Parallel Model Counting with CUDA: Algorithm Engineering for Efficient Hardware Utilization

Johannes K. Fichte
TU Dresden, Germany

Markus Hecher
TU Wien, Austria
Universität Potsdam, Germany

Valentin Roland
TU Dresden, Germany

Abstract

Propositional model counting (MC) and its extensions as well as applications in the area of probabilistic reasoning have received renewed attention in recent years. As a result, also the need for quickly solving counting-based problems with automated solvers is critical for certain areas. In this paper, we present experiments evaluating various techniques in order to improve the performance of parallel model counting on general purpose graphics processing units (GPGPUs). Thereby, we mainly consider engineering efficient algorithms for model counting on GPGPUs that utilize the treewidth of a propositional formula by means of dynamic programming. The combination of our techniques results in the solver GPUSAT3, which is based on the programming framework CUDA that –compared to other frameworks– shows superior extensibility and driver support. When combining all findings of this work, we show that GPUSAT3 not only solves more instances of the recent Model Counting Competition 2020 (MCC 2020) than existing GPGPU-based systems, but also solves those significantly faster. A portfolio with one of the best solvers of MCC 2020 and GPUSAT3 solves 19% more instances than the former alone in less than half of the runtime.

2012 ACM Subject Classification Theory of computation → Logic; Computing methodologies → Massively parallel algorithms; Mathematics of computing → Graph algorithms

Keywords and phrases Propositional Satisfiability, GPGPU, Model Counting, Treewidth, Tree Decomposition

Digital Object Identifier 10.4230/LIPIcs.CP.2021.24

Supplementary Material *Software (Source Code, archived)*: `https://zenodo.org/record/5539470`
Software (Source Code): `https://github.com/daajoe/GPUSAT/releases/tag/v3.000-pre`

Funding *Johannes K. Fichte*: Google Fellowship at the Simons Institute, UC Berkeley.
Markus Hecher: FWF Grants Y698 and P32830 and Grant WWTF ICT19-065.

Acknowledgements Authors are given in alphabetical order. Main work has been carried out while the first two authors were visiting the Simons Institute for the Theory of Computing. The authors gratefully acknowledge the valuable comments made by the anonymous reviewers. In particular, the authors thank Roland Yap for his efforts and feedback on the final version of this paper.

1 Introduction

Counting problems have perceived increasing interest in recent years. One of these problems that is well-studied is MC, which aims at counting the number of satisfying assignments of a given propositional formula. In fact, MC is canonical [3, 46] for the complexity class #P and there are a list of applications and variants thereof. Among those variants, extensions of the problem have been studied that involve, e.g., projecting satisfying assignments to certain variables or weighting variables, which enables applications like quantitative reasoning via Bayesian networks and other structures, e.g., [48, 9, 14]. Interestingly, there are also intensive studies focusing on approximation variants of MC, e.g., [7, 18, 6], whose goal is to approximate the number of satisfying assignments within a certain approximation factor.


© Johannes K. Fichte, Markus Hecher, and Valentin Roland;
licensed under Creative Commons License CC-BY 4.0

There are a list of solvers stemming from different technologies and approaches. These solvers have been pushed towards their limits with the help of a dedicated competition [22] for model counting. Among the different approaches, powerful solvers emerged based on caching, knowledge compilation, and parameterized complexity, e.g., [11, 51, 42, 47, 12, 44, 39, 49, 15]. Notably, for model counting, also techniques for parallel solving proved successful [5, 40]. Among those parallel solvers there are also solvers [28, 29] that utilize modern consumer general purpose graphics processing units (GPGPUs). Those existing GPGPU-based implementations are based on dynamic programming on a tree decomposition of different graph representations of a propositional formula. One particular graph representation [47], namely the so-called primal graph representation[1], proved successful since it is employed in the latest GPGPU-based implementation for MC, referred to by GPUSAT2 [29], but also in other solvers [8, 34, 27, 15, 16]. In this work, we take up this idea and systematically study improvements from the perspective of algorithm engineering in order to significantly speed up counting models of a propositional formula on GPGPUs. Our approach is an implementation that follows the existing implementation GPUSAT2, but we use the GPGPU framework CUDA instead of OPENCL since CUDA enables a more detailed and systematic performance analysis and hardware monitoring capabilities.

Contributions. We present a novel system GPUSAT3[2] that comprises the following new techniques and contributions. (i) We introduce a new clause representation, called *compact clause form (CCF)*, that allows us to check whether a partial assignment satisfies a clause by only two binary bit operations that can be efficiently implemented in hardware. (ii) Then, we enhance dynamic programming, which is designed to combine results (tables) of "local" computations, by *global caching*. Thereby we maintain a global cache on the GPGPU that is shared among different local tables in order to prevent copy overhead between the GPGPU memory and main memory (RAM) whenever possible. (iii) Further, we show the benefit of CUDA framework tuning, where we focus on quantifying the effect of *pinned memory*, a technique designed for reducing transfer overhead between GPGPU memory and RAM. (iv) Finally, we perform a study over existing *libraries for computing tree decompositions* in order to quantify the effect of those decomposers on the actual solving performance. These techniques and variants are systematically analyzed and presented individually. Then, the performance results involving the full system GPUSAT3 is given at the end. For the sake of presentation, *related work* is discussed in-place where suitable and applicable.

2 Preliminaries

Let α be a *bit vector* $\langle b_0, \ldots, b_{n-1} \rangle$, which is a sequence consisting of n many *bits* that are integers between 0 and 1. Then, we refer to the i-th bit for position $0 \leq i < n$ by $\alpha_i := b_i$. We use the bit-wise XOR operator $\oplus$ and the bit-wise AND operator & in the usual meaning. Further, we define the integer *value* of α by $\mathrm{val}(\alpha) := \sum_{0 \leq i \leq n-1} 2^i \cdot \alpha_i$.

Propositional Satisfiability (SAT)

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. Then, we refer to a

[1] The primal graph of a propositional formula distinguishes as vertices the variables of the formula and there is an edge between two variables, whenever those variables appear together in at least one clause.
[2] GPUSAT3 including benchmark data [26] is open source; available at github.com/vroland/GPUSAT.

set $S \subseteq F$ by a *sub-formula (of F)*. For a clause $c \in F$, let $var(c)$ consist of all variables occurring in c and $var(F) := \bigcup_{c \in F} var(c)$. Without loss of generality, we assume for every $c \in F$ that $|c| = |var(c)|$, i.e., no variable appears twice in a clause. An *assignment (over $V \subseteq var(F)$)* is a mapping $A : V \to \{0,1\}$. The formula $F[A]$ *under* A is obtained by removing all clauses from F that contain a literal set to 1 by A and removing from remaining clauses all literals set to 0 by A. An assignment A is *satisfying* if $F[A] = \emptyset$. The problem MC asks to count the number of satisfying assignments over $var(F)$ of a formula F.

▶ **Example 1.** Consider the formula $F := \{c_1, c_2, c_3\}$ with $c_1 := a \vee b \vee \neg c$, $c_2 := \neg b \vee \neg a$, and $c_3 := a \vee \neg d$. Then, $A_1 := \{a \mapsto 1, b \mapsto 0\}$ and $A_2 := \{a \mapsto 0, c \mapsto 0, d \mapsto 0\}$ are satisfying, i.e., $F[A_1] = F[A_2] = \emptyset$. In total, there are 7 satisfying assignments over $\{a, b, c, d\}$ of F.

Tree Decompositions (TDs), Treewidth, and Dynamic Programming

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. The *primal graph* G_F [47] 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 $t \in T$ of the primal graph G_F, we let $F_t := \{\, c \mid c \in F, var(c) \subseteq \chi(t) \,\}$ 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 containing all clauses entirely covered by a bag $\chi(s)$ for s and any of its descendant nodes.

▶ **Example 2.** Recall F from Example 1. From the primal graph P_F of F a TD $\mathcal{T}$ of P_F with nodes t_1, t_2, t_3 can be constructed, where t_3 with $\chi(t_3) = \{a\}$ joins t_1, t_2 with $\chi(t_1) = \{a, b, c\}$ and $\chi(t_2) = \{a, d\}$. Intuitively, $\mathcal{T}$ allows to evaluate F in parts. So, when evaluating $F = F_{\leq t_3}$, we split into $F_{\leq t_1}$ and $F_{\leq t_2}$, which refer to $\{c_1, c_2\}$ and $\{c_3\}$, respectively.

