Weighted Model Counting on the GPU by Exploiting Small Treewidth

Johannes K. Fichte

International Center for Computational Logic, TU Dresden, 01062 Dresden, Germany
johannes.fichte@tu-dresden.de
https://orcid.org/0000-0002-8681-7470

Markus Hecher

Institute of Logic and Computation, TU Wien, Favoritenstraße 9-11, 1040 Wien, Austria
hecher@dbai.tuwien.ac.at
https://orcid.org/0000-0003-0131-6771

Stefan Woltran

Institute of Logic and Computation, TU Wien, Favoritenstraße 9-11, 1040 Wien, Austria
woltran@dbai.tuwien.ac.at
https://orcid.org/0000-0003-1594-8972

Markus Zisser

Institute of Logic and Computation, TU Wien, Favoritenstraße 9-11, 1040 Wien, Austria
markus.zisser@student.tuwien.ac.at

Abstract

We propose a novel solver that efficiently finds almost the exact number of solutions of a Boolean formula (#Sat) and the weighted model count of a weighted Boolean formula (WMC) if the treewidth of the given formula is sufficiently small. The basis of our approach are dynamic programming algorithms on tree decompositions, which we engineered towards efficient parallel execution on the GPU. We provide thorough experiments and compare the runtime of our system with state-of-the-art #Sat and WMC solvers. Our results are encouraging in the sense that also complex reasoning problems can be tackled by parameterized algorithms executed on the GPU if instances have treewidth at most 30, which is the case for more than half of counting and weighted counting benchmark instances.

2012 ACM Subject Classification Theory of computation → Parameterized complexity and exact algorithms, Theory of computation → Complexity theory and logic, Computer systems organization → Single instruction, multiple data, Hardware → Theorem proving and SAT solving, Computing methodologies → Graphics processors

Keywords and phrases Parameterized Algorithms, Weighted Model Counting, General Purpose Computing on Graphics Processing Units, Dynamic Programming, Tree Decompositions, Treewidth

Digital Object Identifier 10.4230/LIPIcs.ESA.2018.28

Funding The work has been supported by the Austrian Science Fund (FWF), Grants Y698 and P26696, and the German Science Fund (DFG), Grant HO 1294/11-1. The first and second author are also affiliated with the University of Potsdam, Germany.

1 Introduction

Many computational problems in modern society account to probabilistic reasoning, statistics, and combinatorics. Examples of such problems are identifying the reliability of energy infrastructure [16] or learning preference distributions [10]. Several of these real-world problems can be solved by *representing* the question in (Boolean) formulas [42, 15, 47] and associating the number of solutions of the formula directly with the answer to the question. The task to compute the number of solutions of a formula is usually referred to as the problem #Sat, which is theoretically of high worst case complexity (#P-hard [38]), and generalizes the problem of deciding whether a formula has a solution (Sat). If in addition each literal in the formula has an associated weight and we are interested in the sum of weights of all solutions, where the weight of a truth assignment is the product of the weights of its literals, we speak about weighted model counting (WMC).

One approach to tackle these problems origins in parameterized algorithms, which are based on the assumption that certain structural restrictions in the input allow for efficient solving of problems that are hard in general. A seminal example in this direction is to exploit small treewidth for Sat and #Sat [40]. *Treewidth* roughly measures the tree-likeness of an input graph and is defined in terms of certain decompositions of the graph. For Boolean formulas one takes a graph representation of the input formula, namely the *primal* or *incidence* graph. In order to solve #Sat, dynamic programming on a tree decomposition of the graph representation [40] is used. There one traverses the decomposition in post-order (bottom-up traversal) and computes at each node information stored in a *table*. The runtime heavily depends on the size of the table, which is bounded by a function in the treewidth. Recent competitions in parameterized complexity [14] reveal that exact parameterized algorithms are not just a vibrant theoretical research area, but their implementations are also able to outperform up-to-date Sat solvers when determining treewidth.

State-of-the-art #Sat or WMC engines so far rely on standard techniques from Sat-solving [44, 41, 26], knowledge compilation [33], or approximate solving [7, 8] by means of sampling using Sat solvers. There is few work on parallelizing certain aspects of modern Sat solving on Graphics Progressing Units (GPUs), e.g., [11]. However, a core technique of Sat solving, conflict driven clause learning (CDCL), has inherent sequential aspects and does not parallelize well [3, 22, 24, 34]. In contrast, many problems in artificial intelligence and machine learning have significantly benefited from parallelization. In particular, running algorithms on GPUs or using special purpose processing units such as Tensor Processing Units (TPUs) can speedup standard AI tasks by more than two orders [28].

Parallel algorithms can be implemented on shared-memory or distributed-memory machines. Shared-memory based systems concern parallelizing one machine, whereas distributed-memory based systems involve several machines. Compared to distributed-memory based systems (as for example dCountAntom [6]) consisting of a massive amount of units, we rely on shared-memory based (used for instance in countAntom [5]) techniques, i.e., in particular plain consumer processors and graphics cards. Distributed units build on fast communication networks, and when designing such systems, the goal is to avoid communication overhead where possible to reduce the bottleneck induced by the transport channel. Shared-memory systems on the other hand – though limited by synchronization necessities – do not directly suffer from this issue and are in a sense incomparable to distributed-memory based systems. Consequently, we purposely focus on shared-memory based systems in this paper.

New Contribution

In this paper, we show that computationally involved problems such as #Sat or WMC benefit in practice from parallelization when the input instance has small treewidth. To this end, we implement the aforementioned dynamic programming approach for the first time on a GPU and provide an experimental evaluation. More specifically, our contributions are:

1. We engineer a novel architecture for GPU-based parameterized algorithms that allow for *parallel solving* of #Sat and WMC and where the runtime depends on the size of the computed decomposition of the graph representation of the formula. To this end, we traverse a tree decomposition similar to a sequential algorithm, but distribute the computation of tables among different computation units such that each potential row runs in one thread of the GPU, which is key for an efficient parallelization in practice.
2. We provide an OpenCL implementation gpusat[1] of two parameterized algorithms for the GPU. We highlight crucial algorithm engineering steps such as handling non-nice tree decompositions and specialized procedures that adjust the table sizes to the available number of computation units.
3. We provide rigorous experimental work where we consider an extensive number of dedicated #Sat and WMC instances and compare gpusat with a wide range of related solvers. We present upper bounds on the primal and incidence treewidth for our entire set of benchmark instances and compare the solving time with state-of-the-art solvers. In particular, our results show that gpusat is the fastest, *precise* solver for instances of treewidth up to 30 and is even able to solve certain instances of treewidth up to 45.

