gpusat2 – An Improved GPU Model Counter*

Johannes K. Fichte[1], Markus Hecher[2,3], and Markus Zisser[1]

[1] TU Dresden, Germany
[2] Technische Universität Wien, Austria
[3] University of Potsdam, Germany
{markus.hecher,markus.zisser}@tuwien.ac.at
johannes.fichte@tu-dresden.de

Abstract In this paper, we present and evaluate a new parallel propositional model counter, called gpusat2, which is based on dynamic programming (DP) on tree decompositions using log-counters. gpusat2 implements the principle of single instructions on multiple threads (SIMT) for parallel programming on a GPU using the vendor-independent programming framework OpenCL. We introduce a novel architecture that includes variable data storages and compressing solution parts. We outline results to experiments where we compare the runtime of our system with state-of-the-art model counters on standard benchmark sets for model counting. In particular, we test also other parallel solvers and the predecessor of gpusat2. One major outcome is that the novel architecture significantly improves our solver over its predecessor. As a side result, we observe that state-of-the-art preprocessors allow to produce tree decompositions of significantly smaller width.

1 Introduction

The *model counting problem* (#Sat) asks to compute the number of solutions of a propositional formula and is complete for the class #·P [30]. #Sat and its generalization WMC, which also takes weights for the literals into account, have a variety of applications to real-world questions in modern society, to probabilistic reasoning, statistics, and combinatorics [9,13,14,34,37]. In this paper, we consider #Sat from the practical perspective. We present and evaluate a new parallel model counter, called gpusat2, which is based on *dynamic programming (DP)* on tree decompositions [32]. gpusat2 significantly improves over its predecessor gpusat1 [18]. Its underlying ideas are as follows. A tree decomposition 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.

* The paper is a preliminary workshop version of a paper that has been accepted for publication at the CP'2019 conference. 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.

Intuitively, the width of a tree decomposition indicates how many variables have to be considered exhaustively together during the computation. Here, we use the so-called primal graph [32] as graph representation, even though the incidence graph [32] theoretically allows for smaller width (off by one). The main reason for using the primal graph is because previous practical work indicated that the simpler solving algorithms for the primal graph often outweigh the benefits of potential smaller width [16,18]. Hence, we focus on the primal graph only. Our solver implements just as its predecessor 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 another assignment to these variables during dynamic programming. We implement the solver using the vendor-independent programming framework OpenCL [28].

For our solver gpusat2, we introduce an innovative architecture for dynamic programming that includes using customized tree decompositions [2], storing solutions to parts of the input instance during the computation variably in arrays or binary search trees depending on the width of the decomposition, and compressing sets of assignments. In addition, we avoid data transfer between the RAM and the Video RAM (VRAM) whenever possible and employ extended preprocessing by means of the state-of-the-art preprocessors B+E [21] and pmc [22] for model counting and with a subset of reduction rules for weighted model counting. In order to increase the accuracy and applicability of our solver to instances with very high solution count, we store the model count during the computation by floating *log-counters*. Therefore, we take our counters during the dynamic programming in relation to an exponent e to the power of 2 and dynamically increase the exponent during the computation. Storing values by the log of the value are a common technique in the domain of probabilistic inference.

Finally, we present experimental work. We compare in detail the runtime of our system with state-of-the-art model counters on benchmarks for model counting and weighted model counting. In particular, we also test various other parallel solvers [4]. One major outcome is that the novel architecture significantly improves our solver over its predecessor, which especially becomes visible when we also take preprocessing for both versions into consideration. Then, gpusat2 is able to solve the third most instances over all considered solvers whereas its predecessor was limited to little more than 50% of the instances. As a side result, we observe that state-of-the-art preprocessors allow to produce tree decompositions of significantly smaller width. Since our techniques also work for WMC, we consider also WMC in the practical evaluation part and the predecessor of gpusat2 [18]. Our system is publicly available on github[4].

Related Work. In the past, a variety of model counters and weighted model counters have been implemented based on several different techniques. We list them in details in Section 5. However, here we want to highlight a few differences

[4] gpusat2 is publicly available under GPL3 license at github.com/daajoe/gpusat.

between our technique and knowledge compilation-based techniques as well as distributed computing. The solver d4 [23], 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 propositional 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 [24] 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 [29].

2 Preliminaries

Propositional Satisfiability. A literal is a propositional variable x or its negation $\neg x$. A *clause* is a finite set of literals, interpreted as the disjunction of these literals. A *(CNF) formula* is a finite set of clauses, interpreted as a conjunction of the 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 $\mathrm{var}(F) := \bigcup_{c \in F} \mathrm{var}(c)$. A *(partial) assignment* is a mapping $\sigma : \mathrm{var}(F) \to \{0, 1\}$. The formula $F(\sigma)$ *under assignment* σ is obtained by removing all clauses c from F that contain a literal set to 1 by σ and removing from the remaining clauses all literals set to 0 by σ. An assignment σ is *satisfying* if $F(\sigma) = \emptyset$. The 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 *bag* $\chi(t) \subseteq V(G)$ 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 $r, s, t \in T$, such that s lies on the path from r to t, we have $\chi(r) \cap \chi(t) \subseteq \chi(s)$. The *width* $\mathrm{width}(\mathcal{T})$ of $\mathcal{T}$ is $\max_{t \in V(T)} |\chi(t)| - 1$. The *treewidth* $\mathrm{tw}(G)$ of G is the minimum $\mathrm{width}(\mathcal{T})$ over all tree decompositions $\mathcal{T}$ of G. The *primal graph* P_F [32] 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 brevity, we refer by *treewidth of* a *formula* to the treewidth of its primal graph. For a given node t of a TD (T, χ) of the primal graph P_F, we let $F_t := \{ c \mid c \in F, \mathrm{var}(c) \subseteq \chi(t) \}$ be the clauses entirely covered by $\chi(t)$. The formula $F_{\leq s}$ denotes the union over all F_t for all

descendant nodes $t \in V(T)$ of s. In other words, $F_{\leq s}$ is the sub-formula of F that contains all clauses that have been entirely covered by a bag $\chi(s)$ for t and any of its descendant nodes.