A solver based on *dynamic programming* for formulas evaluates the input formula F in parts along a given TD of the primal graph G_F. For each node t of the TD, results are usually stored in a *table* τ_t. The approach works in four steps as follows:

1. Construct the primal graph G_F of the input formula F.
2. Heuristically compute a tree decomposition $\mathcal{T} = (T, \chi)$ of the primal graph G_F.
3. Traverse the nodes in $V(T)$ in post-order. Thereby, at every node t, run an algorithm for computing table τ_t. This algorithm takes as input the bag $\chi(t)$, the sub-formula F_t, and previously computed tables for the child nodes of t. Such a table τ_t comprises rows of the form $\langle A, c \rangle$, where $A : \chi(t) \to \{0,1\}$ is an assignment and c is an integer used for counting. Each row $\langle A, c \rangle$ indicates that there are c many satisfying assignments over $var(F_{\leq t})$ of $F_{\leq t}$ that extend A. These pairs are carefully maintained for all the different types of nodes; for details we refer to the literature [47, 29].
4. Print the model count by interpreting the result τ_n for the root n of T.

The worst-case runtime of such an algorithm for model counting is in $\mathcal{O}(2^k k d N)$, where k denotes the width of the primal graph, N refers to the number of nodes in the tree decomposition, and d denotes the maximum number of occurrences in the clauses of the input formula F over all variables of F [47].

General Purpose Graphics Processing Units (GPGPUs)

General purpose computing on graphics processing units (*GPGPU*) is the practice of exploiting the massively parallel computing capabilities of graphics cards for non-graphical and scientific applications. Historically, graphics hardware and drivers have been built with only graphics rendering pipelines in mind [52]. But with a growing demand for accelerating data-parallel programs, GPGPU frameworks like CUDA and OPENCL emerged. These offer APIs for writing GPGPU programs (functions) without relying on graphics primitives. Although frameworks aim to make programming applications for CPU and GPGPU more seamless, their memory spaces and instructions are distinct: While the CPU can access the *host memory (RAM)* and executes the *host program*, the GPGPU can only access *GPGPU memory (GPGPU RAM)* and execute functions called *kernels*. The functionality of CUDA and OPENCL largely overlaps. However, OPENCL is standardized and supports running CPUs as well. In contrast, CUDA is a proprietary platform for NVIDIA GPGPUs only, but CUDA is often perceived as a more mature ecosystem. This comprises tools like profilers and runtime checkers, learning resources, but also driver support.

Dynamic Programming With GPGPUs. Dynamic programming as described above is implemented in the solver GPUSAT2 [29], which heavily uses OPENCL and introduces a framework for efficiently solving MC in parallel on GPGPUs. For GPUSAT2, besides a simple implementation of storing tables via a fixed pre-allocated memory block, called ARRAY data structure, the authors have proposed an advanced data structure, referred to by TREE, which is a binary search tree that can be manipulated in parallel by the GPGPU. This solver also provides a heuristic that is used internally in order to decide whether to use ARRAY or TREE based on a tree decomposition width threshold.

Benchmarks

In order to systematically analyze performance, we use the following instances and hardware.

Benchmark Instances. We use the 200 instances of track 1 (private and public) of the 1st International Competition on Model Counting (MCC 2020) [22] as the MCC2020-TRACK1 benchmark. For our final evaluation, we additionally incorporate the instances of track 2 of MCC 2020 with variable weights stripped, making them unweighted. The 400 instances of this combined benchmark set are referred to as MCC2020-TRACK1+2.

Benchmark Hardware. We run benchmarks on three different machines. **Server**: 2x Intel Xeon Silver 4112@2.60GHz 128GB RAM, NVIDIA GeForce GTX 1180 8GB GPGPU RAM, Ubuntu 18.04 LTS, CUDA 9.1.85. **Desktop**: Intel Core i7-9700@3.00GHz 16GB RAM, NVIDIA Quadro RTX 4000 8GB GPGPU RAM, Ubuntu 18.04 LTS, CUDA 11.0. **Cluster**: Cluster of 44 nodes; 2x Intel Xeon E5-2680v3@2.50GHz, 256GB RAM, RHEL 7.9.

SERVER is used for running full benchmarks with various decomposers, as it provides an environment that is not shared with other users and enough memory resources. For detailed profiling, DESKTOP is employed due to the availability of a more current driver and local access to the machine. CLUSTER is chosen for comparing different tree decomposition libraries since it provides the resources needed to finish a massive amount of runs quickly.

3 Algorithm Engineering and Hardware Utilization

While the existing GPGPU-based system GPUSAT2 delivers decent performance compared with state-of-the-art model counters, a number of possible improvements as well as hardware-specific potentials are left unexplored. In this section, we outline several algorithmic as well as implementation-specific improvements for Step 3 of dynamic programming on tree decompositions as defined in the preliminaries. We then systematically evaluate the impact of these improvements, which finally leads to a *new system GPUSAT3*. Since GPUSAT3 is based on CUDA, we can leverage tools and suitable workflows for a systematic analysis.

Next, we introduce a compact representation of clauses as bit vectors that allows us to efficiently check for satisfiability on GPGPU hardware. Then, we describe a global caching scheme for result tables such that GPUSAT3 avoids superfluous transfers between host memory (RAM) and *GPGPU memory (VRAM)*. Lastly, we show that using the so-called pinned memory of CUDA, while introducing some overhead, benefits performance by increasing data transfer speed between host and GPGPU.

Compact Clause Form (CCF)

Clearly we cannot hope that GPUSAT3 solves instances of arbitrarily large treewidth. Since GPUSAT3 is based on dynamic programming aiming for utilizing reasonably small treewidth, we therefore restrict ourselves to instances below a reasonable threshold like treewidth 64. This allows us to build a clause data structure that is more optimized for satisfiability testing on GPGPUs. Assume a formula F, a TD $\mathcal{T} = (T, \chi)$ of G_F, and a node t of T.

Interestingly, every clause $c \in F_t$ can then be represented with two bit vectors, namely an occurrence vector occ and a sign vector sign, which both combined correspond to the *compact clause form (CCF)* of c. To the end of defining this compact representation, let $\mathrm{idx} : \chi(t) \to \{0, \ldots, |\chi(t)|-1\}$ be a bijective function that assigns each variable $v \in \chi(t)$ a *positional index* from 0 to the number $|\chi(t)|-1$ of variables minus one, thereby adhering to some fixed total ordering of variables in $\chi(t)$. Since idx is bijective, we denote the inverse of idx by idx^{-1}. Then, the *occurrence vector* $\mathrm{occ}(c)$ for c is a sequence consisting of $|\chi(t)|$ many bits such that whenever $v \in \mathit{var}(c)$, the corresponding bit $\mathrm{occ}(c)_{\mathrm{idx}(v)}$ is set to 1 (and to 0 otherwise). The *sign vector* $\mathrm{sign}(c)$ for c is of the same form as the occurrence vector such that $\mathrm{sign}(c)_{\mathrm{idx}(v)}$ is set to 1 whenever $\neg v \in c$. Otherwise, bit $\mathrm{sign}(c)_{\mathrm{idx}(v)}$ is set to 0.

In order to test if an assignment satisfies a set of clauses in CCF, assignments must be in a compact representation as well. Let A be an assignment over $\chi(t)$. Then, we compactly represent A as an *assignment vector* $\vec{A}$ such that the i-th bit $\vec{A}_i$ of A for $0 \leq i < |\chi(t)|$ corresponds to the truth value of the variable at position i in A, i.e., $\vec{A}_i = A(\mathrm{idx}^{-1}(i))$.

