FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Solving #SAT on the GPU with Dynamic Programming and OpenCL

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Markus Zisser, BSc
Matrikelnummer 01125404

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Zweitbetreuung: Projektass.(FWF) Dipl.-Inf. Dr.rer.nat. Johannes Klaus Fichte
Mitwirkung: Projektass.(FWF) Dipl.-Ing. Markus Hecher, BSc

Wien, 2. Mai 2018

Markus Zisser	Stefan Woltran

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Solving #SAT on the GPU with Dynamic Programming and OpenCL

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Markus Zisser, BSc
Registration Number 01125404

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Second advisor: Projektass.(FWF) Dipl.-Inf. Dr.rer.nat. Johannes Klaus Fichte
Assistance: Projektass.(FWF) Dipl.-Ing. Markus Hecher, BSc

Vienna, 2nd May, 2018

Markus Zisser

Stefan Woltran

Erklärung zur Verfassung der Arbeit

Markus Zisser, BSc
8232 Grafendorf, Bahnhofstraße 223

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 2. Mai 2018

Markus Zisser

Acknowledgements

I like to thank my advisors Stefan Woltran and Johannes Klaus Fichte for their help. Together with Markus Hecher they provided me with constructive suggestions, and feedback during the course of this thesis. I also want to thank my parents Renate and Karl Zisser. They always supported me during my study and throughout my life.

Kurzfassung

Im Bereich der Informatik gibt es viele rechnerisch schwere Probleme. Eine Vielzahl dieser Probleme kann mit Hilfe von aussagenlogischen Formeln dargestellt, und die Antwort auf manche dieser Probleme kann direket oder indirekt auf die Anzahl der Lösungen zurückgeführt werden. Die Aufgabe, alle Lösungen einer aussagenlogischen Formel zu finden wird auch als das #SAT Problem bezeichnet.

Es hat sich gezeigt, dass verschiedene Algorithmen aus dem Bereich der künstlichen Intelligenz und des maschinellen Lernens von der enormen Parallelisierung auf der Grafikkarte (GPU) profitiert haben. Viele aktuelle #SAT Löser hängen jedoch von Methoden, die beim Lösen des Erfüllbarkeitsproblems verwendet werden, ab oder sind Löser die nur eine Annäherung an den korrekten Wert berechnen. Eine zentrale Methode die beim Lösen des Erfüllbarkeitsproblems verwendet wird ist Conflict Driven Clause Learning (CDCL), welches jedoch Schritte beinhaltet die nur schwer oder nicht parallelisierbar sind. Eine Parallelisierung auf der Grafikkarte funktioniert allerdings nur wenn ein Algorithmus viele ähnliche voneinander unabhängige Schritte beinhaltet, deshalb kann man CDCL schwer oder nur eingeschränkt auf die GPU bringen. Bei der dynamischen Programmierung auf Basis von Baumzerlegungen hingegen sind viele ähnliche, voneinander unabhängige Operationen auszuführen, daher ist sie für die Grafikkarte gut geeignet. Im Gegensatz zu anderen Verfahren ist die Effizienz der Dynamische Programmierung von einer niedrigen Weite der Baumzerlegungen abhängig.

Im Zuge dieser Arbeit haben wir gpusat entwickelt, einen #SAT Löser basierend auf der dynamischen Programmierung auf Baumzerlegungen. gpusat wurde mit Hilfe von OpenCL implementiert, ein offener Standard für die GPU Programmierung. Als Basis für den Algorithmus von gpusat können Baumzerlegungen des primal, incidence und dual Graphens verwendet werden. gpusat ist auch in der Lage den Weighted Model Count (WMC) einer Formel zu berechnen. Bei WMC bekommt jedes Literal der Formel ein Gewicht und das Gewicht einer Lösung ist das Produkt der Gewichte ihrer Literale.

Um gpusat mit anderen #SAT Lösern zu vergleichen haben wir #SAT und WMC Instanzen aus unterschiedlichen Quellen gesammelt. Für diese Instanzen haben wir dann Baumzerlegungen des primal, incidence und dual Graphen generiert. Danach haben wir die Laufzeit von gpusat mit anderen aktuellen #SAT und WMC Lösern auf den zuvor gesammelten Instanzen verglichen.

Bei unseren Tests hat sich gezeigt, dass gpusat für Instanzen mit einer Baumweite von bis zu 30 mit anderen Lösern konkurrieren kann und es war uns möglich einzelne Instanzen mit einer Baumweite von bis zu 45 zu lösen. Ungefähr die Hälfte der Instanzen hatte eine Weite bis zu 30.

Abstract

There are many computational hard problems in computer science and a variety of these problems can be expressed via Boolean formulas. For some of these problems, the number of satisfying assignments can be directly linked to the solution. The task to compute the number of solutions of a Boolean formula is called the #SAT problem.

Algorithms in artificial intelligence and machine learning tasks have profited from the massive parallelism provided by Graphic Processing Units (GPUs). However many current #SAT solvers rely on techniques from Satisfiability solving or approximate solving based on sampling of the search space. A central method for solving the Satisfiability problem is Conflict Driven Clause Learning (CDCL), but CDCL contains parts which are hard to or not parallelizable at all. CDCL does not work well on the GPU since parallelization on the GPU requires an algorithm with many similar independent steps. Dynamic Programming (DP) on tree decompositions on the other hand parallelizes well as we can execute many similar operations which are independent from another. For DP we need tree decompositions with a sufficiently small width.

In the course of this thesis we developed gpusat, a #SAT solver which is based on dynamic programming on tree decompositions with OpenCL, an open standard that can be used to parallelize tasks on the GPU. We use tree decompositions of the primal, incidence and dual graph as base for our dynamic programming algorithms. gpusat is also able to solve Weighted Model Counting (WMC) where a weight is assigned to every literal and the weight of a solution is the product of its literal weights.

To compare gpusat with other solvers we collected #SAT and WMC instances from different sources, and generated tree decompositions of the primal, incidence and dual graph for each of these instances. Then we compared the runtime of gpusat with other state of the art #SAT and WMC solvers on our instances. To get a better understanding of our benchmark set we also generated an overview of the tree width for our benchmark instances.

Our experiments have shown that gpusat is competitive with other solvers for a tree width of up to 30. We were also able to solve some instances with a tree width of up to 45. About half of the instances had a width of 30 or below.

Contents

CHAPTER 1

Preface

1.1 Motivation

Many computational hard problems in computer science amount to combinatorics. A variety of these problems, which arise in the real world, can be solved via checking if a Boolean formula is satisfiable (SAT) or not. If solutions require statistical or probability measures, then the number of satisfying assignments of these formulas can be directly related to the solution. The problem of counting the solutions of a Boolean formula is referred to as #SAT which is a generalization of the SAT problem and is #P-hard [Rot96]. Examples for these problems are calculating the reliability of energy infrastructure [DMPV17] or learning preference distributions [CdBD15]. A problem related to #SAT is Weighted Model Counting (WMC) which asks for the weighted model count of a formula. In WMC a weight is assigned to every literal, and the weight of a solution is the product of its literal weights, the WMC is the sum of the weight of all solutions.

In recent decades there has been an increasing interest in using Graphic Processing Units (GPUs) for general purpose computing with parallelizeable algorithms. It has been shown that GPU computing can speed up tasks in artificial intelligence and machine learning by more than two orders of magnitude [JYP+17]. Most modern #SAT solvers rely on techniques from SAT-solving [Thu06, SBB+04, GSS09], knowledge compilation [LM17, Dar04] or approximate #SAT solving [CFM+14, Dar11] based on sampling the search space with SAT solvers. Conflict Driven Clause Learning (CDCL) is a main technique used in modern SAT solvers, but CDCL does not parallelize well [Man16, FF12, BSS+12].

Parallelization on the GPU works best if there are a lot of independent, similar tasks. Hence alternative methods for #SAT and WMC are required. A promising approach in this context is to employ dynamic programming (DP). A solver which is based

on dynamic programming evaluates the input formula on parts along a given tree decomposition, and stores the results in tables. Graph representations of the input formula, such as the primal, incidence or dual graph are used for the decompositions. The runtime of dynamic programming algorithms depend on the size of these tables, and the size of the result table grows exponentially with respect to the tree width of the input formula.

1.2 Methodology and Research Question

Our main research questions we want to answer with this thesis are:

1. How to parallelize dynamic programming on the GPU? What challenges, and problems are there when parallelizing algorithms on the GPU? How can we overcome these challenges and problems?

2. How does dynamic programming on the GPU compare to serial and parallel #SAT and WMC techniques? How does dynamic programming on the GPU compare to dynamic programming on the CPU?

3. What is the maximum width we can solve with our approach? What is the maximum width at which we are competitive to other solvers? How many instances of standard #SAT benchmark sets are we able to solve and are in our future reach?

To answer these questions, we have implemented gpusat, a #SAT and WMC solver which is based on dynamic programming on tree decompositions. The CPU code is written in C++ and OpenCL was used for the GPU code.

gpusat works as follows:

1. We generate the primal, incidence or dual graph of the input formula.

2. We heuristically generate a tree decomposition of the graph.

3. The solver takes the tree decomposition, and then computes the number of solutions for each possible variable assignment on the GPU, based on [SS10].

4. Then, the solver sums up the number of satisfying assignments in the last bag.

5. The solver prints the number of satisfying assignments.

We have collected #SAT and WMC benchmark sets from different sources for benchmarking. Then we generated the primal, incidence, dual graphs and the tree decompositions of the graphs for our benchmark set. The tree decompositions were generated with

htd [AMW17] which heuristically generates a tree decomposition of a graph, as generating a tree decomposition with minimal width is NP-hard [Bod05].

Then we benchmarked gpusat and the other #SAT and WMC solvers on our benchmark set. In the end we generated an overview of the widths for each benchmark set, and compared the runtime of gpusat to other state of the art #SAT and WMC solvers.

1.3 Contributions/Publications

The main contributions of this thesis are as follows:

- We implemented a #SAT solver based on dynamic programming on tree decompositions which utilizes the massive parallelization and computational power of the GPU with OpenCL. Our solver can be found at:

  `https://github.com/Budddy/GPUSAT`

- We carried out extensive benchmarking to compare our solver with 11 other state of the art #SAT and WMC solvers.

- We generated a classification of our benchmark set according to the width, of the primal, incidence, and dual graphs.

The author has contributed to the following publications:

Johannes Fichte, Markus Hecher, Markus Zisser, and Stefan Woltran. Weighted Model Counting on the GPU by Exploiting Small Treewidth. Submitted to ESA 2018, under review.

1.4 Related Work

To the best of our knowledge there are no other #SAT solvers which utilize the GPU for solving, therefore we take a look at other serial and parallel #SAT solvers which run on the CPU as well as SAT solvers which run on the GPU.

Many current #SAT solvers such as sharpSAT [Thu06], sharpCDCL [KMM13] and Dsharp [MMBH12] rely on techniques from SAT solving such as Conflict Driven Clause Learning (CDCL), an approach which does not parallelize well [Man16, FF12, BSS+12]. The clause learning and conflict analysis add irregularities to the execution which becomes expensive due to the SIMT architecture of the GPUs and also adds additional effort for synchronization on parallel architectures [Cos13]. Other solvers such as c2d [Dar04] and d4 [LM17] rely on knowledge compilation.

Fichte et. al. [FHMW17] used dynamic programming for solving the #ASP problem in their solver called dynasp. Charwat and Woltran [CW16] developed a solver called

dynQBF, which uses dynamic programming to solve QBF formulas. Both of these solvers work serial on the CPU.

Burchard et al. [BSB15] introduce a parallel #SAT solver called countAntom which runs on the CPU. As basis for their solver they used the Davis-Putnam-Logemann-Loveland (DPLL) algorithm which was introduced by Davis et al. [DLL62].

Different approaches to solve the SAT problem on the GPU exist, one approach is from Deleau et al. [DJK08]. They developed a solver called GPU4SAT which is based on matrix multiplications. For solving they use two matrices, the boolean formula is encoded into the first matrix, and multiple variable assignments are encoded into the second matrix, then the two matrices are multiplied. For each cell in the resulting matrix a thread is started on the GPU. Another approach is by Palù et al. [PDFP15] who developed a system called CUD@SAT. Similar to Burchard at al. [BSB15] they modified the DPLL algorithm for their solver. Costa [Cos13] used minisat [ES03] as base for his solver, and partially parallelized it on the GPU. Our approach evaluates a SAT formula along the path of a tree decomposition in contrast to the methods mentioned before. For all of these SAT solvers CUDA was used, which is a proprietary standard from the NVIDIA Corporation, and can only be used on their GPUs in contrast to OpenCL which was used for our solver.

1.5 Thesis Outline

The remainder of this thesis is structured as follows: Chapter 2 provides the reader with the necessary background for this thesis. In Chapter 3 we describe our dynamic programming approach and the algorithms used, starting with the primal algorithm, then the incidence and the dual algorithms. Next, Chapter 4 provides the reader with an in-depth view of our implementation, challenges we faced and the techniques we used to overcome these challenges. Chapter 5 contains a description of our benchmark setting, the benchmark results and a short discussion. In the final chapter of this thesis we conclude our work and give hints for further work.

CHAPTER 2

Background

This chapter provides the reader with a brief background for this work. We start with basics on #SAT and WMC, followed by the definition of Tree Decompositions (TDs) and the different graph types used as basis for our solving algorithms. At the end of the chapter we also provide an introduction to OpenCL.

2.1 #SAT and WMC

A *literal* is a boolean variable v or its negation $\neg v$. A *clause* is a finite set of literals. A clause is interpreted as the disjunction of its literals. A clause c is called *unit* if $|c| = 1$. A CNF *formula* is a set of clauses and is interpreted as the conjunction of its clauses. In Example 2.1 we can see an example of a SAT formula. We define $var(C)$ as the set of variables contained in the clause or clause set C. An *assignment* α maps variables in a formula to 0 or 1, $\alpha : var(C) \rightarrow 0, 1$. A clause c is satisfied by an assignment if for some variable $v \in var(c)$ we have $v \in c$ and $\alpha(v) = 1$ or $\neg v \in c$ and $\alpha(v) = 0$. Otherwise the assignment falsifies the clause. An assignment satisfies a formula if each clause in the formula is satisfied by the assignment. A set C of clauses is *unfalsifiable* if there is no truth assignment that falsifies all clauses in C. C is only unfalsifiable if there exists a variable $a \in var(C)$ such that $a \in C$ and $\neg a \in C$. C is *falsifiable* if there is a truth assignment that falsifies all clauses in C. C is *satisfiable* if there is a truth assignment that satisfies all clauses in C. C is *unsatisfiable* if there does not exist a truth assignment that satisfies all clauses in C.

Example 2.1. Assume the CNF formula $C = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8\}$ with the clauses $c_1 = \{v_1, v_3, \neg v_4\}$, $c_2 = \{\neg v_1, v_6\}$, $c_3 = \{\neg v_2, \neg v_3, \neg v_4\}$, $c_4 = \{\neg v_2, v_6\}$, $c_5 = \{\neg v_3, \neg v_4\}$, $c_6 = \{\neg v_3, v_5\}$, $c_7 = \{\neg v_5, \neg v_6\}$, $c_8 = \{v_5, v_7\}$. If we take the clauses c_8 and c_6 then a truth assignment that falsifies the two clauses is $\alpha(v_5) = 0$ and $\alpha(v_7) = 0$ and $\alpha(v_3) = 1$. An example for a satisfying assignment of c_8 and c_6 would be $\alpha(v_5) = 1$ and $\alpha(v_7) = 0$ and

v_1	v_2	v_3	v_4	v_5	v_6	v_7
1	0	0	0	0	1	1
0	0	0	0	0	1	1
0	0	0	0	1	0	1
0	0	0	0	0	0	1
0	0	1	0	1	0	1
1	0	0	1	0	1	1
0	0	0	0	1	0	1
0	0	1	0	1	0	0
1	1	0	0	0	1	0
0	1	0	0	0	1	1
1	1	0	1	0	1	1

Table 2.1: The satisfying assignments for the formula from Example 2.1.

$\alpha(v_3) = 1$. Therefore the set $\{c_8, c_6\}$ is falsifiable and satisfiable, but neither unfalsifiable nor unsatisfiable. If we take the clause set $\{c_1, c_2\}$ the set would be unfalsifiable as the literals $v_1 \in c_1$ and $\neg v_1 \in c_2$ are in the clause set. As the set is unfalsifiable it is also satisfiable. The clause set $\{\{\neg v_1, \neg v_2\}, \{v_1, v_2\}, \{\neg v_1, v_2\}, \{v_1, \neg v_2\}\}$ is unsatisfiable and falsifiable as there is no assignment of the variables v_1 and v_2 that satisfies every clause.

The #SAT Problem.

Input	A formula C.
Question	How many assignments do the occurring variables satisfy in C?
Example	The formula from Example 2.1 has 11 satisfying assignments, which can be seen in Table 2.1.

In Weighted Model Counting (WMC) each literal is assigned a weight. The weight of an assignment $w(\alpha)$ is the product of all weights w of its literals. The *weighted model count* of a formula is the sum of the weights of all satisfying assignments of the formula.

Example 2.2. We assign each literal from Example 2.1 a weight. $w(v_1) = 1 - w(\neg v_1) = 0.8$, $w(v_2) = 1 - w(\neg v_2) = 0.2$, $w(v_3) = 1 - w(\neg v_3) = 0.1$, $w(v_4) = 1 - w(\neg v_4) = 0.7$, $w(v_5) = 1 - w(\neg v_5) = 0.4$, $w(v_6) = 1 - w(\neg v_6) = 0.5$. Then the resulting weighted model count of the formula is 0.13218.

The *primal graph* of a SAT formula contains a vertex for each variable of the SAT formula, and has an edge between two vertices if the node variables occur in the same clause. The *incidence graph* of a SAT formula contains a node for each variable and clause in the SAT formula, and an edge between a variable node and a clause node if the variable

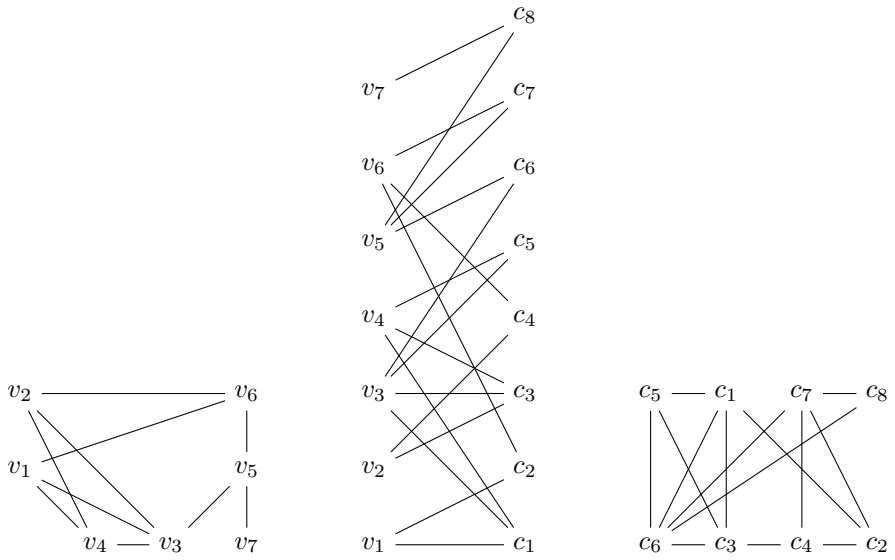


Figure 2.1: The primal (left), incidence (middle) and dual (right) graph for the SAT formula in Example 2.1

occurs in the clause. The *dual graph* of a SAT formula contains a node for each clause in the SAT formula, and has an edge between two clauses, if the two clauses have at least one common variable.

The three different graph types for the formula from Example 2.1 can be seen in Figure 2.1. The primal graph is on the left side, the incidence graph is in the middle and the dual graph is on the right side.

2.2 Tree Decompositions

Tree decompositions were originally introduced by Robertson and Seymour [RS84]. A *tree decomposition* (TD) of a graph G is a pair (T, χ). T is a tree and χ is a mapping which assigns each node $n \in V(T)$ a set $\chi(n) \subseteq V(G)$ called a *bag*. Then (T, χ) is a tree decomposition if the following conditions hold:

1. for each vertex $v \in V(G)$ there is a node $n \in V(T)$ such that $v \in \chi(n)$,
2. for each edge $(x, y) \in E(G)$ there is a node $n \in V(T)$ such that $x, y \in \chi(n)$, and
3. if $x, y, z \in V(T)$ and y lies on the path from x to z then $\chi(x) \cap \chi(z) \subseteq \chi(y)$.

The width of a tree decomposition $width(T)$ is $max_{n \in V(T)}(|\chi(n)|) - 1$. The tree width of a graph is the minimal width over all tree decompositions of the graph. An Example of a tree decomposition for the graphs from Figure 2.1 can be seen in Figure 2.2 for the primal graph, in Figure 2.3 for the incidence graph and in Figure 2.4 for the dual graph.

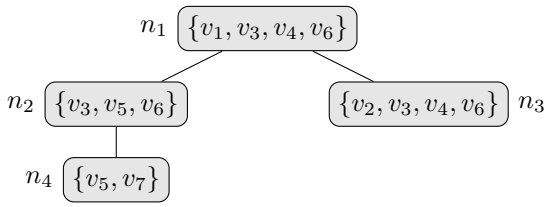


Figure 2.2: A possible decomposition of the primal graph from the formula in Example 2.1.

