/sec.) is achieved on SylvesterGraph, but this graph has relatively few vertices.

We caution the reader that the graph names are somewhat ambiguous. For instance, the queen7_7 instance is from libtw and has treewidth 35. The 2016 PACE instances include a graph called dimacs_queen7_7 which only has treewidth 28. The instances used in our evaluation are available from our GitHub repository [9].

4.3 Work Size and Global v.s. Shared Memory

In this section, we study the effect of work size and whether shared or global memory is used on the running time of our implementation.

Recall that shared memory is a small amount (in our case, 96KiB) of memory that is physically close to each Streaming Multiprocessor, and is therefore in principle faster than the (much larger, off-chip) global memory. We would therefore expect that our implementation is faster when used with shared memory.

Each SM contains 128 CUDA cores, and thus 4 warps of 32 threads each can be executed simultaneously on each SM. The work size (which should be a multiple of 32), represents the number of threads we assign to each SM. If we set the work size larger than 128, more threads than can physically be executed at once are assigned to one SM. The SM can then switch between executing different warps, for instance to hide latency of memory accesses. If the work size is smaller than 128, a number of CUDA cores will be unutilized.

In Table 2, we present some experiments that show running times on several graphs, depending on whether shared memory or global memory is used, for several sizes of work group (which is the number of threads allocated to a single SM).

There is not much difference between running the program using shared or global memory. In most instances, the shared memory version is slightly faster. Surprisingly, it also appears that the work size used does not affect the running time significantly. This suggests that our program is limited by the throughput of memory, rather than being computationally-bound.

Table 3 The effect of using the Minor-Min-Width Heuristic. Time is in seconds. Global memory, work size 128.

Name	$\|V\|$	tw	With MMW		Without MMW	
			Time	Exp	Time	Exp
1e0b_graph	55	24	2750	1660 $\times 10^6$	779	1730 $\times 10^6$
1fjl_graph*	57	26	timeout	3260 $\times 10^6$	1730	3680 $\times 10^6$
1igd_graph	59	25	471	235 $\times 10^6$	107	261 $\times 10^6$
1ubq*	47	~~11~~	2010	1500 $\times 10^6$	1130	2300 $\times 10^6$
8x6_torusGrid*	48	~~7~~	1350	1300 $\times 10^6$	1110	2100 $\times 10^6$
BN_98	47	21	1480	1440 $\times 10^6$	689	1590 $\times 10^6$
contiki_dhcpc_handle_dhcp*	39	6	2670	2900 $\times 10^6$	1490	2930 $\times 10^6$
DoubleStarSnark	30	6	38,3	76,0 $\times 10^6$	34,5	87,6 $\times 10^6$
KneserGraph_8_3*	56	~~24~~	1330	1220 $\times 10^6$	1730	4130 $\times 10^6$
myciel5*	47	19	2550	3200 $\times 10^6$	2000	4000 $\times 10^6$
NonisotropicUnitaryPolarGraph_3_3	63	53	3,36	1,30 $\times 10^6$	1,16	1,56 $\times 10^6$
queen8_8	64	45	83,5	51,1 $\times 10^6$	26,3	57,9 $\times 10^6$
RandomBarabasiAlbert_100_2*	41	12	2390	2840 $\times 10^6$	1610	3280 $\times 10^6$
RandomBoundedToleranceGraph_60	59	30	0,630	0,0478 $\times 10^6$	0,274	0,0560 $\times 10^6$
SylvesterGraph	36	15	274	503 $\times 10^6$	248	632 $\times 10^6$
te*	62	~~10~~	2260	1690 $\times 10^6$	1170	2160 $\times 10^6$

4.4 Minor-Min-Width

In Table 3, we list results obtained when using Minor-Min-Width to prune states.

The computational expense of using MMW is comparable to that of the initial computation (for determining the degree of vertices): the algorithm does a linear search for a minimum degree vertex (using the precomputed degree values), and then does a graph traversal (using BFS) to find a minimum degree neighbour (recall that we do not store the intermediate graph, and use only a single copy of the original graph). Once such a neighbour is found, the contraction is performed (by updating the disjoint set data structure) and another graph traversal is required (to compute the number of common neighbours, and thus update the degree of the vertex).

The lower bound given by MMW does not appear to be very strong, at least for the graphs considered in our experiment: the reduction in number of states expanded is not very large (for instance, from 1730 million states to 1660 million for `1e0b`, or from 1590 million to 1480 million for `BN_98`). The largest reductions are visible for graphs on which we run out of memory (for instance, from 4130 million to 1330 million for `KneserGraph_8_3`), but this is likely because the search is terminated before we reach the actual treewidth (so we avoid the part of our search where using a heuristic is least effective) and there are no graphs on which we previously ran out of memory for which MMW allows us to determine the treewidth (the biggest improvement is that we are able to determine that `te` has treewidth at least 10, up from treewidth at least 7).