▶ **Proposition 3** (Correctness of CCF). *Assume a formula F, a TD $\mathcal{T} = (T, \chi)$ of G_F as well as a node t of T. Let further $c \in F_t$ be a clause and A be any assignment over variables $\chi(t)$.*

Then, A satisfies c if and only if $\mathrm{val}((\vec{A} \oplus \mathit{sign}(c)) \,\&\, \mathit{occ}(c)) \geq 1$, *where $\oplus$ and $\&$ denotes the bit-wise XOR and AND operator, respectively.*

▶ **Example 4.** Consider the clauses from Ex. 1: $c_1 = a \vee b \vee \neg c$, $c_2 = \neg b \vee \neg a$, and $c_3 = a \vee \neg d$. Observe that for the given total ordering $\langle a, b, c, d \rangle$, there is only one unique positional index function idx, defined by $\mathrm{idx}(a) := 0$, $\mathrm{idx}(b) := 1$, $\mathrm{idx}(c) := 2$, and $\mathrm{idx}(d) := 3$. Then, the corresponding bit vectors are $\mathrm{occ}(c_1) = 1110$, $\mathrm{occ}(c_2) = 1100$, $\mathrm{occ}(c_3) = 1001$ and sign vectors are $\mathrm{sign}(c_1) = 0010$, $\mathrm{sign}(c_2) = 1100$, $\mathrm{sign}(c_3) = 0001$.

Now, let us check the assignment $A = \{a \mapsto 0, b \mapsto 1, c \mapsto 0, d \mapsto 1\}$. In bit vector representation, this corresponds to $\vec{A} = 0101$. For c_1, we have that $(\vec{A} \oplus \text{sign}(c_1)) \,\&\, \text{occ}(c_1) = (0101 \oplus 0010)\&1110 = 0110$. Since $\text{val}(0110) = 6 \geq 1$, A satisfies c_1. Conversely, $(\vec{A} \oplus \text{sign}(c_3) \,\&\, \text{occ}(c_3) = (0101 \oplus 0001)\&1001 = 0000$ indicates that A does not satisfy c_3.

Observe that if we restrict ourselves to instances of treewidth below 64, the length of all involved vectors discussed above is bounded by 64 as well. So, testing if a truth assignment satisfies a clause can be efficiently implemented using 64-bit integers and bit-wise logic operators on top. More concretely, a satisfiability check can be implemented on any 64-bit hardware by only using one bit-wise XOR and one bit-wise AND operation for each clause in F_t. Interestingly, both occurrence and sign vectors can be computed once per TD node t and clause in F_t before actually invoking the GPGPU.

By carefully choosing the variable ordering (idx), we can ensure that the unique id of each parallel GPGPU computation unit performing such checks is a prefix of the assignment vector A. Consequently, the assignments tested in a single such computation unit only differs by a few of their least significant bits, allowing the assignment vector to be efficiently constructed by combining the unique id with a counter variable.

Achieving a form of parallelism using bit-wise instructions was used in the context of SAT solving [35]. There, a single instruction operates on multi-bit variable values representing multiple assignments. In our #SAT solver, multiple instruction operate on multiple assignments where each thread works on exactly one assignment. We obtain the assignments immediately from the thread id. Our compact representation of clauses minimizes thread divergence by taking a constant number of instructions for varying number of literals in a clause. In fact, small thread divergence is important for effective performance when running massive parallel execution on the GPGPU and low overhead of the used caches.

Reducing GPGPU Copy Overhead via Global Caching

In the preliminaries, we outline a model counting algorithm based on dynamic programming. The implementation in GPUSAT3 contains some steps that are performed on the GPGPU and others on the host. Thus, for each node in the tree decomposition, one or more *kernels* are executed which compute a table associated with the current node. Intuitively, if a kernel invocation uses a table produced by a previous kernel, this data can remain in VRAM and does not have to be copied to the host. If ideally VRAM was unlimited, no intermediate memory transfers to the host memory (RAM) would be needed, except for the final result. However, when tables become too large to store in VRAM alongside the next table, tables are divided into multiple *table chunks* per node in order to make solving such nodes feasible. Table chunks that are not currently needed are moved to main (host) memory, which is typically larger than the available VRAM.

Copying tables from and back to the VRAM can take a significant portion of the overall execution time. This leads to the idea of *global caching*: GPUSAT3 tries to keep whole tables or table chunks in a cache that is managed globally, in the sense that it potentially contains tables for several tree decomposition nodes at the same time. Thus, in the case that some or all child tables chunks already reside in VRAM when solving a node, i.e., they are *cached*, GPUSAT3 does not need to transfer them from host memory for solving. After a kernel execution, the result is left in the cache if the table is needed later in the solving process. More precisely, GPUSAT3 prefers to not transfer table chunks to the RAM until solving is completed; instead tracking a repository of chunks in VRAM.

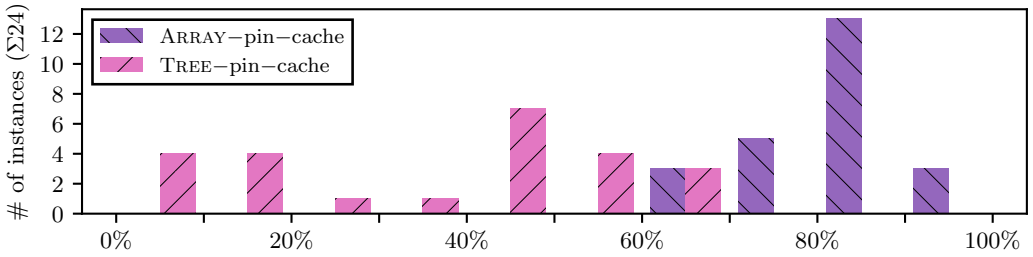

Figure 1 Histogram of percentages of GPGPU runtime spent on data transfers (memcpy) grouped in steps of 10% without global caching, using either TREE or ARRAY data structure.

A chunk is deleted from the cache if (a) it is either no longer needed for solving subsequent nodes or (b) the VRAM is used otherwise. The latter (b) is the case if a new table needs more VRAM than available. In this scenario, all cache entries are evicted, since as much VRAM as possible is needed for solving the current node. This effectively degrades the cache to a local one, as only the currently needed table chunks can be kept. However, if tables are in relation to the available VRAM sufficiently small, the algorithm can benefit from keeping tables from both branches of a *join* node in the VRAM during the traversal. Ideally, this prevents intermediate transfers from the VRAM to host memory, except for the final results.

Evaluation. To investigate the need for global caching, we analyze the following hypothesis.

▶ **Hypothesis 1.** *GPUSAT3 spends large portions of runtime on GPGPU data transfers.*

To evaluate Hypothesis 1, we use the NVPROF profiler to determine how much GPGPU runtime is spent for copying data to and from the GPGPU. We relate this time to the total time spent in GPGPU functions during the solver run, as recorded by NVPROF. This represents the proportion of GPGPU runtime spent on data transfers instead of kernel executions. For small instances, this ratio may not be representative, as their data structures are typically very small. Constant costs like runtime initialization and memory allocation could further distort the results in such cases. Consequently, we only consider instances with a total solver runtime of at least 5s in one of the configurations. In each following comparison, only instances successfully solved by both compared configurations are included.