Nice Tree Decomposition: In order to simplify the cases in our algorithm we use so called nice tree decompositions. A tree decomposition T is called a *nice tree decomposition* if the root and leaf nodes are empty, and the type of each bag is one of:

- **leaf**: a node $n \in V(T)$ is called a leaf node if n has no child nodes
- **join**: a node n is called a join node if n has two child nodes n' and n'' with $\chi(n) = \chi(n') = \chi(n'')$
- **introduce**: a node n is called an introduce node if it has a child node n', $\chi(n) \supseteq \chi(n')$ and $|\chi(n)| = |\chi(n')| + 1$
- **forget**: a node n is called a forget node if it has a child node n', $\chi(n) \subseteq \chi(n')$ and $|\chi(n)| = |\chi(n')| - 1$

For every tree decomposition a nice tree decomposition can be computed within linear time without increasing the width [Klo94]. An example of a nice tree decomposition for the graph in Figure 2.2 can be seen in Figure 2.5.

Arnborg et. al. [ACP87] has shown, that finding a tree decomposition of at most width k is NP-complete. Therefore, computing minimal width tree decompositions is only feasible for small graphs. There are exact methods for obtaining minimal width tree decompositions, e.g. [GD04, BB06]. In this thesis we use heuristics for generating tree decompositions, therefore the width might not be minimal. An overview of different heuristic algorithms for computing the tree width is provided by Bodlaender and Koster [BK11, BK10] and Hammerl et. al. [HMS15].

2.3 OpenCL

In this section we give a brief introduction to OpenCL (Open Computing Language) version 1.2. For further details on OpenCL we refer to the OpenCL specification [Mun11]. OpenCL is an open standard used for parallel programming on heterogeneous platforms consisting of central processing units (CPU) and graphic processing units (GPU). It defines an application programming interface (API) and a programming language. Each hardware vendor has to provide drivers for their hardware. The OpenCL programming

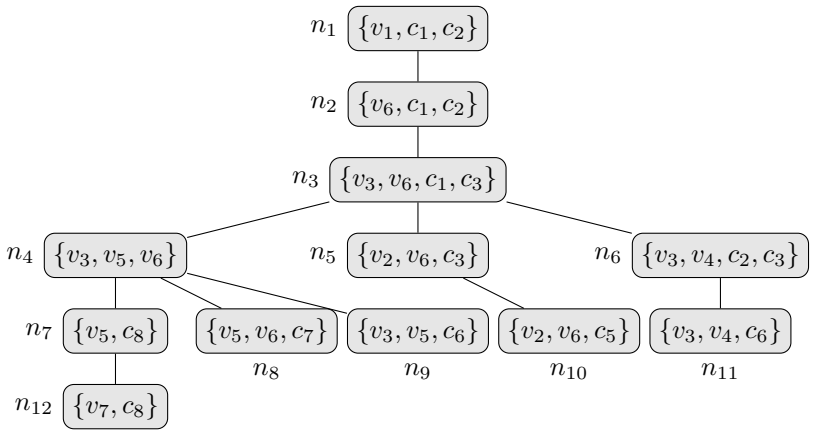


Figure 2.3: A decomposition of the incidence graph from the formula in Example 2.1.

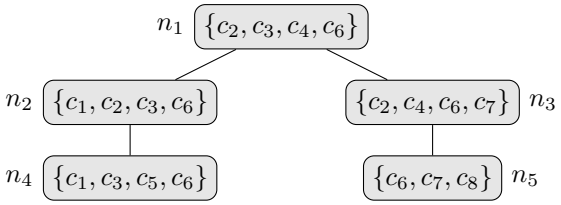


Figure 2.4: A decomposition of the dual graph from the formula in Example 2.1.

language is a subset of the ISO C99 standard, with additional functions needed for synchronization and image processing. OpenCL was released 2009 and belongs to the Khronos Group [1]. The most recent version of OpenCL is 2.2, but AMD and NVIDIA only support version 1.2 in their recent drivers. There exist older driver versions for AMD GPUs, which support OpenCL version 2.0.

A *kernel* is a function which can be executed on an OpenCL device. An OpenCL *device* can be a GPU or CPU. Each OpenCL device consists of one or more *compute units*. Each compute unit has its own local memory and one or more processing elements. Figure 2.6 illustrates a compute unit and its processing elements. *Processing elements* are virtual processors on which a command is executed. A *work-group* is a collection of work items which are executed on the same compute unit and share the same local memory. A *work item* is one execution of an OpenCL kernel. Each work item can be distinguished by its ID and can be executed on one or more processing elements.

There are two different parallel architectures in OpenCL, Single Instruction Multiple Data (SIMD) and Single Program Multiple Data (SPMD) architecture. In SIMD each thread on a compute unit shares the same instruction pointer and works on different data, that means if two threads take a different execution path they have to wait for the

[1] See: `https://www.khronos.org/`

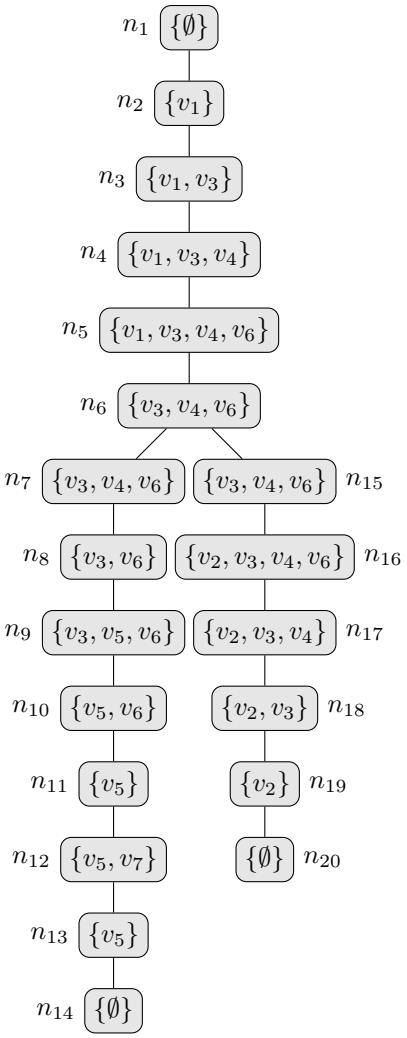


Figure 2.5: A nice tree decomposition of the decomposition in Figure 2.2

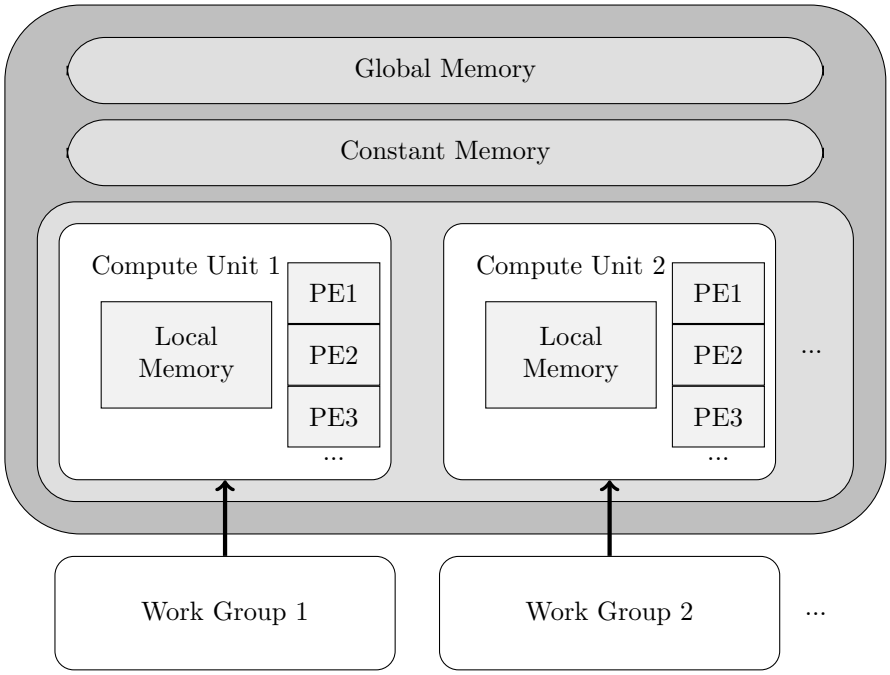


Figure 2.6: Hardware model of an OpenCL device.

other. In SPMD each thread has its own instruction pointer. GPUs work with the SIMD mode. Therefore, parallelization on the GPU works best if algorithms execute the same set of instructions.

Synchronization in OpenCL can be done via work-group barriers or command queue barriers. *Work-group barriers* are a mechanism to synchronize threads within a work group, such that all threads have to reach the barrier before any thread can advance further. *Command queue barriers* are for synchronization of jobs in the command queue all jobs before the barrier have to be finished, before the next job can be executed. Atomic operations can be used for synchronizing memory access.

There are 4 distinct memory regions defined by the OpenCL standard, which can be accessed by the kernel. These memory regions are:

1. **Global Memory**: The global memory is a memory region to which all threads have read and write access, it has to be allocated beforehand. On the GPU it corresponds to the local memory of the GPU (VRAM), it is also the slowest memory region to access on the GPU.

2. **Constant Memory**: The constant memory is a memory region to which all threads have read access, it has to be allocated and initialized beforehand. This region is used for information which does not change during the kernel execution.

3. **Local Memory**: The local memory is a memory region to which all threads of a work-group have access. It can be used to share information between threads of a work-group. Depending on the hardware the local memory can be accessed faster than the main memory, e.g., GPUs have a cache for each compute unit, but its size is much smaller than the global memory.

4. **Private Memory**: The private memory can only be accessed by the thread it belongs to. Local variables are stored in this memory region. This is the fastest memory region.

Buffered objects are used to copy data into the global and constant memory. They contain the start address and size of the memory region to copy, therefore only connected regions from the main memory can be copied into the global and constant memory.

In Listing 2.2 we can see a C++ program that calls an OpenCL kernel which adds two vectors, the kernel code can be seen in Listing 2.1, the Example was taken from `gist.github.com/ddemidov/2925717`. For better readability we have separated the GPU and CPU code.

The CPU code starts by searching for OpenCL GPU devices. The *cl::Platform::get* function queries all available OpenCL platforms, the *getDevices* function then returns all GPU devices. Then a command queue is created and the kernel is compiled. In the next step, buffers are created, which are used to copy chunks of memory from the RAM to the VRAM. enqueueNDRangeKernel then submits the kernel for execution. For each element in the first vector a thread on the GPU is started. In the first step, the kernel function queries its id and then adds the vector values, for which the vector index is equal to the id of the kernel. enqueueReadBuffer waits until the kernel is finished and then copies the resulting vector from the VRAM to the RAM.

```
__kernel void add(
    __global const float *a,
    __global const float *b,
    __global float *c
  )
{
  // get the kernel id
  size_t i = get_global_id(0);
  // sum up the vectors
  c[i] = a[i] + b[i];
}
```

Listing 2.1: An OpenCL kernel which adds two vectors - (a,b) and saves the result in the third vector - c. The Example is from `gist.github.com/ddemidov/2925717`.

```
// Get list of OpenCL platforms.
std::vector<cl::Platform> platform;
cl::Platform::get(&platform);

```

```
5  // Get first available GPU device.
6  cl::Context context;
7  std::vector<cl::Device> device;
8  for(auto p = platform.begin();
9    device.empty() && p != platform.end(); p++)
10 {
11   std::vector<cl::Device> pldev;
12
13   // Get a GPU
14   p->getDevices(CL_DEVICE_TYPE_GPU, &pldev);
15
16   for(auto d = pldev.begin();
17     device.empty() && d != pldev.end(); d++)
18   {
19     device.push_back(*d);
20     context = cl::Context(device);
21   }
22 }
23
24 // Create command queue.
25 cl::CommandQueue queue(context, device[0]);
26
27 // Compile OpenCL program for found device.
28 cl::Program program(context,
29   cl::Program::Sources(1, std::make_pair(source, strlen(source))))
       ;
30 program.build(device);
31
32 cl::Kernel add(program, "add");
33
34 // Prepare input data.
35 std::vector<float> a(1 << 20, 1);
36 std::vector<float> b(1 << 20, 2);
37 std::vector<float> c(1 << 20);
38
39 // Allocate device buffers and transfer input data to device.
40 cl::Buffer A(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
41   a.size() * sizeof(float), a.data());
42 cl::Buffer B(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
43   b.size() * sizeof(float), b.data());
44 cl::Buffer C(context, CL_MEM_READ_WRITE,
45   c.size() * sizeof(float));
46
47 // Set kernel parameters.
48 add.setArg(0, A);
49 add.setArg(1, B);
50 add.setArg(2, C);
51
52 // Launch kernel on the compute device.
```

```
53 queue.enqueueNDRangeKernel(add, cl::NullRange, N,
54   cl::NullRange);
55
56 // Get result back to host.
57 queue.enqueueReadBuffer(C, CL_TRUE, 0, c.size() * sizeof(float),
58   c.data());
```

Listing 2.2: Example for a programm that calls the Kernel from Listing 2.1. The Example is from `gist.github.com/ddemidov/2925717`.

CHAPTER 3

Dynamic Programming

In this chapter, we describe the dynamic programming algorithms we used to solve the #SAT problem. We start by describing how the dynamic programming approach on tree decompositions works in general. Then we introduce the algorithms used for solving, starting with the algorithm for the primal graph, followed by the incidence graph algorithm and the dual graph algorithm. For further information on the used algorithms we refer to the original source [SS10].

3.1 Dynamic Programming on Tree Decompositions

In dynamic programming a problem is split up into parts, and the parts are then solved individually. In our case, we use tree decompositions to split up the search space of the #SAT problem. The solving algorithm then evaluates the formula along the path of the tree decomposition in a bottom up order starting in one of the leafs. The solution count for each assignment in the bag is then stored in a *table*. The size of the table is exponential to the size of the bag. The tables of the child nodes can be deleted as soon as the table of the current node is generated.

The dynamic programming approach on the CPU for a given SAT formula F works as follows:

1. Generate the primal, incidence, or dual graph G of the SAT formula F.
2. Create a tree decomposition $\mathcal{T}$ of the graph.
3. Apply dynamic programming for each bag in bottom up order:
   a) visit the next node n of the tree decomposition,
   b) apply the solving algorithm $\mathbb{A}$ to the bag F_n, and

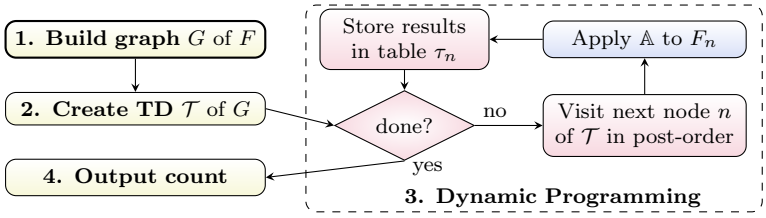


Figure 3.1: DP Algorithm on the CPU

c) store the results in a table τ_n.

4. Print the result based on the table of the root node of the tree decomposition.

3.2 Dynamic Programming Algorithms for #SAT

In this section, we describe the algorithms for each graph type, starting with the primal, then the incidence, and in the end the dual algorithm. The operations depend on the type of the tree decomposition node currently processed.

3.2.1 Primal Graph

The primal algorithm stores for a partial assignment α the number i of assignments which extend α and satisfy all clauses containing only variables from the subtree induced by the current nodes to the leaf nodes. The leaf operation is applied to the empty set of clauses which has one solution. In the introduce operation we expand the current assignment by a new variable, and check if the assignment satisfies all clauses which only contain variables from the current bag. If we forget a variable, we have already checked all clauses containing the variable. Hence, once a variable is forgotten it is not of further concern for the remaining tree decomposition nodes in the traversal. The join operation connects two subtrees with each subtree containing a different set of variables and clauses. If there are solutions for the current variable assignment in both subtrees then we know that the current variable assignment satisfies all clauses from both subtrees.

In Algorithm 3.1 we can see each operation of the primal algorithm. The algorithm gets as input the current node n of the tree decomposition, the bag χ_n containing the variables from the node, the clauses F_n which only contain variables from χ_n and the child tables *C-Tabs* of the current node. The solutions of bag χ_n are stored in table τ_n together with the variable assignment. For a formula F and an assignment α, $F(\alpha)$ returns all clauses which are not satisfied by the variable assignment α.

Example 3.1. We have a SAT formula with the following set of clauses: $C = \{ c_1 = \{\neg v_1, v_2, \neg v_4\}, c_2 = \{v_1, \neg v_3, v_5\}, c_3 = \{v_2, v_4\}, c_4 = \{v_2, \neg v_5\}, c_5 = \{v_3, \neg v_5\}\}$, the decomposition of the primal graph can be seen in Figure 3.2. The tables generated

Algorithm 3.1: Table algorithm $\mathbb{PRIM}(n, \chi_n, F_n, \text{C-Tabs})$.

Data: Node n, bag χ_n, clauses F_n, C-Tabs of node n. **Out:** Tab τ_n.

1 **if** type(n) = *leaf* **then** $\tau_n \leftarrow \{\langle \emptyset, 1 \rangle\}$;
2 **else if** type(n) = *intr*, $a \in \chi_n$ *is introduced*, $\tau' \in$ *C-Tabs* **then**
3 $\quad \tau_n \leftarrow \{\langle \alpha^+_{a \mapsto 1}, i \rangle \mid \langle \alpha, i \rangle \in \tau', F_n(\alpha^+_{a \mapsto 1}) = \emptyset\} \cup \{\langle \alpha^+_{a \mapsto 0}, i \rangle \mid \langle \alpha, i \rangle \in \tau', F_n(\alpha^+_{a \mapsto 0}) = \emptyset\}$
4 **else if** type(n) = *forg*, $a \notin \chi_n$ *is forgotten*, $\tau' \in$ *C-Tabs* **then**
5 $\quad \tau_n \leftarrow \{\langle \alpha^-_a, \sum_{\langle \beta, i \rangle \in \tau' : \alpha^-_a = \beta^-_a} i \rangle \mid \langle \alpha, \cdot \rangle \in \tau'\}$
6 **else if** type(n) = *join*, $\tau', \tau'' \in$ *C-Tabs s.t.* $\tau' \neq \tau''$ **then**
7 $\quad \tau_n \leftarrow \{\langle \alpha, i' \cdot i'' \rangle \mid \langle \alpha, i' \rangle \in \tau', \langle \alpha, i'' \rangle \in \tau''\}$
8 **return** τ_n

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

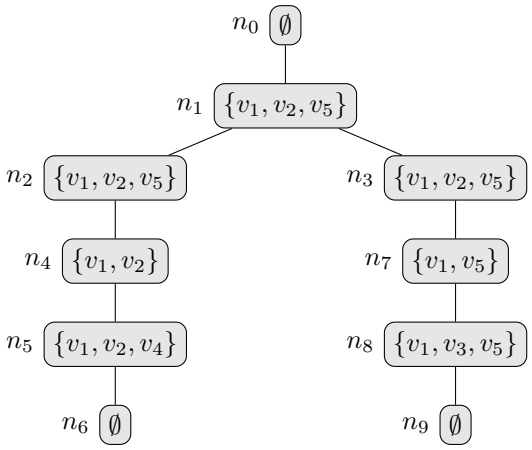


Figure 3.2: The tree decomposition of the primal graph of formula C from Example 3.1.

during solving can be seen in Figure 3.3. "n Sol" is the current number of solutions for the assignment. The number of satisfying assignments for the formula is 11 and can be seen in table τ_0.

3.2.2 Incidence Graph

The incidence algorithm stores a partial assignment α that can be extended to an assignment β that satisfies all already forgotten clauses from the subtree induced by the current node to the leaf nodes. Further, each row additional contains the clauses C from the current bag which are satisfied by β and the number i of these assignments. The leaf operation is applied to the empty set of clauses which has one solution. In the introduce clause operation we extend the current set of clauses by a new clause and check if the current assignment satisfies the clause. The introduce variable operation adds a new variable to the bag. The assignment of the new variable can satisfy a previously unsatisfied clause, therefore we need to sum up the model count for the newly satisfied

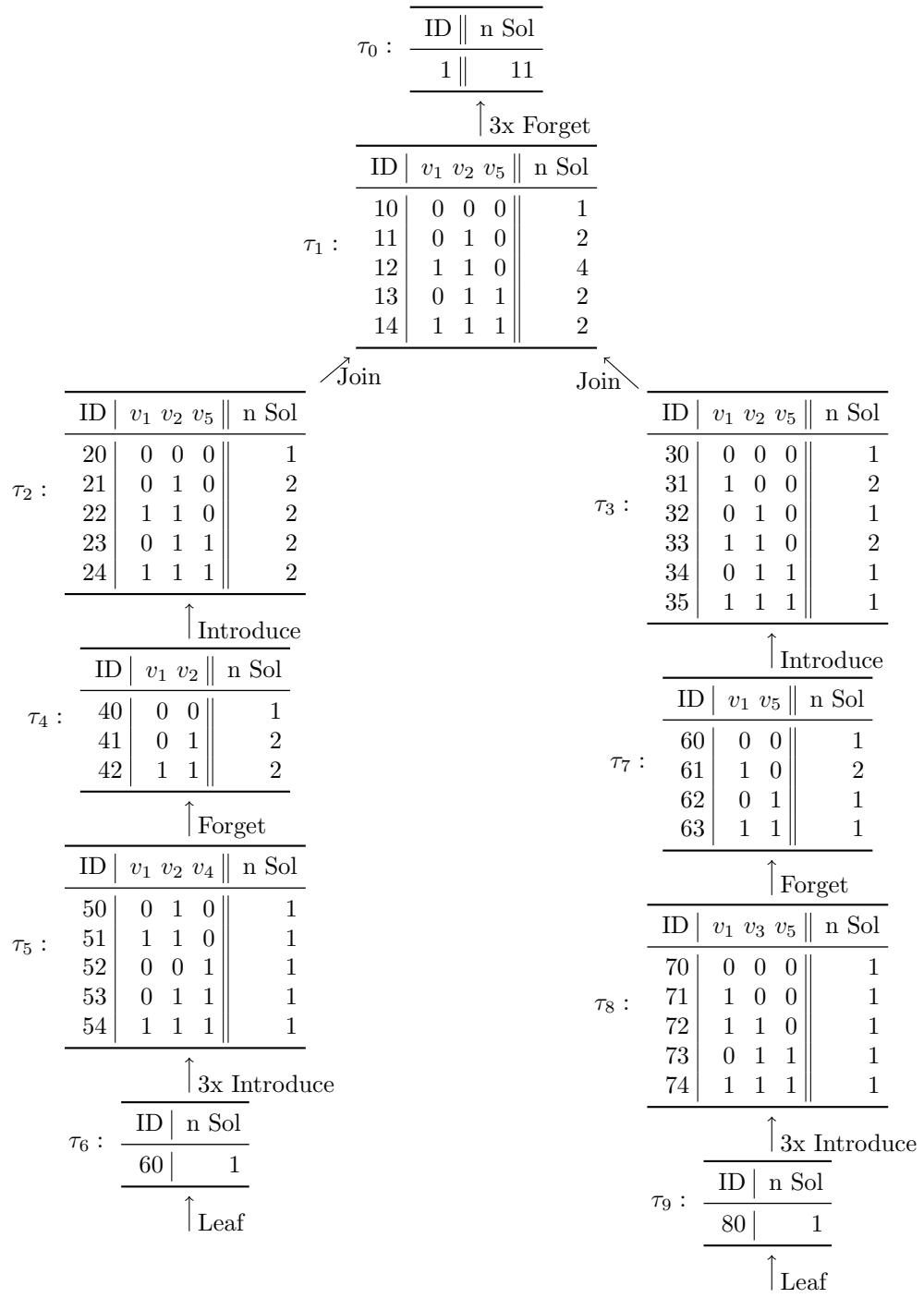


Figure 3.3: The primal algorithm solutions for the formula in Example 3.1.