Consistent with the relatively low reduction in the number of states expanded, we see the computation using MMW typically takes around $2-3$ times longer. On the graphs considered here, the reduction in search space offered by MMW does not offset the additional cost of computing it.

Again, the GPU version is significantly faster than executing the same algorithm on the CPU: we observed a 55× speedup for `queen8_8`. Still, given what we observed in Section

4.3, it is not clear whether our approach of not storing the intermediate graphs explicitly is indeed the best approach. Our main motivation for taking this approach was to be able to store the required data structures entirely in shared memory, but our experiments indicate that for MMW, using global memory gives better performance than using shared memory. However, the relatively good performance of global memory might be (partially) due to caching and the small amount of data transferred, so it is an interesting open question to determine whether the additional memory costs of using more involved data structures is compensated by the potential speedup.

4.5 Loop Unnesting

Finally, we experimented with another technique, which aims to increase parallelism (and thus speedup) by limiting branch divergence. However, as the results were discouraging, we limit ourselves to a brief discussion.

The algorithm of Listing 1 consists of a loop (lines 5–22) over the (not yet eliminated) vertices, inside of which is a depth-first search (which computes the degree of the vertex, to determine whether it can be eliminated). The depth-first search in turn consists of a loop which runs until the stack becomes empty (lines 10–19) inside of which is a final loop over the neighbours of the current vertex (lines 12–18). This leads to two sources of branch divergence:

- First, if the graph is irregular, all threads in a warp have to wait for the thread that is processing the highest degree vertex, even if they only have low-degree vertices.
- Second, all threads in a warp have to wait for the longest of the BFS searches to finish before they can start processing the next vertex.

To alleviate this, we proposed a technique which we call *loop unnesting*: rather than have 3 nested loops, we have only one loop, which simulates a state machine with 3 states: (1) processing the adjacency list of a vertex, (2) having finished processing of an adjacency list and being ready to pop a new vertex off the queue, or (3) having finished a BFS, and being ready to begin computing the degree of a new vertex.

We considered a slightly more general version of this idea: in an (x, y)-unnesting of our program, after every x iterations of the inner loop (exploring neighbours of the current vertex) one iteration of the middle loop is executed (if exploring the adjacency list is finished, get a new vertex from the queue), and for every y iterations of the middle loop, one iteration of the outer loop is executed (begin processing an entirely new vertex). Thus, a $(1, 1)$-unrolling corresponds to the state machine simulation described above, and an (∞, ∞)-unrolling corresponds to the original program.

Picking the right values for x, y means finding the right trade-off between checking frequently enough whether a thread is ready to start working on another vertex, and the cost of performing those checks. What we observed was surprising: while $(1, 1)$, $(3, 2)$ and $(1, \infty)$-unrollings gave reasonable results, the best results were obtained with (∞, ∞)-unrollings (i.e. the original, unmodified algorithm) and the performance of $(\infty, 1)$-unrollings was abysmal.

We believe that a possible explanation may be that loop unnesting does work to some extent, but not unnesting the loops has the advantage that all BFS searches running simultaneously start from the same initial vertex, and (up to differences caused by different sets S being used) will access largely the same values from the adjacency lists at the same time, which may increase the efficiency of read operations. On the other hand, $(\infty, 1)$-unnesting can not take advantage of either phenomenon: different initial vertices may be processed at any given time (so there is little consistency in memory accesses) and the inner loop

is not unnested at all so there is no potential to gain speedup there either. Perhaps for larger graphs, where the difference in length of adjacency lists may be more pronounced, or the amount of time a BFS takes varies more strongly with the initial vertes and S, loop unnesting does provide speed up, but for the graphs considered here it does not appear to be a beneficial choice.

5 Conclusions

We have presented an algorithm that computes treewidth on the GPU, achieving a very large speedup over running the same algorithm on the CPU. Our algorithm is based on the classical $O^*(2^n)$-time dynamic programming algorithm [4] and our results represent (promising) first steps in speeding up dynamic programming for treewidth on the GPU. The current best known practical algorithm for computing treewidth is the algorithm due to Tamaki [22]. This algorithm is much more complicated, and porting it to the GPU would be a formidable challenge but could offer an extremely efficient implementation for computing treewidth.

Given the large speedup achieved, we are no longer mainly limited by computation time. Instead, our ability to solve larger instances is hampered by the memory required to store the very large lists of partial solutions. Using minor-min-width did not prove effective in reducing the number of states considerably, so it would be interesting to see how other heuristics and pruning rules (such as simplicial vertex detection) could be implemented on the GPU.