Dynamic Programming on TDs. A solver based on *dynamic programming (DP)* for propositional formulas such as `gpusat1` evaluates the input formula F in parts along a given tree decomposition of the primal graph P_F. For each node t of the tree decomposition results are usually stored in a *local storage* ρ_t. The approach works in four steps as follows:

1. Construct the primal graph P_F of the input formula F.
2. Heuristically compute a tree decomposition $\mathcal{T} = (T, \chi)$ of the primal graph P_F.
3. DP: Traverse the nodes in $V(T)$ in post-order O.
   At every node $t \in O$, run an algorithm K that takes as input only the sub-formula F_t and previously computed results of its children and stores the results in ρ_t, which in turn is used by the algorithm at the parent (if exists).
4. Print the (weighted) model count by interpreting the result ρ_n, which has been computed for the root $n \in T$.

For details of the algorithm in Step 3, we refer to the literature [18,32]. Even so, we would like to mention that the algorithm intuitively returns in storage ρ_t only (weighted) model counts for the sub-formula $F_{\leq t}$ with respect to assignments to the variables in $\mathrm{var}(F_t)$, i.e., ρ_t contains a compact representation of counts up to the node t. In order to parallelize the computation of the counts in ρ_t it is sufficient to observe that the counters are entirely independent of each other and each counter in ρ_t depends only on results previously computed at the children. Since we have $2^{|\,\mathrm{var}(F_t)|}$ assignments at each node, for which we can compute the (potentially zero) counters by the very same operations, we can immediately parallelize the operations on the GPU [18] by employing a *single instruction on multiple threads (SIMT)* computation model. More detailed, the procedure K in Step 3 refers to a small program that can be executed on the GPU taking a small set of instructions but multiple input data. Such a procedure is also called *(GPU-)kernel* for this hardware architecture. The simplest possible data structure is an *array* that just contains the counts, where an assignment is addressed by the memory address of an entry in the array. However, this data structure has to be allocated on the *video RAM (VRAM)* prior to running the kernel on the GPU. This results in the situation that one might easily run out of memory due to huge space requirements.

3 An Improved GPU-based DP Architecture

In this section, we present an innovative architecture for parallel dynamic programming on the GPU. Later in Section 5, we will see that this also pays off in our implementation. Novel parts of the architecture are the *preprocessors*, tree *decomposition selection* heuristics (customized TDs), generalization to allow for adaptable, more advanced *data structures*, *caching* intermediate results on the

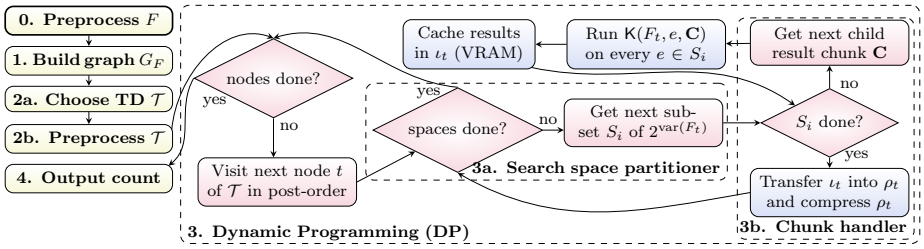

Figure 1: Architecture of our DP-based solver for parallel execution. Yellow colored boxes indicate tasks that are required as initial step for the DP-run or to finally read the model count from the computed results. The parts framed by a dashed box illustrate the DP-part. Boxes colored in red indicate computations that run on the CPU. Boxes colored in blue indicate computations that are executed on the GPU.

GPU, and the idea of *compressing* counters for assignments. Figure 1 outlines the steps of this innovative architecture. Note that the architecture is independent from the underlying data structures, i.e., in Step 3 we refer by ρ_t to a storage for data, which can be an array or another data structure. Analogously, ι_t also denotes a storage, that caches results in the VRAM for GPU computations. In the following, we discuss the novel steps of the architecture, whereas implementation details are presented in Section 4.

Step 0: Instance Preprocessing Before we decompose our instance, we *simplify the formula* F by a preprocessor for propositional formulas. There, we preserve the number of satisfying assignments or the weighted model count, respectively, and potentially decrease the treewidth of the instance F.

Step 2: Tree Decomposition In Step 2a, we heuristically compute a tree decomposition for the dynamic programming. Various recent literature suggests [19,8,3] that tree decompositions for practical solving require in addition to "small" width other criteria to speed up the performance of a solver. Such tree decompositions are frequently called *customized tree decompositions*. Therefore, we compute m different tree decompositions via heuristics [2] and then we select among the m decompositions one according to a selection criterion. In the implementation, we use the library *htd* version 1.2 with default settings [2] where $m = 30$. The selection criterion is follows. We first try to minimize the width. Then, if several decompositions of the same width are found, we select the decomposition with the smallest maximal cardinality $v(\mathcal{T})$ of the intersection of bags of any node with its children, i.e., $v(\mathcal{T}) = \max\{\, |\chi_t \cap (\chi_{t_1} \cup \chi_{t_2} \cup \ldots \cup \chi_{t_\ell})| \mid t \in V(T), t_1, t_2, \ldots, t_\ell \text{ are children of } t \text{ in } T \,\}$ where $\mathcal{T} = (T, \chi, n)$. The idea of the selection is to balance the *trade-off* between runtime and space requirements in the worst-case as outlined in earlier work [19]. In that way, we first improve on the worst-case *runtime (and VRAM consumption)* and then on the number of IO operations required to copy data between RAM and VRAM. After the selection of tree decomposition $\mathcal{T}$, we *preprocess* $\mathcal{T}$ (Step 2b). There, we combine nodes to obtain bags of size s, which is the largest number such that on the chosen

hardware 2^s GPU threads can still run in parallel. This is in order to reduce the overhead of copying data onto the VRAM and GPU thread allocation.