Algorithm 3.2: Table algorithm $\mathbb{INC}(\chi_n, F_n, \text{C-Tabs})$.

Data: Bag χ_n, bag-clauses F_n, C-Tabs of node n. **Out:** Tab τ_n.

if type(n) = *leaf* **then** $\tau_n \leftarrow \{\langle \emptyset, \emptyset, 1\rangle\}$;
else if type(n) = *intr, variable a is introduced,* $\tau' \in$ *C-Tabs* **then**
  $\tau_n \leftarrow \{\langle \alpha^+_{a\mapsto 1}, C \cup D, \Sigma_{\langle \alpha, C', i\rangle \in \tau' : (C' \cup D) = (D \cup C)} i\rangle \mid \langle \alpha, C, \cdot\rangle \in \tau', D = SatCl(F_n, \alpha^+_{a\mapsto 1})\} \cup$
    $\{\langle \alpha^+_{a\mapsto 0}, C \cup D, \Sigma_{\langle \alpha, C', i\rangle \in \tau' : (C' \cup D) = (D \cup C)} i\rangle \mid \langle \alpha, C, \cdot\rangle \in \tau', D = SatCl(F_n, \alpha^+_{a\mapsto 0})\}$
else if type(n) = *intr, clause c is introduced,* $\tau' \in$ *C-Tabs* **then**
  $\tau_n \leftarrow \{\langle \alpha, C \cup SatCl(\{c\}, \alpha), i\rangle \mid \langle \alpha, C, i\rangle \in \tau'\}$
else if type(n) = *forg, variable a is forgotten,* $\tau' \in$ *C-Tabs* **then**
  $\tau_n \leftarrow \{\langle \alpha^-_a, C, \Sigma_{\langle \beta, C, i\rangle \in \tau' : \alpha^-_a = \beta^-_a} i\rangle \mid \langle \alpha, C, \cdot\rangle \in \tau'\}$
else if type(n) = *forg, clause c is forgotten,* $\tau' \in$ *C-Tabs* **then**
  $\tau_n \leftarrow \{\langle \alpha, C \setminus \{c\}, i\rangle \mid \langle \alpha, C, i\rangle \in \tau', c \in C\}$
else if type(n) = *join,* $\tau', \tau'' \in$ *C-Tabs s.t.* $\tau' \neq \tau''$ **then**
  $\tau_n \leftarrow \{\langle \alpha, C' \cup C'', \Sigma_{\langle \alpha, C', i'\rangle \in \tau', \langle \alpha, C'', i''\rangle \in \tau'' : C = C' \cup C''} i' \cdot i''\rangle \mid \langle \alpha, C', \cdot\rangle \in \tau', \langle \alpha, C'', \cdot\rangle \in \tau''\}$
return τ_n

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

clauses. If we forget a clause, we have already checked all variables which are contained in the clause and only keep the number of assignments for which the clause was satisfied. If we forget a variable, we have already checked all assignments of the variable for each clause which contains the variable and need to sum up the assignments where the variable occurs positively and negatively. The current variable assignment can then occur multiple times in the table with a different clause set. In the join operation we connect two subtrees with different variables and clauses. Clauses can be satisfied by the partial assignment in one tree, but not yet satisfied in the other. Therefore we need to collect the models for which the clauses are satisfied by the assignment in one of the subtrees.

In Algorithm 3.2 we can see how each operation of the incidence algorithm works. The solutions for the bag χ_n are stored in table τ_n together with the variable assignments and the satisfied clauses. The algorithm gets as input the current node n of the tree decomposition, the bag χ_n containing the variables and clauses, the clauses F_n from the current node and the child tables C-Tabs of the child nodes of the current node. $SatCl(C, \alpha)$ returns all clauses from the set C which are satisfied by the assignment α.

Example 3.2. We have a SAT formula with the following clause set: $C = \{ c_1 = \{\neg v_1, v_2, \neg v_3\}, c_2 = \{\neg v_2, v_3\}\}$. The decomposition of the incidence graph can be seen in Figure 3.4. Each table generated during solving can be seen in Figure 3.5. S means that a clause is satisfied, "n Sol" is the current number of solutions for the assignment. The number of satisfying assignments for the formula is 5 and can be seen in table τ_0.

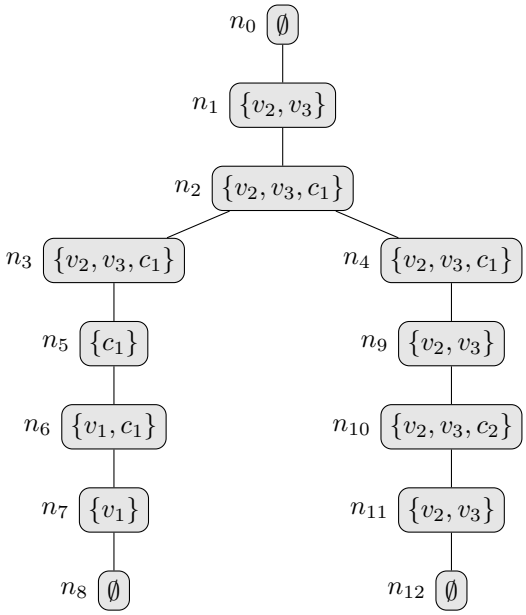


Figure 3.4: The tree decomposition of the incidence graph of formula C from Example 3.2.

3.2.3 Dual Graph

The dual algorithm saves a clause set ("falsified clauses") restricted to the current bag and the number of variable assignments restricted to the variables of the clauses in the subtree below, which falsify the clause set and satisfy all already forgotten clauses. In the leaf operation we set the model count to 1 since the empty interpretation (uniquely) falsifies the empty clause set and satisfies the (zero) already forgotten clauses. If we introduce a new clause we decide if we add it to the set of falsified clauses or not. If we do not add it do the falsified clauses we need to multiply the model count by the number of all possible assignments for the variables which are in the new clause, but were not in a previous clause. If the new clause is introduced as falsifiable we need to rule out all assignments for the variables from the new clause which were already in a clause from the bag, but not in one of the falsified clauses. In the forget operation we need to remove the number of assignments for which the forgotten clause is falsified. In the join operation we combine two subtrees of the tree decomposition. We need to remove the variable assignments which are in both subtrees, otherwise these assignments would be counted twice.

In Algorithm 3.3 we can see each operation of the dual algorithm. The solutions for the bag χ_n are stored in table τ_n together with the falsified clauses. The algorithm gets as input the current node n of the tree decomposition, the clauses F_n from the current bag and the tables C-Tabs of the child nodes of the current node.

Example 3.3. We have a SAT formula with the following clause set: $C = \{c_1 = \{v_1, v_2, \neg v_3\}, c_2 = \{v_1, \neg v_4\}, c_3 = \{v_2, \neg v_3\}, c_4 = \{\neg v_3, \neg v_4\}, c_5 = \{v_1\}\}$, the decomposition of the dual

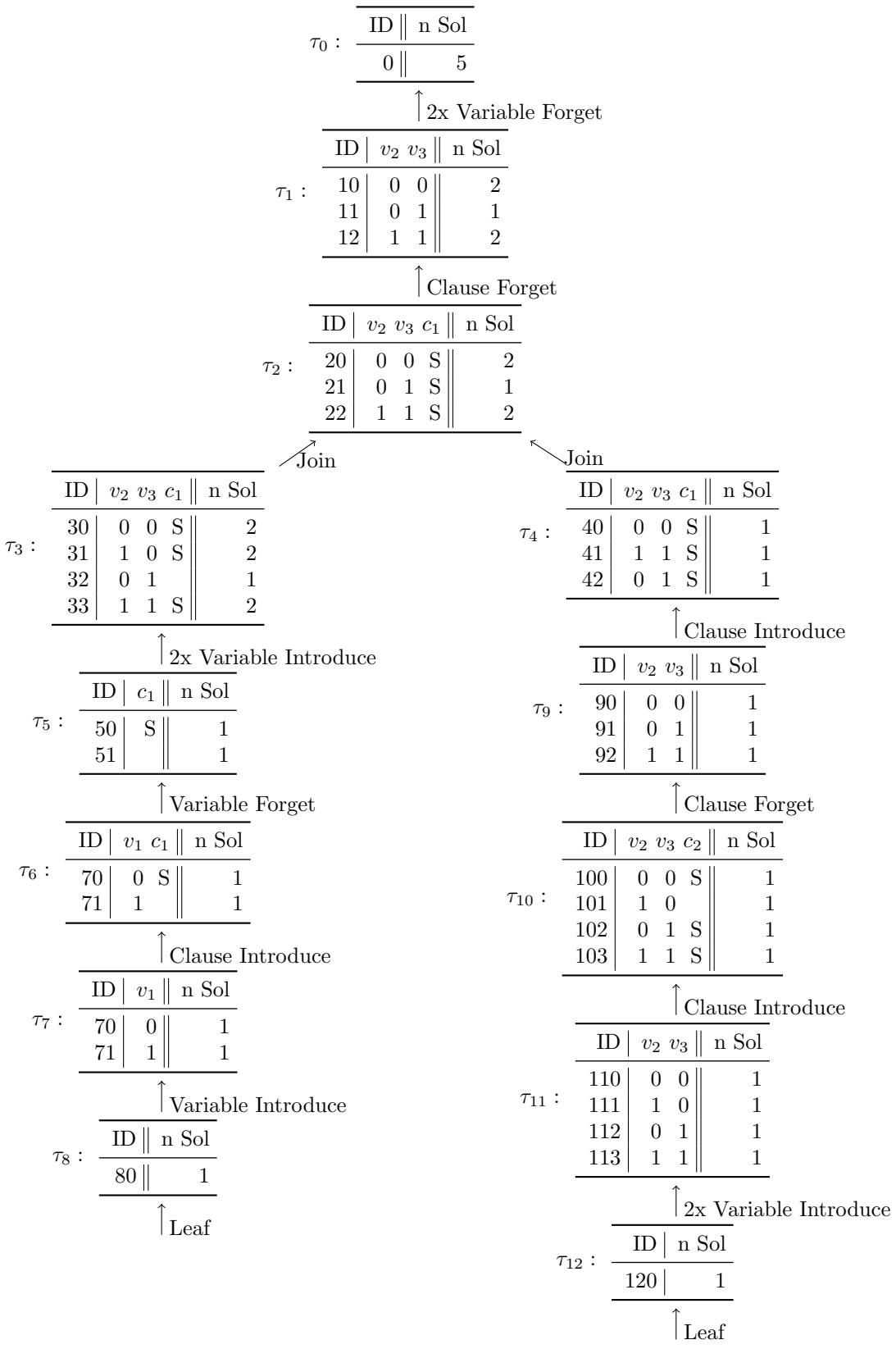


Figure 3.5: The incidence algorithm solutions for Example 3.2.

Algorithm 3.3: Table algorithm $\mathbb{DUAL}(t, \chi_n, F_n, \text{C-Tabs})$.

Data: Node n, bag χ_n, bag-clauses F_n, C-Tabs of node n. **Out:** Tab τ_n.

1 **if** type(n) = *leaf* **then** $\tau_n \leftarrow \{\langle \emptyset, 1 \rangle\}$;
2 **else if** type(n) = *intr, clause c is introduced, $\tau' \in$ C-Tabs* **then**
3 $\quad \tau_n \leftarrow \{\langle C, i * 2^{|var(c) \setminus var(\chi_{n'})|} \rangle \quad | \langle C, i \rangle \in \tau', C \text{ is } falsifiable\} \cup$
4 $\quad\quad \{\langle C \cup \{c\}, \frac{i}{2^{|var(c) \cap (var(\chi_{n'}) \setminus var(C))|}} \rangle \quad | \langle C, i \rangle \in \tau', C \cup \{c\} \text{ is } falsifiable\}$
5 **else if** type(n) = *forg, clause c is forgotten, $\tau' \in$ C-Tabs* **then**
6 $\quad \tau_n \leftarrow \{\langle C, i' - i'' \rangle \quad | \langle C, i' \rangle \in \tau', \langle C \cup \{c\}, i'' \rangle \in \tau'\}$
7 **else if** type(n) = *join, $\tau', \tau'' \in$ C-Tabs s.t. $\tau' \neq \tau''$* **then**
8 $\quad \tau_n \leftarrow \{\langle C, \frac{i' \cdot i''}{2^{|var(\chi_n) \setminus var(C)|}} \rangle \quad | \langle C, i' \rangle \in \tau', \langle C, i'' \rangle \in \tau''\}$
9 **return** τ_n

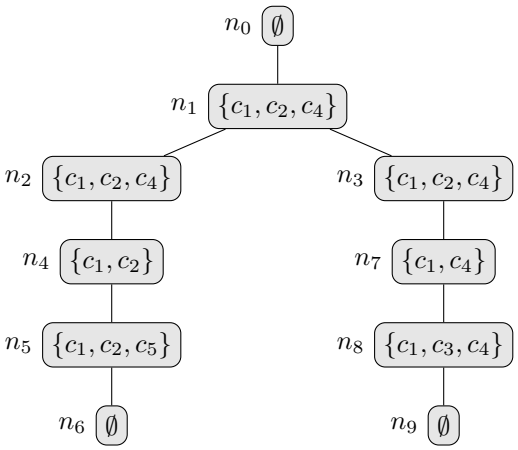


Figure 3.6: The tree decomposition of the dual graph for of formula from Example 3.3

graph can be seen in Figure 3.6. Each table generated during the solving algorithm can be seen in Figure 3.7. S means that a clause is satisfied, "n Sol" is the number of solutions for each clause set. In our example we do not store lines where "n Sol" is 0. The number of solutions for the formula is 5 and can be seen in table τ_0.

3.2.4 Dynamic Programming Algorithms for WMC

With some modifications the primal and incidence graph algorithms from the last section can also be used for weighted model counting. The current version of the dual graph algorithm can not be used for weighted model counting as it does not distinguish different literals in different clauses.

Primal Graph For the primal graph algorithm we need to modify the introduce and the join operations. In the introduce operation we have to multiply the current model count by the literal weight of the newly introduced variable. In case of an join operation

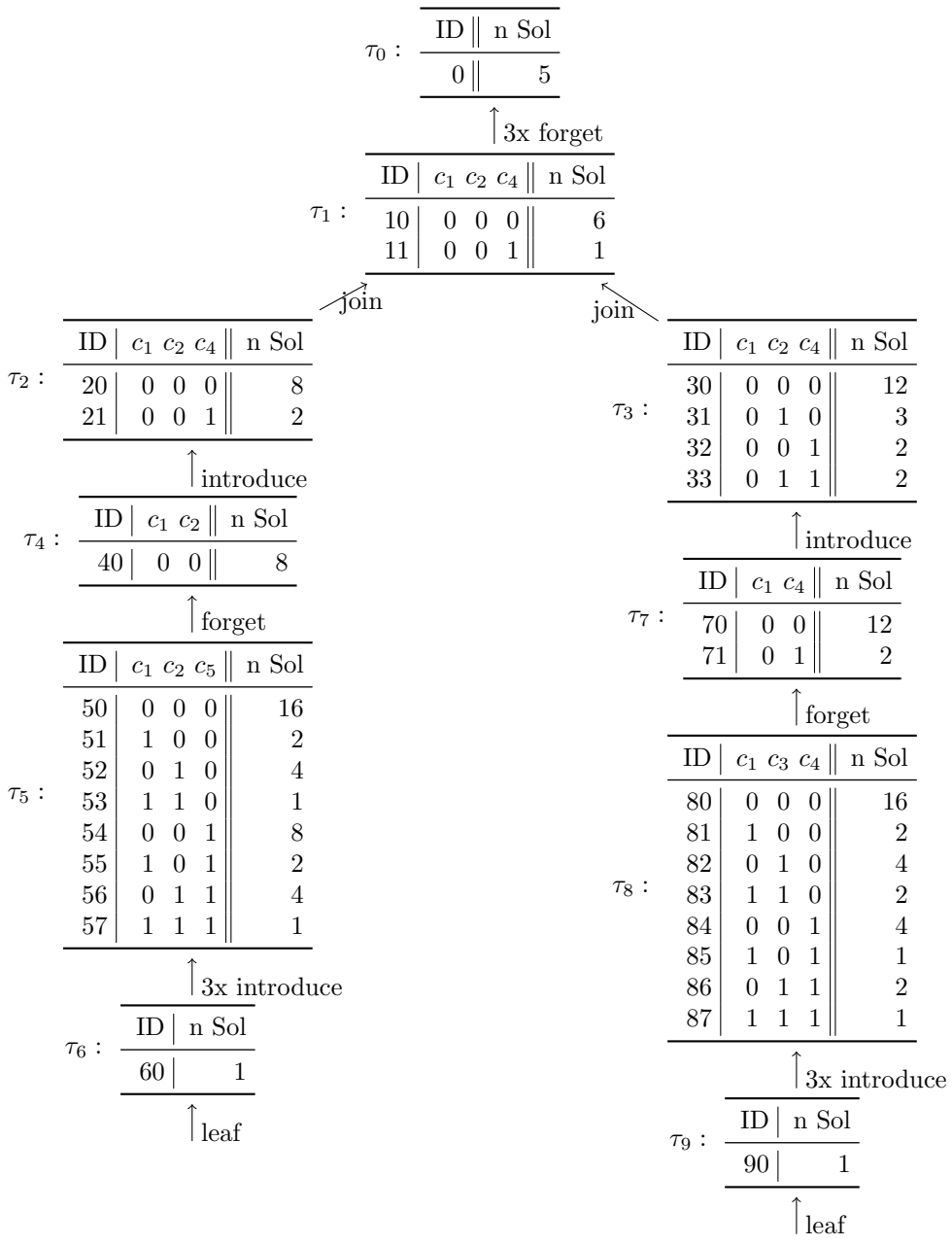


Figure 3.7: The dual algorithm solutions for Example 3.3.

we need to divide the model count by the weight of the literals of the variables in the current bag as these variables were introduced in both branches of the decomposition therefore the product of the literals for the assignment contains the literal weights twice.

Incidence Graph For the incidence graph algorithm we need to modify the join and the introduce variable operations. For the introduce variable operation we need to multiply the current model count with the literal weight of the newly introduce variable. For the join operation we need to divide the model count by the literal weight of the variables which are in the current bag as the literal weight is contained in both subtrees.

CHAPTER 4

Implementation

In this chapter we describe our implementation. We start with the challenges, one is faced with when implementing an algorithm on the GPU. Then we describe our techniques for dynamic programming on the GPU. The last part of this chapter is about the architecture of our program.

4.1 Challenges

1. OpenCL does not allow function calls as they are not supported by most GPUs. It is possible to define functions in the kernel code, but these functions are then inlined by the compiler. Therefore we need non recursive versions of the algorithms or we can not execute them on the GPU.

2. Only primitive data types are supported by OpenCL. Advanced data structures such as maps and tables or arbitrary precision types are not supported. Especially the lack of a big number type for saving the solutions is a problem. Therefore, we need to implement our own data structures.

3. The SIMT execution model on the GPU. We need to write our code in a way such that each thread on the GPU has the same execution path, otherwise the threads have to wait for each other.

4. In general the VRAM is smaller than the RAM. Therefore we need to split up tables if they are too big for the VRAM.

5. Only connected memory regions can be copied between the VRAM and RAM. Therefore, we can not copy structures that contain pointers from the RAM to the VRAM.

6. The kernel itself can not allocate memory. Therefore, all the memory the kernel uses has to be allocated before its execution by the CPU code.

7. Kernels should access the same memory region, but not the same address of the global memory. A memory region is cached if a thread accesses it. Therefore, consecutive accesses to the same memory region are faster, but only one thread can access the same address at once.