First, we compare this ratio for GPUSAT3 with caching and pinned memory disabled for both the ARRAY and TREE data structures, executed on DESKTOP with an arbitrary but fixed decomposition seed. Pinned memory is a technique to speed up data transfers between GPGPU and host, which is disabled here and will be explained in more detail in the next subsection. MCC2020-TRACK1 with a timeout of 600s is chosen as it contains many instances with sufficient treewidth to necessitate transfers between GPGPU and host memory. This allows us to get a baseline for the cost of data transfers without any optimizations. In Fig. 1, we show the distribution of GPGPU runtime in data transfers among the applicable instances of MCC2020-TRACK1 as a histogram. Clear differences are visible between the TREE and ARRAY data structures: With TREE, two clusters are visible at $\leq 20\%$ and $50\% - 70\%$. No instance uses more than 70% of GPGPU runtime in memcpy. With ARRAY, at least 60% is used, with the highest number of instances spending $80\% - 90\%$. Based on this experiment, we can confirm that a large portion of runtime is taken up by data transfers, assuming most work is done on the GPGPU. The TREE structure results in smaller relative

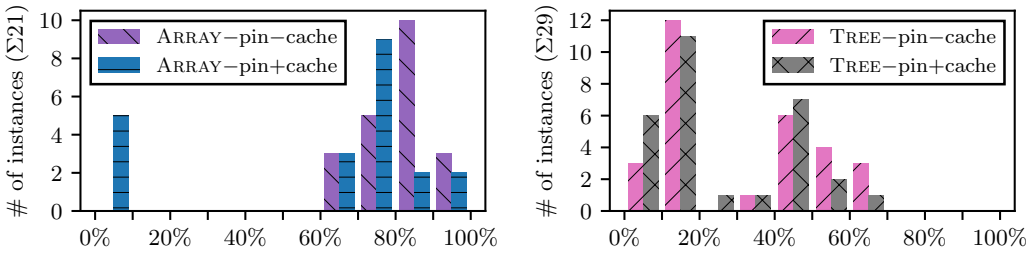

Figure 2 Histograms of percentages of GPGPU runtime spent in memcpy with and without caching. Only instances of MCC2020-TRACK1 which are solved in both configurations with a runtime of at least 5s in one are considered. Results are given for the ARRAY and TREE data structure, (left) and (right), respectively.

transfer times than using ARRAY. This could be due to their smaller average size [29], longer kernel runtimes or a combination of both. Nevertheless, a large portion of instances spend a proportion of at least 40%, as well.

To alleviate this issue, we propose global caching as described above. In order to evaluate the effectiveness of our global cache, which avoids data transfers and reuses (cached) tables within the GPGPU memory, we consider the following hypothesis.

▶ **Hypothesis 2.** *Caching reduces the proportion of time GPUSAT3 spends on data transfers.*

We assess Hypothesis 2 with the same method as previously used for Hypothesis 1, where we run GPUSAT3 for both data structures, with and without caching. The results are presented in Fig. 2, which confirms this hypothesis: Considering the ARRAY configuration, the number of instances spending $80\% - 90\%$ in memcpy is greatly reduced, many presumably shifting to the $70\% - 80\%$ bucket. Interestingly, while without caching no instance spent less than 60% of its runtime on data transfers, a number of instances achieves using less than 10% with caching. In these cases, global caching avoids most transfers altogether. This is in-line with our expectations: While the usefulness of the global cache is degraded when all GPGPU memory is needed for solving, instances which mostly produce small tables can benefit significantly. With the TREE data structure, we see a similar trend of instances spending less time in memcpy. However, the effect is not as strong as with ARRAY. Again, we believe this is due to the TREE already using GPGPU memory more efficiently, leading to smaller transfers which take less time relative to kernel executions.

The Effect of Cuda Pinned Memory

To speed up data transfer between host memory and GPGPU memory, the CUDA driver offers an API that allows the use of *pinned memory* pages (also known as page-locked memory) when allocating host memory [43, 10]. Pinned memory pages reside in a fixed physical location of the host memory and cannot be moved, e.g., swapped out, by the *operating system (OS)*. This guarantee allows the CUDA driver to perform data transfers to and from these regions through its *direct memory access (DMA)* engine. Through DMA hardware, neither the CPU, nor the OS are involved in transferring data. So, no checks for the validity of memory pages through the OS kernel are needed, since physical page locations are fixed.

Pinned memory has already been utilized in the literature. Quirem et al. [45] implement a GPGPU-accelerated version of an algorithm used in the HMMER framework [30] for identifying homologous protein sequences using CUDA. The authors report a 20% speed

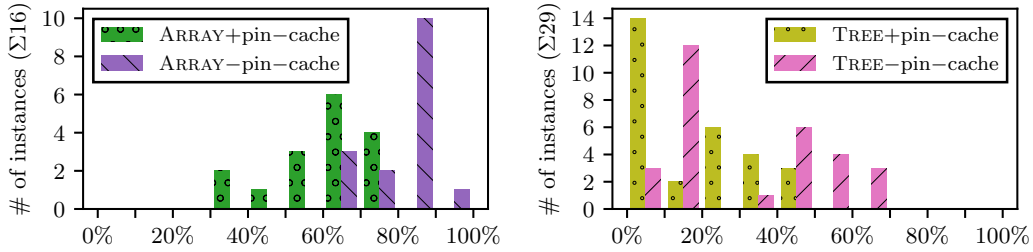

Figure 3 Histograms of percentages of GPGPU runtime spent on data transfers (memcpy) grouped in steps of 10%, with and without pinned memory. Only instances of MCC2020-TRACK1 which are solved in both configurations with a runtime of at least 5s in one are considered. Results are given for both the ARRAY (left) data structure as well as the TREE (right) data structure.

up by using pinned memory. Similarly, Fatica [20] demonstrates significant performance improvements for LINPACK with pinned memory. However, allocating pinned memory incurs additional overhead compared to regular allocations of main memory [32]. Additionally, if a large portion of the physical system memory is pinned, overall performance is degraded. Moreover, when using pinned memory, the CUDA driver enforces a shared virtual address space for allocations in host memory and GPGPU memory, which is referred to as CUDA Unified Memory. Aside from pinned memory, this feature is used for handling data transfers between CPU and GPGPU memory implicitly by the driver, as a convenience for the programmer. Consequently, the overhead of unified memory applies as well. Jarzabek et al. [37] investigated the performance of unified memory, overall finding it to have only a small impact. Nevertheless, especially many small allocations increased the performance overhead. We mitigate the performance degradation of repeatedly allocating and freeing pinned memory by employing a so-called sub-allocator. This *sub-allocator* is responsible for caching pinned memory allocations, handing out memory allocations from an existing pool and only allocating additional memory when needed [36].

Evaluation

We evaluate the potential for performance improvements through achieving faster data transfers which is counteracted by the additional overhead of pinned memory allocation.

The Potential of Pinned Memory. As a first step, we consider the potential speedup of pinned memory without considering the cost for its allocation.

▶ **Hypothesis 3.** *With pinned memory, GPUSAT3 reduces the proportion of time spent on copying data to and from the GPGPU.*

In order to analyze this hypothesis, we measure the proportion of time the GPGPU spends for memory transfers with and without pinned memory for both the ARRAY and TREE data structures. The experiment is conducted on DESKTOP with a maximal runtime of 600s for each instance of MCC2020-TRACK1. Chunk caching, i.e., leaving solution data in GPGPU memory if possible, is disabled to measure the impact of pinned memory in isolation. The results are shown in Fig. 3 as histograms of the ratio of GPGPU runtime used for copying memory to the total GPGPU runtime, given in percent. We apply the same criteria for selecting instances for the comparison as in Sect. 3. For the ARRAY data structure (left), we see that without pinned memory, most memory transfers take up as much as 80%

to 90% of the GPGPU runtime on some instances. All instances spend at least 60% for copying memory by means of memcpy. With pinned memory, the majority consumes around 50% to 60% GPGPU runtime for copying, some less, with at most $\approx 80\%$. When using the Tree data structure (right) and no pinned memory, the proportion is generally lower and corresponds to the baseline distribution established in Fig. 1. With pinned memory, the distribution shifts to spend significantly less time in memcpy. Additionally, many instances have a copy to execution time ratio below 10%, while no instance has more than 50%. In conclusion, we see that pinned memory reduces the time used for data transfers significantly regardless of data structure. The smaller size of the Tree data structure compared to Array results in a smaller proportion of GPGPU runtime spent copying.

The Benefit of Pinned Memory. For the benefit of faster memory transfers to result in improved solver performance, it needs to outweigh the overhead of pinned memory allocation.

▶ **Hypothesis 4.** *Employing pinned memory decreases the runtime of the solver GPUSAT3.*

To test this hypothesis, we consider the runtime for the instances of MCC2020-Track1 with and without pinned memory. We run GPUSAT3 for both the Array and the Tree data structure on MCC2020-Track1 for up to 600s on Server. Global caching is enabled to determine if pinned memory further improves solver runtime on top of caching. In Fig. 4, we compare the differences in runtime on a per-instance basis with an arbitrary but fixed decomposition seed. For instances where the runtime differs by more than $\geq 1\%$ of the unpinned runtime, the difference is marked by an arrow as described in the figure caption.

