Exploiting Database Management Systems and Treewidth for Counting*

Johannes K. Fichte[1][0000−0002−8681−7470], Markus Hecher[2,3][0000−0003−0131−6771], Patrick Thier[2][TODO0000−0003−0131−6771], and Stefan Woltran[2][TODO0000−0003−0131−6771]

[1] TU Dresden, Germany johannes.fichte@tu-dresden.de
[2] TU Wien, Austria {hecher,woltran,thier}@dbai.tuwien.ac.at
[3] University of Potsdam, Germany hecher@uni-potsdam.de

Abstract Bounded treewidth is one of the most cited combinatorial invariants, which was applied in the literature for solving several counting problems efficiently. Counting problems in the counting complexity class #P are considered to be extremely hard, since one can solve any problem of the polynomial hierarchy by means of a polynomial number of calls to a #P oracle. A canonical #P-complete problem is #Sat, which asks to count the satisfying assignments of a propositional formula. Recent work shows that benchmarking instances for #Sat often have reasonably small treewidth. Hence, algorithms that exploit small treewidth are particularly suited to solve #Sat. This paper deals with counting problems for instances of small treewidth. We introduce a general framework to solve counting questions based on state-of-the-art database management systems (DBMS). Our framework takes explicitly advantage of small treewidth by solving instances using dynamic programming (DP) on tree decompositions (TD). Therefore, we implement the concept of DP into a DBMS (PostgreSQL), since DP algorithms are already often given in terms of table manipulations in theory. This allows for elegant specifications of DP algorithms and the use of SQL to manipulate records and tables, which gives us a natural approach to bring DP algorithms into practice. To the best of our knowledge, we present the first approach to employ a DBMS for algorithms on TDs. A key advantage of our approach is that DBMS naturally allow to deal with huge tables with a limited amount of main memory (RAM), parallelization, as well as suspending and resuming computation.

Keywords: Dynamic Programming · Parameterized Algorithmics · Bounded Treewidth · Counting · Database Management Systems

1 Introduction

The *model counting problem* (#Sat) asks to compute the number of solutions of a propositional formula. A natural generalization of #Sat is weighted model

write following part before "Contribution"

* Our tool gpusat2 is available under GPL3 license at github.com/hmarkus/dp_on_dbs.

counting (WMC), where formulas are extended by weights. Both #SAT and WMC are special cases of the weighted constraint satisfaction problem [**?**,**?**]. Nonetheless, they can already be used to solve a variety of applications to real-world questions in modern society, reasoning, and combinatorics [**?**,**?**,**?**,**?**,**?**]. Both #SAT and WMC are known to be complete for the class #P [**?**,**?**].

In this paper, we consider both problems from the practical perspective. We present and evaluate a new version of a parallel model counter, called gpusat2, which is based on *dynamic programming (DP)* on tree decompositions (TDs) [**?**]. The idea of solving #SAT decomposing graph representations of the formula and applying DP on them is in fact quite well-known [**?**] and has earlier already been introduced for the constraint satisfaction problem (CSP) by Kask et al. [**?**]. Its underlying ideas are as follows. A TD of a propositional formula F is defined on a graph representation of F and formalizes a certain static relationship of the variables of F among each other. The decomposition then gives rise to an evaluation order and to sets of variables, which define which variables have to be evaluated together when solving the given formula. Intuitively, the width of a TD indicates how many variables have to be considered exhaustively together during the computation. Our previous solver gpusat1 already implements DP-based weighted model counting and model counting using uniform weights on a GPU [**?**]. Prior to this, Fioretto et al. [**?**] presented an approach and implementation to compute one solution in weighted CSP, which could also be extended to solve the sum-of-products problem[4]. Here, we focus on an efficient computation and implementation of #SAT solving by introducing a novel architecture in our solver gpusat2. We focus on the so-called primal graph as graph representation, even though the incidence graph [**?**] theoretically allows for smaller width (off by one), mainly because simplicity of algorithms on the primal graph often outweighs the benefits of potential smaller width [**?**,**?**]. Our solver implements the principle of parallel programming of single instructions on multiple threads (SIMT) on a GPU. Therefore, we parallelize by executing the computation of variables that have to be considered exhaustively together on multiple threads, since the computation of an assignment to these variables is independent of other assignments during DP.

Contribution. We implement a solver dpdb for solving counting problems based on dynamic programming on tree decompositions, and present the following contributions. (i) Our solver dpdb uses database management systems to efficiently handle table operations needed for performing dynamic programming efficiently. The system dpdb is written in Python and employs PostgreSQL as DBMS, but can work with other DBMSs easily. (ii) The architecture of dpdb allows to solve general problems of bounded treewidth that can be solved by by means of table operations (in form of SQL) on tree decompositions. As a result, dpdb is a generalized framework for dynamic programming on tree decompositions, where one only needs to specify the essential and problem-specific parts of dynamic programming in order to solve (counting) problems. (iii) Finally, we show how to

[4] The sum-of-product problem is often also referred to as weighted counting, partition function, or probability of evidence.

solve the canonical problem #SAT with the help of dpdb, where it seems that the architecture of dpdb is particularly well-suited. In particular, we compare the runtime of our system with state-of-the-art model counters, where we observe competitive behavior.