Step 3: Dynamic Programming As the architecture of Step 3 is more involved and consists of multiple parts, we present the step in more details. The main reason is that we are interested in getting things work also on GPUs, which are available for the consumer at home or with a small office computer at uni. In Figure 1 we marked parts colored by red and blue to distinguish between CPU and GPU computation.

Step 3a: Search Space Partitioning. As described in the preliminaries the DP proceeds by traversing a tree decomposition in post-order. At each node, we consider assignments restricted to the variables in $\mathrm{var}(F_t)$ and its corresponding counters. Overall we can have at most $2^{|\mathrm{var}(F_t)|}$ assignments ("local search space" at a node). Thus, the number of assignments can simply be too large to even store just one counter per assignment in the VRAM. In practice, we would expect that plenty of these assignments result in a counter that is zero and hence we could actually avoid the out-of-memory issue as data can be compressed. However, on the VRAM we have to allocate memory prior to the computation and hence it would require to detect the point where we run out-of-memory then to copy the data back to the RAM resulting in turn in an unutilized GPU. To avoid this situation, we simply split in Step 3a all possible assignments that are considered together at once on the GPU into several disjoint subsets $S_1, S_2, \ldots, S_k$ of $2^{|\mathrm{var}(F_t)|}$, which we call *search space partitioning*. On these grounds, we do the solution space splitting before the GPU kernels are even executed to ensure that no out-of-memory issue occurs. The splitting is independent of the actually used data structure and can be used if we store counters in an array similar to `gpusat1` or other data structures.

Step 3b: Splitting Input Result from Children and Compression. In the next step, we systematically process each set S_i for $1 \leq i \leq k$. Therefore, we consider the assignments in S_i and the corresponding counters for the children, i.e., the counters and corresponding assignments at the children which we need to compute the counter for an assignment at the currently considered node t. Since we have both to copy these relevant assignments of the children onto the VRAM and still allocate enough VRAM for S_i, we might run into the situation that both would not fit into the VRAM. Hence, we need to split for the counts and its corresponding assignments in S_i the relevant results in $\rho_{t_1}, \ldots, \rho_{t_\ell}$ computed at the child nodes $t_1, \ldots, t_\ell$ of t into subsets $C_1 \subseteq \rho_{t_1}$, …, $C_\ell \subseteq \rho_{t_\ell}$. We call a pair $\boldsymbol{C} = \langle C_1, \ldots, C_\ell \rangle$ of these subsets *chunk*. Then, the *chunk handler* systematically takes each pair $\boldsymbol{C}$ relevant for S_i and executes a kernel in a GPU thread for each element in S_i using $\boldsymbol{C}$. Subsequently, the resulting counts are summed up accordingly and kept inside cache storage ι_t on the VRAM. This allows to reduce the number of IO operations between RAM and VRAM for tree decompositions of larger width. Finally, if all chunks are processed for S_i, the memory region ι_t is merged into the RAM at node t. There, depending on the

data structure, it can be beneficial to merge and *compress* resulting solutions obtained for two different solution spaces S_i and S_j. The resulting merged subsets of ρ_t might be later reused when spawning kernels for the parent node of t, where one might prevent splitting results from children.

4 Implementation Details

We implemented our solver gpusat2[4] based on the architecture presented in the previous section. In this section, we describe advancements in the implementation of the solver such as *data structures* optimized for GPUs, improved accuracy in form of *log-counters*, and details on the *kernels*.

4.1 Binary Search Tree on the GPU

A naive approach to store counters on the VRAM is simply to exhaustively consider all possible assignments in $2^{\mathrm{var}(F_t)}$ and store for each assignment a counter, even if zero, in an array. In order to compactly store assignments at a node t in the VRAM, we propose a new data structure, which is in a broader sense a *binary search tree (BST)* for assignments on a very low level architecture. The binary search tree contains only assignments to F_t whose corresponding counter is non-zero, in other words, only counters for assignments that can be extended[5] to satisfying assignments of $F_{\leq t}$. The BST data structure allows us to allocate memory on the VRAM in advance, which is required by OpenCL 1.2, as kernels itself are not allowed to allocate memory on the VRAM during the execution. Internally, a BST consists of a continuous sequence of *cells* that are implemented as 64-bit unsigned integers. There are three types of cells, namely *empty cells*, *value cells*, and *index cells*. An empty cell contains a zero whereas a value cell contains an integer greater than zero. For value cells the 64-bit integer corresponds to a counter that is internally actually interpreted as a double floating point type. We discuss details in the next paragraph. Index cells have either one or two successors in the tree and refer to a value or index cell. For an index cell, the *lower 32 bits* of the integer represent the *index* to the next cell, where a corresponding variable is set to false. Symmetrically, the *upper 32 bits* form an index to the cell if the variable is set to true. Note that either the lower or upper bits can be zero, indicating that the respective index is empty.

Example 1. Figure 2 (left) illustrates a binary search tree $\mathcal{B}$, where value cells are depicted in **bold** face. Both empty cells and empty indices are represented by the symbol "ε". In the BST $\mathcal{B}$ we assume $x < y$. In Figure 2 (right), we can see the BST $\mathcal{B}'$ obtained by inserting the assignment $\alpha := \{x \mapsto 0, y \mapsto 1\}$ into $\mathcal{B}$. The insertion algorithm works as follows. We recursively search for the assignment α in $\mathcal{B}$ by traversing $\mathcal{B}$ according to the variable order, beginning at start index 0. Then, depending on the assignment of the variable at index 0 in α,

[5] Extending an assignment can be done by recursively considering previously computed assignments at the children that correspond to an assignment at the node.