With both the Array and Tree data structures, especially long-running instances benefit from pinned memory, while the allocation overhead amounts to an overall longer runtime on small instances. The configuration using the Array data structure (left) benefits more from pinned memory on instances with long runtimes compared to the Tree (right) configuration. We attribute this to the capability of the Tree structure to grow as needed, leading to smaller transfers than with the Array data structure on average. Conversely, the size of Arrays grows with the number of variables in a node regardless of its solution count. Moreover, fewer small instances are negatively impacted by pinned memory when using Tree, presumably because of smaller allocations incurring less overhead.

Overall, we have shown that pinned memory can speed up the solving process for instances where large memory transfers are needed. For small instances and depending on the data structure, its additional allocation overhead can outweigh the faster transfer times however. By comparing the improvement in data transfer times seen in Fig. 3 with overall solver performance in Fig. 4, we see that this mostly translates to improvements in runtime for larger instances, where the sub-allocator can serve most allocations. Thus, the addition of pinned memory is beneficial in most cases, although using the Tree data structure often mitigates the need for large memory transfers, lessening the impact of pinned memory.

4 The Influence of Decomposition Libraries

In this section, we focus on Step 2 of the dynamic programming approach, as defined in the preliminaries. Thereby, we compare several available implementations for finding tree decompositions and their effect on solver runtime in order to efficiently employ these libraries.

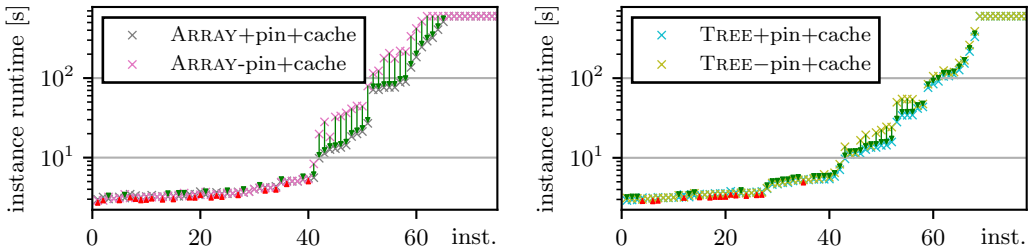

Figure 4 Comparison of runtime over the instances of the MCC2020-TRACK1 data set with and without pinned memory. Downward (green) arrows denote an improvement in runtime with pinned memory, upward (red) arrows indicate a longer runtime with pinned memory. Results for the ARRAY and TREE data structure are given (left) and (right) respectively.

Finding Tree Decompositions

Quickly finding a tree decomposition with a small width is crucial for the performance of GPUSAT3. Utilizing a tree decomposition of smaller width not only improves worst-case runtime of our algorithm, but also exponentially decreases memory requirements. This is paramount for the practical efficiency: Once certain tables do not fit into the GPGPU memory, larger chunks of data have to be swapped to the host memory (RAM), thereby increasing processing time. Inconveniently, finding the treewidth of a graph represents a NP-hard problem itself [2]. An algorithm for obtaining tree decompositions of small, bounded width in linear time has been developed, but its runtime complexity contains constant factors too large for practical use [4]. To the best of our knowledge, there are no practically feasible, fast algorithms with such low time complexity. Thus, the time spent on finding a tree decomposition of low treewidth and running the dynamic programming algorithm must be balanced. To find a suitable decomposer for computing tree decompositions we compared the 3 top-ranked submissions to the heuristic competition of track A of the "Parameterized Algorithms and Computational Experiments Challenge" in 2017 (PACE17) [13]: tamaki by Keitaro Makii, Hiromu Ohtsuka, Takuto Sato, Hisao Tamaki (Meiji University), github.com/TCS-Meiji/PACE2017-TrackA, flowcutter [50] by Ben Strasser (Karlsruhe Institute of Technology), and htd [1] by Michael Abseher, Nysret Musliu, Stefan Woltran (TU Wien). As a first step, we compare the obtained widths and speed of the three decomposers above.

▶ **Hypothesis 5.** *Given a long processing time, the rank by best decomposition width reflects the placement in PACE17 in MCC2020-TRACK1: 1. tamaki, 2. flowcutter, and 3. htd.*

In Fig. 5 (left), we compare the lowest width found by the implementations in 600s for the MCC2020-TRACK1 instances on CLUSTER. For each instance and each decomposer, we obtain 10 decompositions with varying seeds. In the following analysis, the best result of these runs is considered. Although flowcutter ranked better than htd in PACE17, it consistently produces higher widths than htd and tamaki in our benchmark. Note however, that we use the more recent htd 1.2. The decomposer tamaki generates lower widths than htd for most instances, with some outliers. For small widths up to ≈ 30, the difference between htd and tamaki is noticeable but small for most instances. With wider widths, flowcutter performs much worse than htd and tamaki. Overall, the relative performance of htd and tamaki matches our expectations of Hypothesis 5. However, flowcutter performs significantly worse than its competitors, which might be due to the size and structure of our instances.

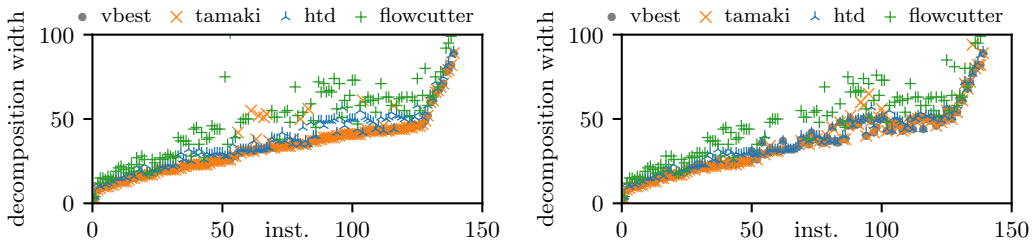

Figure 5 Best decomposition width for MCC2020-Track1 instances of different implementations; ordered by asc. best width. The plots show the widths obtained after 600s (left) and 15s (right).

As htd was chosen in GPUSAT2 for being fast, we suspect the above results to change when restricting the implementations to shorter runtimes. Thus, we repeat the above analysis in Fig. 5 (right), but consider the respective best result found after only 15 seconds. GPUSAT3 with htd usually takes less than one second for computing the tree decomposition for most solvable instances of MCC2020-Track1. Nonetheless, we believe that 15s would be a realistic time budget for implementations that cannot be tightly integrated into the solver as a library. Compared with Fig. 5 (left), we see the advantage of tamaki over htd shrinking. For some instances, htd produces smaller decompositions than tamaki, which generates either a very wide or no decomposition at all. Decomposer flowcutter still produces the widest decompositions for most instances. Consequently, tamaki generates the best results for most instances, even with constrained time. However, there are cases where only htd produces a decomposition of usable width. Since the advantage of tamaki is small at low runtimes and htd is available as a C++ library, we keep it as the primary implementation in GPUSAT3 for convenience. In the future, a portfolio of implementations with a tuned heuristic of the time spent searching for a decomposition could yield better results [17].

The Performance Impact of Decompositions

Recall that we have already investigated the benefit of finding tree decompositions of small width, based on worst-case time and space bounds. To justify this claim, we now explore the performance impact of the chosen tree decomposition during solving in practice.