GPUs are traditionally used to solve easy (e.g. linear time) problems on very large inputs (such as the millions of pixels rendered on a screen, or exploring a graph with millions of nodes), but clearly, the speedup offered by inexpensive GPUs would also be very welcome in solving hard (NP-complete) problems on small instances. Exploring how techniques from FPT and exact algorithms can be used on the GPU raises many interesting problems - not only practical ones, but also theoretical: how should we model complex devices such as GPUs, with their many types of memory and branch divergence issues?

Acknowledgements. We thank Jacco Bikker for discussions on the architecture of GPUs, and Gerard Tel for discussions on hash functions.

Source Code and Instances. We have made our source code, as well as the graphs used for the experiments, available on GitHub [9].

References

1 Austin Appleby. SMHasher. Accessed 2017-04-12. URL: `https://github.com/aappleby/smhasher`.
2 Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
3 Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25:1305–1317, 1996.
4 Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. On exact algorithms for treewidth. *ACM Trans. Algorithms*, 9(1):12:1–12:23, 2012.
5 Hans L. Bodlaender and Arie M.C.A. Koster. Safe separators for treewidth. *Discrete Mathematics*, 306(3):337–350, 2006.
6 Hans L. Bodlaender and Arie M.C.A. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2008.

7 Hans L. Bodlaender and Arie M.C.A. Koster. Treewidth computations II. Lower bounds. *Information and Computation*, 209(7):1103–1119, 2011.

8 Hans L. Bodlaender and T. C. van der Zanden. BZTreewidth. Accessed 2017-04-11. URL: `https://github.com/TomvdZanden/BZTreewidth`.

9 Hans L. Bodlaender and T. C. van der Zanden. GPGPU treewidth. Accessed 2017-04-21. URL: `https://github.com/TomvdZanden/GPGPU-Treewidth`.

10 François Clautiaux, Jacques Carlier, Aziz Moukrim, and Stéphane Nègre. New lower and upper bounds for graph treewidth. In Klaus Jansen, Marian Margraf, Monaldo Mastrolilli, and José D. P. Rolim, editors, *Experimental and Efficient Algorithms: Second International Workshop, WEA 2003*, pages 70–80. Springer, 2003.

11 Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*. Springer, 1st edition, 2015.

12 Holger Dell, Thore Husfeldt, Bart M. P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. The parameterized algorithms and computational experiments challenge (PACE). Accessed 2017-04-05. URL: `https://pacechallenge.wordpress.com/pace-2016/track-a-treewidth/`.

13 Holger Dell, Thore Husfeldt, Bart M. P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. The first parameterized algorithms and computational experiments challenge. In Jiong Guo and Danny Hermelin, editors, *11th International Symposium on Parameterized and Exact Computation, IPEC 2016, August 24-26, 2016, Aarhus, Denmark*, volume 63 of *LIPIcs*, pages 30:1–30:9. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. `doi:10.4230/LIPIcs.IPEC.2016.30`.

14 P. Alex Dow. *Search Algorithms for Exact Treewidth*. PhD thesis, 2010.

15 P. Alex Dow and Richard E. Korf. Best-first search for treewidth. In *Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, pages 1146–1151. AAAI Press, 2007.

16 Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence*, UAI '04, pages 201–208. AUAI Press, 2004.

17 Alexander Hein and Arie M. C. A. Koster. An experimental evaluation of treewidth at most four reductions. In Panos M. Pardalos and Steffen Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental and Efficient Algorithms, SEA 2011*, volume 6630 of *LNCS*, pages 218–229. Springer, 2011.

18 M. Held and R. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10:196–210, 1962.

19 Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In Yossi Azar and Thomas Erlebach, editors, *Algorithms – ESA 2006: 14th Annual European Symposium*, pages 456–467. Springer, 2006.

20 NVIDIA. NVIDIA GeForce GTX 1080 Whitepaper. Accessed 2017-04-10. URL: `http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf`.

21 NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: FERMI. Accessed 2017-04-12. URL: `http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf`.

22 Hisao Tamaki. Positive-instance driven dynamic programming for treewidth. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, volume 87 of *LIPIcs*, pages 68:1–68:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. `doi:10.4230/LIPIcs.ESA.2017.68`.

23 Tom C. van der Zanden and Hans L. Bodlaender. Computing Treewidth on the GPU. Preprint, 2017. `arXiv:arXiv:1709.09990`.

24 Thomas C. van Dijk, Jan-Pieter van den Heuvel, and Wouter Slob. Computing treewidth with libtw, 2006. Accessed 2017-06-16. URL: `http://www.treewidth.com/treewidth`.

25 Y. Yuan. A fast parallel branch and bound algorithm for treewidth. In *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, pages 472–479, 2011.

26 Rong Zhou and Eric A. Hansen. Combining breadth-first and depth-first strategies in searching for treewidth. In *Proceedings of the 21st International Joint Conference on Artifical Intelligence*, IJCAI'09, pages 640–645. Morgan Kaufmann Publishers Inc., 2009.