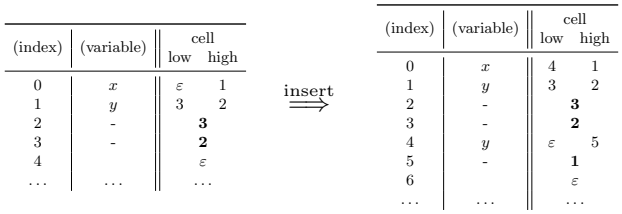

Figure 2: Initial BST $\mathcal{B}$ (left) and BST $\mathcal{B}'$ (right), which was obtained after inserting assignment $\{x \mapsto 0, y \mapsto 1\}$. Value cells are depicted in bold. Both empty cells and empty indices are indicated by the symbol ε. Note: We store *only* cells ("low", "high").

we continue searching using the next variable at the respective index. As soon as an empty index is found during the search for the assignment, new index cells for the remaining variables are subsequently inserted, followed by an inserted value cell of value **1**. As a result, the search for α in $\mathcal{B}$ stops at the lower 32 bits of the index cell at index 0. In turn, these bits refer to a new index cell for y at index 4, whose upper 32 bits point to a new value cell at index 5. ■

Note that we need some *fixed order on the variables* in $\mathrm{var}(F_t)$, to distinguish index and value cells in order to search, insert, update, and delete counters for a given assignment over the variables. The binary search tree enables us to address $2^{32}-1$ many 64-bit integers, which can be changed to relative indices (offsets) if more address space is required. Further, for a given number b of variables, the tree requires in the worst-case at most $2^{b+1}-1$ many 64-bit integers, since there are at most 2^b many value cells (all assignments have non-zero counters) and 2^b-1 index cells (perfectly balanced BST) needed. Note that our data structure has to be manipulated by several GPU threads in parallel. Our strategy to prevent side effects by different threads lies in additionally keeping track of the memory area, where we stored non-empty cells. Therefore, it is sufficient to simply consider the *size* of this memory area. In more details, we keep track of the number of non-empty cells of the tree and prevent writing to a non-empty index again if its index is below the size or the cell is not an empty cell. However, (i) inserting into empty indices and (ii) synchronized updates on existing value cells is allowed. In the actual implementation this is efficiently done by *atomic operations* for 32-bit and 64-bit data types provided by the OpenCL framework [26]. In Case (i) we rely on atomic_cmpxchg for atomically inserting into the index if it is empty. In Case (ii) we use atomic_add for concurrently updating counters.

4.2 Accuracy of Large-Scale Counters

In the previous paragraph, we described that value cells are interpreted as 64-bit floating point numbers. IEEE 64-bit floating point numbers allow to represent values below 10^{308} [1]. However, counters can have a significantly higher value. Therefore, we implement floating point log-counters, where values are stored in relation to 2^e for a 64-bit integer e. In the implementation, we choose the exponent e dynamically at a node t such that every value for an assignment at

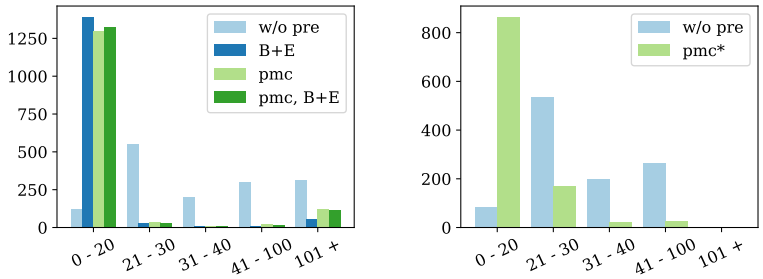

Figure 3: Width distribution of #SAT instances (left) before and after preprocessing (using both B+E and pmc). Width distribution of WMC instances (right) before and after preprocessing using pmc*. Results are based on the primal treewidth and presented in intervals. X-axis labels the intervals, y-axis labels the number of instances.

the node can be represented with the exponent while still keeping e small in order to obtain high accuracy. We normalize the largest counter c in τ_t at a node t to a binary floating point number $c = 1.x \cdot 2^e$, where e is chosen accordingly, and $1.x$ is stored instead of c. We call the resulting e the *largest exponent for* t. Note that all other counters in τ_t are represented with the same exponent e, i.e., we need only one exponent per node. The largest exponent is carefully maintained on the GPU during computation of ρ_t and passed along to parent nodes. For the largest exponents at a node, which has more than one child (join node), we may have to combine counters with respect to different largest exponents for child nodes of t.

4.3 GPU Kernels

When programming for GPUs, small procedures, which are compiled separately from the CPU code and later executed on the GPU, are called *kernels* or shaders [26]. These kernels are spawned on the GPU with massive input data and executed in parallel. Usually, the programming framework takes care of the execution scheduling [26]. In our solver `gpusat2` we developed two different types of GPU kernels, namely, compute kernels and support kernels.

Compute kernels run the actual computation of counters at a node t and are invoked during the traversal of the tree decomposition inside Step 3b when computing the counters for the considered assignments to F_t. Before a compute kernel K is invoked, it obtains as parameters the formula F_t, its preceding results ρ_1, ρ_2, ... computed at its children, and the sum e of the largest exponents e_1, e_2, ... of its children, which represent the counters. Exponent e is then also used to represent counters at t, however, it might increase during the computation and is therefore returned by the kernel after the computation.