▶ **Hypothesis 6.** *The solving time of GPUSAT3 strongly correlates with the decomposition width and only to a lesser extent depends on the instance.*

We investigate the connection of decomposition width and solving time by creating a set of different tree decompositions for select instances. To find suitable instances, we first compute 10 tree decompositions per instance of MCC2020-Track1 with each of the implementations: tamaki, flowcutter and htd. Each run is allowed 600s of wall clock time on Cluster. From MCC2020-Track1, we select a subset of instances where we find at least one decomposition with a width between 25 and 35. These bounds are chosen because instances with a width larger than 35 are mostly unsolvable on our hardware. Conversely, if the decomposition width is too low, the solving process is usually very fast and dominated by set-up time. Thus, measuring runtime differences is not meaningful outside of this range. By applying this criterion, we obtain a set of 62 instances for our evaluation. Then, we measure the solving time for each decomposition of the selected instances with GPUSAT3 on Server. The version of GPUSAT3 used in this experiment already includes the improvements introduced in the course of this paper. For each instance, 10 decompositions are obtained by each

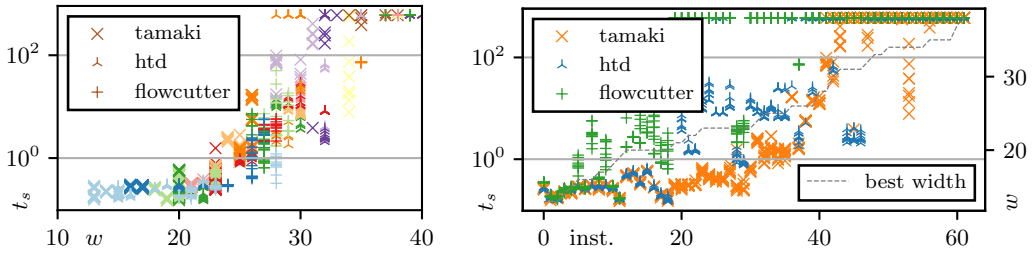

Figure 6 GPUSAT3 solving time t_s in seconds by decomposition width w (left) and by instance (right). On the left, runs of the same instance are colored in the same color, but different instances may have the same color. On the right, instances are ordered by their respective lowest decomposition width, which is marked as a dashed grey line.

decomposer, amounting to 1860 decompositions in total. Since we are interested in the behaviour of the GPGPU solving algorithm rather than overall runtime compared to other solvers, we use the time spent in the solving step as reported by GPUSAT3. This time is referred to by *solving time* and excludes parsing and preprocessing steps.

In Fig. 6 (left), we show the results of this experiment as a plot of solving time and decomposition width, colored by instance. We observe a general trend of increasing solving time with larger decomposition width. Moreover, the plot can be roughly divided in three sections: Up to a width of ≈ 22, the solving time consistently stays below one second, without major variations for different widths. For widths between ≈ 23 and ≈ 38, solving time correlates with decomposition width. However, large differences in solving time occur among decompositions of equal widths, sometimes by multiple orders of magnitude. For larger decompositions, all runs either time out or exhaust the resources of SERVER, which is also shown as a timeout. This observation supports our focus on instances with decompositions of widths between 25 and 35 for this experiment: The runtime for small decompositions is dominated by set-up costs and parallel GPGPU resources are not saturated, thus it does not vary significantly. For decompositions of high width, resource and time limits are quickly exceeded. While this experiment clearly demonstrates a correlation between decomposition width and solving time as indicated by theoretical bounds, there is a large spread of runtimes for decompositions of the same width. For example, runtimes for the width 28 span from sub-1s to almost 100s. When looking at the distribution of instances, which are indicated by color, we see that runs of the same instance often form clusters. Through the different marker symbols, we see that distinct clusters of the same color mostly originate from different composers. Conversely, most clusters only contain runs of one decomposer and instance. This indicates that the variance in runtime is low for the same instance and decomposer. However, this view does not immediately reveal solving times of all runs of the same instance.

Thus, we visualize this perspective in Fig. 6 (right). The plot shows solving times for every generated decomposition of each instance as specified in the figure caption. Similarly to the by-width perspective Fig. 6 (left), runtimes for decompositions generated by the same decomposer often appear clustered. However, this effect appears less pronounced for decompositions generated by flowcutter. For many instances, tamaki appears to generate the best-performing decompositions. This is in line with our findings above. Some instances, especially those of smaller width, show very similar runtimes for all decompositions. However, in most cases, runtimes are clearly separated for the decomposers. As instances are sorted by their best generated decomposition width, a trend of increasing runtime with width is visible.

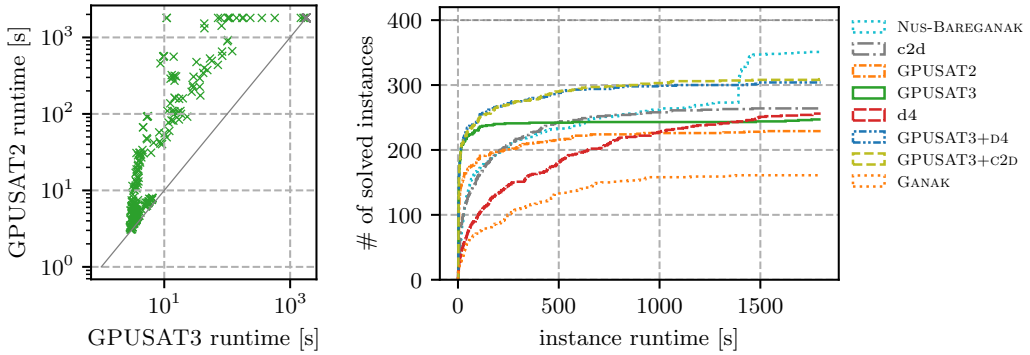

Figure 7 Performance of GPUSAT3 compared GPUSAT2 and other state-of-the-art systems. In the cactus plot (right) instances are sorted for each solver individually, according to ascending runtime. The scatter plot (left) compares instance runtimes of GPUSAT3 with GPUSAT2.

This indicates that the width of a decomposition is a better predictor for solver runtime than the given instance, as stated in Hypothesis 6. Nonetheless, runtimes vary among decompositions of the same width, so treewidth is not the sole estimate for instance hardness of GPUSAT3. Thus, detailed structural studies are left for further research, cf. [41].

5 Overall Results

Next, we evaluate the combined result of the presented techniques above by comparing the performance of GPUSAT3 to GPUSAT2, d4 [39], c2d [11], nus-bareganak [22], and ganak [49]. The systems d4, c2d, and nus-bareganak are among the best solvers of the Model Counting Competition 2020 [22] (MCC 2020). At time of submission, the full instance set of the 2021 competition was not publicly available [21]. d4 and c2d are based on knowledge compilation, while nus-bareganak is a portfolio of solvers: First, they run the B+E preprocessor [38], followed by ganak. If ganak does not produce a result in a chosen timeout, ApproxMC [7] is used. As no external preprocessing is used with the other solvers, pure ganak is included for reference. We run all solvers for up to 1800s for each instance of MCC2020-Track1+2 on Server. All solvers are used in their default configuration: c2d and d4 ran with flags to enable counting, otherwise no additional arguments where supplied. The number of solved instances, ordered by instance runtime, is shown in Fig. 7 (right).

The total number of solved instances per solver is listed in Tab. 1. For low runtimes, GPUSAT3 establishes a clear lead over the other solvers. Given more time, GPUSAT2 approaches GPUSAT3 and d4, c2d surpass GPUSAT3. nus-bareganak delivers similar performance to c2d until ApproxMC is used, where it surpasses the other solvers. Note that the results of ApproxMC are within ±30% of the correct count with 80% probability [7].

In practice, not only the number of eventually solved instances is relevant, but also the time in which they are solved. Thus, we define a *baseline* set of benchmarks, which are the instances that are solved by all solvers except pure ganak, thereby enabling meaningful comparisons of solving time. ganak is excluded to maintain a meaningful baseline set size. Tab. 1 lists the accumulated runtime for each solver, for 50%, 90%, 95% and 100% of instances of both the baseline set and all of a solver's respective solved instances. When comparing with respect to the baseline set, we obtain an 8x speedup in accumulated runtime of GPUSAT3 over GPUSAT2, 11x over c2d, and over 10x speedup over nus-bareganak.

Table 1 Solved instances and accumulated runtimes for the fastest n% of solved instances for each surveyed solver. In the second row for each solver, accumulated runtime is compared with respect to a baseline set of 161 instances, which are solved by all except GANAK. * the portfolio includes an approximate solver, for more details see above.