2 Solving #SAT by Dynamic Programming

Boolean Satisfiability and Weighted Model Counting

A literal is a Boolean variable x or its negation $\neg x$. A *clause* is a finite set of literals, interpreted as the disjunction of these literals. We say that a clause is *unit* if it is singleton. A *(CNF) formula* is a finite set of clauses, interpreted as the conjunction of its clauses. Let F be a formula. A *sub-formula* S of F consists of subsets of clauses of F. For a clause $c \in F$, $\mathrm{var}(c)$ consists of all variables that occur in c and $\mathrm{var}(F) := \bigcup_{c \in F} \mathrm{var}(c)$. An *assignment* is a mapping $\alpha : \mathrm{var}(F) \to \{0,1\}$ and $2^{\mathrm{var}(F)}$ the set of all assignments of F. $F(\alpha)$ is the formula F *under assignment* α obtained by removing all clauses c from F that contain a literal set to 1 by α and removing from the remaining clauses all literals set to 0 by α. An assignment α is *satisfying* if $F(\alpha) = \emptyset$. The problem #Sat asks to output the number of satisfying assignments of a formula. Let w be function that maps each literal of F to a real between 0 and 1. We call $w(\ell)$ the *weight* of literal ℓ. The *weight* of α is the product over the weights of its literals, i.e., $w(\alpha) := \Pi_{v \in \alpha^{-1}(1)} w(v) \cdot \Pi_{v \in \alpha^{-1}(0)} w(\neg v)$. The *weighted model count* of F is the sum of weights over all its satisfying assignments, i.e., $\Sigma_{\alpha \in 2^{\mathrm{var}(F)}, F(\alpha) = \emptyset} w(\alpha)$. The problem WMC asks to output the weighted model count of F.

Tree Decomposition and Treewidth

A *tree decomposition (TD)* of a graph G is a pair $\mathcal{T} = (T, \chi)$ where T is a rooted tree (arborescence) and χ is a mapping that assigns to each node $t \in V(T)$ a set $\chi(t) \subseteq V(G)$, called a *bag*, such that the following conditions hold: (i) $V(G) = \bigcup_{t \in V(T)} \chi(t)$ and $E(G) \subseteq$

[1] Our solver is available at `github.com/daajoe/GPUSAT`.

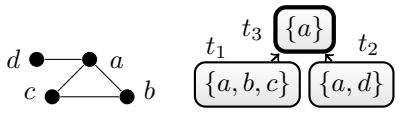

Figure 1 Primal graph P_F of F from Example 1 (left) with a tree decomposition $\mathcal{T}$ of the graph P_F (right).

$\bigcup_{t \in V(T)} \{ \{u, v\} \mid u, v \in \chi(t) \}$; and (ii) for each $r, s, t \in T$, such that s lies on the path from r to t, we have $\chi(r) \cap \chi(t) \subseteq \chi(s)$. The *width* of $\mathcal{T}$, denoted width($\mathcal{T}$), is $\max_{t \in V(T)} |\chi(t)| - 1$. The *treewidth* tw($G$) of G is the minimum width($\mathcal{T}$) over all tree decompositions $\mathcal{T}$ of G. For arbitrary but fixed $w \geq 1$, it is feasible in linear time to decide if a graph has treewidth at most w and, if so, to compute a tree decomposition of width w [4]. Graphs that originate in the real-world often admit tree decompositions of small width [14]. Interestingly, one can use GPU-based implementations to compute the treewidth [46]. However, we use *htd* together with min-fill heuristics to compute TDs [1]. In that case, the width might not be minimal. In order to simplify cases in the theoretical algorithms, one uses for theoretical descriptions so-called nice TDs, which we can compute in linear time without increasing the width [29].

We need dedicated graph representations for satisfiability problems. The *primal graph* of a formula F has as vertices its variables and two variables are joined by an edge if they occur together in a clause of F. For a given node s of a TD (T, χ) of the primal graph of F, we let $F_s := \{ c \mid c \in F, \text{var}(c) \subseteq \chi(s) \}$, i.e., clauses entirely covered by $\chi(s)$. The set $F_{\leq s}$ denotes the union over F_t for all descendant nodes $t \in V(T)$ of s. The *incidence graph* of a formula F is the bipartite graph on the clauses and variables of F, where a clause and a variable are joined by an edge if the variable occurs in the clause. We call the treewidth of the primal or incidence graph the *primal treewidth* or *incidence treewidth*, respectively.

▶ **Example 1.** Consider the formula $F := \{c_1 := a \vee b \vee \neg c, c_2 := \neg b \vee \neg a, c_3 := a \vee \neg d\}$. The primal graph P_F of formula F and a TD $\mathcal{T}$ of P_F are depicted in Figure 1. Intuitively, $\mathcal{T}$ allows to evaluate formula F in parts. Later when evaluating $F_{\leq t_3}$, we split into $F_{\leq t_1}$ and $F_{\leq t_2}$, which refer to $\{c_1, c_2\}$ and $\{c_3\}$, respectively.

Dynamic Programming on TDs

A #SAT or WMC solver based on *dynamic programming (DP)* evaluates the input formula F in parts along a given TD of F. For each node of the tree decomposition results are stored in *tables*. The algorithm works as outlined in Figure 2 and performs the following steps:

1. Construct a primal graph or incidence graph G of F.
2. Heuristically compute a tree decomposition (T, χ) of G.
3. DP: For every node t in post-order of $V(T)$, we run an algorithm $\mathbb{A} \in \{\mathbb{PRIM}, \mathbb{INC}\}$ that outputs a table τ_t and takes as input the node t, its bag $\chi(t)$, sub-formula F_t, and previously computed child tables C-Tabs of t (empty at the leaves).
4. Print the result by interpreting the table for root n of T.

We provide a brief intuition on $\mathbb{PRIM}$. For details and algorithm $\mathbb{PRIM}$ and $\mathbb{INC}$, we refer to the original source [40]. The main idea of $\mathbb{PRIM}$ is to store in table τ_t only assignments, which are restricted to bag χ_t depending on nice case distinctions of the node type, and its counters. From the count stored together with an assignment in the table at node t, we can read the number of satisfying assignments of the formula $F_{\leq t}$ for the induced sub-tree of T rooted at t. In the end, we can simply read the solution from the table at the root. $\mathbb{INC}$

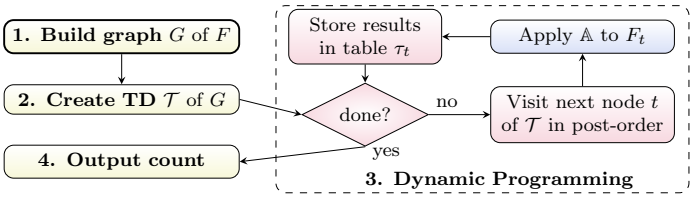

Figure 2 Architecture of solvers based on dynamic programming on the CPU where an algorithm $\mathbb{A}$ modifies tables.