Support kernels are used to reorganize data and to carry out memory management tasks. For example, to run the compression in Step 3b we use a support kernel. In more detail, this support kernel compactly merges results temporarily stored in ι_t into ρ_t, which in turn might decrease the number of child chunks needed at the parent of t. Further, we need a support kernel for computing results at a node with more than one child. Here, in order to faster combine previous counts

over multiple stores ρ_{t_1}, ρ_{t_2}, ... of child nodes t_1, t_2, ... we internally use arrays (that fit into VRAM) in a compute kernel first and then convert the data into a BST by a support kernel to have a compact representation again at the parent.

5 Experiments

We conducted a series of experiments using several benchmark sets for model counting. Both benchmark sets[6] and our results[7]are publicly available.

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 would like to emphasis that we benchmarked our GPU-based solvers on cheap consumer hardware, whereas all other solvers ran on recent server hardware. We set a timeout of 900s 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[8](1480 instances), and `c2d` benchmarks[9] (14 instances). For WMC, we used the overall 1091 instances from the `Cachet` benchmark set[10].

Benchmarked Solvers. In our experimental work, we present results for the most recent versions of publicly available #SAT solvers, namely, *c2d* 2.20 [11], *d4* 1.0 [23], *DSHARP* 1.0 [25], *miniC2D* 1.0.0 [27], cnf2eadt 1.0 [20], *bdd_minisat_all* 1.0.2 [36], and *sdd* 2.0 [12] (based on knowledge compilation techniques). We also considered rather recent approximate solvers ApproxMC2, ApproxMC3 [5] and *sts* 1.0 [15], as well as CDCL-based solvers *Cachet* 1.21 [33], sharpCDCL[11], and *sharpSAT* 13.02 [35]. Finally, we also included multi-core solver *countAntom* 1.0 [4] 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.

[6] See: tinyurl.com/gpusat2Instances
[7] See: tinyurl.com/gpusat2Results
[8] See: tinyurl.com/countingbenchmarks
[9] See: reasoning.cs.ucla.edu/c2d
[10] See: cs.rochester.edu/u/kautz/Cachet
[11] See: tools.computational-logic.org

prob	pre	vMdn	cMdn	t[s] Mdn	to	t[s] Mdn pre	to	Mdn	50%	80%	90%	95%	min	max	mdn	mean
#SAT	w/o pre	637	810	0.07	**6**	n/a	n/a	31	31	166	378	922	n/a	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	**31.9**
	pmc	**231**	189	0.03	**6**	**0.03**	**103**	**3**	4	19	228	**823**	-1839	547	**23**	23.1
	B+E	**231**	**185**	**0.02**	**6**	0.04	189	**3**	**3**	18	**192**	**823**	**-2**	633	**23**	31.7
WMC	w/o pre	**200**	519	0.04	**0**	n/a	n/a	28	28	40	43	54	n/a	n/a	n/a	n/a
	pmc*	**200**	**300**	**0.03**	**0**	**0.03**	**0**	**11**	**11**	**20**	**25**	**30**	**0**	**330**	**16**	**18.8**

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

For gpusat2 we considered the *array (A)* and *BST (B)* data structure. Further, we designed a combination referred to by *A+B* aiming to minimize the drawbacks of both variants, which uses the array structure for tree decompositions with width less than 30 and runs with the BST otherwise.

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 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[12]. 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 the vendor-independent OpenCL 1.2 framework. The system operated on Ubuntu 18.04.1 LTS with kernel 4.15.0-34.

Results

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

Treewidth Analysis. We computed upper bounds on the primal treewidth for our benchmarks before and after preprocessing and state them in intervals. For model-count preserving preprocessing we explored both B+E Apr2016 [21] and pmc 1.1 [22]. 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 3 (left) presents the width distribution of number of instances (y-axis) and their

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

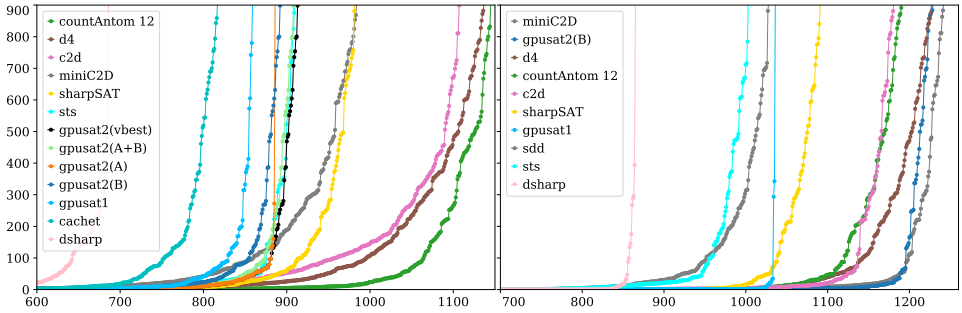

Figure 4: Runtime for the top 10 solvers over all the #SAT instances without preprocessing (left), and with pmc preprocessor (right). The x-axis refers to the number of instances; y-axis depicts the runtime sorted in ascending order for each solver individually.

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 treewidth upper bounds of the WMC instances could be reduced with pmc* as depicted in Figure 3 (right). In more detail, after preprocessing 95% of the instances have primal treewidth below 30, c.f., Table 1.