Related Work. lots of it

2 Preliminaries

Boolean Satisfiability. We define Boolean formulas and their evaluation in the usual way, cf., [**?**,**?**]. A literal is a Boolean variable x or its negation $\neg x$. A *CNF formula* φ is a set of *clauses*, interpreted as conjunction, which are sets of literals interpreted as disjunction. For a formula or clause X, we abbreviate by $\mathrm{var}(X)$ the variables that occur in X. An *assignment* of φ is a mapping $I : \mathrm{var}(\varphi) \to \{0, 1\}$. The formula $\varphi(I)$ *under assignment* I is obtained by removing every clause c from φ that contains a literal set to 1 by I, and removing from every remaining clause of φ all literals set to 0 by I. An assignment I is *satisfying* if $\varphi(I) = \emptyset$. *Problem* #SAT asks to output the number of satisfying assignments of a formula.

Tree Decomposition and Treewidth. A *tree decomposition (TD)* of a given graph G is a pair $\mathcal{T} = (T, \chi)$ where T is a rooted tree and χ is a mapping which assigns to each node $t \in V(T)$ a set $\chi(t) \subseteq V(G)$, called *bag*, such that (i) $V(G) = \bigcup_{t \in V(T)} \chi(t)$ and $E(G) \subseteq \{\, \{u, v\} \mid t \in V(T), \{u, v\} \subseteq \chi(t) \,\}$; and (ii) for each $r, s, t \in V(T)$, such that s lies on the path from r to t, we have $\chi(r) \cap \chi(t) \subseteq \chi(s)$. We let $\mathrm{width}(\mathcal{T}) := \max_{t \in V(T)} |\chi(t)| - 1$. The *treewidth* $\mathrm{tw}(G)$ of G is the minimum $\mathrm{width}(\mathcal{T})$ over all TDs $\mathcal{T}$ of G. For a node $t \in V(T)$, we say that $\mathrm{type}(t)$ is *leaf* if t has no children and $\chi(t) = \emptyset$; *join* if t has children t' and t'' with $t' \neq t''$ and $\chi(t) = \chi(t') = \chi(t'')$; *intr* ("introduce") if t has a single child t', $\chi(t') \subseteq \chi(t)$ and $|\chi(t)| = |\chi(t')| + 1$; *rem* ("removal") if t has a single child t', $\chi(t') \supseteq \chi(t)$ and $|\chi(t')| = |\chi(t)| + 1$. If for every node $t \in N$, $\mathrm{type}(t) \in \{\mathit{leaf}, \mathit{join}, \mathit{intr}, \mathit{rem}\}$, then the TD is called *nice*.

Relational Algebra. We use relational algebra [**?**] for manipulation of relations, which forms the theoretical basis of its the well-known implementation database standard *Structured Query Language (SQL)* [] on tables. An *attribute* a is of a certain finite *domain* $\mathrm{dom}(a)$. Then, a *tuple* r over set $\mathrm{att}(r)$ of attributes, is a set of pairs of the form (a, v) with $a \in \mathrm{att}(r), v \in \mathrm{dom}(a)$ s.t. for each $a \in \mathrm{att}(r)$, there is exactly one $v \in \mathrm{dom}(a)$ with $(a, v) \in r$. A *relation* R is a finite set of tuples r over set $\mathrm{att}(R) := \mathrm{att}(r)$ of attributes. Given a relation R over $\mathrm{att}(R)$. Then, we let $\mathrm{dom}(R) := \bigcup_{a \in \mathrm{att}(R)} \mathrm{dom}(a)$, and let relation R *projected to* $A \subseteq \mathrm{att}(R)$ be given by $\Pi_A(R) := \{r_A \mid r \in R\}$, where $r_A := \{(a, v) \mid (a, v) \in r, a \in A\}$. We define *renaming* of R given set A of attributes, and a bijective mapping $m : \mathrm{att}(R) \to A$ s.t. $\mathrm{dom}(a) = \mathrm{dom}(m(a))$ for $a \in \mathrm{att}(R)$, by $\rho_m(R) := \{(m(a), v) \mid (a, v) \in R\}$. *Selection* of rows in R according to a given Boolean formula φ mh:extended projection

Listing 2: Table algorithm $\mathsf{S}(t, \chi(t), \varphi_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for #SAT using nice TDs.