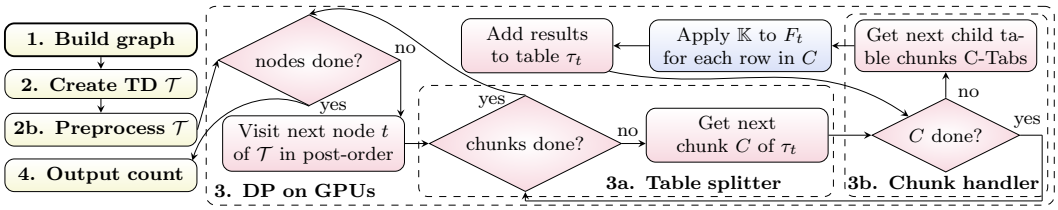

Figure 3 Architecture of our dynamic programming solver on the GPU where kernel $\mathbb{K}$ modifies each row individually and in parallel.

works similar, but requires more complex data structures. Both algorithms can be modified for computing the weighted model count.

3 GPU-based DP Architecture

Over the last decade there has been significant effort in the consumer market on graphics processing units (GPU) dedicated to render 3D graphics. GPUs are highly specialized in processing geometry and image information independent and in parallel. When one compares the actual computation power of such units to CPUs, GPUs are extremely cost efficient [28]. Recently, there is also increasingly strong interest in using such units for general purposes of parallelizable tasks in artificial intelligence and computation intensive applications such as number crunching [43].

In this section, we present an architecture for parallel dynamic programming on the GPU. In the dynamic programming algorithm, as outlined in Figure 2, nodes only depend on child nodes and in the table algorithm ($\mathbb{PRIM}$) rows in a table are entirely independent of each other. Consequently, there are two imminent ways to parallelize the execution. The first way is to compute tables for multiple nodes in parallel. This, however, does not allow for immediate massive parallelization due to dependencies to the child nodes. The second way is to distribute rows among different computation units. This allows with the right hindsight for massive parallelization, in particular, because the computation of a specific row is independent of any other row in the same table.

We would like to emphasize that the crucial tricks are (i) the way *how we parallelize* and (ii) a direct way to *represent potential assignments* (as explained below). Implementation techniques on the GPU and its parallelization follow a straight-forward programming paradigm and require in contrast to distributed-memory based systems [6] no parameter tuning.

The Kernel

Figure 3 outlines our dynamic programming approach on the GPU. It replaces Step 3 in the sequential dynamic programming approach above. The core of our solver is the procedure $\mathbb{K}$,

which considers all possibly resulting rows at node t; even rows where the assignment in the row might not satisfy the sub-formula F_t, as we do not know the satisfiability in advance. For a node t, we call the table that consists of all possible rows *exhaustive table* (at node t). On the GPU a potential row can be seen as an output pixel that has to be computed. For all rows we take as common input the sub-formula F_t and specific to the row (assignment) corresponding rows in the tables of children of t. In terms of the methodology of programming on the GPU, procedure $\mathbb{K}$ is called a *kernel*. In our case, we spawn a (computation) thread at the GPU for each potential row of the exhaustive table with kernel $\mathbb{K}$. All threads have the same instructions $\mathbb{K}$, but start on different data. The underlying principle of the architecture is usually called single instruction multiple threads (SIMT). The kernel $\mathbb{K}$ depends on the type of the node just as before. For example, from the algorithm $\mathbb{PRIM}$ would still obtain several case distinctions but only for the different node types. In practice, however, we do not work on nice tree decompositions and therefore have case distinctions of mixed form. Further, it is crucial to tune our implementation towards simplicity and efficiency, which requires extensive *bit-twiddling* [2]. In particular, we need to reduce the number of execution paths, which we obtain by avoiding conditional jumps if possible. In other words, we prefer bit operations over if then else constructions to optimize for the underlying hardware. The GPU computation outputs counts of assignments that satisfy the sub-formula F_t. Processing all rows at once on the GPU allows us to compute the entire table in one GPU call, if the number of threads on the GPU and the available *video RAM (VRAM)* suffices, otherwise we run multiple "rounds" of computation.

Table Splitting

Even though running the kernel on the GPU allows us to obtain a parallel version of dynamic programming, our *main memory (RAM)* requirements are quite extensive and the required RAM might exceed the capacity of the VRAM on the GPU. Hence, we need to *split* large tables into smaller partitions of exhaustive tables (*chunks*). For a node t, this affects tables of the children of t as well as the exhaustive table at node t. We split the exhaustive table by a *table splitter* in Step 3a of Figure 3. A *chunk handler* then takes relevant chunks of the exhaustive table and spawns kernels depending on chunks of corresponding child tables as in Step 3b of Figure 3. The resulting counts for one exhaustive table chunk of this step are summed up accordingly and stored in table τ_t as previously explained.

TD Preprocessing

Orthogonally, in order to utilize the entire computation power of one cycle on the GPU, we merge several nodes of the tree decomposition into one node to obtain larger exhaustive tables. This reduces overhead caused by IO operations between the RAM and the VRAM and caused by spawning and deallocating GPU threads. Therefore, we run a *preprocessing* operation on the tree decomposition that merges small bags. This step may result in a tree decomposition that is not nice. Hence, we need to implement *more complex kernel* algorithms. Further, we obtain an even better GPU utilization by handling certain cases (introduce, remove, and leaf [40]) in one case and merging small bags, which share introduced and removed variables or clauses.

Data Types and Precision

In contrast to programs that are executed on the CPU, the instruction set for procedures on the GPU (kernels) is very limited and only a few data structures are available. In particular,

there is no established data type for storing big numbers as directly offered in common programming languages [27, 48]. Still, we need dedicated data types to represent large numbers to express counts of satisfying assignments. Unfortunately, storing the exact number of solutions in each row of each table can be too expensive on the VRAM. Instead, we use the data type *double*. Hence, we cannot expect an exact solution when solving #Sat or WMC. However, we can use an extended type (*double4*) that combines four plain double types to increase the precision. Then, we can balance between a faster running time or higher precision. When solving #SAT we may run into a double or double4 overflow. Then, we can *relax* the instance into a weighted model count instance where all literals have the same weight, but less than 1, and reconstruct the original count at the end of the computation.