Solving Performance Analysis. Figure 4 illustrates the top ten solvers without preprocessing (left) and with preprocessor pmc (right) and the variants of gpusat in a cactus-like plot. Note that regardless of the overall scoring, we included at least one solver from each of the five categories of solving techniques (knowledge compilation, approximate, CDCL, multi-core, and dynamic programming). We can observe a slight improvement between gpusat2 and its predecessor when disallowing preprocessing. From Table 2 we can observe that the variant gpusat2(A) performs particular well for instances below width 30, since the compression is relatively expensive. Whereas gpusat2(B) performs well for instances for width at least 30. Since we can dynamically initialize the data structure, we designed a variant gpusat2(A+B) where we use array below 30

solver	racc	0-20	21-30	31-40	41-50	51-60	>60	best	unique	∑	time[h]
countAntom 12	**0**	118	511	139	**175**	**21**	**181**	318	15	**1145**	**96.64**
d4	**0**	124	514	148	162	**21**	168	69	15	1137	104.94
c2d	**0**	119	525	**165**	161	18	120	48	15	1108	110.53
miniC2D	**0**	122	514	128	149	9	62	0	0	984	141.22
sharpSAT	**0**	100	467	124	156	12	123	**390**	4	982	135.41
sts	⋄1.02	⋄118	⋄466	⋄75	⋄**196**	⋄11	⋄44	⋄217	⋄**58**	⋄910	⋄150.79
gpusat2(vbest)	9.8E-18	**125**	**539**	98	141	0	0	97	**21**	903	149.75
gpusat2(A+B)	9.8E-18	**125**	**539**	96	138	0	0	94	19	898	151.16
gpusat2(A)	9.8E-18	**125**	**539**	83	139	0	0	96	19	886	153.28
gpusat2(B)	9.8E-18	**125**	523	96	138	0	0	78	17	882	155.43
gpusat1	1.4E-10	**125**	524	67	140	0	0	82	9	856	162.03
cachet	**0**	99	430	71	152	8	57	3	0	817	176.26
dsharp	4.4E-6	100	382	57	135	7	5	73	0	686	205.31

Table 2: Number of #SAT instances (grouped by treewidth upper bound intervals) solved by the top ten counting solvers without preprocessing. The uniquely solved instances for the gpusat configurations do not count other gpusat1 and gpusat2 runs; racc is the model count error in relation to the correct value (lower is better); symbol ⋄ indicates high inaccuracy; time[h] is the total wall clock time in hours, where unsolved instances are counted as 900s.

and BST for at least 30. This variant performs quite well and is close to the virtual best version of gpusat2(*vbest*) between array and BST. Interestingly, when considering the results on preprocessing in Table 3 (top) and Figure 4 (right) we observe that the architectural improvements quite pay off. gpusat2(*B*) can solve the vast majority of the instances and ranks second place. If one uses B+E preprocessor shown in Table 3 (bottom), gpusat2 solves even more instances as well as the other solvers. However, our solver still solves the most instances with width below 30 and overall manages to get quite close to the others.

Due to space reasons we omit figures on WMC, which we provide in the supplemental material. When considering the runtime without preprocessing, our solver gpusat2(*A+B*) shows significant improvement over its predecessor when the width of the instance is between 31 and 40. The new version solves about 42 instances more than the previous version gpusat1 and only 16 instances less than d4, which is the best solver. However, when it comes to preprocessing using pmc* we obtain a different picture. gpusat solves >100 instances more than d4, but 10 instances less than the best solver miniC2D. What seems a little surprising at first is that gpusat1 solves 2 instances more than gpusat2(*A+B*). So one might ask whether the new version actually provides an improvement over the old one. However, if we also take Figure 3 into account, we easily observe that the preprocessing allows us to obtain tree decompositions of significantly smaller width. In particular, almost all instances are now in the width interval 0–30 and the width interval 31–40 in which the improvements pay off particularly well. Hence, from the presented experiments we will not see any improvement, in particular, since selecting many tree decompositions and the data structure is in practice more expensive than the much simpler technique used in gpusat1.

Currently, we are unable to measure the speed-up of the implementations in terms of the used cores, mainly due to the fact that drivers for OpenCL do not

solver	racc	0-20	21-30	31-40	41-50	51-60	>60	best	unique	∑	time[h]
miniC2D	**0**	1193	29	**10**	2	1	7	13	0	**1242**	**68.77**
gpusat2(B)	4.7E-15	**1196**	**32**	1	0	0	0	250	**8**	1229	71.27
d4	**0**	1163	20	**10**	2	**4**	28	52	1	1227	76.86
countAntom 12	**0**	1141	18	**10**	**5**	**4**	13	101	0	1191	84.39
c2d	**0**	1124	31	**10**	3	3	10	20	0	1181	84.41
sharpSAT	**0**	1029	16	**10**	2	**4**	**30**	**253**	1	1091	106.88
gpusat1	1.7E-13	1020	16	0	0	0	0	106	7	1036	114.86
sdd	**0**	1014	4	7	1	0	2	0	0	1028	124.23
sts	◇2.7	◇927	◇4	◇8	◇**7**	◇**5**	◇**52**	◇73	◇**21**	◇1003	◇128.43
dsharp	4.4E-6	853	3	7	2	0	0	84	0	865	157.87
c2d	**0**	1199	24	**9**	0	2	23	14	0	**1257**	**63.46**
miniC2D	**0**	1203	**27**	8	0	2	12	8	0	1252	64.92
d4	**0**	1182	15	**9**	**1**	**3**	31	79	1	1241	69.32
countAntom 12	**0**	1177	14	8	0	2	34	100	0	1235	69.79
gpusat2(B)	6.4E-16	**1204**	26	1	0	0	0	150	**3**	1231	68.15
sdd	**0**	1106	11	4	**1**	1	4	0	0	1127	100.48
gpusat1	9.9E-12	1037	16	0	0	0	0	87	**3**	1053	110.87
sts	◇1.3	◇943	◇10	◇5	◇**1**	◇**3**	◇**49**	◇21	◇**15**	◇1011	◇125.58
bdd_minisat_all	**0**	926	6	3	**1**	1	0	101	0	937	140.59
sharpSAT	**0**	842	14	8	0	2	**35**	**197**	1	901	153.65
solver	racc	0-20	21-30	31-40	41-50	51-60	>60	best	unique	∑	time[h]

Table 3: Number of #SAT instances (grouped by treewidth upper bound intervals) solved by the top ten counting solvers with preprocessor pmc (top) and B+E (bottom). The number of unique solved instances for the gpusat configurations is without counting in other gpusat1 and gpusat2 runs; racc is the error of model count in relation to the correct value (lower is better); time[h] is the total wall clock time in hours, where unsolved instances are counted as 900 seconds.

support disabling certain cores on the GPU. Therefore, we aim as future work for a new implementation in CUDA [10].