In: Node t, bag $\chi(t)$, clauses φ_t, sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables. **Out:** Table τ_t.

if $\mathrm{type}(t) = \mathit{leaf}$ **then** $\tau_t := \{\langle \emptyset, 1 \rangle\}$
else if $\mathrm{type}(t) = \mathit{intr}$, *and* $a \in \chi(t)$ *is introduced* **then**
  $\tau_t := \{\langle J, c \rangle \mid \langle I, c \rangle \in \tau_1, J \in \{I^+_{a \mapsto 0}, I^+_{a \mapsto 1}\}, \varphi_t(J) = \emptyset\}$
else if $\mathrm{type}(t) = \mathit{rem}$, *and* $a \notin \chi(t)$ *is removed* **then**
  $\tau_t := \{\langle I^-_a, \Sigma_{\langle J, c \rangle \in \tau_1 : I^-_a = J^-_a} c \rangle \mid \langle I, \cdot \rangle \in \tau_1\}$
else if $\mathrm{type}(t) = \mathit{join}$ **then**
  $\tau_t := \{\langle I, c_1 \cdot c_2 \rangle \mid \langle I, c_1 \rangle \in \tau_1, \langle I, c_2 \rangle \in \tau_2\}$

$I^-_e := I \setminus \{e \mapsto 0, e \mapsto 1\}$, $I^+_e := I \cup \{e\}$.

with equality[5] is defined by $\sigma_\varphi(R) := \{r \mid r \in R, \varphi(r^E) = \emptyset\}$, where r^E is a truth assignment over $\mathrm{var}(\varphi)$ such that for each $v, v', v'' \in \mathrm{dom}(R) \cup \mathrm{att}(R)$ (1) $r^E(v \approx v') = 1$ if $(v, v') \in r$, (2) $r^E(v \approx v) = 1$, (3) $r^E(v \approx v') = r^E(v' \approx v)$, and (4) if $r^E(v \approx v') = 1$, and $r^E(v' \approx v'') = 1$, then $r^E(v \approx v'') = 1$. Given a relation R' with $\mathrm{att}(R') \cap \mathrm{att}(R) = \emptyset$. Then, we refer to the *Cartesian product* by $R \times R' := \{r \cup r' \mid r \in R, r' \in R'\}$. Further, wet let *$\theta$-join* correspond to $R \bowtie_\varphi R' := \sigma_\varphi(R \times R')$.

mh: group by

3 Dynamic Programming on TDs with Relational Algebra

A solver based on *dynamic programming (DP)* evaluates the input $\mathcal{I}$ in parts along a given TD of a graph representation G of the input. Thereby, for each node t of the TD, intermediate results are stored in a *table* τ_t. This is achieved by running a so-called *table algorithm* A, which is designed for a certain graph representation, and stores in τ_t results of problem parts of $\mathcal{I}$, thereby considering tables $\tau_{t'}$ for child nodes t' of t. The DP approach works for many problems $\mathcal{P}$ as follows.

1. Construct a graph representation G of the given input instance $\mathcal{I}$.
2. Heuristically compute a tree decomposition $\mathcal{T} = (T, \chi)$ of G.
3. Traverse the nodes in $V(T)$ in post-order, i.e., perform a bottom-up traversal of T. At every node t during post-order traversal, execute a table algorithm A that takes as input t, bag $\chi(t)$, a *local problem* $\mathcal{P}(t, \mathcal{I})$ depending on $\mathcal{P}$, as well as previously computed tables of children of t and stores the result in τ_t.
4. Interpret table τ_n for the root n of T in order to output the solution of $\mathcal{I}$.

For solving problem $\mathcal{P} = \#\textsc{Sat}$, we need the following graph representation. The *primal graph* G_φ [**?**] of a formula φ has as vertices its variables, where two variables are joined by an edge if they occur together in a clause of φ. Given formula φ, a TD $\mathcal{T} = (T, \chi)$ of G_φ and a node $t \in T$. Then, we let local problem $\#\textsc{Sat}(t, \varphi)$ be $\varphi_t := \{\, c \mid c \in \varphi, \mathrm{var}(c) \subseteq \chi(t) \,\}$, which are the clauses entirely covered by $\chi(t)$.

[5] We allow for φ to contain expressions $v \approx v'$ as variables for $v, v' \in \mathrm{dom}(R) \cup \mathrm{att}(R)$, and we abbreviate for $v \in \mathrm{att}(R)$ with $\mathrm{dom}(v) = \{0, 1\}$, $v \approx 1$ by v and $v \approx 0$ by $\neg v$.

Listing 3: Table algorithm $\mathsf{S}_{\mathsf{RAlg}}(t, \chi(t), \varphi_t, \langle \tau_1, \ldots, \tau_\ell \rangle)$ for #SAT using nice TDs.