Implementation

We implemented our approach for dynamic programming on the GPU and kernels for the table algorithms $\mathbb{PRIM}$ and $\mathbb{INC}$ into our prototypical solver *gpusat*. We used OpenCL1.2 [37], which is a universal vendor and hardware independent computation framework, and C++11 for our implementation. Currently, we only use very limited formula preprocessing and simplifications during the search. Prior solving, we once propagate unit clauses in the usual way. If there is a table that does not contain any solution, we terminate and output that there is no solution. At a node t, we compute the sub-formula F_t using the CPU and start one GPU thread for each possible assignment. Kernels are compiled only once. The assignment is tied to the memory address, which then requires only memory for counts on the VRAM. We statically split tables based on the available memory on the GPU. We merge bags of small size as long as we obtain at most 14 variables in one bag.

4 Experimental Results

We performed an extensive series of experiments using several benchmark sets among them instances that originate in model counting and weighted model counting questions. All benchmarks as well as detailed results including raw data are publicly available[2]. Theoretically, we *do not* expect to solve formulas with graph representations of high treewidth. Therefore, we restricted the sets to instances where we were able to find tree decompositions of width below 30 using standard heuristic decomposers [1]. Nonetheless, we provide upper bounds on the treewidth for all instances of our benchmark sets. Since our benchmarks require entirely different type of hardware, we can only use wall clock time as a time measurement. Note that we used cheap consumer hardware for gpusat; whereas we used a very recent server hardware configuration for all other solvers.

Hardware

Our results were gathered on Ubuntu 16.04 LTS Linux machines kernel 4.4.0-101 and 4.14.0-041400, respectively, both pre-Spectre and pre-Meltdown kernels[3]. We ran non-GPU solvers on a cluster of 9 nodes. Each node is equipped with two Intel Xeon E5-2650 CPUs consisting of 12 physical cores each at 2.2 GHz clock speed and 256 GB RAM. Hyper threading was disabled. For gpusat we used a machine equipped with a consumer GPU: Intel Core i3-3245 CPU operating at 3.4 GHz, 16 GB RAM, and one Sapphire Pulse ITX Radeon RX 570 GPU

[2] See: Benchmark repository (including used tree decompositions) [19] and results/raw data [20].
[3] See: `spectreattack.com`

Table 1 Overview on upper bounds of the primal treewidth for considered benchmarks. # represents counting and W represents weighted model counting, number N of instances, number n of variables, median t Mdn of the runtime in seconds, maximum runtime t, and median Mdn and percentiles of the upper bounds on the treewidth.

set	origin	N	n Mdn	t[s] Mdn	(max.)	Mdn	50%	80%	95%
W *Dqmr*	`Cachet`	660	140	0.0	(1.6)	28	28	42	44
W *Grid*	`Cachet`	420	1825	0.2	(1.3)	29	29	39	71
W *Plan*	`Cachet`	11	812	2.9	(9.3)	73	85	399	na
# *Mixed*	`c2d`	14	1287	3.8	(15.9)	57	63	399	540
# *Basic*	`fre/meel`	92	604	1.0	(9.3)	26	37	64	352
# *Proj.*	`fre/meel`	308	62586	120.3	(880.4)	273	328	1084	na
# *Weig.*	`fre/meel`	1080	200	0.1	(1.6)	28	28	40	48

running at 1.24 GHz with 32 compute units, 2048 shader units, and 4GB VRAM using driver amdgpu-pro 17.10.

Solvers

We benchmarked c2d [12], d4 [33], DSHARP [35], miniC2D [36], cnf2eadt [30], bdd_minisat_all [45], and sdd [13], which are based on knowledge compilation techniques. We also included recent approximate solvers ApproxMC [7] and sts [17], as well as pure CDCL-based solvers Clasp [25], Cachet [41], sharpCDCL[4] and sharpSAT [44]. Further, we considered the recent multi-core solver countAntom [5] utilizing exclusively all 12 physical cores, and DP based solvers on tree decompositions from related domains that allow with slight modifications for #SAT solving, i.e., dynasp [18] and dynQBF 1.1.1 [9]. We used all solvers with default options and ran gpusat with uniform weights 0.78 for #SAT experiments. All solvers allow for #SAT solving and sts, gpusat, miniC2D, and Cachet in addition support WMC[5].

Setup and Limits

In order to draw conclusions about the efficiency of gpusat, we mainly inspected the wall clock time *including decomposition time* and number of timeouts. We set a timeout of 900 seconds and limited available RAM to 8 GB per instance. For each instance we only used one tree decomposition, which was obtained by setting a random seed for the decomposer. All the tree decompositions together with the experimental data are provided as well[2]. Note that we avoid IO access on the CPU solvers whenever possible, i.e., we extract instances into the RAM before starting solving.

Benchmark Instances

We considered a selection of 2585 instances from various publicly available benchmark sets for model counting and weighted model counting, consisting of `Cachet` benchmarks[6] (1091 instances), `fre/meel` benchmarks[7](1451 instances), and `c2d` benchmarks[8] (14 instances).

[4] See: `tools.computational-logic.org`
[5] Note that in principle using a d-DNNF reasoner one can also use c2d and d4 to solve WMC.
[6] See: `cs.rochester.edu/u/kautz/Cachet`
[7] See: `tinyurl.com/countingbenchmarks`
[8] See: `reasoning.cs.ucla.edu/c2d`

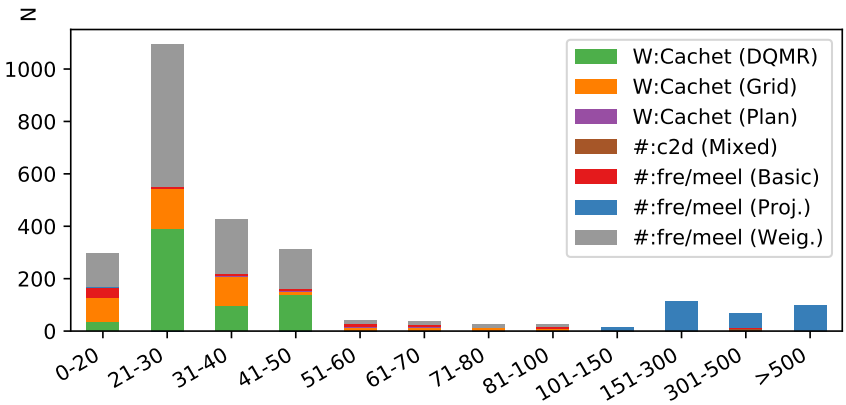

Figure 4 Distribution of instances in upper bound intervals on the primal treewidth over our benchmarks. The x-axis labels the intervals. The y-axis labels the number of observed instances.