4.2 Techniques

Our dynamic programming approach on the GPU works slightly different than the approach we presented in Chapter 3. In contrast to the algorithms described before, we also save negative cases as OpenCL does not offer advanced data structures such as tables or maps. A general outline of our approach can be seen in Figure 4.1. The GPU algorithm works as follows:

1. The primal, incidence, or dual graph of the formula is generated.
2. Then the tree decomposition $\mathcal{T}$ of the graph is generated and the tree decomposition and formula are preprocessed. The preprocessing consists of two parts:
   a) Unit clauses are removed from the formula together with the variables contained in the clauses.
   b) Small bags of the decomposition are combined to better utilize the GPU.
3. In the next step we iterate over all nodes of the tree decomposition in depth first order and for each bag:
   a) We split up the result table τ_t into chunks C if the table containing the model counts does not fit into the main memory of the GPU (VRAM), then for each chunk:
      i. Get the next chunk of the child table and execute the kernel $\mathbb{K}$ for each row in the current chunk.
      ii. Add the resulting model count to the current model count
4. Then we calculate the model count based on the contents of the last table.

4.2.1 Merge Operations

The algorithms are easier to explain with nice tree decompositions, but non nice tree decompositions are faster for solving. Therefore we combined the leaf, introduce and forget operations into one introduce forget (IF) operation. Thereby, reducing the overhead for memory operations on the GPU and the overhead for copying and retrieving tables from the GPU. Another advantage is that we can reduce the size of the tables we need to

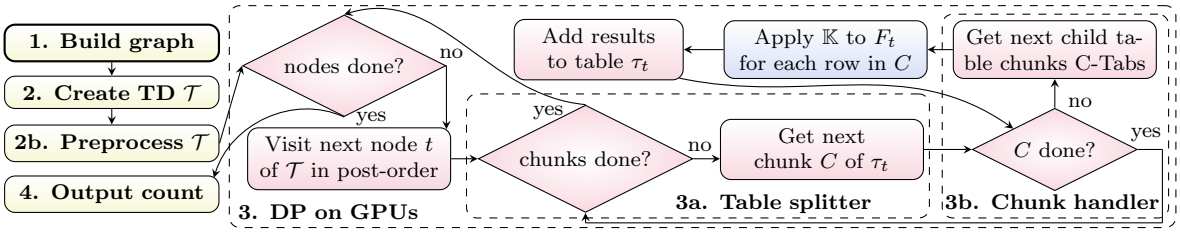


Figure 4.1: DP Algorithm on the GPU

store by forgetting the variables which do not occur in the next bag after each introduce operation.

The **introduce forget** (IF) operation for bag b works as follows:

1. We generate a new bag b' which contains the elements $b' = b \cap b_p$ with b_p being the parent of b.
2. Then we introduce the new elements from bag b and forget the elements which are not in b_p.
3. In the end bag b is replaced by b'.

Our **join** operation on bag b works as follows:

1. We take the first child bag b' and second child bag b'' and generate a new bag b_n containing the elements $b_n = b' \cup b''$.
2. Then we introduce the elements which are missing in the child bags and join them together into bag b_n.
3. For each remaining child bag b_c we execute the following:
   a) We generate a new bag b' with the elements $b' = b_c \cup b_n$ with b_n being the bag generated in the last step.
   b) Then we introduce the elements which are missing in the bags b_c and b_n and join them together into bag b'.
4. In the end we execute an IF operation on the bag which was generated in the last step to introduce the elements which occur in b, but not in its child bags.

Example 4.1. Example for the primal algorithm. We have a SAT formula with the following clause set: $C = \{c_1 = \{v_1, v_4, v_6\}, c_2 = \{v_1, \neg v_5\}, c_3 = \{\neg v_1, v_7\}, c_4 = \{v_2, v_3\}, c_5 = \{v_2, v_5\}, c_6 = \{v_2, \neg v_6\}, c_7 = \{v_3, \neg v_8\}, c_8 = \{v_4, \neg v_8\}, c_9 = \{\neg v_4, v_6\}, c_{10} = \{\neg v_4, v_7\}\}$, the decomposition of the primal graph can be seen in Figure 4.2. The tables generated during solving can be seen in Figure 4.3. The variable assignment is encoded in the binary

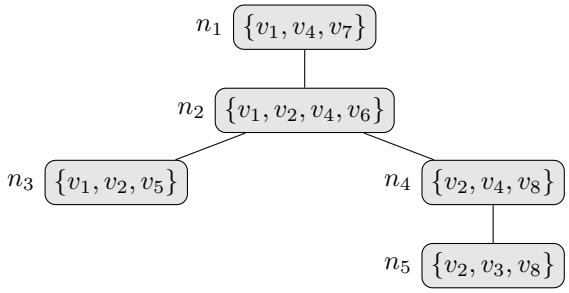


Figure 4.2: The tree decomposition of the primal graph of the formula from Example 4.1.

representation of the ID, "n Sol" is the current number of solutions for the assignment. The tables in brackets represent temporary results. The solving works as follows:

1. The solving algorithm starts at node n_3 with an IF operation. The IF operation generates a new bag containing the elements $\{v_1, v_2\}$. The result of the IF operation can be seen in table τ_5. Table τ_6 contains the temporary results generated by the introduce operation. The new bag then replaces the bag of node n_3.

2. Next, node n_5 will be solved with an IF operation. For the IF operation a new bag is generated containing the elements $\{v_2, v_8\}$. The new bag then replaces the bag of node n_5. The result of the IF operation can be seen in table τ_9. Table τ_{10} contains the temporary results generated by the introduce operation.

3. Then node n_4 is solved with an IF operation, the solutions of the operation can be seen in table τ_7. The temporary results of the introduce operation are in table τ_8. The bag generated by the IF operation contains the elements $\{v_2, v_4\}$. The new bag then replaces the bag of node n_4.

4. In the next step, node n_2 is solved. As the bag of n_2 contains elements which are not in its child nodes, we create a new bag containing the elements $\{v_1, v_2, v_4\}$ and execute a join operation. The result of the join can be seen in table τ_4. Then we use an IF operation on node n_2 to introduce v_6 and forget $\{v_2, v_6\}$. The result of the IF operation is in table τ_2 with the temporary results in table τ_3. the bag of node n_2 is then replaced by the new bag which contains the elements $\{v_1, v_4\}$.

5. The last node which remains to solve is node n_1 which is solved with an IF operation, the result of the operation can be seen in table τ_1.

In the end, the solutions of the last bag are summed up and the final solution count is 22.

Example 4.2. Example for the incidence algorithm. We have a SAT formula with the following clause set: $C = \{c_1 = \{\neg v_1, \neg v_2, \neg v_3\}, c_2 = \{\neg v_1, v_4\}, c_3 = \{v_2, \neg v_3\}, c_4 = \{\neg v_2, \neg v_4\}\}$. The decomposition of the incidence graph can be seen in Figure 4.4. Each

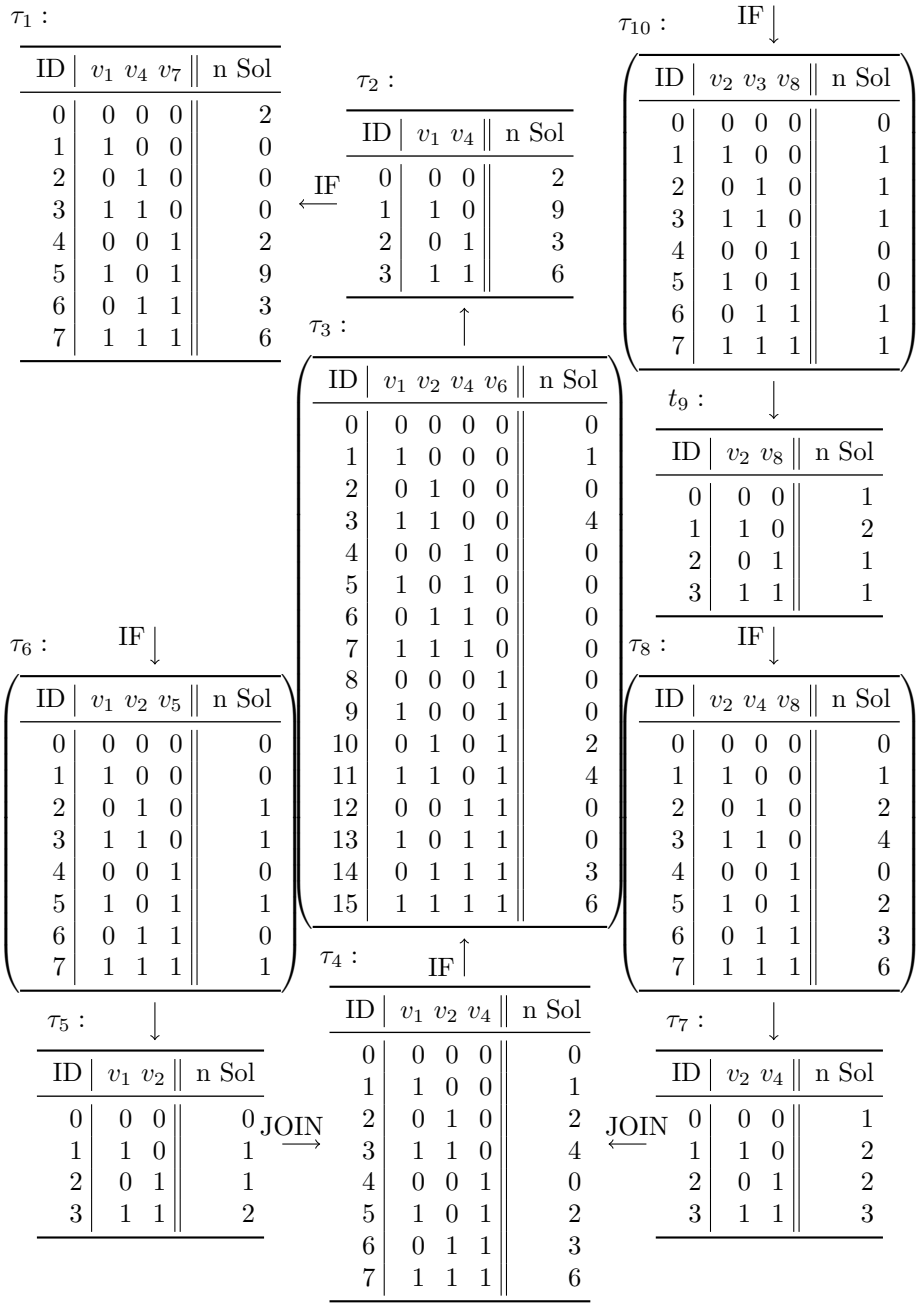


Figure 4.3: The primal algorithm solutions for the formula in Example 4.1.

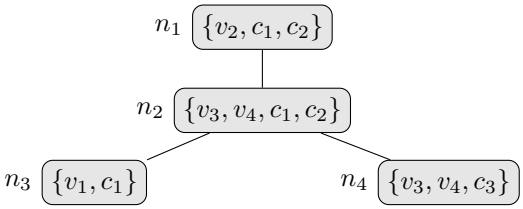


Figure 4.4: The tree decomposition of the incidence graph of the formula from Example 4.2.

table generated during solving can be seen in Figure 4.5. The assignment for the variables and clauses is encoded in the binary representation of the ID, "n Sol" is the number of solutions for each assignment. The tables in brackets represent temporary results. Lines which we did not need to calculate are marked with a "-" in the temporary tables. The solving works as follows:

1. The solving starts with node n_3 with an IF operation. The result of the IF operation can be seen in table τ_7 and the temporary result of the introduce is in table τ_8. The new bag containing $\{c_1\}$ then replaces the bag of node n_3.

2. Next is node n_4 which is solved with an IF operation. The result is in table τ_{10} and the temporary results of the introduce operation are in table τ_{11}. The new bag generated during the IF operation, which contains the elements $\{v_3, v_4\}$ then replaces the bag of node n_4.

3. Next is the join of node n_3 and node n_4 into node n_2. Therefore, we create a new node n' with the bag of n' containing the elements of the bag from node n_3 and bag of node n_4. Then we introduce the elements and clauses which are in the bag of n', but not in the bag of node n_3 into node n_3 which generates a new bag containing the elements of the bag of n', the new bag then replaces the bag of node n_3. The results of the introduce can be seen in table τ_6. The same is done with node n_4 with the results in table τ_9. Then node n_4 and node n_3 are joined into node n'. The result of the join can be seen in table τ_5. After the join we execute an IF operation on n'. The result of the IF operation can be seen in table τ_3 and the temporary results of the introduce operation are in table τ_4.

4. The last node to solve is n_1 with an IF operation. The result of the IF operation is in table τ_1. In the last node we always forget the remaining clauses to reduce the table size.

In the end we need to sum up the solution count from table τ_1 and the final result is 12.

Example 4.3. Example for the dual algorithm. We have a SAT formula with the following clause set: $C = \{c_1 = \{v_1, \neg v_2\}, c_2 = \{v_1, v_6\}, c_3 = \{v_1, v_7\}, c_4 = \{\neg v_2, v_3\}, c_5 = \{v_2, \neg v_5\},$

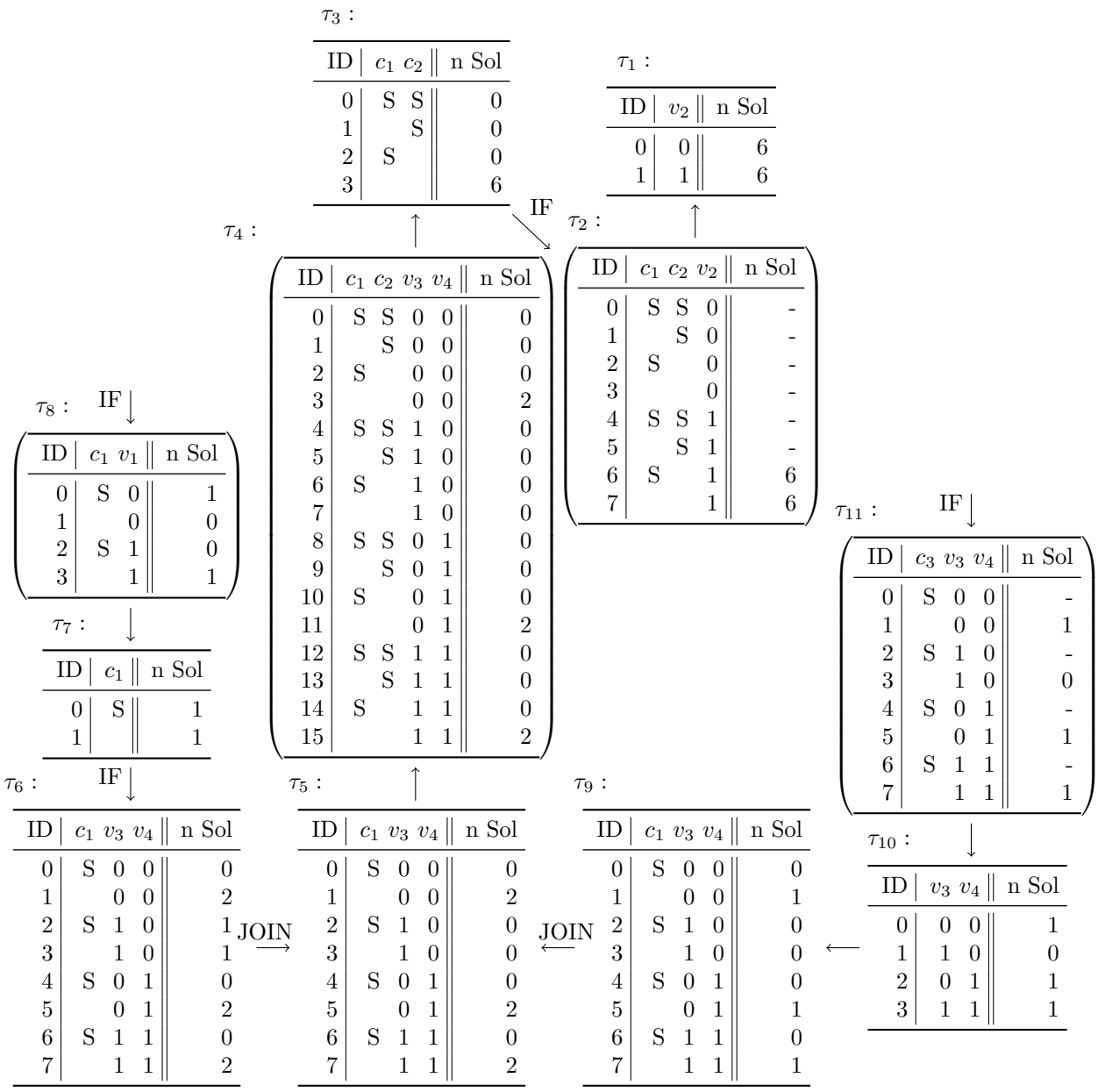


Figure 4.5: The incidence algorithm solutions for Example 4.2.

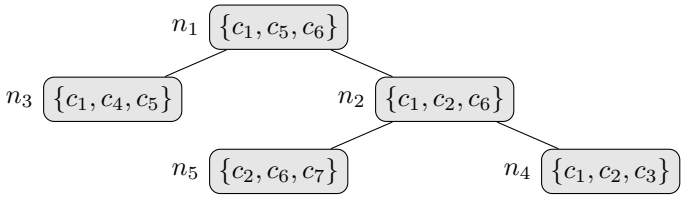


Figure 4.6: The tree decomposition of the dual graph of the formula from Example 4.3

$c_6 = \{v_4, \neg v_5\}$, $c_7 = \{\neg v_4, v_6\}\}$, the decomposition of the dual graph can be seen in Figure 4.6. Each table generated during the solving algorithm can be seen in Figure 4.7. The clause set is encoded into the binary representation of the ID of each table, 0 means that the clause is not in the set and 1 means that the clause is in the set, "n Sol" is the number of solutions for each clause set. The tables in brackets represent temporary results. The solving works as follows:

1. The algorithm starts by solving node n_3 which is solved with an IF operation. The resulting bag of the IF operation contains the clauses $\{c_1, c_5\}$. The result of the operation can be seen in table τ_2 and the temporary results of the introduce are in table τ_3.

2. Next, node n_5 is solved with an IF operation. The resulting bag which replaces the bag of node n_5 contains the clauses $\{c_2, c_6\}$. The result of the operation can be seen in table τ_7 with the temporary results of the introduce in table τ_8.

3. In the next step we solve node n_4 with an IF operation. The resulting bag contains the clauses $\{c_1, c_2\}$. The result of the IF operation can be seen in table τ_6.

4. Then we join the nodes n_4 and n_5 together into node n_2, the resulting bag contains the clauses $\{c_1, c_2, c_6\}$. The result can be seen in table τ_6. Afterwards, we execute an IF operation on n_2 and forget the clauses which are not in the bag of node n_1, the new bag created by the IF operation contains the clauses $\{c_1, c_6\}$. The result of the IF operation is in table τ_4.

5. In the last step we join node n_3 and n_2 into n_1, the result of the join is in table τ_1.

The final solution count is 48 - 4 - 16 + 0 - 15 + 1 + 10 - 0 = 24.

4.2.2 Table Splitting

If the table for a node in the tree decomposition is too large for the VRAM, we need to split the table into parts and compute the result for each part individually. The kernel is then executed with all possible combinations of parent and child table parts.

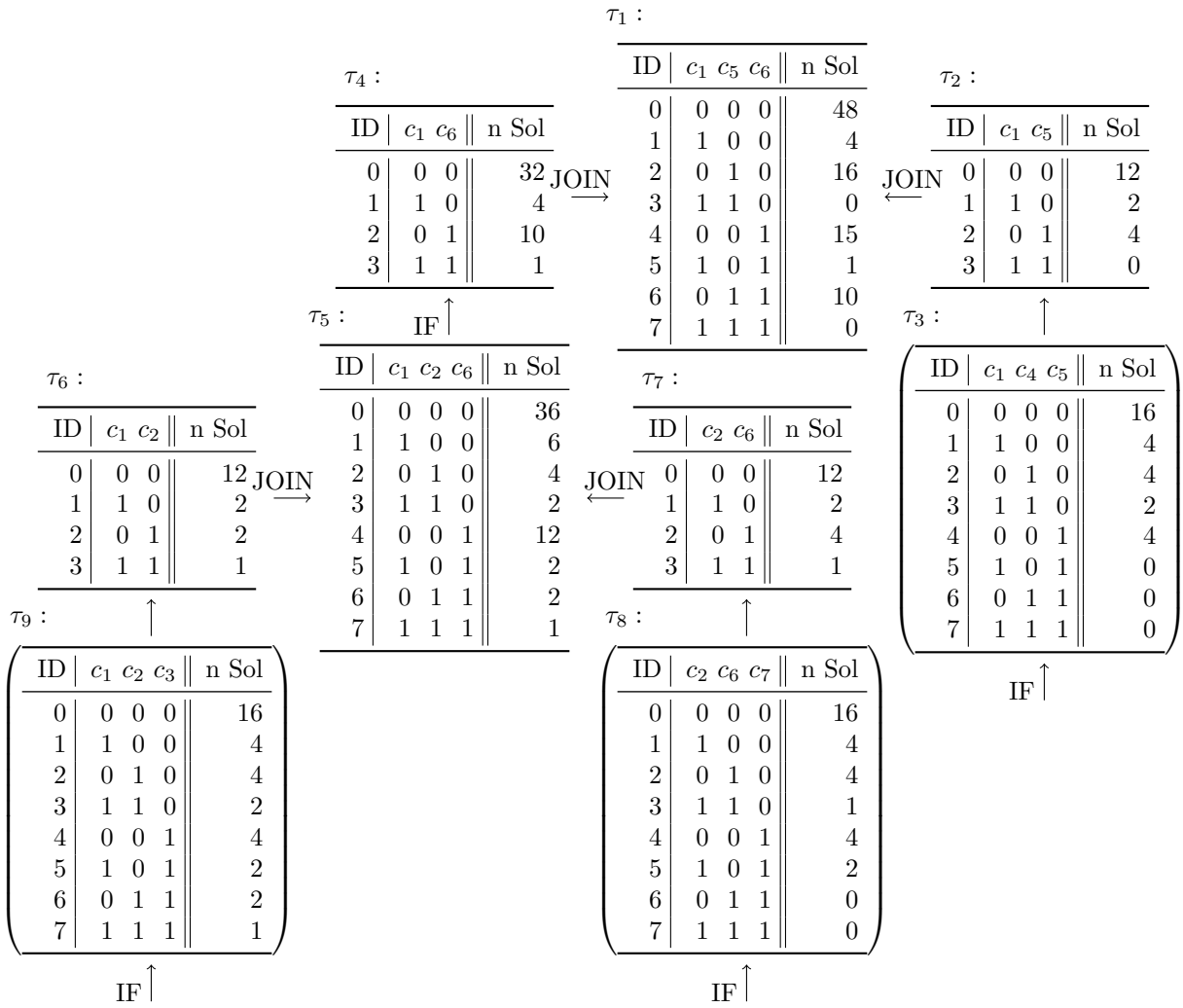
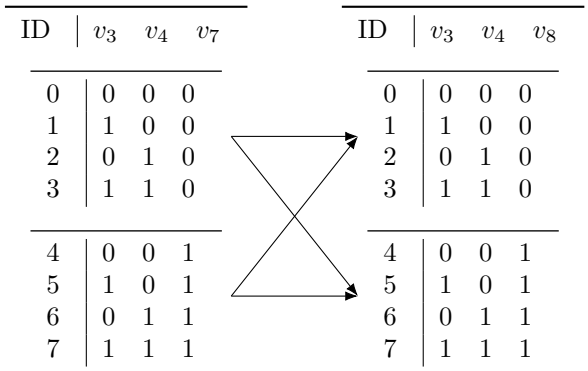


Figure 4.7: The dual algorithm solutions for Example 4.3.