In: Node t, bag $\chi(t)$, clauses φ_t, sequence $\langle \tau_1, \ldots \tau_\ell \rangle$ of child tables. **Out:** Table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t := \{\{(cnt, 1)\}\}$

2 **else if** type$(t) =$ *intr, and* $a \in \chi(t)$ *is introduced* **then**

3 $\quad \tau_t := \tau_1 \bowtie_{\varphi_t} \{\{([\![a]\!], 0)\}, \{([\![a]\!], 1)\}\}$

4 **else if** type$(t) =$ *rem, and* $a \notin \chi(t)$ *is removed* **then**

5 $\quad \tau_t := {}_{\chi(t)}G_{\mathrm{SUM}(cnt)}(\Pi_{\chi(t)^+_{cnt}} \tau_1)$

6 **else if** type$(t) =$ *join* **then**

7 $\quad \tau_t := \Pi_{\chi(t)^+_{cnt}}[\dot{\Pi}_{\{cnt \leftarrow cnt_1 \cdot cnt'\}}(\rho_{\{cnt \mapsto cnt_1\}}\tau_1 \bowtie_{\bigwedge_{a \in \chi(t)} a \approx a'} \rho_{\bigcup_{a \in \chi(t)^+_{cnt}} \{[\![a]\!] \mapsto [\![a]\!]'\}}\tau_2)]$

$I^+_e := I \cup \{e\}$.

Example 1. Table algorithm S as presented in Listing 2 shows all the cases that are needed to solve #SAT by means of DP of nice TDs. Each table τ_t consist of rows of the form $\langle \alpha, c \rangle$, where α is an assignment of φ_t and c is a counter. Nodes t with type$(t) =$ *leaf* consist of the empty assignment and counter 1, cf., Line 1. For a node t with introduced variable $a \in \chi(t)$, we guess in Line 3 for each assignment β of the child table, whether a is set to true or to false, and ensure that φ_t is satisfied. When an atom a is removed in node t, we project assignments of child tables to $\chi(t)$, cf., Line 5, and counters of the same assignments are summed up. For join nodes t, counters of common assignments in the child tables are multiplied as in Line 7. ■

Towards Relational Algebra. Instead of using set theory to describe how tables are obtained during dynamic programming are performed, one could alternatively use relational algebra. There, tables τ_t for each TD node t are pictured as relations, where τ_t distinguishes a unique column (attribute) $[\![x]\!]$ for each $x \in \chi(t)$. Further, there might be additional attributes required depending on the problem at hand, e.g., we need an attribute *cnt* for counting in #SAT, or an attribute for modeling costs or weights in case of optimization problems. Listing 3 presents a table algorithm for problem #SAT that is equivalent to Listing 2, but relies on relational algebra only for computing tables. This step from set notation to relational algebra is driven by the observation that in these table algorithms one can identify recurring patterns. In particular, one typically derives for nodes t with type$(t) =$ *leaf*, a fresh initial table τ_t, cf., Line 1 of Listing 3. Then, whenever an atom a is introduced, such algorithms often use θ-joins with a fresh initial table for the introduced variable a that represent potential values a can have. In Line 2 the selection of the θ-join is performed by ensuring φ_t, corresponding to the local problem of #SAT. Further, for nodes t with type$(t) =$ *rem*, these table algorithms oftentimes need projection. In case of Listing 3, Line 4 also needs grouping in order to maintain the counter, as several rows of τ_1 might collapse in τ_t. Finally, for a node t with type$(t) =$ *join*, we use again θ-joins (and extended projection for maintaining counters), which allows us to leverage database technology of the last decades.

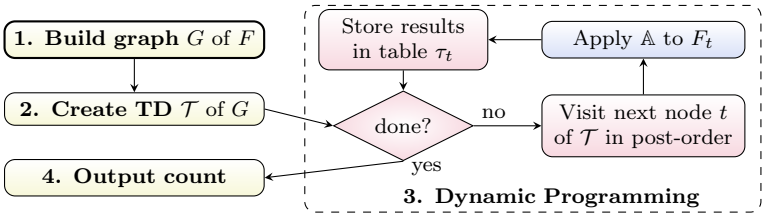

Figure 1: Architecture of Dynamic Programming with Databases.

4 Dynamic Programming on TDs using Databases & SQL

In this section, we present a general architecture to model table algorithms by means of a procedural *generator template* that describes how to dynamically obtain SQL code, which is a specific implementation standard of relational algebra, for each node t, which heavily depends on $\chi(t)$. The generated SQL code is then used for manipulation of tables τ_t during the tree decomposition traversal of dynamic programming.

Problem

- events: before solve (table erstellen), after solve (table drop)
- dynamic sql: introduce (table), filter (where condition), join (where inside), assignment view (for forgetting), candidate extra cols (describe how to add them)
- setup: prepare input / graph, prepare tables
- candidates: grundmenge

5 System `dpdb`: Dynamic Programming with Databases

We implemented the proposed architecture of the previous section in the `dpdb` system.

In the following, we discuss implementation specifics that are crucial.

Target Database.

Normalization. First of all, for reasons of solving performance later, we need to interleave the/

Node Workers.

JOIN Kinder.

Query Plan Optimizer.

6 Experiments

We conducted a series of experiments using several benchmark sets for model counting and weighted model counting. Benchmark sets [?] and our results [?] are publicly available and also on github at daajoe/gpusat_experiments.