Table 2 Number of WMC instances solved. Intervals are given with respect to primal graph. abs err indicates the absolute error. best indicates the number of instances the solver solved the fastest. † absolute weighted model counts were rounded to 3 decimal places. * indicates a significant (≥ 0.2 on average) absolute error.

solver	abs err	0-20	21-30	31-40	41-50	51-60	>60	best	$\sum$
Cachet	**0.0**	92	448	*108*	*105*	*2*	**9**	**476**	764
gpusat(i)	**±0.0**	*127*	487	83	101	0	0	42	798
gpusat(i4)	**±0.0**	*127*	432	75	90	0	0	0	724
gpusat(p)	**±0.0**	**128**	**526**	88	104	0	0	296	*846*
gpusat(p4)	**±0.0**	127	478	80	96	0	0	0	781
miniC2D	†**±0.0**	126	*513*	**143**	**110**	**5**	*6*	143	**903**
sts*	†±0.2	121	**533**	**200**	**152**	1	*6*	*na	***1013**

Treewidth

We computed upper bounds on the primal and incidence treewidth for our benchmarks. The sets contain instances that have the same graph representation. Upper bounds on the treewidth and running times to obtain a decomposition were quite similar for both the primal graph and the incidence graph, except for instances of the set *Proj*. Hence, we focus on an upper bound of the treewidth of the primal graph only and state them in intervals. Table 1 provides statistics on the benchmarks, including runtime of the decomposer to obtain a decomposition. Further, the decomposer ran 0.034s in median (max 1.57s) for instances of width 0–30, 0.132s (max 2.503s) for instances of width 31–40, and 0.054s (max 900.0s) over all instances. The decomposer did not output a decomposition within 900 seconds for 41 instances. Table 1 also states the median of the width of the obtained decompositions and its percentiles, which is the width below a given percentage the instances have. When considering the set *Dqmr*, even 99% of the instances have treewidth below 45. In contrast, the decomposer outputted only decompositions of very high width for instances from the set *Proj*. Figure 4 illustrates the distribution of number of instances (y-axis) and their respective upper bounds (x-axis) for primal treewidth. Considering all sets 54% of the instances have primal treewidth below 30, 70% of the instances have treewidth below 40, and 88% of the instances have treewidth below 150, and for 1% of the instances we obtained no result within the limit.

Table 3 Number of counting instances solved by sum of the top ten counting solvers and gpusat. The symbol * indicates that this gpusat configuration was not among the top ten.

solver	0-20	21-30	31-40	41-50	51-60	>60	best	$\sum$
c2d	164	*519*	**175**	116	*20*	118	120	*1112*
Cachet	133	421	91	109	8	58	13	820
d4	**169**	510	*156*	*119*	**23**	**162**	191	**1139**
gpusat(i)	**169**	490	79	97	0	0	1	835
gpusat(i4)	168	427	70	89	0	0	1	*761
gpusat(p)	**169**	**523**	79	104	0	0	88	875
gpusat(p4)	**169**	478	79	97	0	0	0	823
miniC2D	*167*	491	137	103	8	67	2	973
sharpSAT	136	465	136	112	11	*124*	**483**	984
sts	162	448	101	**146**	10	45	*252*	912

Solved Instances, Runtime, and Error (WMC)

Table 2 gives an overview on the number of solved instances for weighted model counting benchmarks (`Cachet`) and the average error on the weighted model count of the solver. The *absolute error* is the difference of the weighted model count of the solver and the one obtained by Cachet. The configuration gpusat(p) and gpusat(i) refer to the primal and incidence graph implementation, respectively. gpusat(i4) or gpusat(p4) indicates that this configuration uses extended data type precision (double4). gpusat(p) solved the most instances in interval 0–20 and second most instances in interval 21–31 on benchmark sets for WMC; in interval 0–30 gpusat(p) solved the same number of instances as sts. However, gpusat(p) produced almost no absolute error on average ($\pm 1.42 \cdot 10^{-5}$). sts produced a very high absolute error on average ($\pm$ 0.2, stdev 0.8; avg relative error 1037) and had a relative error of more than one order on 56 instances (even when rounding weighted model counts to 3 decimal places). For example, sts outputted a weighted model count of 1.5 (0.873 Cachet) on instance *90-12-3-q.cnf* and 0.316 (0.001 Cachet) on instance *or-50-5-4-UC-20.cnf*. Slightly increasing the number of sampling iterations and samples per level resulted in slower runtimes than gpusat at similar error. Considering all instances gpusat(p) still solved the second most instances at sufficiently high accuracy. The double4 precision versions solved 65 and 74 less instances at negligible accuracy improvement, both versions provide at least the precision that Cachet offers. Figure 5 (top) illustrates runtime results on weighted model counting instances of width between 0 and 30 as cactus plot. When we directly compare gpusat(i) and gpusat(p), gpusat(i) solved 18 instances, which could not be solved by gpusat(p), and 70 instances vice versa. gpusat(i) was on 120 instances faster than gpusat(p) and 815 vice versa.

Solved Instances, Runtime, and Error (#SAT)

Table 3 gives an overview on the number of solved counting instances. gpusat(p) solved the most instances in interval 0–30. Considering all instances gpusat(p) solved the sixth most instances and surprisingly many instances in the interval 31–50. The double4 precision versions solved 52 (p) and 74 (i) less instances. In our experiments we observed on average an error of $4 \cdot 10^{-13}$ for double and $2 \cdot 10^{-32}$ for double4 when comparing to sharpSAT. Hence, we consider the precision error negligible. Without using a uniform weight for gpusat, we ran 80 (p) and 56 (i) times into a double overflow at similar runtime. Figure 5 (bottom) illustrates runtime results (in seconds) on instances of interval 0–30 as cactus plot.