ID	v_3	v_4	v_7
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
4	0	0	1
5	1	0	1
6	0	1	1
7	1	1	1

ID	v_3	v_4	v_8
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
4	0	0	1
5	1	0	1
6	0	1	1
7	1	1	1

Figure 4.8: Example of splitting for an introduce forget operation with splitting at bag size 2.

call #	current table	child table
1	part 1	part 1
2	part 1	part 2
3	part 2	part 1
4	part 2	part 2

Table 4.1: The combinations of each call for the introduce forget operation for Figure 4.8.

In the case of an introduce forget operation we have to combine each part of the child table with each part of the current table. In Figure 4.8 we can see an example of splitting for an introduce forget operation. Our current table is on the left side and our child table is on the right side. The combinations of tables we need to call our kernel with can be seen in Table 4.1.

In case of a join operation on the primal or dual graph we have to combine each part of the current node with each part of the child nodes. In Figure 4.9 we can see an example for splitting during a join operation on the decomposition of the primal or dual graph. The table in the middle with the variables (v_3,v_4,v_5,v_6) is our current table. The combinations of kernel executions can be seen in Table 4.2.

In case of a join operation on the incidence graph we have to combine each part of the current node with each combination of child node parts. In Figure 4.10 we can see an example for splitting during a join operation on the decomposition of the incidence graph, the table in the middle is our current table. The combinations of kernel executions can be seen in Table 4.3.

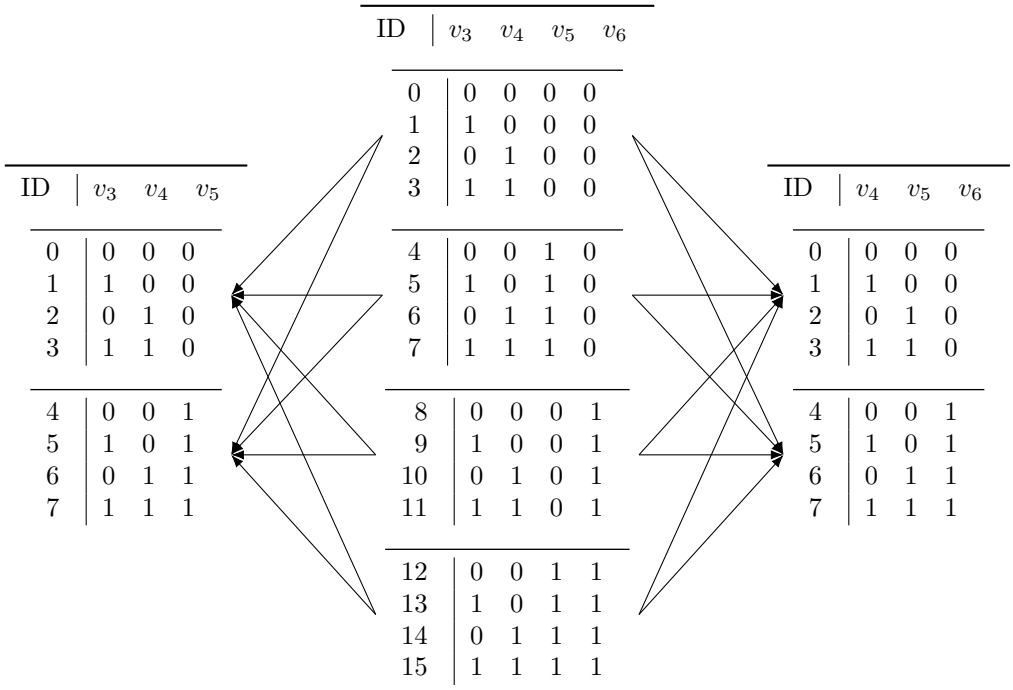


Figure 4.9: Example of splitting for a join operation of the primal and dual algorithm with splitting at bag size 2.

call #	current table	child table 1	child table 2
1	part 1	part 1	part 1
2	part 1	part 2	part 2
3	part 2	part 1	part 1
4	part 2	part 2	part 2
5	part 3	part 1	part 1
6	part 3	part 2	part 2
7	part 4	part 1	part 1
8	part 4	part 2	part 2

Table 4.2: In this table we can see the combinations of each call for the primal and dual join operation for Figure 4.9.

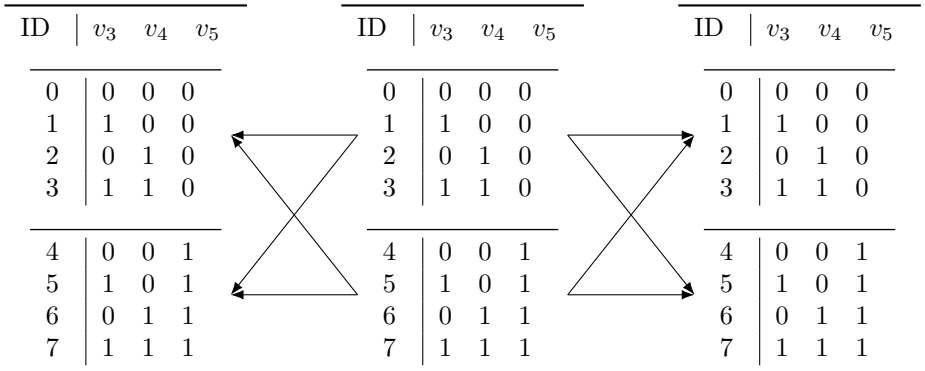


Figure 4.10: Example of splitting for a join operation with splitting at bag size 2.

call #	current table	child table 1	child table 2
1	part 1	part 1	part 1
2	part 1	part 1	part 2
3	part 1	part 2	part 1
4	part 1	part 2	part 2
5	part 2	part 1	part 1
6	part 2	part 1	part 2
7	part 2	part 2	part 1
8	part 2	part 2	part 2

Table 4.3: In this table we can see the combinations of each call for the incidence join operation for a table with 4 variables/clauses with table splitting at width 2 for Figure 4.10.

4.2.3 Preprocessing

Our preprocessing consists of two steps, first we remove unit clauses and then we combine small bags of the tree decomposition.

Remove Unit Clauses We iterate over the formula and remove each unit clause from the formula and the literals which are contained in the clauses. If the literal occurs in a clause we remove the clause and if the literal occurs negated in a clause then we remove the literal from the clause. The variables from the literals and clauses also need to be removed from the tree decomposition. There is a problem in the case of Weighted Model Counting, as the weights of the literals would be lost. Therefore we generate the product of the weights of the literals which are removed and then multiply the resulting weighted model count with the weights.

Example 4.4. We have the SAT formula $\{\{v_1, v_9\}, \{v_1, v_{10}, v_{12}\}, \{v_1, \neg v_6\}, \{v_2, v_9, \neg v_4\}, \{v_2, v_{11}\}, \{\neg v_2, \neg v_7\}, \{\neg v_3, \neg v_8\}, \{v_3, v_7\}, \{v_4, \neg v_9\}, \{v_4, \neg v_{10}\}, \{v_5, \neg v_9\}, \{\neg v_5, v_{12}\}, \{\neg v_6, \neg v_8\}, \{\neg v_6, v_{11}\}, \{\neg v_{10}, \neg v_{12}\}, \{v_1\}, \{v_2\}\}$. After preprocessing the formula looks as follows:

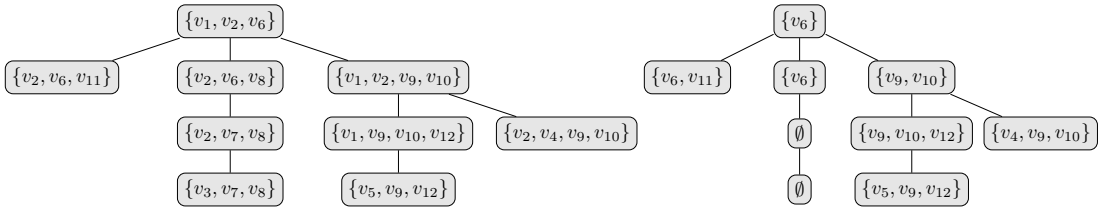


Figure 4.11: The tree decomposition of the formula from Example 4.4 on the left side before and on the right side after unit preprocessing.

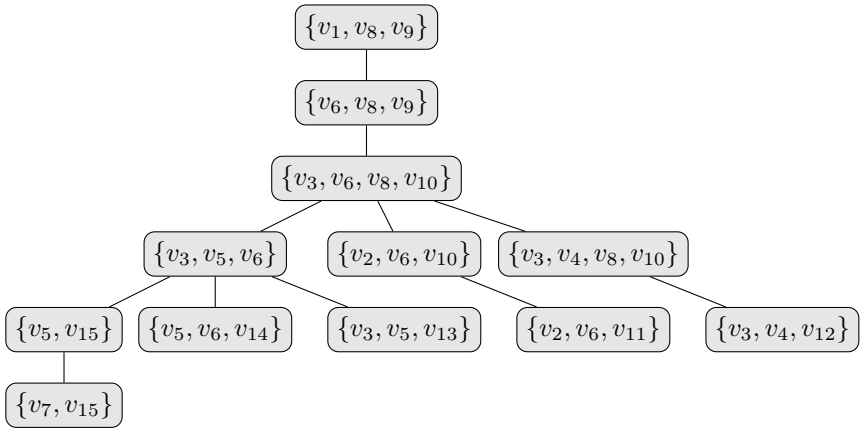


Figure 4.12: Tree decomposition before preprocessing.

$\{\{v_4, \neg v_9\}, \{v_4, \neg v_{10}\}, \{v_5, \neg v_9\}, \{\neg v_5, v_{12}\}, \{\neg v_6, v_{11}\}, \{\neg v_{10}, \neg v_{12}\}\}$. In Figure 4.11 we can see the tree decomposition before preprocessing on the left side and after preprocessing on the right side. The empty bags are then removed in the next preprocessing step.

Combine Small Bags To fully utilize the computational power of the GPU we need a certain amount of threads. The number of threads depends on the size of the current bag in the tree decomposition and if the bag is too small then there are not enough threads for the GPU. In cases where we have small bags we combine consecutive bags of the tree decomposition as long as we are below a certain threshold. The default value for the threshold is 10, but a different threshold can be set via the command line parameter -w,--combineWidth . In Figure 4.12 we can see a tree decomposition before preprocessing and in Figure 4.13 we can see the tree decomposition after preprocessing.

4.2.4 Increase Precision

OpenCL doesn't offer big number or arbitrary precision types. The type with the highest precision is a 64Bit floating point type (double). Therefore we adapted the quad double type from `https://github.com/scibuilder/QD` which links 4 double types together to

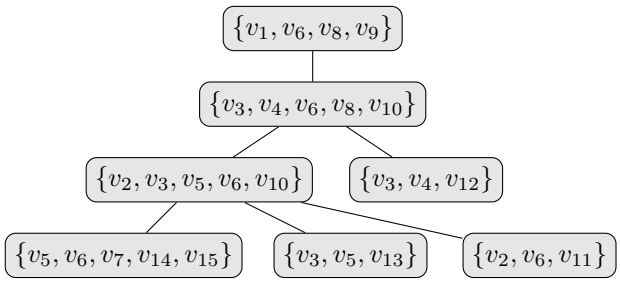


Figure 4.13: The tree decomposition from Figure 4.12 after preprocessing.

increase the precision and create a new type which has the precision of a 256 Bit floating point type. Internally the type is an array consisting of 4 double types, by summing up all 4 doubles we get the original number. By using the double 4 type we need 4 times as much memory compared to using the double type and operations on the double 4 type need considerably more time.

4.2.5 Increase Range

The exponent of the double type can range from 308 to -307. In the case of #SAT the exponent is always positive, therefore the exponent range from 0 to -307 will not be used. To utilize the full double range we use WMC instead of #SAT with a constant weight of 0.78 for every literal and divide the resulting model count by 0.78^{nVars} with nVars being the number of variables in the SAT formula.

4.3 Architecture

4.3.1 Input Format

We use the DIMACS [1] cnf format as input format for the SAT formulas. The DIMACS format consists of three different line types, the comment lines, the problem line and the clause lines. Comment lines start with a c. There has to be a problem line before the clause lines start. The problem line starts with *p cnf nvars nclauses*, *cnf* indicates that the file is in CNF format, *nvars* is the number of variables and *nclauses* is the number of clauses. After the problem line is a clause line for each clause from the formula. The clause lines contain a positive number if a variable occurs positive in the clause and a negative number if the corresponding variable occurs negative in the clause. The variable number has to be between 1 and nvars. Clause lines end with a 0.

The format for WMC extends the DIMACS format by adding weight lines. Weight lines are of the form *w var weight*, *var* is the number of the variable and *weight* is the weight of the variable if it occurs in a positive literal and 1-weight is the weight of the variable if it occurs in a negative literal. The weight has to be a real between 0 and 1.

[1] See: `http://www.satcompetition.org/2009/format-benchmarks2009.html`

Example 4.5. In Listing 4.1 we can see a possible cnf representation of the following SAT formula: $\{c_1 = \{v_1, \neg v_2, v_3, \neg v_6\},\ c_2 = \{\neg v_1, \neg v_5\},\ c_3 = \{v_2, v_3, \neg v_4, \neg v_6\},\ c_4 = \{\neg v_3, \neg v_4, v_6\},\ c_5 = \{\neg v_3, \neg v_5\},\ c_2 = \{\neg v_4, v_6\}\}$ with the weights $w(v_1) = 0.7$, $w(v_2) = 0.6$, $w(v_3) = 0.9$, $w(v_4) = 0.8$, $w(v_5) = 0.6$ and $w(v_6) = 0.1$.

```
c This is a comment line.
c start of the weights
w 1 0.7
w 2 0.6
w 3 0.9
w 4 0.8
w 5 0.6
w 6 0.1
c the problem line
p cnf 6 6
c the clause lines
1 -2 3 -6 0
-1 -5 0
2 3 -4 -6 0
-3 -4 6 0
-3 -5 0
-4 6 0
```

Listing 4.1: Example for a weighted CNF file.

Our input format for the tree decomposition is the DIMACS td format used in the PACE challenge [DKTW17] [2]. The td format consists of optional comment lines which can be at the beginning of a file. Comment lines start with a *c*. After the comment lines is a start line which consist of *s td nbags maxBagSize nvars*. *nbags* is the number of bags in the tree decomposition, *nvars* is the number of variables and *maxBagSize* is the maximal number of vertices in a bag. The start line is followed by a bag line for each bag in the tree decomposition. A bag line starts with *b bId vertices*, *bId* is the id of the bag and *vertices* is a list of vertices which are contained in the bag. The bag lines are followed by edge lines. An edge line contains the id of the first and second bag which are connected by the edge. In Listing 4.2 we can see the DIMACS td representation of the tree decomposition from Figure 4.14.

```
s td 9 4 12
b 1 6 7 9 10
b 2 2 7 9
b 3 4 6 9 10
b 4 3 7 9 10
b 5 4 6 12
b 6 3 7 8
b 7 1 7 8
b 8 3 5 8
```

[2] See: `https://pacechallenge.wordpress.com/`

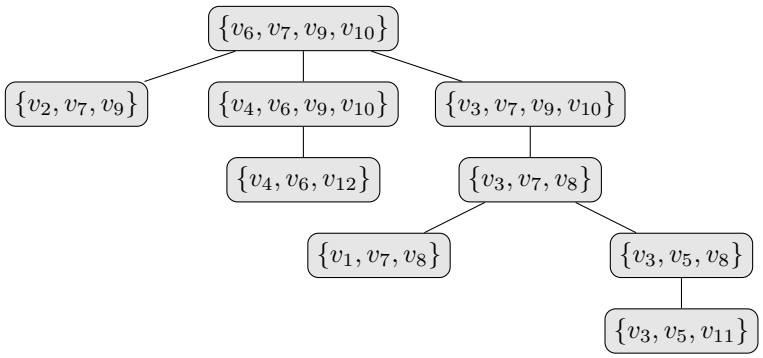


Figure 4.14: Example tree decomposition for Listing 4.2

```
10 b 9 3 5 11
11 1 2
12 1 3
13 1 4
14 3 5
15 4 6
16 6 7
17 6 8
18 8 9
```

Listing 4.2: Example of a td file.

Output Format: The output is in JSON [Cro06] format, the different output parameters are:

- **Model Count**: is the model count or the weighted model count
- **Time / Solving**: is the time needed for solving
- **Time / Parsing**: is the time needed to parse and preprocess the SAT formula and tree decomposition
- **Time / Build_Kernel**: is the time needed to build the kernel
- **Time / Init_OpenCL**: is the time needed for the first OpenCL operation
- **Time / Total**: is the total time needed for the execution
- **Statistics / Num Join**: is the number of executed join operations
- **Statistics / Num Introduce/Forget**: is the number of executed introduce forget operations

An example output can be seen in Listing 4.3.

```
{
  "Model Count": 7973665505280
  ,"Time":{
    "Solving": 8.761
    ,"Parsing": 0.003
    ,"Build_Kernel": 0.001
    ,"Generate_Model": 0.001
    ,"Init_OpenCL": 0.077
    ,"Total": 8.974
  }
  ,"Statistics":{
    "Num Join": 78
    ,"Num Forget": 174
  }
}
```

Listing 4.3: Example Output of gpusat.

Command Line Options The different command line arguments are:

- **-s,--formula <formulaPath>**: the path to the SAT formula
- **-f,--decomposition <decompositionPath>**: the path to the tree decomposition
- **-w,--combineWidth <width>**: the threshold for combining bags
- **-m,--maxBagSize <size>**: the maximal bag size before splitting
- **-c,--kernelDir <path to kernel Directory>**: the path to the directory which contains the kernel files
- **-g,--graph <0|1|2>**: sets the type of the input graph
- **--noFactRemoval**: deactivates the removal of unit clauses
- **--weighted**: activates the weighted model count
- **--CPU**: used to run OpenCL on the CPU instead of the GPU

4.3.2 CPU

The CPU code was written in C++11. It consists of the parser for CNF files, the parser for the tree decompositions, and a solver for each graph type. The class diagram can be seen in Figure 4.15, we have omitted the function parameters to increase the readability.