6 Conclusion and Future Work

We presented an improved OpenCL-based solver gpusat2 for solving model counting. Compared to its predecessor gpusat2 implements adapted memory management, specialized data structures on the GPU, improved data type precision handling, and an initial approach to use customized tree decompositions. We carried out rigorous experimental work, including establishing upper bounds for treewidth after preprocessing of commonly used benchmarks and comparing to most recent solvers. The results of this paper 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 tree decompositions of smaller width, in particular for instances of projected model counting. Since projected model counting is complexity-wise also harder than model counting [17], it would be interesting whether it can also benefit from a dedicated parallel implementation. This would be particularly interesting for practical applications, e.g., infrastructure reliability [6]. An interesting further research direction is to study whether efficient data representation techniques can be combined with dynamic programming in order to lift our solver to QSAT [7] or CSP. Finally, alternative parameters [31] might be also interesting for a parallel implementation.

References

1. IEEE standard for floating-point arithmetic. IEEE Std 754-2008 pp. 1–70 (Aug 2008), 10.1109/IEEESTD.2008.4610935
2. Abseher, M., Musliu, N., Woltran, S.: htd – a free, open-source framework for (customized) tree decompositions and beyond. In: Salvagnin, D., Lombardi, M. (eds.) Proceedings of the 14th International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR'17). Lecture Notes in Computer Science, vol. 10335, pp. 376–386. Springer Verlag, Padova, Italy (Jun 2017), 10.1007/978-3-319-59776-8_30
3. Abseher, M., Musliu, N., Woltran, S.: Improving the efficiency of dynamic programming on tree decompositions via machine learning. J. Artif. Intell. Res. **58**, 829–858 (2017)
4. Burchard, J., Schubert, T., Becker, B.: Laissez-faire caching for parallel #SAT solving. In: Heule, M., Weaver, S. (eds.) Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT'15). Lecture Notes in Computer Science, vol. 9340, pp. 46–61. Springer Verlag, Austin, TX, USA (2015), 10.1007/978-3-319-24318-4_5
5. Chakraborty, S., Fremont, D.J., Meel, K.S., Seshia, S.A., Vardi, M.Y.: Distribution-aware sampling and weighted model counting for SAT. In: Brodley, C.E., Stone, P. (eds.) Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14). pp. 1722–1730. The AAAI Press, Québec City, QC, Canada (2014)
6. Chakraborty, S., Meel, K.S., Vardi, M.Y.: Improving approximate counting for probabilistic inference: From linear to logarithmic sat solver calls. In: Kambhampati, S. (ed.) Proceedings of 25th International Joint Conference on Artificial Intelligence (IJCAI'16). pp. 3569–3576. The AAAI Press, New York City, NY, USA (Jul 2016), https://bitbucket.org/kuldeepmeel/approxmc
7. Charwat, G., Woltran, S.: Dynamic programming-based QBF solving. In: Lonsing, F., Seidl, M. (eds.) Proceedings of the 4th International Workshop on Quantified Boolean Formulas (QBF'16). vol. 1719, pp. 27–40. CEUR Workshop Proceedings (CEUR-WS.org) (2016), co-located with 19th International Conference on Theory and Applications of Satisfiability Testing (SAT'16)
8. Charwat, G., Woltran, S.: Expansion-based QBF solving on tree decompositions. In: RCRA@AI*IA. CEUR Workshop Proceedings, vol. 2011, pp. 16–26. CEUR-WS.org (2017)
9. Choi, A., Van den Broeck, G., Darwiche, A.: Tractable learning for structured probability spaces: A case study in learning preference distributions. In: Yang, Q. (ed.) Proceedings of 24th International Joint Conference on Artificial Intelligence (IJCAI'15). The AAAI Press (2015)
10. Cook, S.: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2013)
11. Darwiche, A.: New advances in compiling CNF to decomposable negation normal form. In: López De Mántaras, R., Saitta, L. (eds.) Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04). pp. 318–322. IOS Press, Valencia, Spain (2004)
12. Darwiche, A.: SDD: A new canonical representation of propositional knowledge bases. In: Walsh, T. (ed.) Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11). pp. 819–826. AAAI Press/IJCAI, Barcelona, Catalonia, Spain (Jul 2011)