Measure, Setup, and Resource Enforcements. As we use different types of hardware in our experiments and other natural measures such as power consumption cannot be recorded with current hardware, we compare wall clock time and number of timeouts. In the time we include, if applicable, *preprocessing time* as well as *decomposition time* for computing 30 decompositions with a random seed and decomposition selection time. However, we avoid IO access on the CPU solvers whenever possible, i.e., we load instances into the RAM before we start solving. For parallel CPU solvers we allow access to 12 or 24 physical cores on machines where hyperthreading was disabled. We set a timeout of 900 seconds and limited available RAM to 14 GB per instance and solver.

Benchmark Instances. We considered a selection of overall 1494 instances from various publicly available benchmark sets for model counting consisting of `fre/meel` benchmarks[6](1480 instances), and `c2d` benchmarks[7] (14 instances). For WMC, we used the overall 1091 instances from the `Cachet` benchmark set[8].

Benchmarked Solvers. In our experimental work, we present results for the most recent versions of publicly available #SAT solvers, namely, *c2d* 2.20 [?], *d4* 1.0 [?], *DSHARP* 1.0 [?], *miniC2D* 1.0.0 [?], *cnf2eadt* 1.0 [?], *bdd_minisat_all* 1.0.2 [?], and *sdd* 2.0 [?] (based on knowledge compilation techniques). We also considered rather recent approximate solvers *ApproxMC2*, *ApproxMC3* [?] and *sts* 1.0 [?], as well as CDCL-based solvers *Cachet* 1.21 [?], *sharpCDCL*[9], and *sharpSAT* 13.02 [?]. Finally, we also included multi-core solvers *gpusat* 1.0 [?], as well as *countAntom* 1.0 [?] on 12 physical CPU cores, which performed better than on 24 cores. Note that we benchmarked additional solvers, which we omitted from the presentation here and where we placed results online in our result data repository. For WMC, we considered the following solvers: *sts*, *gpusat1*, gpusat2, *miniC2D*, *Cachet*, *d4*, and *d-DNNF reasoner* 0.4.180625 (on top of d4 as underlying knowledge compiler). All experiments were conducted with default solver options. For solver gpusat2, we also benchmarked variant gpusat2(A+B) where we used 30 as threshold above which we apply the BST.

Benchmark Machines. The non-GPU solvers were executed 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. The results

[6] See: tinyurl.com/countingbenchmarks
[7] See: reasoning.cs.ucla.edu/c2d
[8] See: cs.rochester.edu/u/kautz/Cachet
[9] See: tools.computational-logic.org

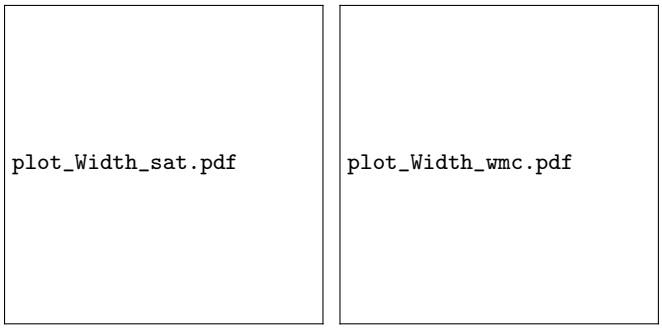

Figure 2: Width distribution of #SAT instances (left) before and after preprocessing (using both B+E and pmc). Width distribution of WMC instances (right) before and after preprocessing using pmc*. Results are based on the primal treewidth and presented in intervals. X-axis labels the intervals, y-axis labels the number of instances.

prob	pre	vMdn	cMdn	t[s] Mdn	to	t[s] Mdn pre	to	Mdn	50%	80%	90%	95%	min	max	mdn
#SAT	w/o pre	637	810	0.07	**6**	n/a	n/a	31	31	166	378	922	n/a	n/a	n/a
	pmc, B+E	**231**	350	0.02	**6**	0.06	192	**3**	**3**	**17**	201	**823**	-72	**755**	22
	pmc	**231**	189	0.03	**6**	**0.03**	**103**	**3**	4	19	228	**823**	-1839	547	**23**
	B+E	**231**	**185**	**0.02**	**6**	0.04	189	**3**	**3**	18	**192**	**823**	**-2**	633	**23**
WMC	w/o pre	**200**	519	0.04	**0**	n/a	n/a	28	28	40	43	54	n/a	n/a	n/a
	pmc*	**200**	**300**	**0.03**	**0**	**0.03**	**0**	**11**	**11**	**20**	**25**	**30**	**0**	**330**	**16**

Table 1: Overview on upper bounds of the primal treewidth for considered #SAT and WMC benchmarks before and after preprocessing. vMdn median of variables, cMdn median of clauses, t[s] Mdn of the decomposition runtime in seconds, maximum runtime t[s] Max, median Mdn and percentiles of upper bounds on treewidth, and min/max/mdn of the width improvement after preprocessing. Negative values indicate worse results.

were gathered on Ubuntu 16.04.1 LTS machines with disabled hyperthreading on kernel 4.4.0-139, which is already a post-Spectre and post-Meltdown kernel[10]. For `gpusat1` and `gpusat2` 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 running at 1.24 GHz with 32 compute units, 2048 shader units, and 4GB VRAM using driver amdgpu-pro-18.30-641594 and OpenCL 1.2. The system operated on Ubuntu 18.04.1 LTS with kernel 4.15.0-34.