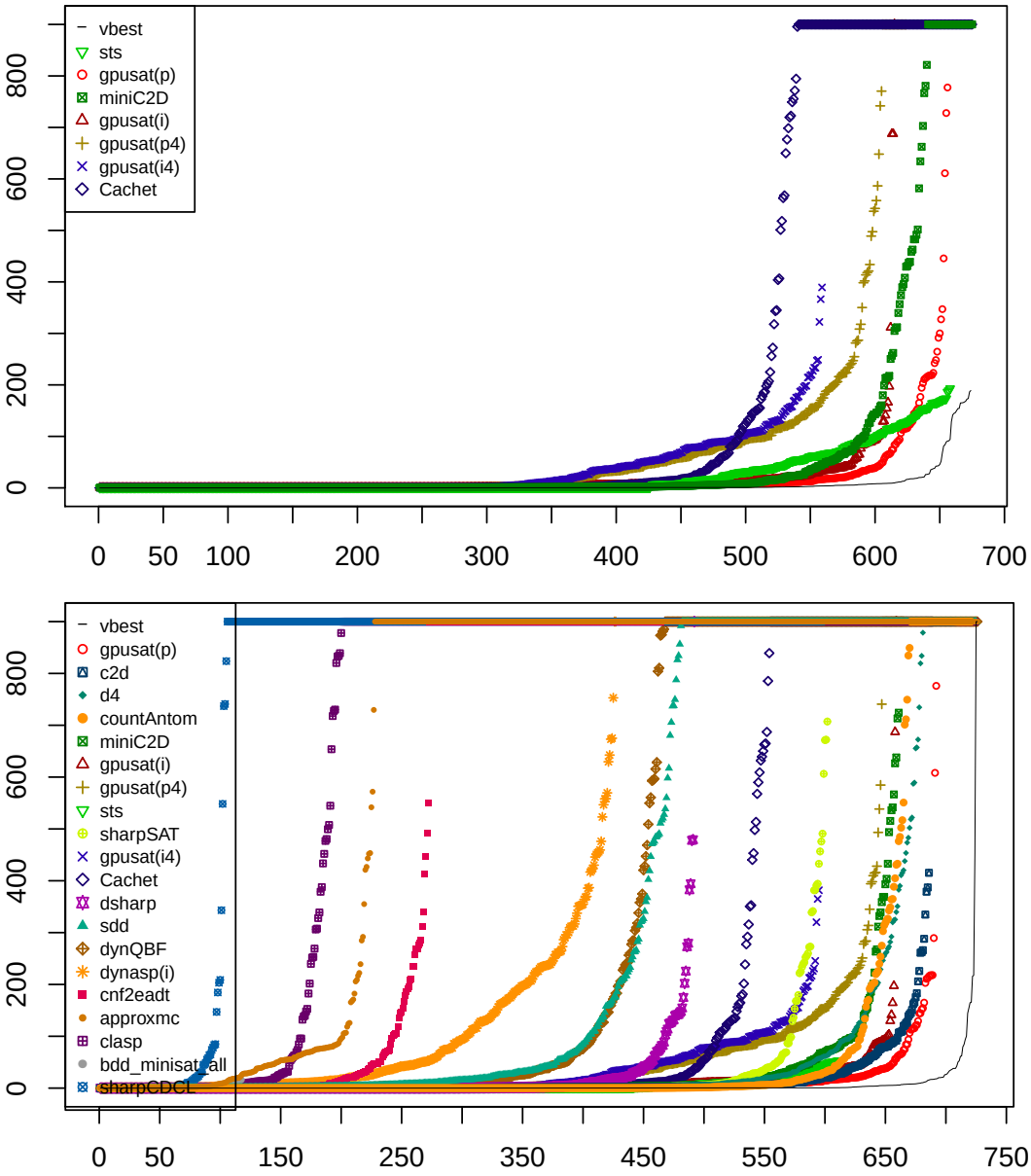

Figure 5 (Top): Runtime on WMC instances (`Cachet`) of primal treewidth at most 30 as cactus plot. (Bottom): Runtime on counting instances (`c2d`, `fre/meel`) of primal treewidth at most 30 as a cactus plot. vbest refers to the virtual best solver, i.e., the best runtime result among all solvers. The x-axis labels consecutive integers that identify instances. The instances are ordered by running time, individually for each solver. Hence, the figure does not provide insights on the solving time of the individual instances and solvers might solve instances fast, which is usually indicated by the virtual best solver. The y-axis labels the runtime (in seconds).

Runtime deviation

We tested gpusat with five different TDs (computed via *htd* [1]) to draw conclusions about runtime stability. The results indicate that the best, the average, and the median among those five tree decomposition still yield good runtime results. Regarding the number of tested instances it is practically quite unlikely to obtain the worst case behavior.

Discussion and Summary

Our results on upper bounds of the primal and incidence treewidth of WMC and #Sat benchmark instances, show that more than half of the instances have treewidth below 30 and more than two third have treewidth below 40. We observed that table splitting was necessary at width above 26. Since gpusat solved the vast majority of the instances in interval 0–30 (only 22 of the 670 WMC instances and 23 of the 721 #Sat instances were not solved), gpusat is highly suitable for the majority of the instances. It turns out that instances in interval 30–40 are still in reach for our solver, even certain instances of width upper bound 45 were solved. Overall gpusat was the fastest virtually exact solver in interval 0–30 for considered WMC and #Sat instances. Our results show that gpusat(p) solves more instances than gpusat(i) and instances often faster, which indicates that gpusat(p) benefits from its simpler algorithms. Using data types of higher precision does obviously not pay off. However, relaxing a #Sat instance into a WMC instance with uniform weights gives almost no precision loss. From our analysis, gpusat is not yet a general propose solver, but highly competitive if the treewidth is below 30. Since we can often find tree decompositions of small width in well below a second, it makes gpusat perfectly suitable for a portfolio approach.

5 Conclusion & Future Work

We introduced the OpenCL-based solver gpusat, which allows for solving #Sat and WMC using dynamic programming on tree decompositions running on consumer GPUs. Our solver parallelizes the computation of each table, vaguely speaking, a partial model count is represented by a pixel. Further, we provide insights on tuning parameterized algorithms for the GPU, including balancing VRAM utilization. We carried out rigorous experimental work, including establishing upper bounds for treewidth of commonly used benchmarks and comparing to most recent solvers. Our findings indicate that a majority of benchmark instances have treewidth below 30. Then, we can also heuristically compute tree decompositions in less than a second. Since gpusat is competitive on those instances, we show that implementations of parameterized algorithms on the GPU are a promising attempt to solve WMC. Hence, those algorithms are not just an interesting theoretical research direction, but its implementations are also competitive in practice. In our opinion, a wide range of applications [8, 15], even suggests to establish dedicated #Sat or WMC competitions, in particular, to obtain a wider picture on which method pays off for which domain.