Solver	# inst.	∑t 100%	∑t 95%	∑t 90%	∑t 50%
GPUSAT2	229	5:56:06	3:05:47	1:51:28	0:09:13
. . . on baseline	161	2:26:36	1:00:06	0:36:32	0:06:38
GPUSAT3	247	**3:05:58**	**0:43:40**	**0:27:12**	**0:07:01**
. . . on baseline	161	**0:18:08**	**0:13:00**	**0:11:03**	**0:04:51**
D4	256	1 day, 2:30:28	20:57:09	16:39:14	1:59:09
. . . on baseline	161	15:54:58	12:09:37	9:21:47	0:53:41
C2D	265	12:25:56	8:20:19	6:21:38	0:39:15
. . . on baseline	161	3:29:07	2:16:12	1:40:38	0:13:16
NUS-BAREGANAK	**351***	1 day, 21:47:59	1 day, 14:21:25	1 day, 7:41:05	1:33:55
. . . on baseline	161	3:12:16	1:57:25	1:30:17	0:17:53
GANAK	161	11:26:48	9:05:15	7:28:31	0:53:24
GPUSAT3+D4	304	7:36:44	3:36:43	1:58:31	0:09:05
. . . on baseline	161	**0:18:08**	**0:13:00**	**0:11:03**	**0:04:51**
GPUSAT3+C2D	309	8:45:15	4:30:15	2:35:57	0:09:23
. . . on baseline	161	**0:18:08**	**0:13:00**	**0:11:03**	**0:04:51**

To combine the capability of D4 and C2D with the speed of GPUSAT3, we define the portfolios GPUSAT3+D4 and GPUSAT3+C2D to use GPUSAT3 for instances with a decomposition width of ≤ 35 and D4 resp. C2D otherwise. The time to calculate the width is negligible and therefore not included in the runtime of instances solved with the portfolio solvers. To generate the decomposition we use htd as used in GPUSAT3. As shown in Tab. 1, the portfolio solvers are very successful: Not only do they solve significantly more instances than D4 and C2D alone, but accomplish this in significantly less accumulated runtime. As expected, GPUSAT3+D4 and GPUSAT3+C2D achieve the same performance on the baseline set as GPUSAT3, which overall either solves instances extremely fast or fails. With the availability of advanced hardware with larger GPGPU memory, we expect that due to global caching, GPUSAT3 solves instances that currently reach a timeout. External preprocessing as used in NUS-BAREGANAK could further improve the results, cf. [29].

6 Conclusion and Future Work

Efficiently solving problems related to propositional model counting is critical for a range of applications such as probabilistic reasoning. To accelerate solving, the massively parallel computing capabilities of general purpose GPUs (GPGPUs) can be leveraged by a dynamic programming based algorithm. Our system GPUSAT3 builds on top of ideas from GPUSAT2, where we implement algorithmic improvements and techniques for better hardware utilization. We describe a new, hardware-friendly compact clause form, a global caching strategy, as well as pinned memory, and systematically evaluate impacts on solving performance. Additionally, we survey a range of libraries for generating tree decompositions

and show their performance impact as well. Compared to GPUSAT2, our overall results show that GPUSAT3 solves all instances faster, sometimes by an order of magnitude. While GPUSAT3 is designed for bounded treewidth, it complements D4 and C2D in a portfolio approach; significantly enhancing the overall performance compared to the individual solvers.

In the future, we plan on migrating portions of the solver code to GPGPU kernel code. Additionally, we are interested in the scalability of GPUSAT3 when using multiple GPGPUs. To the best of our knowledge, currently there is no way to limit the number of parallel compute cores a program can utilize, preventing such experiments with a single device. Alternatives include frequency and voltage scaling [31] and dynamic transformation to a CPU program [19], both options have probably different scaling characteristics than the addition of parallel compute cores. Furthermore, techniques used in other dynamic programming based solvers such as ADDMC [15] could be brought to the GPGPU. Conversely, integrating GPUSAT3 into other solvers for solving sub-problems of small treewidth might be beneficial. Finally, GPGPU-based approaches might be applicable to formalisms such as argumentation [23], logic programming [33], or description logics [24], despite strong theoretical limits [25].

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 AI and OR Techniques in Constraint Programming (CPAIOR 2017), Padua, Italy, June 5-8, 2017*, volume 10335 of *Lecture Notes in Computer Science*, pages 376–386. Springer, 2017. doi:10.1007/978-3-319-59776-8_30.

2 Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987. doi:10.1137/0608024.

3 Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and complexity results for #SAT and bayesian inference. In *Proceedings of the 44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA*, pages 340–351. IEEE Computer Soc., 2003. doi:10.1109/SFCS.2003.1238208.

4 Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996. doi:10.1137/S0097539793251219.

5 Jan Burchard, Tobias Schubert, and Bernd Becker. Laissez-faire caching for parallel #SAT solving. In Marijn Heule and Sean A. Weaver, editors, *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT 2015), Austin, TX, USA, September 24-27, 2015*, volume 9340 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2015.

6 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 2014), July 27 -31, 2014, Québec City, Québec, Canada*, pages 1722–1730. AAAI Press, 2014. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8364.

7 Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls. In Subbarao Kambhampati, editor, *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016), New York, NY, USA, 9-15 July 2016*, pages 3569–3576. IJCAI/AAAI Press, July 2016. URL: http://www.ijcai.org/Abstract/16/503.

8 Günther Charwat and Stefan Woltran. Expansion-based QBF solving on tree decompositions. *Fundamenta Informaticae*, 167(1-2):59–92, 2019. doi:10.3233/FI-2019-1810.

9 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 and Michael J. Wooldridge, editors, *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015), Buenos Aires, Argentina, July 25-31, 2015*, pages 2861–2868. AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/405.

10 Shane Cook. *CUDA programming: A developer's guide to parallel computing with GPUs*. Applications of GPU Computing Series. Morgan Kaufmann, Boston, 2013. doi:10.1016/B978-0-12-415933-4.02001-9.

11 Adnan Darwiche. New advances in compiling CNF to decomposable negation normal form. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence (ECAI 2004), including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 318–322. IOS Press, 2004.

12 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 2011), Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 819–826. AAAI Press/IJCAI, 2011. doi:10.5591/978-1-57735-516-8/IJCAI11-143.

13 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 2017), September 6-8, 2017, Vienna, Austria*, volume 89 of *LIPIcs*, pages 30:1–30:12. Dagstuhl Publishing, 2017. doi:10.4230/LIPIcs.IPEC.2017.30.

14 Carmel Domshlak and Jörg Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research*, 30:565–620, 2007. doi:10.1613/jair.2289.

15 Jeffrey M. Dudek, Vu Phan, and Moshe Y. Vardi. ADDMC: weighted model counting with algebraic decision diagrams. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020), New York, NY, USA, February 7-12, 2020*, pages 1468–1476. AAAI Press, 2020. URL: https://aaai.org/ojs/index.php/AAAI/article/view/5505.

16 Jeffrey M. Dudek, Vu H. N. Phan, and Moshe Y. Vardi. DPMC: weighted model counting by dynamic programming on project-join trees. In Helmut Simonis, editor, *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP 2020), Louvain-la-Neuve, Belgium, September 7-11, 2020*, volume 12333 of *Lecture Notes in Computer Science*, pages 211–230. Springer, 2020. doi:10.1007/978-3-030-58475-7_13.

17 Jeffrey M. Dudek and Moshe Y. Vardi. Parallel weighted model counting with tensor networks. *CoRR*, abs/2006.15512, 2020. arXiv:2006.15512.

18 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 31st AAAI Conference on Artificial Intelligence (AAAI 2017), February 4-9, 2017, San Francisco, California, USA*, pages 4488–4494. AAAI Press, 2017. URL: http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14870.

19 Naila Farooqui, Andrew Kerr, Gregory Frederick Diamos, Sudhakar Yalamanchili, and Karsten Schwan. A framework for dynamically instrumenting GPU compute applications within GPU ocelot. In *Proceedings of the 4th Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2011), Newport Beach, CA, USA, March 5, 2011*, page 9. ACM, 2011. doi:10.1145/1964179.1964192.

20 Massimiliano Fatica. Accelerating linpack with CUDA on heterogenous clusters. In David R. Kaeli and Miriam Leeser, editors, *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2009), Washington, DC, USA, March 8, 2009*, volume 383 of *ACM International Conference Proceeding Series*, pages 46–51. ACM, 2009. doi:10.1145/1513895.1513901.