6.1 Results

First, we present how existing preprocessors for #SAT and equivalence-preserving preprocessors for WMC influence the treewidth on the considered instances.

Treewidth Analysis. We computed upper bounds on the primal treewidth for our benchmarks before and after preprocessing and state them in intervals. For

[10] Details on spectre and meltdown: spectreattack.com.

model-count preserving preprocessing we explored both B+E Apr2016 [?] and pmc 1.1 [?]. For WMC, we used pmc with documented options −vivification −eliminateLit −litImplied −iterate = 10 to preserve all the models, which we refer to by *pmc**. In this experiment, we used different timeouts. We set the timeout of the preprocessors to 900 seconds and allowed further 1800 seconds for the decomposer to get a detailed picture of treewidth upper bounds. Figure 2 (left) presents the width distribution of number of instances (y-axis) and their corresponding upper bounds (x-axis) for primal treewidth, both before and after preprocessing using B+E, pmc, and both preprocessors in combination (first pmc, then B+E) for #SAT. Table 1 (top) provides statistics on the benchmarks combined, including runtime of the preprocessor, runtime of the decomposer to obtain a decomposition, upper bounds on primal treewidth, and its improvements before and after preprocessing. Further, the table also lists the median of the widths of the obtained decompositions and their percentiles, which is the treewidth upper bound a given percentage of the instances have. Interestingly, overall we have that a majority of the instances after preprocessing has width below 20. In more details, more than 80% of the #SAT instances have primal treewidth below 19 after preprocessing, whereas 90% of the instances have treewidth below 192 for B+E. With pmc we observed a corner case where the primal treewidth upper bound increased by 1839, however, on average we observed a mean improvement on the upper bound of slightly above 23. The best improvement among the widths of all our instances was achieved with the combination of pmc and B+E where we improved the width by 755. Overall, both B+E and pmc managed to *drastically reduce* the widths, the decomposer ran below 0.1 seconds in median. Interestingly, even the upper bounds on the treewidth of the WMC instances reduced with pmc* as depicted in Figure 2 (right). In more detail, after preprocessing 95% of the instances have primal treewidth below 30, c.f., Table 1 (bottom).

Solving Performance Analysis. Figure 3 illustrates the top five sequential solvers, and all parallel counting solvers with preprocessor pmc in a cactus-like plot. Table 2 presents detailed runtime results for #SAT with preprocessors pmc, B+E, and without preprocessing, respectively. Since the solver sts produced results that varied from the correct result on average more than the value of the correct result, we excluded it from the presented results. If we disallow preprocessing, gpusat2 and gpusat1 perform only slightly better in the overall standing of the solvers. But gpusat2 solves 42 instances more and requires about 10 hours less of wallclock time. Further, we can observe, that the variant gpusat2(*A*+*B*) performs particular well, mainly since for instances below width 30, the BST compression seems relatively expensive compared to the array data structure. Interestingly, when considering the results on preprocessing in Table 2 (top, mid) and Figure 3 we observe that the architectural improvements pay off quite well. gpusat2 can solve the vast majority of the instances and ranks second place. If one uses the B+E preprocessor shown in Table 2 (mid), gpusat2 solves even more instances as well as the other solvers. Still, it ranks fifth solving only 26 instances less

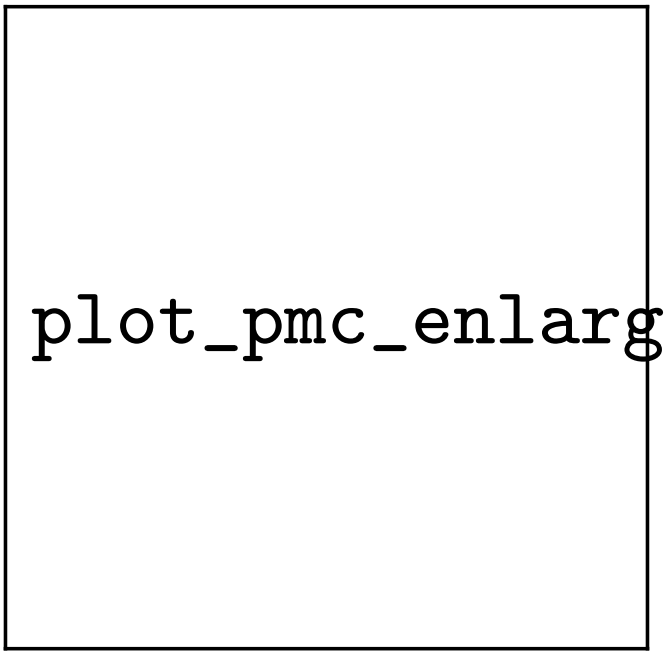

Figure 3: Runtime for the top 5 sequential and all parallel solvers over all the #SAT instances with pmc preprocessor. The x-axis refers to the number of instances and the y-axis depicts the runtime sorted in ascending order for each solver individually.

than the best solver and 10 less than the third best solver and solves the most instances having width below 30.