The results of this paper give rise to several research questions. For instance, it would be interesting to determine the effect of formula preprocessing [32, 31] on the treewidth and solver runtimes. We conducted initial experiments, which suggest that preprocessors might drastically reduce the treewidth and hence increase the applicability of gpusat. Further, it might be fruitful to investigate on obtaining decompositions that have smaller width [21] or that are customized to improve efficiency of the dynamic programming algorithm [1]. An interesting further research direction is to study whether efficient data representation techniques can be combined with dynamic programming similar to techniques for QBF [9] and even be run in parallel on the GPU. Concerning potential overflows of counters for counting-only problems, we aim at analyzing and implementing further improvements as for example storing logarithmic counters [23]. At the same time we want to elaborate on ways to provide high-precision counter (libraries). Finally, parameterized algorithmics suggests recent parameters similar to treewidth [39], which can however be arbitrarily smaller than treewidth. We also aim for implementing these algorithms in OpenCL.

References

1 Michael Abseher, Nysret Musliu, and Stefan Woltran. htd – a free, open-source framework for (customized) tree decompositions and beyond. In Domenico Salvagnin and Michele Lombardi, editors, *Proceedings of the 14th International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR'17)*, volume 10335 of *Lecture Notes in Computer Science*, pages 376–386, Padova, Italy, jun 2017. Springer Verlag. doi:10.1007/978-3-319-59776-8_30.

2 Sean Eron Anderson. Bit twiddling hacks. https://graphics.stanford.edu/~seander/bithacks.html, 2009.

3 Sander Beckers, Gorik De Samblanx, Floris De Smedt, Toon Goedemé, Lars Struyf, and Joost Vennekens. Parallel hybrid SAT solving using OpenCL. In Nico Roos, Mark Winands, and Jos Uiterwijk, editors, *Proceedings of the 24th Benelux Conference on Artificial Intelligence (BNAIC'12)*, pages 11–18, Maastricht, The Netherlands, 2012. Maastricht University.

4 Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

5 Jan Burchard, Tobias Schubert, and Bernd Becker. Laissez-faire caching for parallel #SAT solving. In Marijn Heule and Sean Weaver, editors, *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT'15)*, volume 9340 of *Lecture Notes in Computer Science*, pages 46–61, Austin, TX, USA, 2015. Springer Verlag. doi:10.1007/978-3-319-24318-4_5.

6 Jan Burchard, Tobias Schubert, and Bernd Becker. Distributed parallel #sat solving. In Bronis R. de Supinski, editor, *Proceedings of the 2016 IEEE International Conference on Cluster Computing (CLUSTER'16)*, pages 326–335, 2016. doi:10.1109/CLUSTER.2016.20.

7 Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. Distribution-aware sampling and weighted model counting for SAT. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14)*, pages 1722–1730, Québec City, QC, Canada, 2014. The AAAI Press.

8 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Improving approximate counting for probabilistic inference: From linear to logarithmic sat solver calls. In Subbarao Kambhampati, editor, *Proceedings of 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*, pages 3569–3576, New York City, NY, USA, jul 2016. The AAAI Press. URL: https://bitbucket.org/kuldeepmeel/approxmc.

9 Günther Charwat and Stefan Woltran. Dynamic programming-based QBF solving. In Florian Lonsing and Martina Seidl, editors, *Proceedings of the 4th International Workshop on Quantified Boolean Formulas (QBF'16)*, volume 1719, pages 27–40. CEUR Workshop Proceedings (CEUR-WS.org), 2016. co-located with 19th International Conference on Theory and Applications of Satisfiability Testing (SAT'16).

10 Arthur Choi, Guy Van den Broeck, and Adnan Darwiche. Tractable learning for structured probability spaces: A case study in learning preference distributions. In Qiang Yang, editor, *Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*. The AAAI Press, 2015.

11 Alessandro Dal Palu, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Cud@SAT: SAT solving on GPUs. *Journal of Experimental & Theoretical Artificial Intelligence*, 27(3), 2015.

12 Adnan Darwiche. New advances in compiling CNF to decomposable negation normal form. In Ramon López De Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, pages 318–322, Valencia, Spain, 2004. IOS Press.

13 Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In Toby Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 819–826, Barcelona, Catalonia, Spain, jul 2011. AAAI Press/IJCAI.

14 Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The pace 2017 parameterized algorithms and computational experiments challenge: The second iteration. In Daniel Lokshtanov and Naomi Nishimura, editors, *Proceedings of the 12th International Symposium on Parameterized and Exact Computation (IPEC'17)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 30:1—-30:13. Dagstuhl Publishing, 2017. doi:10.4230/LIPIcs.IPEC.2017.30.

15 Carmel Domshlak and Jörg Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research*, 30, 2007. doi:10.1613/jair.2289.

16 Leonardo Dueñas-Osorio, Kuldeep S. Meel, Roger Paredes, and Moshe Y. Vardi. Counting-based reliability estimation for power-transmission grids. In Satinder P. Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17)*, pages 4488–4494, San Francisco, CA, USA, feb 2017. The AAAI Press.

17 Stefano Ermon, Carla P. Gomes, and Bart Selman. Uniform solution sampling using a constraint solver as an oracle. In Nando de Freitas and Kevin Murphy, editors, *Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence (UAI'12)*, pages 255–264, Catalina Island, CA, USA, aug 2012. AUAI Press.

18 Johannes K. Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. Answer set solving with bounded treewidth revisited. In Marcello Balduccini and Tomi Janhunen, editors, *Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17)*, volume 10377 of *Lecture Notes in Computer Science*, pages 132–145, Espoo, Finland, jul 2017. Springer Verlag. doi:10.1007/978-3-319-61660-5_13.

19 Johannes K. Fichte, Markus Hecher, Stefan Woltran, and Markus Zisser. A Benchmark Collection of #SAT Instances and Tree Decompositions (Benchmark Set), 2018. doi:10.5281/zenodo.1299752.

20 Johannes K. Fichte, Markus Hecher, Stefan Woltran, and Markus Zisser. Analyzed Benchmarks and Raw Data on Experiments for gpusat (Dataset), jun 2018. doi:10.5281/zenodo.1299742.

21 Johannes K. Fichte, Neha Lodha, and Stefan Szeider. Sat-based local improvement for finding tree decompositions of small width. In Serge Gaspers and Toby Walsh, editors, *Proceedings on the 20th International Conference on Theory and Applications of Satisfiability Testing (SAT'17)*, pages 401–411, Melbourne, VIC, Australia, aug 2017. Springer Verlag. doi:10.1007/978-3-319-66263-3_25.