21 Johannes K. Fichte and Markus Hecher. The model counting competition 2021. https://mccompetition.org/past_iterations, 2021.

22 Johannes K. Fichte, Markus Hecher, and Florim Hamiti. The model counting competition 2020. *ACM Journal of Experimental Algorithmics*, 2021. In press.

23 Johannes K. Fichte, Markus Hecher, and Arne Meier. Counting complexity for reasoning in abstract argumentation. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAA'19)*, Honolulu, Hawaii, USA, 2018.

24 Johannes K. Fichte, Markus Hecher, and Arne Meier. Knowledge-base degrees of inconsistency: Complexity and counting. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI'21)*, pages 6349–6357, 2021.

25 Johannes K. Fichte, Markus Hecher, and Andreas Pfandler. Lower Bounds for QBFs of Bounded Treewidth. In Naoki Kobayashi, editor, *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'20)*, pages 410–424. Assoc. Comput. Mach., New York, 2020.

26 Johannes K. Fichte, Markus Hecher, and Valentin Roland. GPUSAT3 benchmark data and source code. *Zenodo*, 2021. doi:10.5281/zenodo.5159903.

27 Johannes K. Fichte, Markus Hecher, Patrick Thier, and Stefan Woltran. Exploiting database management systems and treewidth for counting. In Ekaterina Komendantskaya and Y. Annie Liu, editors, *Proceedings of the 22nd International Symposium on Practical Aspects of Declarative Languages (PADL'20)*, volume 12007 of *Lecture Notes in Computer Science*, pages 151–167. Springer, 2020. doi:10.1007/978-3-030-39197-3_10.

28 Johannes K. Fichte, Markus Hecher, Stefan Woltran, and Markus Zisser. Weighted model counting on the GPU by exploiting small treewidth. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *Proceedings of the 26th Annual European Symposium on Algorithms (ESA 2018), August 20-22, 2018, Helsinki, Finland*, volume 112 of *LIPIcs*, pages 28:1–28:16. Dagstuhl Publishing, 2018. doi:10.4230/LIPIcs.ESA.2018.28.

29 Johannes K. Fichte, Markus Hecher, and Markus Zisser. An improved GPU-based SAT model counter. In Thomas Schiex and Simon de Givry, editors, *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP 2019), Stamford, CT, USA, September 30 - October 4, 2019*, volume 11802 of *Lecture Notes in Computer Science*, pages 491–509. Springer, 2019. doi:10.1007/978-3-030-30048-7_29.

30 Robert D. Finn, Jody Clements, and Sean R. Eddy. HMMER web server: interactive sequence similarity searching. *Nucleic Acids Research*, 39(Web-Server-Issue):29–37, 2011. doi:10.1093/nar/gkr367.

31 Rong Ge, Ryan Vogt, Jahangir Majumder, Arif Alam, Martin Burtscher, and Ziliang Zong. Effects of dynamic voltage and frequency scaling on a K20 GPU. In *Proceedings of the 42nd International Conference on Parallel Processing (ICPP 2013), Lyon, France, October 1-4, 2013*, pages 826–833. IEEE Computer Society, 2013. doi:10.1109/ICPP.2013.98.

32 GNU Project. GNU libc manual, 3.5.2 locked memory details. URL: https://www.gnu.org/software/libc/manual/html_node/Locked-Memory-Details.html.

33 Markus Hecher. Treewidth-aware reductions of normal ASP to SAT - is normal ASP harder than SAT after all? In Diego Calvanese, Esra Erdem, and Michael Thielscher, editors, *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12-18, 2020*, pages 485–495, 2020. doi:10.24963/kr.2020/49.

34 Markus Hecher, Patrick Thier, and Stefan Woltran. Taming high treewidth with abstraction, nested dynamic programming, and database technology. In Luca Pulina and Martina Seidl, editors, *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT 2020), Alghero, Italy, July 3-10, 2020*, volume 12178 of *Lecture Notes in Computer Science*, pages 343–360. Springer, 2020. doi:10.1007/978-3-030-51825-7_25.

35 Marijn Heule and Hans van Maaren. Parallel SAT solving using bit-level operations. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):99–116, 2008. doi:10.3233/sat190040.

36 Jared Hoberock, Nathan Bell, and Thrust Contributors. Thrust API documentation. URL: https://thrust.github.io/doc/classthrust_1_1mr_1_1disjoint__unsynchronized__pool__resource.html.

37 Lukasz Jarzabek and Pawel Czarnul. Performance evaluation of unified memory and dynamic parallelism for selected parallel CUDA applications. *Journal of Supercomputing*, 73(12):5378–5401, 2017. doi:10.1007/s11227-017-2091-x.

38 Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Improving model counting by leveraging definability. In Subbarao Kambhampati, editor, *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016), New York, NY, USA, 9-15 July 2016*, pages 751–757. The AAAI Press, July 2016. URL: http://www.ijcai.org/Abstract/16/112.

39 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 2017), Melbourne, Australia, August 19-25, 2017*, pages 667–673. The AAAI Press, 2017. doi:10.24963/ijcai.2017/93.

40 Jean-Marie Lagniez, Pierre Marquis, and Nicolas Szczepanski. DMC: a distributed model counter. In Jérôme Lang, editor, *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018), July 13-19, 2018, Stockholm, Sweden*, pages 1331–1338. The AAAI Press, 2018. doi:10.24963/ijcai.2018/185.

41 Silviu Maniu, Pierre Senellart, and Suraj Jog. An experimental study of the treewidth of real-world graph data. In Pablo Barceló and Marco Calautti, editors, *Proceedings of the 22nd International Conference on Database Theory (ICDT 2019), March 26-28, 2019, Lisbon, Portugal*, volume 127 of *LIPIcs*, pages 12:1–12:18. Dagstuhl Publishing, 2019. doi:10.4230/LIPIcs.ICDT.2019.12.

42 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 Advances in Artificial Intelligence Artificial Intelligence (Canadian AI 2012), Toronto, ON, Canada, May 28-30, 2012*, volume 7310 of *Lecture Notes in Computer Science*, pages 356–361. Springer, 2012. doi:10.1007/978-3-642-30353-1_36.

43 NVIDIA Corporation. Application note – CUDA 2.2 pinned memory APIs. URL: http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/simpleZeroCopy/doc/CUDA2.2PinnedMemoryAPIs.pdf.

44 Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015), Buenos Aires, Argentina, July 25-31, 2015*, pages 3141–3148. The AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/443.

45 Saddam Quirem, Fahian Ahmed, and Byeong Kil Lee. CUDA acceleration of P7Viterbi algorithm in HMMER 3.0. In Sheng Zhong, Dejing Dou, and Yu Wang, editors, *Proceedings of the 30th IEEE International Performance Computing and Communications Conference (IPCCC 2011), Orlando, Florida, USA, November 17-19, 2011*, pages 1–2. IEEE, 2011. doi:10.1109/PCCC.2011.6108104.

46 Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996. doi:10.1016/0004-3702(94)00092-1.

47 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.

48 Tian Sang, Paul Beame, and Henry A. Kautz. Performing bayesian inference by weighted model counting. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings of the 20th National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 475–482. AAAI Press / The MIT Press, 2005. URL: http://www.aaai.org/Library/AAAI/2005/aaai05-075.php.

49 Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A scalable probabilistic exact model counter. In Sarit Kraus, editor, *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019), Macao, China, August 10-16, 2019*, pages 1169–1176, 2019. doi:10.24963/ijcai.2019/163.

50 Ben Strasser. Computing tree decompositions with flowcutter: PACE 2017 submission. *CoRR*, abs/1709.08949, 2017. arXiv:1709.08949.

51 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 on Theory and Applications of Satisfiability Testing (SAT 2006), Seattle, WA, USA, August 12-15, 2006*, volume 4121 of *Lecture Notes in Computer Science*, pages 424–429. Springer, 2006. doi:10.1007/11814948_38.

52 Richard Vuduc and Jee Choi. *A Brief History and Introduction to GPGPU*, pages 9–23. Springer US, Boston, MA, 2013. doi:10.1007/978-1-4614-8745-6_2.