While we focus on #SAT with our implementation, we also conducted the experiments with WMC in order to compare our solver with gpusat1 in the setting for which it was designed. Table 3 (top) lists results of the top five best solvers capable of solving WMC on our instances. Compared to gpusat1, our solver gpusat2 shows an improvement when the width of the instance is between 21 and 40, in more detail gpusat2 solves 44 instances more. After preprocessing with pmc*, one can observe that the majority of the instances has width below 20, c.f., Table 3 (bottom). As a result, gpusat2 does not provide significant improvement over gpusat1 there apart from small runtime improvements.

Currently, we are unable to measure the speed-up of the implementations in terms of the used cores, mainly due to the fact that we aimed for an implementation that is close to gpusat1 so that the improvements are actually from the architectural changes and not just from the different framework or drivers. Note that OpenCL does not support disabling certain cores on the GPU. We also benchmarked gpusat2 on an Nvidia GPU, whose runtimes are quite similar. We also provide preliminary data online with the experiments; which are however not conclusive yet. However, we we ran into bugs, which seems to be attributed to the OpenCL1.2 Nvidia driver. Therefore, we aim as future work for a new implementation in CUDA [**?**].

	solver	0-20	21-30	31-40	41-50	51-60	>60	best	unique	∑	time[h]
pmc preprocessing	miniC2D	1193	29	**10**	2	1	7	13	0	**1242**	**68.77**
	gpusat2	**1196**	32	1	0	0	0	250	**8**	1229	71.27
	d4	1163	20	**10**	2	**4**	28	52	1	1227	76.86
	gpusat2(A+B)	1187	18	1	0	0	0	120	7	1206	74.56
	countAntom 12	1141	18	**10**	**5**	**4**	13	101	0	1191	84.39
	c2d	1124	31	**10**	3	3	10	20	0	1181	84.41
	sharpSAT	1029	16	**10**	2	**4**	**30**	**253**	1	1091	106.88
	gpusat1	1020	16	0	0	0	0	106	7	1036	114.86
	sdd	1014	4	7	1	0	2	0	0	1028	124.23
	solver	0-20	21-30	31-40	41-50	51-60	>60	best	unique	∑	time[h]
B+E preprocessing	c2d	1199	24	**9**	0	2	23	14	0	**1257**	**63.46**
	miniC2D	1203	**27**	8	0	2	12	8	0	1252	64.92
	d4	1182	15	**9**	**1**	**3**	31	79	1	1241	69.32
	countAntom 12	1177	14	8	0	2	**34**	100	0	1235	69.79
	gpusat2	**1204**	26	1	0	0	0	**150**	**3**	1231	68.15
	gpusat2(A+B)	1201	21	1	0	0	0	67	**3**	1223	70.39
	sdd	1106	11	4	**1**	1	4	0	0	1127	100.48
	gpusat1	1037	16	0	0	0	0	87	**3**	1053	110.87
	bdd_minisat_all	926	6	3	**1**	1	0	101	0	937	140.59
	solver	0-20	21-30	31-40	41-50	51-60	>60	best	unique	∑	time[h]
without preprocessing	countAntom 12	118	511	139	**175**	**21**	**181**	318	15	**1145**	**96.64**
	d4	124	514	148	162	**21**	168	69	15	1137	104.94
	c2d	119	525	**165**	161	18	120	48	15	1108	110.53
	miniC2D	122	514	128	149	9	62	0	0	984	141.22
	sharpSAT	100	467	124	156	12	123	**390**	4	982	135.41
	gpusat2(A+B)	**125**	**539**	96	138	0	0	94	**19**	898	151.16
	gpusat2	**125**	523	96	138	0	0	78	17	882	155.43
	gpusat1	**125**	524	67	140	0	0	82	9	856	162.03
	cachet	99	430	71	152	8	57	3	0	817	176.26
	solver	0-20	21-30	31-40	41-50	51-60	>60	best	unique	∑	time[h]

Table 2: Number of #SAT instances (grouped by treewidth upper bound intervals) solved by sum of the top five sequential and all parallel counting solvers with preprocessor pmc (top), B+E (mid), and without preprocessing (bottom). time[h] is the cumulated wall clock time in hours, where unsolved instances are counted as 900 seconds.