22 Ferdinando Fioretto, Enrico Pontelli, William Yeoh, and Rina Dechter. Accelerating exact and approximate inference for (distributed) discrete optimization with GPUs. *Constraints*, 23(1):1–23, 2017. doi:10.1007/s10601-017-9274-1.

23 Philippe Flajolet. Approximate counting: A detailed analysis. *BIT Numerical Mathematics*, 25(1):113–134, 1985. doi:10.1007/BF01934993.

24 Hironori Fujii and Noriyuki Fujimoto. Gpu acceleration of bcp procedure for sat algorithms. In Hamid R. Arabnia, Hiroshi Ishii, Minoru Ito Kazuki Joe, and Hiroaki Nishikawa, editors, *Proceedings of the 24th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'12)*, pages 10–16, Las Vegas, NV, USA, 2012. CSREA Press.

25 Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012. doi:10.1016/j.artint.2012.04.001.

26 Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Chapter 20: Model counting. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 633–654. IOS Press, Amsterdam, Netherlands, 2009. doi:10.3233/978-1-58603-929-5-633.

27 Torbjörn Granlund, Gunnar Sjödin, Hans Riesel, Richard Stallman, Brian Beuning, Doug Lea, Paul Zimmermann, Ken Weber, Per Bothner, Joachim Hollman, Bennet Yee, Andreas Schwab, Robert Harley, David Seal, Torsten Ekedahl, Linus Nordberg, Kevin Ryde, Kent Boortz, Steve Root, Gerardo Ballabio, Jason Moxham, Niels Möller, Alberto Zanoni, Marco Bodrato, David Harvey, Martin Boij, Marc Glisse, David S Miller, Mark Sofroniou, and Ulrich Weigand. The GNU multiple precision arithmetic library. https://gmplib.org, 2016.

28 Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In David Brooks, editor, *Proceedings of the 44th International Symposium on Computer Architecture (ISCA'17)*, pages 1–12, Toronto, ON, Canada, jun 2017. doi:10.1145/3079856.3080246.

29 Ton Kloks. *Treewidth. Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer Verlag, 1994. doi:10.1007/BFb0045375.

30 Frédéric Koriche, Jean-Marie Lagniez, Pierre Marquis, and Samuel Thomas. Knowledge compilation for model counting: Affine decision trees. In Francesca Rossi and Sebastian Thrun, editors, *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13)*, Beijing, China, aug 2013. The AAAI Press.

31 Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Improving model counting by leveraging definability. In Subbarao Kambhampati, editor, *Proceedings of 25th International Joint Conference on Artificial Intelligence (IJCAI'16)*, pages 751–757, New York City, NY, USA, 2016. The AAAI Press.

32 Jean-Marie Lagniez and Pierre Marquis. Preprocessing for propositional model counting. In Carla E. Brodley and Peter Stone, editors, *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14)*, pages 2688–2694, Québec City, QC, Canada, 2014. The AAAI Press.

33 Jean-Marie Lagniez and Pierre Marquis. An improved decision-DDNF compiler. In Carles Sierra, editor, *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, pages 667–673, Melbourne, VIC, Australia, 2017. The AAAI Press.

34 Norbert Manthey. Towards next generation sequential and parallel SAT solvers. *KI - Kuenstliche Intelligenz*, 30(3-4):339–342, 2016. doi:10.1007/s13218-015-0406-8.

35 Sheila A. Muise, Christian J .and McIlraith, J. Christopher Beck, and Eric I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. In Leila Kosseim and Diana Inkpen, editors, *Proceedings of the 25th Canadian Conference on Artificial Intelligence (AI'17)*, volume

7310 of *Lecture Notes in Computer Science*, pages 356–361, Toronto, ON, Canada, 2012. Springer Verlag. doi:10.1007/978-3-642-30353-1_36.

36 Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pages 3141–3148. The AAAI Press, 2015.

37 Jonathan Passerat-Palmbach and David Hill. *OpenCL: A suitable solution to simplify and unify high performance computing developments*, chapter 8. Saxe-Coburg Publications, 2013.

38 Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1–2), 1996. doi:10.1016/0004-3702(94)00092-1.

39 Sigve Hortemo Sæther, Jan Arne Telle, and Martin Vatshelle. Solving #SAT and MAXSAT by dynamic programming. *Journal of Artificial Intelligence Research*, 54:59–82, 2015.

40 Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50—-64, 2010. doi:10.1016/j.jda.2009.06.002.

41 Tian Sang, Fahiem Bacchus, Paul Beame, Henry Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In Holger H. Hoos and David G. Mitchell, editors, *Online Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, Vancouver, BC, Canada, 2004.

42 Tian Sang, Paul Beame, and Henry Kautz. Performing bayesian inference by weighted model counting. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings of the 29th National Conference on Artificial Intelligence (AAAI'05)*. The AAAI Press, 2005.

43 Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. In Jonathan Katz and Hovav Shacham, editors, *Proceedings of the 37th Annual International Cryptology Conference (Advances in Cryptology – CRYPTO'17)*, volume 10401 of *Lecture Notes in Computer Science*, pages 570–596, Santa Barbara, CA, USA, 2017. Springer Verlag. doi:10.1007/978-3-319-63688-7_19.

44 Marc Thurley. sharpSAT – counting models with advanced component caching and implicit BCP. In Armin Biere and Carla P. Gomes, editors, *Proceedings of the 9th International Conference Theory and Applications of Satisfiability Testing (SAT'06)*, pages 424–429, Seattle, WA, USA, 2006. Springer Verlag. doi:10.1007/11814948_38.

45 Takahis Toda and Takehide Soh. Implementing efficient all solutions SAT solvers. *ACM Journal of Experimental Algorithmics*, 21:1.12, 2015. Special Issue SEA 2014, Regular Papers and Special Issue ALENEX 2013.

46 Tom C. van der Zanden and Hans L. Bodlaender. Computing treewidth on the GPU. In Daniel Lokshtanov and Naomi Nishimura, editors, *Proceedings of the 12th International Symposium on Parameterized and Exact Computation (IPEC'17)*, volume 89 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:13, Dagstuhl, Germany, 2018. Dagstuhl Publishing. doi:10.4230/LIPIcs.IPEC.2017.29.

47 Yexiang Xue, Arthur Choi, and Adnan Darwiche. Basing decisions on sentences in decision diagrams. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI'12)*, Toronto, ON, Canada, 2012. The AAAI Press.

48 Moshe Zadka and Guido van Rossum. PEP 237 – unifying long integers and integers. https://www.python.org/dev/peps/pep-0237/, 2001.