The CPU code works as follows:

1. First we use the CNFParser to parse the input formula.

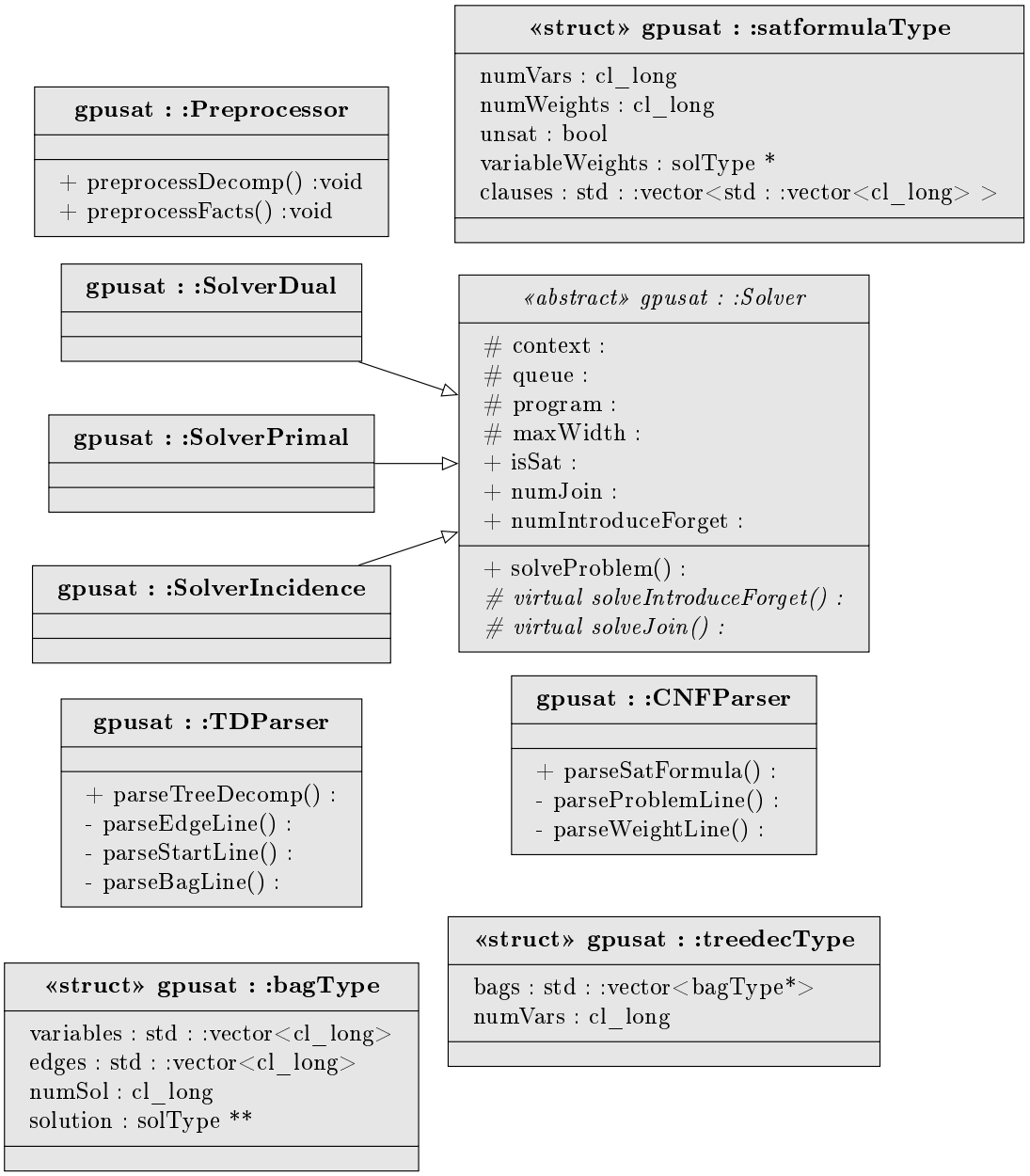


Figure 4.15: Class diagram of gpusat.

2. Then the TDParser parses the tree decomposition.
3. In the next step, the Preprocessor first removes the unit clauses and then preprocesses the tree decomposition.
4. Then we check the graph type of the tree decomposition.
5. If there is no kernel binary we compile the kernel, otherwise the kernel binary is loaded.
6. Then we start the Solver, depending on the tree decomposition we either use the Dual, Primal or Incidence solver. The Solver then executes the introduce forget or join operation for each bag of the tree decomposition.
7. In the end we calculate the final model count and print the solving statistics together with the model count.

The CNFParser is responsible for parsing the input CNF formula, it has three methods:

- The *parseSatFormula* method which parses the sat formula, it calls the methods *parseProblemLine*, *parseWeightLine* and *parseClauseLine* for the respective line in the CNF file.
- The *parseProblemLine* method which parses the problem line of the CNF file.
- The *parseWeightLine* method which parses a weight line of the CNF file.
- The *parseClauseLine* method which parses a clause line of the CNF file.

The TDParser parses the tree decomposition file, it has four methods:

- The *parseTreeDecomp* method which parses the tree decomposition, it calls the methods *parseStartLine*, *parseEdgeLine* and *parseBagLine* for the respective line in the td file.
- The *parseStartLine* method which parses the start line of the td file.
- The *parseEdgeLine* method which parses an edge line of the td file.
- The *parseBagLine* method which parses a bag line of the td file.

The Preprocessor class is responsible for preprocessing the tree decomposition and the SAT formula, it has two main methods:

- The *preprocessDecomp* method which does the preprocessing of the tree decomposition.
- The *preprocessFacts* method which does the unit clause removal.

The Solver class is used to iterate over the tree decomposition and calculate the tables for each bag. It has three methods:

- *solveProblem* is the main method of the Solver, it iterates over each bag of the tree decomposition and calls the *solveJoin* or *solveIntroduceForget* method based on the current bag.
- The *solveJoin* method has to be implemented by the respective solver, it solves a join node.
- The *solveIntroduceForget* method has to be implemented by the respective solver, it solves a introduce and forget node.

The SolverDual class implements the *solveJoin* and *solveIntroduceForget* method of the Solver class and calls the respective function of the dual kernel. **The SolverPrimal** class like the SolverDual class implementes the *solveJoin* and *solveIntroduceForget* methods, but calls the primal kernel. **The SolverIncidence** class is used to call the respective kernels for the incidence algorithm.

Init OpenCL Listing 4.4 contains an example of how we initialize OpenCL. First we get all OpenCL platforms. Then we search if there is a platform that contains either a CPU or a GPU device depending on the **--CPU** command line parameter. If we found a suitable device we create a command queue.

```
1  // get all OpenCL platforms
2  cl::Platform::get(&platforms);
3  std::vector<cl::Platform>::iterator iter;
4
5  // iterate over all platforms
6  for (iter = platforms.begin(); iter != platforms.end(); ++iter)
7  {
8    cl_context_properties cps[3] = {CL_CONTEXT_PLATFORM,
9      (cl_context_properties) (*iter)(), 0};
10
11   if (cpu)
12   {
13     // search for CPU devices
14     context = cl::Context(CL_DEVICE_TYPE_CPU, cps);
15   } else
16   {
17     // search for GPU devices
18     context = cl::Context(CL_DEVICE_TYPE_GPU, cps);
19   }
20
21   cl_int err;
22   devices = context.getInfo<CL_CONTEXT_DEVICES>(&err);
23
```

```
24    // we found a suitable device
25    if (err == CL_SUCCESS)
26    {
27      // create a command queue
28      queue = cl::CommandQueue(context, devices[0]);
29      break;
30    }
31  }
```

Listing 4.4: The initialization of OpenCL.

Compile kernel In Listing 4.5 we can see an example of how we precompile our OpenCL kernel. First we read the kernel source from the file. Then we create a cl::Program::Sources object which holds our source code. Afterwards we create a cl::Program object with the source. We then compile the program with the program.build() command. Afterwards we read the binary from the cl::Program object and combine it in a string. Then we write the string into a file which we can read later. Listing 4.6 shows how to load the precompiled kernel. First we read the binary. The *GPUSATUtils::readBinary* function is a small function that reads a file and returns a string containing the file contents. Then we create a binary object with the kernel data and an OpenCL program from the binary.

```
1  // read kernel source from file
2  std::string kernelStr = GPUSATUtils::readFile(sourcePath);
3  cl::Program::Sources sources(1, std::make_pair(kernelStr.c_str(),
4    kernelStr.length()));
5  program = cl::Program(context, sources);
6
7  // compile the kernel source
8  program.build(devices);
9
10 // get the size of the compiled sources
11 const std::vector<size_t> binSizes =
12   program.getInfo<CL_PROGRAM_BINARY_SIZES>();
13 std::vector<char> binData((unsigned long long int)
14   std::accumulate(binSizes.begin(), binSizes.end(), 0));
15 char *binChunk = &binData[0];
16
17 std::vector<char *> binaries;
18 for (const size_t &binSize : binSizes)
19 {
20   binaries.push_back(binChunk);
21   binChunk += binSize;
22 }
23
24 // save the kernel source into file for later usage
25 program.getInfo(CL_PROGRAM_BINARIES, &binaries[0]);
26 std::ofstream binaryfile(binPath.c_str(), std::ios::binary);
27 for (unsigned int i = 0; i < binaries.size(); ++i)
```

```
28    binaryfile.write(binaries[i], binSizes[i]);
29  binaryfile.close();
```

Listing 4.5: The compilation of the kernel.

```
1  cl_int err;
2  std::string kernelStr = GPUSATUtils::readBinary(binPath);
3  cl::Program::Binaries bins(1, std::make_pair((const void *)
       kernelStr.data(), kernelStr.size()));
4  program = cl::Program(context, devices, bins, nullptr, &err);
5  program.build(devices);
```

Listing 4.6: The loading of the kernel.

Start kernel In Listing 4.7 we can see a part of the primal introduce forget function. The function contains two loops, one for the split table parts of the current bag and one to iterate over the split table parts of the child bag. The parameters of the kernel are initialized with the kernel.setArg function, the first argument is the position of the parameter in the kernel source and the second parameter is either a buffer containing a region of the RAM or a value. The queue.enqueueNDRangeKernel then starts a kernel for each row in the result table. Afterwards we wait for the execution of the kernel. The queue.enqueueReadBuffer function reads the current table and a flag indicating if there are solutions from the GPU. If the table part contains no solutions we delete it afterwards.

```
1  // iterate over all parts of the current solution table
2  for (int a = 0; a < numIterations; a++)
3  {
4    cl_int solutions = 0;
5    node.solution[a] = new solType[bagSizeForget]();
6    cl_long startIdNode = a * bagSizeForget;
7
8    cl::Buffer buf_solsF(context, CL_MEM_READ_WRITE |
9      CL_MEM_COPY_HOST_PTR, sizeof(solType) * (bagSizeForget),
10     node.solution[a]);
11   kernel.setArg(0, buf_solsF);
12
13   // buffer for the solution table of the curren bag
14   cl::Buffer buf_varsF;
15   if (fVars.size() > 0)
16   {
17     buf_varsF = cl::Buffer(context, CL_MEM_READ_WRITE |
18       CL_MEM_COPY_HOST_PTR, sizeof(cl_long) * fVars.size(),
19       &fVars[0]);
20     kernel.setArg(1, buf_varsF);
21   } else {
22     kernel.setArg(1, NULL);
23   }
```

```
24
25    cl_int bagsolutions = 0;
26    cl::Buffer bufsolBag(context, CL_MEM_READ_WRITE |
27      CL_MEM_COPY_HOST_PTR, sizeof(cl_int), &bagsolutions);
28    kernel.setArg(11, bufsolBag);
29
30    // iterate over all child table parts
31    for (int b = 0; b < numSubIterations; b++)
32    {
33      // buffer for the solution table of the child bag
34      cl::Buffer buf_solsE(context, CL_MEM_READ_WRITE |
35        CL_MEM_COPY_HOST_PTR, sizeof(solType) * (bagSizeEdge),
36        cnode.solution[b]);
37
38      // start the kernel
39      queue.enqueueNDRangeKernel(kernel,
40        cl::NDRange(static_cast<size_t>(startIdNode)),
41        cl::NDRange(static_cast<size_t>(bagSizeForget)));
42
43      // wait for the kernel to finish
44      queue.finish();
45    }
46
47    // check if there are solutions
48    queue.enqueueReadBuffer(bufsolBag, CL_TRUE, 0, sizeof(cl_int),
49      &bagsolutions);
50    solutions += bagsolutions;
51
52    // read the solution table from the GPU
53    queue.enqueueReadBuffer(buf_solsF, CL_TRUE, 0, sizeof(solType) *
54      (bagSizeForget), node.solution[a]);
55
56    // delete table if it contains no solutions
57    if (solutions == 0)
58    {
59      delete[] node.solution[a];
60      node.solution[a] = nullptr;
61      numHpath -= bagSizeForget;
62    } else
63    {
64      this->isSat = 1;
65    }
66  }
```

Listing 4.7: Primal introduce forget function.

4.3.3 GPU

In total we have 12 kernels, for each algorithm there are two different kernels, one for the join and one for the introduce forget operation, each of these kernels has a double and double 4 implementation. The kernel code was written in OpenCL version 1.2.

Join In Listing 4.8 we can see the kernel for our join for the primal graph, the join for the dual graph works similar. The parameters of the join function are the solution table, the variables and the number of Variables of the current node and the first and second child node. The start and end id for the current table and for the table of the two edges. The last parameters are the weights of the literals for weighted model count and a flag which is set if there are solutions in the current bag.

The kernel first queries its id. Then it queries the corresponding model counts of the first and second edge with the solveIntroduce_ method. It then generates the product of the model count of the first and second child node and saves it in the table. The function then divides the model count by the weights of the literals in the bag if weighted model count is enabled.

```
1  long id = get_global_id(0);
2  double tmp, tmp_;
3  double weight = 1;
4
5  // get solution count from first edge
6  tmp = solveIntroduce_(numV, edge1, numVE1, variables,
       edgeVariables1, minId1, maxId1, startIDEdge1, weights, id);
7  // get solution count from second edge
8  tmp_ = solveIntroduce_(numV, edge2, numVE2, variables,
       edgeVariables2, minId2, maxId2, startIDEdge2, weights, id);
9
10 // weighted model count
11 if (weights != 0)
12 {
13     for (int a = 0; a < numV; a++)
14     {
15         weight *= weights[((id >> a) & 1) > 0 ?
16           variables[a] * 2 : variables[a] * 2 + 1];
17     }
18 }
19
20 // we have some nSol in edge1
21 if (tmp >= 0.0)
22 {
23     nSol[id - (startIDNode)] *= tmp;
24     nSol[id - (startIDNode)] /= weight;
25 }
26
27 // we have some solutions in edge2
```

```
28 if (tmp_ >= 0.0)
29 {
30     nSol[id - (startIDNode)] *= tmp_;
31 }
```

Listing 4.8: The kernel of the primal join.

In Listing 4.9 we can see the kernel for the join of the incidence graph. The parameters of the join function are the solution table of the current node and the child nodes. The start and end id for the current table and for the table of the two child nodes. The last parameters are the weights of the literals for weighted model count, a flag which is set to 1 if there are solutions in the current bag, the list of variables from the current bag and the number of clauses in the current bag.

The kernel first calculates the number of solutions we need to check from the child tables and then queries its id. Then it sums up the edge solutions if the union of the clause sets of the child nodes is equal to the clause set of the current node. If weighted model count is enabled it divides the model count by the weights of the literals in the bag and saves the count in the table.

```
1  // get the number of counts we have to check from the edges
2  unsigned long combinations = ((unsigned long) exp2((double)
       numClauses));
3  unsigned long start2 = 0, end2 = combinations - 1;
4
5  // get the id
6  unsigned long id = get_global_id(0);
7  unsigned long mask = id & (((unsigned long) exp2((double)
       numClauses)) - 1);
8  unsigned long templateID = id >> numClauses << numClauses;
9  double tmpSol = 0;
10
11 //sum up the solution count for all subsets of Clauses (A1,A2)
       where the intersection of A1 and A2 = A
12 for (int a = 0; a < combinations; a++)
13 {
14   if ((templateID | a) >= minIDe1 && (templateID | a) < maxIDe1 &&
         e1Sol[(templateID | a) - (startIDe1)] != 0)
15   {
16     for (int b = start2; b <= end2; b++)
17     {
18       if (((a | b)) == mask && ((templateID | b) >= minIDe2 && (
           templateID | b) < maxIDe2) && e2Sol[(templateID | b) - (
           startIDe2)] != 0)
19       {
20         tmpSol += e1Sol[(templateID | a) - (startIDe1)] * e2Sol[(
             templateID | b) - (startIDe2)];
21       }
22     }
```

```
23    }
24  }
25
26  if (tmpSol != 0.0)
27  {
28    // weighted model count is activated
29    if (weights != 0)
30    {
31      // get weight of current bag for WMC
32      double weight = 1;
33      unsigned long assignment = id >> numClauses;
34      for (int a = 0; nVars[a] != 0; a++)
35      {
36        weight *= weights[((assignment >> a) & 1) > 0 ? nVars[a] * 2
              : nVars[a] * 2 + 1];
37      }
38      nSol[id - (startIDn)] += tmpSol / weight;
39    } else
40    {
41      nSol[id - (startIDn)] += tmpSol;
42    }
43  }
```

Listing 4.9: The kernel of the incidence join.

Introduce Forget In Listing 4.10 we can see the kernel for our introduce forget operation of the incidence graph, the introduce forget kernel of the primal and dual graph are similar. The parameters of the introduce forget function are the solution table, the variables, the number of Variables, the clauses and the number of clauses for the current node, the child node and the introduce node. The start and end id for the current table and child table. The last parameters are the weights of the literals for weighted model count and a flag which is set if there are solutions in the current bag.

The kernel first queries its id. Then if there are clauses or variables forgotten it collects all the corresponding values from the child table which are obtained via an introduce. If there are no forget variables or clauses it only executes an introduce.

```
1   unsigned long id = get_global_id(0);
2   unsigned long templateId = 0;
3   unsigned long combinations = (unsigned long) exp2((double) numVI -
        numVF);
4
5   if (numVI != numVF || numCI != numCF)
6   {
7     //generate template for clauses
8     for (unsigned long a = 0, b = 0; a < numCI; a++)
9     {
10      if (fClauses[b] == iClauses[a])
```

```
11      {
12        templateId = templateId | (((id >> b) & 1) << a);
13        b++;
14      } else
15      {
16        templateId = templateId | (1 << a);
17      }
18    }
19
20    //generate template for variables
21    for (unsigned long a = 0, b = 0; a < numVI && b < numVF; a++)
22    {
23      if (varsF[b] == varsI[a])
24      {
25        templateId = templateId | (((id >> (b + numCF)) & 1) << (a +
              numCI));
26        b++;
27      }
28    }
29
30    // iterate through all corresponding assignments from the edge
31    for (unsigned long i = 0; i < combinations; i++)
32    {
33      long b = 0, otherId = templateId;
34      for (a = 0; a < numVI; a++)
35      {
36        if (b >= numVF || varsI[a] != varsF[b])
37        {
38          otherId = otherId | (((i >> (a - b)) & 1) << (a + numCI));
39        } else {
40          b++;
41        }
42      }
43
44      // get solution count from the edge
45      solsF[id - (startIDf)] += solveIntroduceF(solsE, clauses, cLen
            , varsI, varsE, numVI, numVE, iClauses, eClauses, numCI,
           numCE, startIDe, minIDE, maxIDE, weights, otherId);
46    }
47  } else
48  {
49    // solve only introduce if there is no forget
50    solsF[id - (startIDf)] += solveIntroduceF(solsE, clauses, cLen,
         varsI, varsE, numVI, numVE, iClauses, eClauses, numCI, numCE,
          startIDe, minIDE, maxIDE,
51    weights, id);
52  }
```

Listing 4.10: The kernel of the incidence algorithm.

CHAPTER 5

Experiments

In this section we describe our experiments, first starting with the setting. The setting contains the solvers which were used in the benchmarks, then we describe the hardware used to run our benchmarks and the sets which were used for benchmarking. Then we take a look at the results of our benchmarks, starting with the #SAT benchmarks, then the WMC benchmarks and a brief discussion of our results.

5.1 Setting

To compare the different solvers we used the wall clock time and set the timeout of each run to 900s. We also limited the available RAM to 8GB. For gpusat we used one randomly generated tree decomposition.

5.1.1 Solvers

For all solvers the default configuration was used except for countAntom which we set to use 12 cores on the CPU. The solvers we used for benchmarking are can be seen in Table 5.1.

5.1.2 Hardware

In this section we describe the Hardware which was used to run our benchmark set. We used two different machines, one for our GPU experiments and one for the CPU solvers. For our CPU experiments a recent hardware configuration was used and for gpusat we used a cheap consumer hardware. The configuration of our GPU machine can be seen in Table 5.2. For the CPU experiments we used a cluster consisting of 9 nodes, hyper threading was disabled. The configuration of each node can be seen in Table 5.3.

Solver	Description	#SAT	WMC	Reference
c2d	a solver based on knowledge compilation techniques	X		[Dar04]
d4	a solver based on knowledge compilation techniques	X		[LM17]
miniC2D	a solver based on knowledge compilation techniques	X	X	[LM17]
sts	an approximate solver	X	X	[EGS12]
sharpSAT	a solver based on CDCL	X		[Thu06]
Cachet	a solver based on CDCL	X	X	[SBB+04]
DSHARP	a solver based on knowledge compilation techniques	X		[MMBH12]
sdd	a solver based on knowledge compilation techniques	X		[Dar11]
dynQBF 1.1.1	a QBF solver based on dynamic programming; in our experiments we used a modified version for solving the #SAT problem	X		[CW16]
dynasp	a #ASP solver based on dynamic programming; in our experiments we used a modified version which is able to solve the #SAT problem	X		[FHMW17]
cnf2eadt	a solver based on knowledge compilation techniques	X		[KLMT13]
ApproxMC	an approximate solver based on knowledge compilation techniques	X		[CFM+14]
bdd_minisat_all	a solver based on knowledge compilation techniques	X		[TS16]
sharpCDCL	a solver based on CDCL	X		[KMM13]
Clasp	a standard SAT and ASP solver which is also able to solve #SAT and #ASP problem; clasp is based on CDCL	X		[GKS12]
countAntom	a parallel solver based on DPLL	X		[BSB15]

Table 5.1: Overview of the different solvers used in the benchmarks.

Operating System:	Ubuntu 16.04.3 LTS
Linux kernel version:	4.4.0-87
CPU:	Intel Core i3-3245 (dual core - 3.4GHz)
RAM:	16GB
GPU:	Sapire Pulse ITX Radeon RX 570 (32 compute units, 2048 shaders and 4GB VRAM)
GPU driver:	amdgpu-pro 17.10

Table 5.2: The configuration of our GPU machine.

Operating System:	Ubuntu 16.04.01 LTS
Linux kernel version:	4.4.0-101
CPU:	two Intel Xenon E5-2650 CPUs (12 cores per CPU) 2.2GHz
RAM:	256GB

Table 5.3: The configuration of our CPU machine.

Type	set	Origin	N	v Mdn	c Mdn
WMC	DQMR	Cachet	660	140	350
WMC	Grid	Cachet	420	1824.5	2786.5
WMC	Plan	Cachet	11	812	3222
#SAT	Mixed	c2d	14	1286.5	11370
#SAT	Basic	fre/meel	92	604	1680
#SAT	Proj.	fre/meel	308	59032	71456.5
#SAT	Weig.	fre/meel	1080	200	500

Table 5.4: Overview of the used benchmark sets with the number of instances, the median number of clauses (cMdn), and the median number of variables (v Mdn).