7 Related Work and Conclusion

Related Work. Fioretto et al. [**?**] introduced a solver for outputting a solution to the weighted CSP problem using a GPU. Their technique is effectively a version of dynamic programming on tree decompositions also known as bucket-elimination, which they limited to an incomplete elimination by introducing shortcuts and discarding non-optimal solutions in order to speed up the computation for the problem of outputting just one solution. While the underlying idea of a dynamic programming based solver still exists in our solver, gpusat2 is very different when just taking a slightly more detailed look. We approach the *counting question – not just outputting one solution*, which disallows certain simplifications. We can neither apply an incomplete bucket-elimination technique (mini-bucket elimination) nor discard non-optimal solutions. But then, we consider the binary case, which allows us to introduce various simplifications including the way we store the data enabling us to save memory and to avoid copying data. Also, we

	solver	0-20	21-30	31-40	41-50	51-60	>60	best	unique	∑	time[h]
with pmc*	miniC2D	858	**164**	**6**	**0**	**0**	3	13	**8**	**1031**	21.29
	gpusat1	**866**	158	0	**0**	**0**	0	**348**	4	1024	18.03
	gpusat2(A+B)	**866**	156	0	**0**	**0**	0	343	4	1022	**17.86**
	gpusat2	**866**	138	0	**0**	**0**	0	299	4	1004	22.43
	d4	810	106	0	**0**	**0**	0	46	0	916	55.36
	cachet	617	128	1	**0**	**0**	3	106	1	749	93.65
without pre	d4	82	501	**142**	**156**	**10**	**19**	111	**24**	**910**	**53.97**
	miniC2D	84	517	134	152	3	4	19	7	894	59.69
	gpusat2(A+B)	**86**	**527**	98	138	0	0	167	19	849	64.40
	gpusat2	**86**	511	98	138	0	0	131	7	833	68.61
	gpusat1	**86**	513	68	140	0	0	**182**	10	807	73.78
	cachet	60	447	100	145	2	9	118	1	763	89.80

Table 3: Number of WMC instances solved (with)out preprocessing. time[h] is the cumulated wall clock time in hours, where unsolved instances count as 900 seconds.

would like to point out that bucket-elimination is used in the decomposer htd to compute just the tree decomposition. In that way, our architecture is quite general as it completely separates the decomposition and the actual computation part resulting in a framework that can also be used for other problems. Moreover, we use more sophisticated data structures and split data whenever the data does not fit into the VRAM of the GPU. Finally, we balance between small width during the computation and not too small width as we want to employ the full computational power of the parallelization with the GPU. In the past, a variety of model counters and weighted model counters have been implemented based on several different techniques. We listed them in details in Section 6. However, here we want to highlight a few differences between our technique and knowledge compilation-based techniques as well as distributed computing. The solver d4 [**?**], which implements a knowledge compilation-based approach, employs heuristics to compute decompositions of an underlying hypergraph, namely the dual hypergraph, and uses this during the computation. Note that the following relationships are known for treewidth (i.e., the width of a tree decomposition of smallest width) of an arbitrary formula F, $\mathrm{inctw}(F) \leq \mathrm{dualtw}(F) + 1$ and $\mathrm{inctw}(F) \leq \mathrm{primtw}(F) + 1$, where inctw refers to the treewidth of the incidence graph, dualtw of the dual graph, and primtw of the primal graph. However, there is no such relationship between the treewidth of the primal and dual graph. We are currently unaware of how these theoretical results generalize to hypergraphs. Experimentally, it is easy to verify that a decomposition of the dual graph is often not useful in our context as it provides only decompositions of large width. When we consider parallel solving, a few words on distributed counting are in order. In fact, the model counter DMC [**?**] is intended for parallel computation on a cluster of computers using the message passing model (MPI). However, this distributed computation requires a separate setup of the cluster and exclusive access to multiple nodes. We focus on parallel counting with a shared memory model. For details, we refer to the difference between parallel and distributed computation [**?**].

Conclusion. We presented an improved OpenCL-based solver gpusat2 for solving #Sat and WMC. Compared to the weighted model counter gpusat1 that uses the GPU, our solver gpusat2 implements adapted memory management, specialized data structures on the GPU, improved data type precision handling, and an initial approach to use customized TDs. We carried out rigorous experimental work, including establishing upper bounds for treewidth after preprocessing of commonly used benchmarks and comparing to most recent solvers.

Future Work. Our results give rise to several research questions. Since established preprocessors are mainly suited for #Sat, we are interested in additional preprocessing methods for weighted model counting (WMC) that reduce the treewidth or at least allow us to compute TDs of smaller width. It would also be interesting whether GPU-based techniques can successfully be used within knowledge compilation-based model counters. An interesting research direction is to study whether efficient data representation techniques can be combined with dynamic programming in order to lift our solver to counting in WCSP [**?**]. Further, we are also interested in extending this work to projected model counting [**?**,**?**,**?**].