13. Domshlak, C., Hoffmann, J.: Probabilistic planning via heuristic forward search and weighted model counting. Journal of Artificial Intelligence Research **30** (2007), 10.1613/jair.2289
14. Dueñas-Osorio, L., Meel, K.S., Paredes, R., Vardi, M.Y.: Counting-based reliability estimation for power-transmission grids. In: Singh, S.P., Markovitch, S. (eds.) Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17). pp. 4488–4494. The AAAI Press, San Francisco, CA, USA (Feb 2017)
15. Ermon, S., Gomes, C.P., Selman, B.: Uniform solution sampling using a constraint solver as an oracle. In: de Freitas, N., Murphy, K. (eds.) Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence (UAI'12). pp. 255–264. AUAI Press, Catalina Island, CA, USA (Aug 2012)
16. Fichte, J.K., Hecher, M., Morak, M., Woltran, S.: Answer set solving with bounded treewidth revisited. In: Balduccini, M., Janhunen, T. (eds.) Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17). Lecture Notes in Computer Science, vol. 10377, pp. 132–145. Springer Verlag, Espoo, Finland (Jul 2017), 10.1007/978-3-319-61660-5_13
17. Fichte, J.K., Hecher, M., Morak, M., Woltran, S.: Exploiting treewidth for projected model counting and its limits. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) Proceedings on the 21th International Conference on Theory and Applications of Satisfiability Testing (SAT'18). Lecture Notes in Computer Science, vol. 10929, pp. 165–184. Springer Verlag, Oxford, UK (Jul 2018)
18. Fichte, J.K., Hecher, M., Woltran, S., Zisser, M.: Weighted model counting on the GPU by exploiting small treewidth. In: Azar, Y., Bast, H., Herman, G. (eds.) Proceedings of the 26th Annual European Symposium on Algorithms (ESA'18). Leibniz International Proceedings in Informatics (LIPIcs), vol. 112, pp. 28:1–28:16. Dagstuhl Publishing (2018), 10.4230/LIPIcs.ESA.2018.28
19. Jégou, P., Ndiaye, S., Terrioux, C.: Computing and exploiting tree-decompositions for solving constraint networks. In: van Beek, P. (ed.) Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05). Lecture Notes in Computer Science, vol. 3709, pp. 777–781. Springer Verlag, Sitges, Spain (Oct 2005)
20. Koriche, F., Lagniez, J.M., Marquis, P., Thomas, S.: Knowledge compilation for model counting: Affine decision trees. In: Rossi, F., Thrun, S. (eds.) Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13). The AAAI Press, Beijing, China (Aug 2013)
21. Lagniez, J., Lonca, E., Marquis, P.: Improving model counting by leveraging definability. In: Kambhampati, S. (ed.) Proceedings of 25th International Joint Conference on Artificial Intelligence (IJCAI'16). pp. 751–757. The AAAI Press, New York City, NY, USA (Jul 2016)
22. Lagniez, J., Marquis, P.: Preprocessing for propositional model counting. In: Brodley, C.E., Stone, P. (eds.) Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI'14). pp. 2688–2694. The AAAI Press, Québec City, QC, Canada (2014)
23. Lagniez, J.M., Marquis, P.: An improved decision-DDNF compiler. In: Sierra, C. (ed.) Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17). pp. 667–673. The AAAI Press, Melbourne, VIC, Australia (2017)
24. Lagniez, J.M., Marquis, P., Szczepanski, N.: Dmc: A distributed model counter. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI'18. pp. 1331–1338. The AAAI Press (7 2018), 10.24963/ijcai.2018/185

25. Muise, Christian J .and McIlraith, S.A., Beck, J.C., Hsu, E.I.: Dsharp: Fast d-DNNF compilation with sharpSAT. In: Kosseim, L., Inkpen, D. (eds.) Proceedings of the 25th Canadian Conference on Artificial Intelligence (AI'17). Lecture Notes in Computer Science, vol. 7310, pp. 356–361. Springer Verlag, Toronto, ON, Canada (2012), 10.1007/978-3-642-30353-1_36
26. Munshi, A., Gaster, B., Mattson, T.G., Fung, J., Ginsburg, D.: OpenCL Programming Guide. Addison-Wesley, 1st edn. (2011)
27. Oztok, U., Darwiche, A.: A top-down compiler for sentential decision diagrams. In: Yang, Q., Wooldridge, M. (eds.) Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15). pp. 3141–3148. The AAAI Press (2015)
28. Passerat-Palmbach, J., Hill, D.: OpenCL: A suitable solution to simplify and unify high performance computing developments, chap. 8. Saxe-Coburg Publications (2013)
29. Raynal, M.: Parallel computing vs. distributed computing: A great confusion? (position paper). In: Hunold, S., Costan, A., Giménez, D., Iosup, A., Ricci, L., Gómez Requena, M.E., Scarano, V., Varbanescu, A.L., Scott, S.L., Lankes, S., Weidendorfer, J., Alexander, M. (eds.) Proceedings of the Parallel Processing Workshops (Euro-Par'15). Lecture Notes in Computer Science, vol. 9523, pp. 41–53. Springer Verlag (2015), 10.1007/978-3-319-27308-2_4
30. Roth, D.: On the hardness of approximate reasoning. Artificial Intelligence **82**(1–2) (1996), 10.1016/0004-3702(94)00092-1
31. Sæther, S.H., Telle, J.A., Vatshelle, M.: Solving #SAT and MAXSAT by dynamic programming. Journal of Artificial Intelligence Research **54**, 59–82 (2015)
32. Samer, M., Szeider, S.: Algorithms for propositional model counting. Journal of Discrete Algorithms **8**(1), 50—64 (2010), 10.1016/j.jda.2009.06.002
33. Sang, T., Bacchus, F., Beame, P., Kautz, H., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: Hoos, H.H., Mitchell, D.G. (eds.) Online Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04). Vancouver, BC, Canada (2004)
34. Sang, T., Beame, P., Kautz, H.: Performing bayesian inference by weighted model counting. In: Veloso, M.M., Kambhampati, S. (eds.) Proceedings of the 29th National Conference on Artificial Intelligence (AAAI'05). The AAAI Press (2005)
35. Thurley, M.: sharpSAT – counting models with advanced component caching and implicit BCP. In: Biere, A., Gomes, C.P. (eds.) Proceedings of the 9th International Conference Theory and Applications of Satisfiability Testing (SAT'06). pp. 424–429. Springer Verlag, Seattle, WA, USA (2006), 10.1007/11814948_38
36. Toda, T., Soh, T.: 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
37. Xue, Y., Choi, A., Darwiche, A.: Basing decisions on sentences in decision diagrams. In: Hoffmann, J., Selman, B. (eds.) Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI'12). The AAAI Press, Toronto, ON, Canada (2012)