5.1.3 Benchmark Sets

Our benchmark set consists of 2585 instances from #SAT and WMC solving. Our #SAT is made up of 1465 instances, 1451 instances are from the collection of Daniel Fremont[1], and 14 instances are from C2D[2]. Our WMC set contains 1091 instances from cachet[3]

Treewidth For our benchmark sets we generated upper bounds on the tree width for the primal, dual and incidence graph. We used 900s as timeout for the decomposer. In Table 5.5 we can see the number of instances within different width ranges of the incidence graph. Table 5.6 shows the median and maximal time which was needed to

[1]See: `https://github.com/dfremont/counting-benchmarks`

[2]See: `http://reasoning.cs.ucla.edu/c2d/results.html`

[3]See: `http://www.cs.rochester.edu/users/faculty/kautz/Cachet/Model_Counting_Benchmarks/index.htm`

set	0-20	21-30	31-40	41-50	51-60	61-90	91-150	151-300	>300	To
DQMR	35	388	97	140	0	0	0	0	0	0
Grid	90	150	118	12	15	35	0	0	0	0
Plan	1	0	1	1	0	3	0	0	4	1
Mixed	1	1	1	3	2	1	0	0	5	0
Basic	37	9	5	6	11	16	3	0	4	1
Proj.	4	0	2	0	0	2	30	105	140	25
Weig.	125	538	215	152	15	35	0	0	0	0

Table 5.5: Overview of how many instances are in each width range of the incidence graph. The column "To" contains the number of instances, which took more than 900s to decompose.

set	t max	t Mdn	w Mdn	50%	80%	90%	95%
DQMR	0.2	0.0	27.5	28	41	42	44
Grid	1.9	0.2	29	29	38	55	70
Plan	17.6	0.4	74	82	442	445	na
Mixed	22.8	3.8	55	59	442	445	540
Basic	17.6	0.2	26	39	65	86	349
Proj.	833.9	53.7	297	363	2195	4973	na
Weig.	1.9	0.1	28	28	40	43	47

Table 5.6: Overview of the upper bound of the incidence tree width, t is the time needed to generate the decompositions, w Mdn is the median width, and the percentiles of the upper bounds on the tree width.

set	0-20	21-30	31-40	41-50	51-60	61-90	91-150	151-300	>300	To
DQMR	35	390	95	140	0	0	0	0	0	0
Grid	91	153	114	12	15	35	0	0	0	0
Plan	1	0	1	1	1	2	0	0	4	1
Mixed	1	1	2	2	1	2	0	0	5	0
Basic	37	9	5	6	12	15	3	0	4	1
Proj.	4	0	2	0	0	0	12	114	154	22
Weig.	126	543	209	152	15	35	0	0	0	0

Table 5.7: Overview of how many instances are in each width range of the primal graph. The column "To" contains the number of instances, which took more than 900s to decompose.

set	t max	t Mdn	w Mdn	50%	80%	90%	95%
DQMR	0.1	0.0	28	28	41	42	44
Grid	1.2	0.2	29	29	39	55	71
Plan	9.3	2.9	73	85	399	442	na
Mixed	15.8	3.9	57	63	399	442	540
Basic	9.3	0.9	26	37	64	88	352
Proj.	895.3	118.1	324	328	1084	1872	na
Weig.	1.2	0.1	28	28	40	43	48

Table 5.8: Overview of the upper bound of the primal tree width, t is the time in seconds needed to generate the decompositions, w Mdn is the median width, and the percentiles of the upper bounds on the tree width.

set	0-20	21-30	31-40	41-50	51-60	61-90	91-150	151-300	>300	To
DQMR	0	0	0	0	0	152	298	210	0	0
Grid	0	0	0	0	19	80	223	81	17	0
Plan	0	0	0	0	0	2	0	4	1	4
Mixed	0	0	1	0	1	1	2	1	2	6
Basic	1	2	5	1	3	18	11	16	31	4
Proj.	4	0	0	0	0	0	0	0	121	183
Weig.	0	0	0	0	19	232	521	291	17	0

Table 5.9: Overview of how many instances are in each width range of the dual graph. The column "To" contains the number of instances, which took more than 900s to decompose.

set	t max	t Mdn	w Mdn	50%	80%	90%	95%
DQMR	0.7	0.3	110	110	164	169	171
Grid	9.8	0.7	121	121	157	217	279
Plan	3.9	1.5	164	204	na	na	na
Mixed	4.1	0.4	112.5	3908	na	na	na
Basic	107.3	0.7	180.5	187	438	751	3416
Proj.	896.9	311.1	1573	na	na	na	na
Weig.	9.8	0.3	113	113	163	170	191

Table 5.10: Overview of the upper bound of the dual tree width, t is the time needed to generate the decompositions, w Mdn is the median width, and the percentiles of the upper bounds on the tree width.

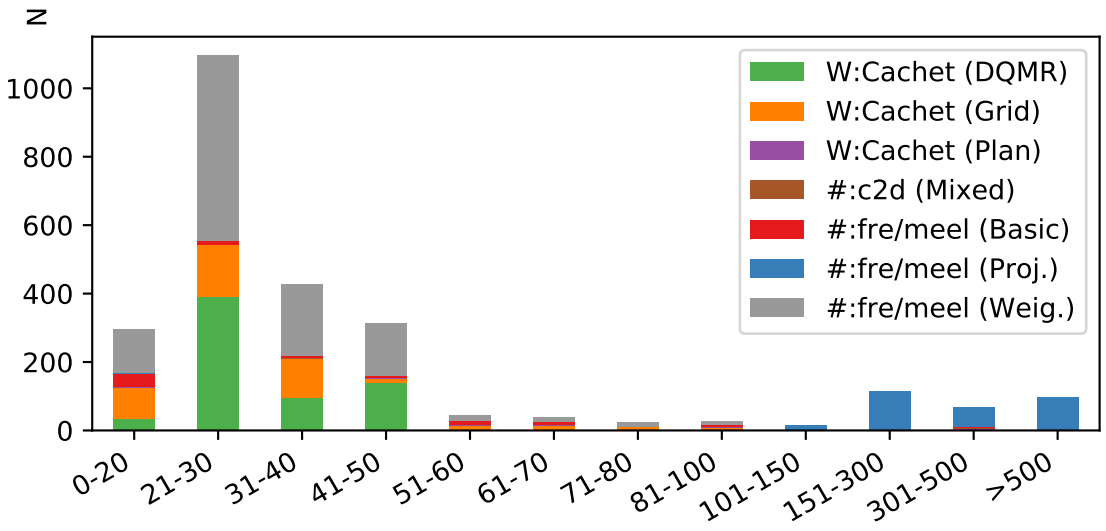


Figure 5.1: Distribution of instances for each interval of upper bounds for the primal width.

generate the decompositions of the incidence graph and the tree width for which 50%, 80%, 90% and 95% of the instances are below of. For #SAT there are 715 instances of 1494 that are in the width range from 0-30, which is 47.8% of the instances. For WMC there are 664 instances of 1091 which are in the width range from 0-30, which is 60.9% of the instances.

In Table 5.7 we can see the number of instances within different width ranges of the primal graph. Table 5.8 shows the median and maximal time which was needed to generate the decompositions of the primal graph and the tree width for which 50%, 80%, 90% and 95% of the instances are below of. For #SAT there are 721 instances of 1494 that are in the width range from 0-30, which is 48.3% of the instances. For WMC there are 670 instances of 1091 which are in the width range from 0-30, which is 61.4% of the instances. Surprisingly, there were more instances for which the primal graph was within our range, as the tree width of the incidence graph can be at most 1 more than the width of the primal and dual graph [KV00]. The reason therefore could be that we only use a heuristic and not an exact solver to generate the tree decompositions.

In Table 5.9 we can see the number of instances within different width ranges of the dual graph. Table 5.10 shows the median and maximal time which was needed to generate the decompositions of the dual graph and the tree width for which 50%, 80%, 90% and 95% of the instances are below of. For #SAT there are only 7 instances of 1494 that are in the width range from 0-30. For WMC there are 0 instances of 1091 which are in the width range from 0-30. Therefore, we did not include the dual algorithm in our benchmarks.

Our decomposer did not generate a decomposition for 27 instances of the incidence graph, 24 instances of the primal graph and 197 instances of the dual graph within 900s. Figure 5.1 shows the distribution of the number of instances on the y axis and the upper

solver	0-10	11-20	21-30	31-40	41-50	51-60	>60	u	0-30	r 0-30	ALL	r ALL
gpusat (p)	16	**153**	**523**	80	104	0	0	0	**692**	1	876	7
c2d	12	152	519	**175**	116	20	118	0	683	2	1112	3
d4	16	**153**	510	156	119	**23**	151	11	679	3	**1128**	1
countAntom	16	141	513	147	129	**23**	**159**	19	670	4	**1128**	1
gpusat (i)	16	152	490	79	97	0	0	0	658	5	834	8
miniC2D	16	151	491	137	103	8	67	0	658	6	973	4
gpusat (p4)	16	**153**	478	79	97	0	0	0	647	7	823	9
sts	16	146	448	101	**146**	10	45	0	610	8	912	6
sharpSAT	12	125	465	136	112	11	100	**23**	602	9	961	5
gpusat (i4)	16	152	427	77	89	0	0	0	595	10	761	11
cachet	16	117	421	91	109	8	56	2	554	11	818	10
dsharp	16	119	356	71	97	7	5	0	491	12	671	13
sdd	16	127	339	113	75	5	4	0	482	13	679	12
dynqbfe	16	119	333	81	67	0	5	0	468	14	621	14
dynqbfa	16	120	331	80	68	0	5	0	467	15	620	15
dynasp (i)	16	151	258	0	0	0	0	0	425	16	425	17
dynasp (p)	16	152	252	0	0	0	0	0	420	17	420	18
cnf2eadt	16	62	193	50	57	8	44	10	271	18	430	16
approxmc	16	37	174	42	43	1	35	10	227	19	348	19
bdd_minisat_all	9	30	95	26	28	1	15	0	134	20	204	20
sharpCDCL	9	13	83	20	24	1	33	10	105	21	183	21

Table 5.11: Number of instances solved in the primal width range #SAT, and the ranking. The column "u" contains instances for which the width is unknown.

bound of the primal width on the x axis for the upper bound of the primal and incidence tree width .

5.2 Results

In this section we describe the findings of our experiments. We start with the #SAT benchmarks, then we describe the WMC benchmarks. The width ranges in the cactus plots are from the incidence and primal graph combined. gpusat (p) and gpusat (i) is gpusat with the primal/incidence algorithm, gpusat (p4) and gpusat (i4) is gpusat width the advanced precision double4 type.

5.2.1 #SAT

In Table 5.11 we can see the number of instances solved for the different primal width ranges. Table 5.13 shows the solved instances for each incidence width range. The number of instances which were solved fastest by each solver can be seen in Table 5.12 for

solver	0-10	11-20	21-30	31-40	41-50	51-60	>60	u	0-30	r 0-30	ALL	r ALL
sharpSAT	**8**	**64**	**189**	45	45	4	42	**13**	**261**	1	**397**	1
countAntom	0	29	98	**59**	34	**17**	**91**	0	127	2	328	2
sts	0	6	113	32	**52**	2	12	0	119	3	217	3
gpusat (p)	0	18	39	1	0	0	0	0	57	4	58	7
dsharp	1	3	41	11	13	0	0	0	45	5	69	5
c2d	0	5	29	33	0	0	0	0	34	6	67	6
cnf2eadt	2	3	19	6	12	1	2	0	24	7	45	8
d4	0	6	9	10	0	2	49	0	15	8	76	4
dynasp (p)	0	13	0	0	0	0	0	0	13	9	13	9
cachet	1	2	4	1	1	0	0	0	7	10	9	10
gpusat (i)	0	0	5	3	0	0	0	0	5	11	8	11
approxmc	2	1	1	0	0	0	0	6	4	12	4	13
bdd_minisat_all	0	1	2	1	0	1	1	0	3	13	6	12
sharpCDCL	1	0	2	0	1	0	0	4	3	14	4	13
dynasp (i)	1	2	0	0	0	0	0	0	3	15	3	15
miniC2D	0	0	1	0	0	0	0	0	1	16	1	16
dynqbfa	0	0	0	0	0	0	0	0	0	17	0	17
dynqbfe	0	0	0	0	0	0	0	0	0	17	0	17
sdd	0	0	0	0	0	0	0	0	0	17	0	17

Table 5.12: Number of #SAT instances solved fastest in the primal width range, and the ranking. The column "u" contains instances for which the width is unknown.

the primal graph, and in Table 5.14 for the incidence graph. gpusat (p) solved the most instances for each width range up to width 30, but sharpSAT solved the most instances the fastest. gpusat (p) solved the sixth most instances when considering the whole benchmark set. With the double 4 type we were not able do solve as many instances as with the normal double type. The primal algorithm solved 45 instances less and the incidence algorithm solved 63 instances less.

Precision In Table 5.15 we can see the precision for the solvers which did not calculate the exact model count, the values are the error in relation to the correct value. Cachet prints the exact number of models, but for some instances the count was slightly off. Dsharp only prints the first 7 digits of the model count. sts and approxmc are both approximate solvers.

5.2.1.1 Variation of Decompositions for #SAT

We tested gpusat on different decompositions, to get an understanding of how different decompositions affect the solving algorithm. In Figure 5.2 and Figure 5.3 we can see a cactus plot containing the best, worst, average and median runtime for each instance

solver	0-10	11-20	21-30	31-40	41-50	51-60	>60	u	0-30	r 0-30	ALL	r ALL
sharpSAT	**7**	**65**	**188**	46	45	2	45	**12**	**261**	1	**397**	1
c2d	0	5	31	31	0	0	0	0	34	6	67	6
d4	0	6	8	10	1	2	49	0	15	8	76	4
cachet	1	2	4	1	1	0	0	0	7	10	9	10
dsharp	1	3	41	11	13	0	0	0	45	5	69	5
approxmc	2	1	1	0	0	0	0	6	4	12	4	13
gpusat (p)	0	18	39	1	0	0	0	0	57	4	58	7
gpusat (i)	0	0	4	4	0	0	0	0	5	11	8	11
sharpCDCL	1	0	2	0	1	0	0	4	3	13	4	13
dynqbfa	0	0	0	0	0	0	0	0	0	17	0	17
dynqbfe	0	0	0	0	0	0	0	0	0	17	0	17
dynasp (i)	1	2	0	0	0	0	0	0	3	13	3	15
dynasp (p)	0	13	0	0	0	0	0	0	13	9	13	9
miniC2D	0	0	1	0	0	0	0	0	1	16	1	16
cnf2eadt	2	3	17	9	11	1	2	0	24	7	45	8
bdd_minisat_all	0	1	2	0	1	2	0	0	3	13	6	12
sdd	0	0	0	0	0	0	0	0	0	17	0	17
sts	0	6	111	33	**52**	3	11	0	118	3	216	3
countAntom	0	28	96	**63**	33	**17**	**91**	0	127	2	328	2

Table 5.13: Number of #SAT instances solved in the incidence width range, and the ranking. The column "u" contains instances for which the width is unknown.

over 5 runs with different decompositions for the primal and incidence graph. The ranking of the runs can be seen in Table 5.16.

5.2.1.2 Width 0 to 30

In Figure 5.4 we can see a cactus plot comparing the different #SAT solvers for the width range 0 to 30. In this width range gpusat (p) solved the most instances with a total of 692, followed by c2d with 686 instances. On the third place is d4 with 681 solved instances. CountAntom is on the fourth place with 670 instances. On the fifth place is gpusat with the incidence algorithm which solved 658 instances.

5.2.1.3 Width 11 to 20

In Figure 5.5 we can see a cactus plot comparing the different #SAT solvers for the width range 11 to 20. In this width range gpusat (p), gpusat (p4) and d4 solved the most instances with a total of 154. gpusat (i), gpusat (i4), miniC2D and c2d are on the fourth place with 153 solved instances.

solver	0-10	11-20	21-30	31-40	41-50	51-60	>60	u	0-30	r 0-30	ALL	r ALL
sharpSAT	7	65	188	46	45	2	45	12	260	1	397	1
c2d	0	5	31	31	0	0	0	0	36	6	67	6
d4	0	6	8	10	1	2	49	0	14	8	76	4
cachet	1	2	4	1	1	0	0	0	7	10	9	10
dsharp	1	3	41	11	13	0	0	0	45	5	69	5
approxmc	2	1	1	0	0	0	0	6	4	11	4	13
gpusat primal	0	18	39	1	0	0	0	0	57	4	58	7
gpusat incidence	0	0	4	4	0	0	0	0	4	11	8	11
sharpCDCL	1	0	2	0	1	0	0	4	3	13	4	13
dynqbfa	0	0	0	0	0	0	0	0	0	17	0	17
dynqbfe	0	0	0	0	0	0	0	0	0	17	0	17
dynasp incidence	1	2	0	0	0	0	0	0	3	13	3	15
dynasp primal	0	13	0	0	0	0	0	0	13	9	13	9
miniC2D	0	0	1	0	0	0	0	0	1	16	1	16
cnf2eadt	2	3	17	9	11	1	2	0	22	7	45	8
bdd_minisat_all	0	1	2	0	1	2	0	0	3	13	6	12
sdd	0	0	0	0	0	0	0	0	0	17	0	17
sts	0	6	111	33	52	3	11	0	117	3	216	3
countAntom	0	28	96	63	33	17	91	0	124	2	328	2

Table 5.14: Number of #SAT instances solved fastest in the incidence width range, and the ranking. The column "u" contains instances for which the width is unknown.

solver	mean	median	min	max
cachet	$1 \cdot 10^{-13}$	0	0	$8.11 \cdot 10^{-11}$
dsharp	$7.98 \cdot 10^{-07}$	$5.07 \cdot 10^{-07}$	0	$4.36 \cdot 10^{-06}$
approxmc	$2.06 \cdot 10^{-01}$	$4.69 \cdot 10^{-02}$	0	$1 \cdot 10^{0}$
sts	$1.50 \cdot 10^{-01}$	$2.65 \cdot 10^{-02}$	0	$1.02 \cdot 10^{0}$
gpusat (p)	$4.08 \cdot 10^{-13}$	$1.45 \cdot 10^{-15}$	0	$1.38 \cdot 10^{-10}$
gpusat (i)	$3.81 \cdot 10^{-13}$	$4.36 \cdot 10^{-15}$	0	$1.38 \cdot 10^{-10}$
gpusat (p4)	$2.19 \cdot 10^{-32}$	0	0	$4.05 \cdot 10^{-31}$
gpusat (i4)	$2.10 \cdot 10^{-32}$	0	0	$5.26 \cdot 10^{-31}$

Table 5.15: The mean, median, minimal and maximal relative error for each #SAT solver.

rank	solver	#
1	gpusat (p) best	722
2	gpusat (p) avg	722
3	gpusat (p) median	718
4	gpusat (i) best	691
5	gpusat (i) avg	691
6	c2d	686
7	d4	681
8	gpusat (p) worst	674
9	countAntom	670
10	gpusat (i) median	667
11	gpusat (i) worst	639

Table 5.16: The ranking for the solver in the variations overview for #SAT with the number of solved instances.

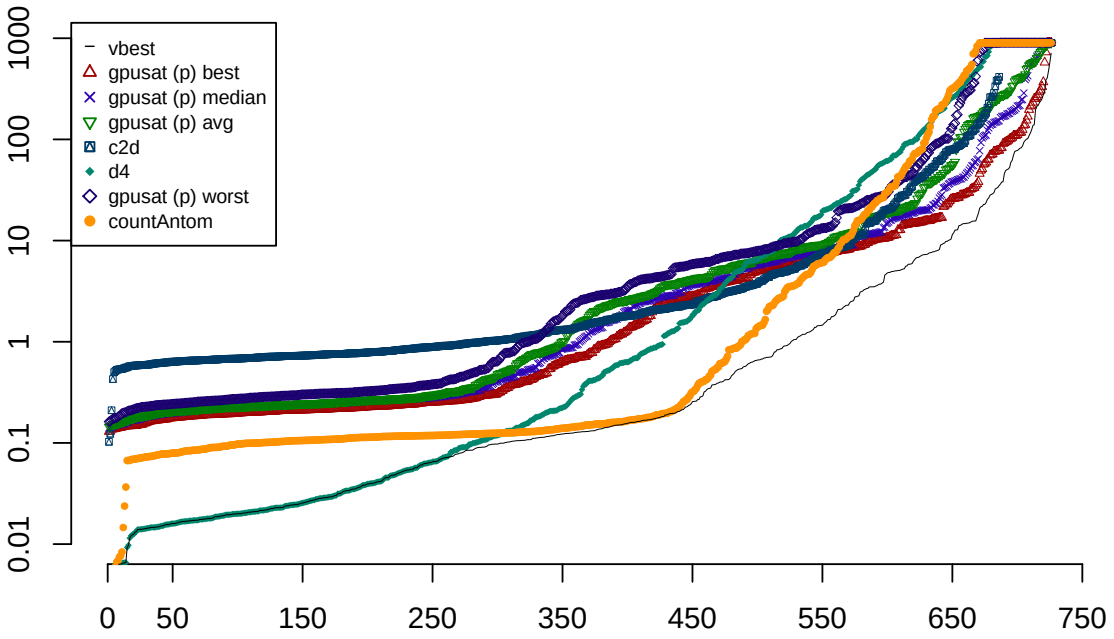


Figure 5.2: Cactus plot containing the min, max, and median of the solving time of the primal algorithm for the #SAT problem.

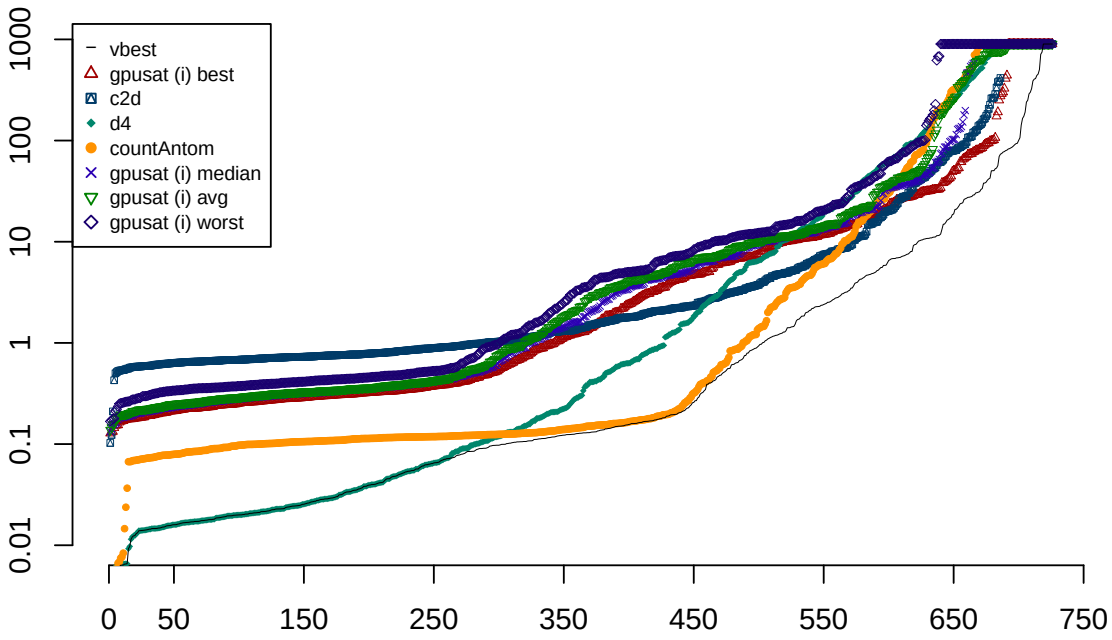


Figure 5.3: Cactus plot containing the min, max, and median of the solving time of the incidence algorithm for the #SAT problem.

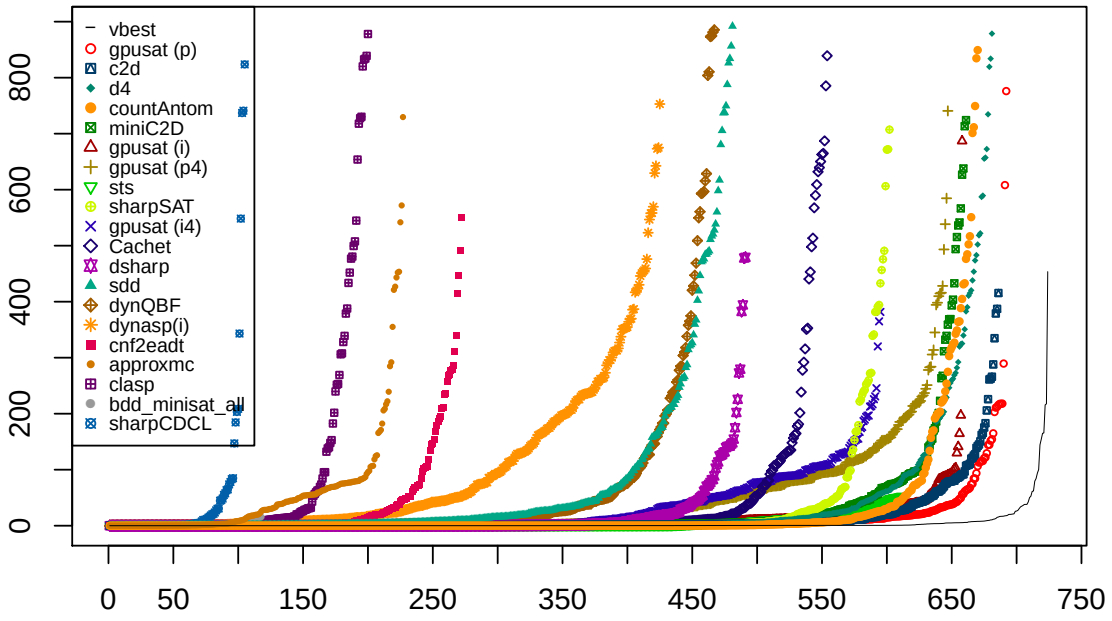


Figure 5.4: Cactus plot for the runtime of the #SAT solvers in the width range 0-30.

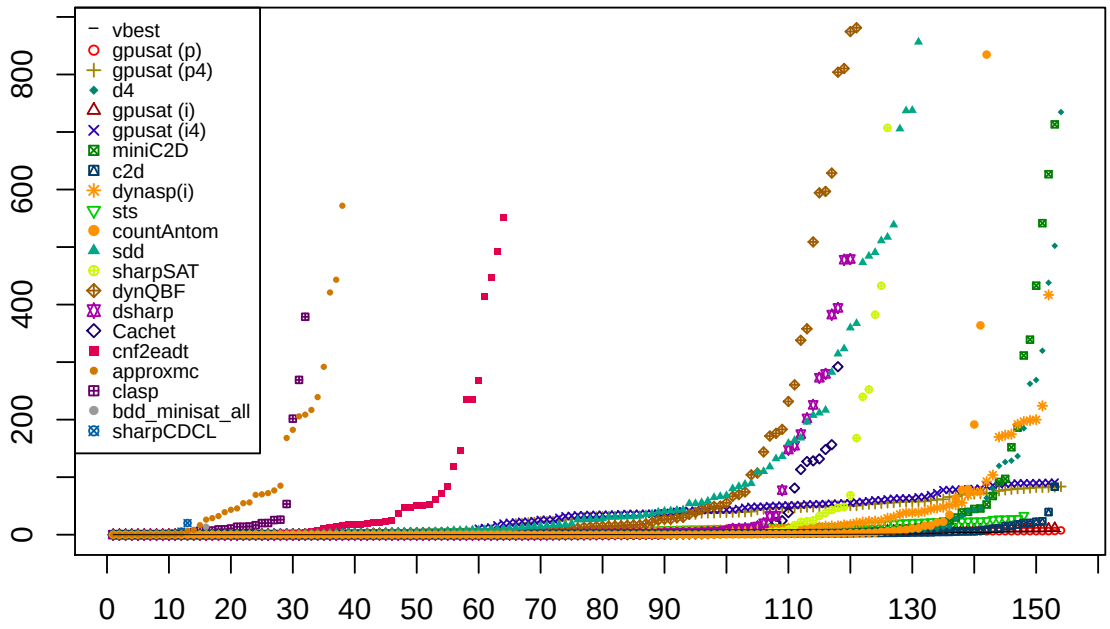


Figure 5.5: Cactus plot for the runtime of the #SAT solvers in the width range 11-20.

5.2.1.4 Width 21 to 30

In Figure 5.6 we can see a cactus plot comparing the different #SAT solvers for the width range 21 to 30. In this width range gpusat (p) solved the most instances with a total of 524, closely followed by c2d with 523 instances. On the third place is countAntom with 514 solved instances. d4 is on the fourth place with 513 instances. On the fifth place is gpusat (i) which solved 491 instances.

5.2.1.5 Width 31 to 45

In Figure 5.7 we can see a cactus plot comparing the different #SAT solvers for the width range 31 to 45. In this width range c2d solved the most instances with a total of 284, followed by countAntom with 271 instances. On the third place is d4 with 268 solved instances. sharpSAT is on the fourth place with 248 instances. On the fifth place is sts which solved 244 instances. gpusat (p) managed to solve 188 instances and gpusat (i) solved 180 instances.

5.2.1.6 Width Total

In Figure 5.8 we can see a cactus plot comparing the different #SAT solvers for all benchmark instances. Overall countAntom solved the most instances with a total of 1147, followed by d4 with 1139 instances. On the third place is c2d with 1112 solved instances. sharpSAT is on the fourth place with 984 instances. On the fifth place is

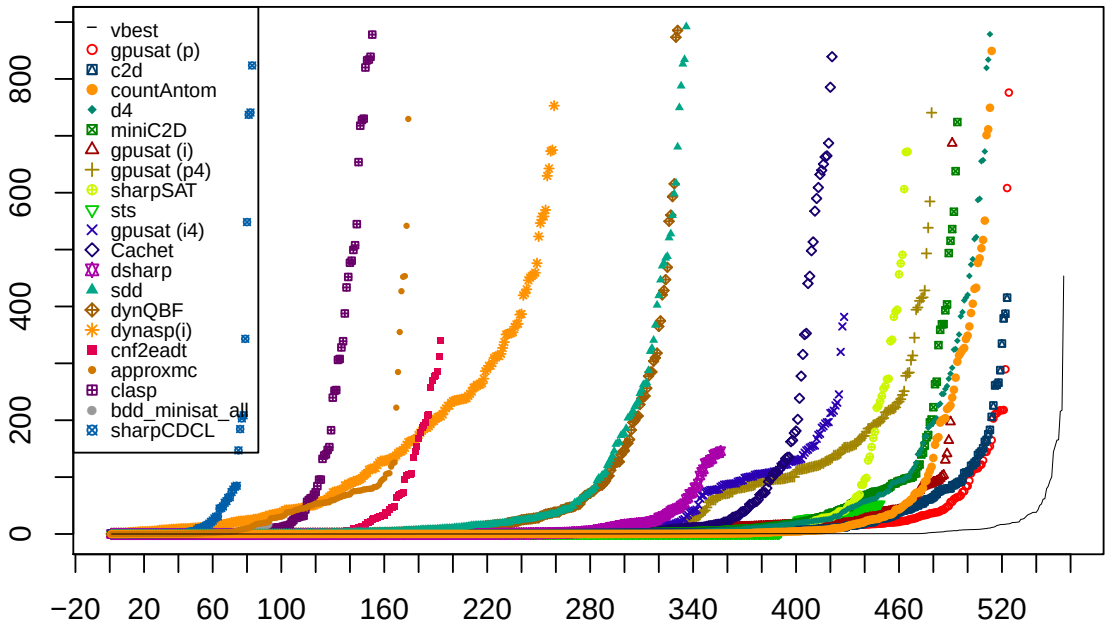


Figure 5.6: Cactus plot for the runtime of the #SAT solvers in the width range 21-30.

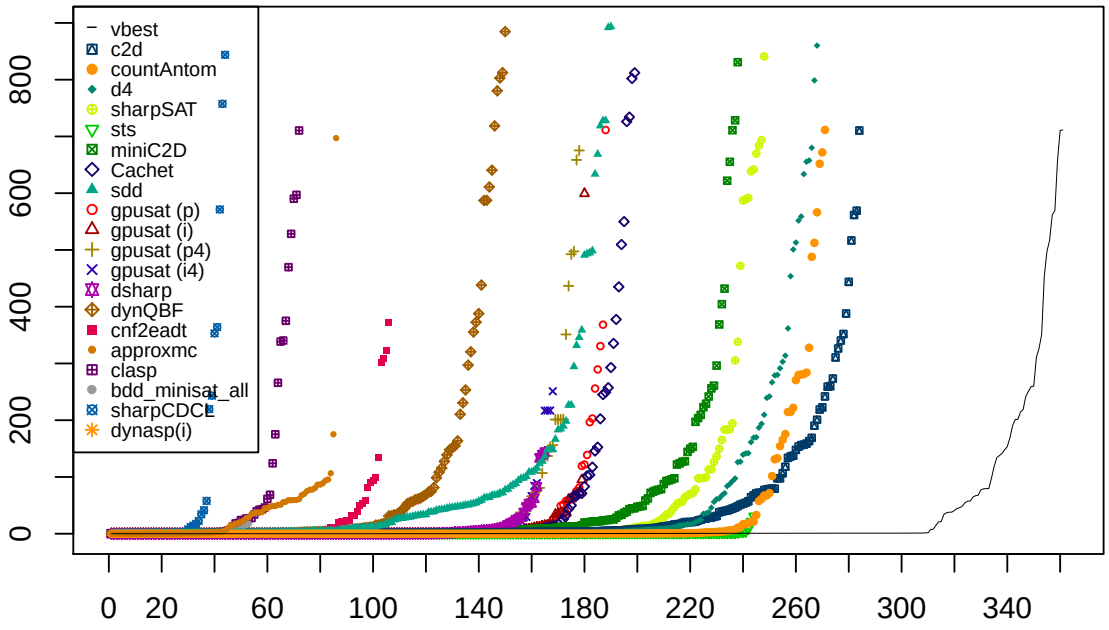


Figure 5.7: Cactus plot for the runtime of the #SAT solvers in the width range 31-45.

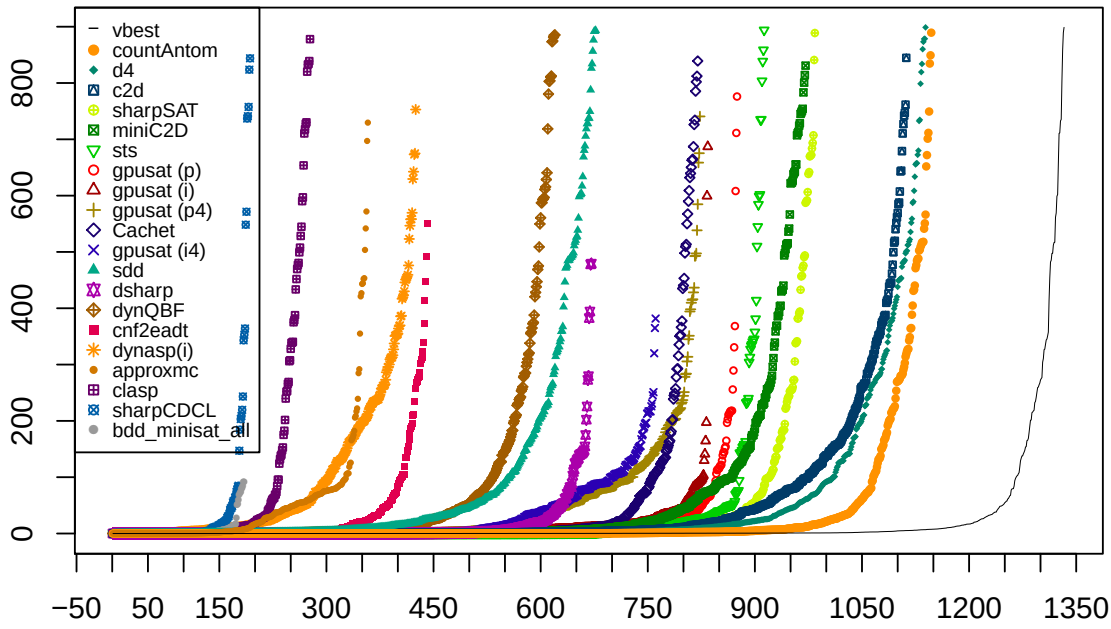


Figure 5.8: Cactus plot for the runtime of the #SAT solvers in the whole width range.

miniC2D which solved 971 instances. gpusat (p) is on the seventh place and managed to solve 875 instances. gpusat (i) is on the eighth place with 834 solved instances.

5.2.2 WMC

Table 5.17 shows the number of instances solved for the different primal width ranges. In Table 5.19 we can see the solved instances within each incidence width range. The number of instances which each solver managed to solve first can be seen in Table 5.18 for the primal, and in Table 5.20 for the incidence width ranges. The most instances in the range from 0-20 were solved by gpusat. sts solved 7 instances more than gpusat in the width range from 21-31, but gpusat produced a small error on average while sts produced a error of 1037 on average.

Precision The only exact solver we could compare with for precision was Cachet, but Cachet only prints 7 digits in the case of WMC. In Table 5.21 we can see the precision, the values are the error in relation to the Cachet value. The error for miniC2d seems quite high, but miniC2D only prints 3 digits after the decimal point.

5.2.2.1 Variation of Decompositions for WMC

We also tested gpusat on different decompositions, for WMC solving. In Figure 5.9 and Figure 5.10 we can see a cactus plot containing the best, worst, average, and median runtime for each instance over 5 runs with different decompositions for the primal and incidence graph. The ranking of the runs can be seen in Table 5.22.

solver	0-10	11-20	21-30	31-40	41-50	51-60	>60	u	0-30	r 0-30	ALL	r ALL
cachet-wmc	0	92	448	108	105	2	**9**	0	540	6	764	6
sts	0	121	**533**	**200**	**152**	1	6	0	**654**	1	**1013**	1
miniC2D	0	126	513	143	110	**5**	6	0	639	2	903	2
gpusat (p)	0	**128**	526	88	104	0	0	0	**654**	1	846	3
gpusat (i)	0	127	487	83	101	0	0	0	614	3	798	4
gpusat (p4)	0	**128**	476	79	96	0	0	0	604	4	779	5
gpusat (i4)	0	127	432	75	90	0	0	0	559	5	724	7

Table 5.17: Number of WMC instances solved in the primal width range, and the ranking. The column "u" contains instances for which the width is unknown.

solver	0-10	11-20	21-30	31-40	41-50	51-60	>60	u	0-30	r 0-30	ALL	r ALL
cachet-wmc	0	**62**	**264**	**71**	**69**	2	**8**	0	**328**	1	**479**	1
miniC2D	0	29	55	46	9	**3**	1	0	84	3	143	3
gpusat (p)	0	37	202	24	33	0	0	0	237	2	293	2
gpusat (i)	0	0	20	13	9	0	0	0	20	4	42	4

Table 5.18: Number of WMC instances fastest solved in the primal width range, and the ranking. The column "u" contains instances for which the width is unknown.

solver	0-10	11-20	21-30	31-40	41-50	51-60	>60	u	0-30	r 0-30	ALL	r ALL
cachet-wmc	0	92	442	114	105	1	**10**	0	540	6	764	6
sts	0	120	**530**	**204**	**152**	0	7	0	**654**	1	**1013**	1
miniC2D	0	125	508	150	109	**3**	8	0	639	2	903	2
gpusat (p)	0	**127**	523	92	104	0	0	0	**654**	1	846	3
gpusat (i)	0	126	483	87	102	0	0	0	614	3	798	4
gpusat (p4)	0	**127**	475	81	96	0	0	0	604	4	779	5
gpusat (i4)	0	126	431	77	90	0	0	0	559	5	724	7

Table 5.19: Number of WMC instances solved in the incidence width range, and the ranking. The column "u" contains instances for which the width is unknown.

solver	0-10	11-20	21-30	31-40	41-50	51-60	>60	u	0-30	r 0-30	ALL	r ALL
cachet-wmc	0	**62**	**262**	**76**	**69**	1	**9**	0	**328**	1	**479**	1
miniC2D	0	29	53	49	8	**2**	2	0	84	3	143	3
gpusat (p)	0	36	203	21	33	0	0	0	237	2	293	2
gpusat (i)	0	0	18	15	9	0	0	0	20	4	42	4

Table 5.20: Number of WMC instances fastest solved in the incidence width range, and the ranking. The column "u" contains instances for which the width is unknown.

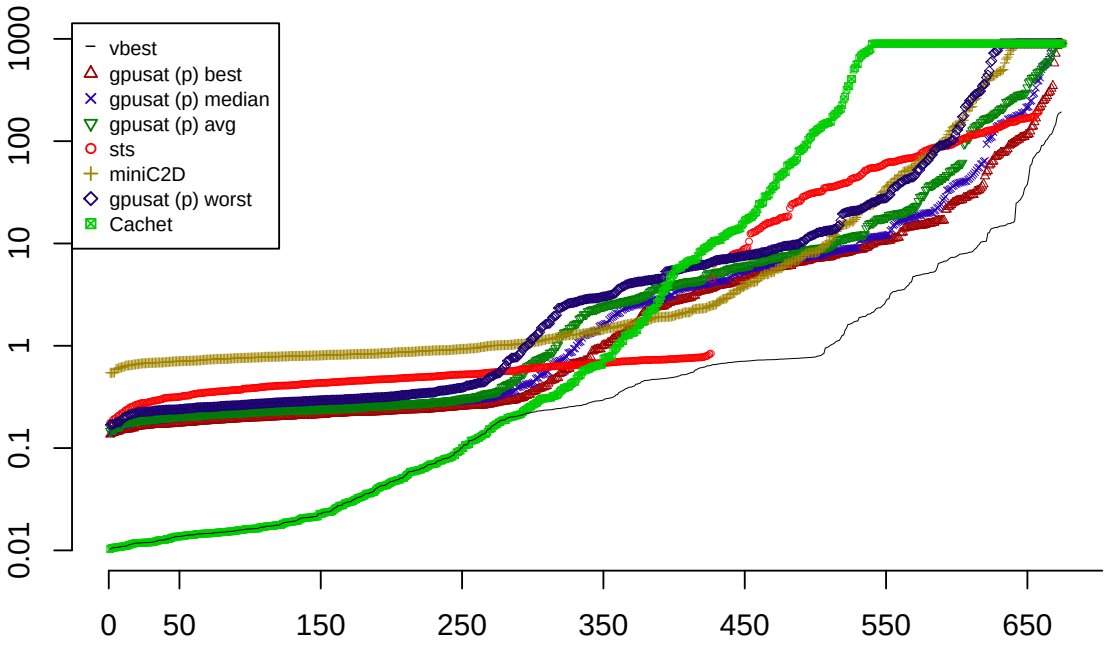


Figure 5.9: Cactus plot containing the min, max, and median of the solving time of the primal algorithm for the WMC problem.

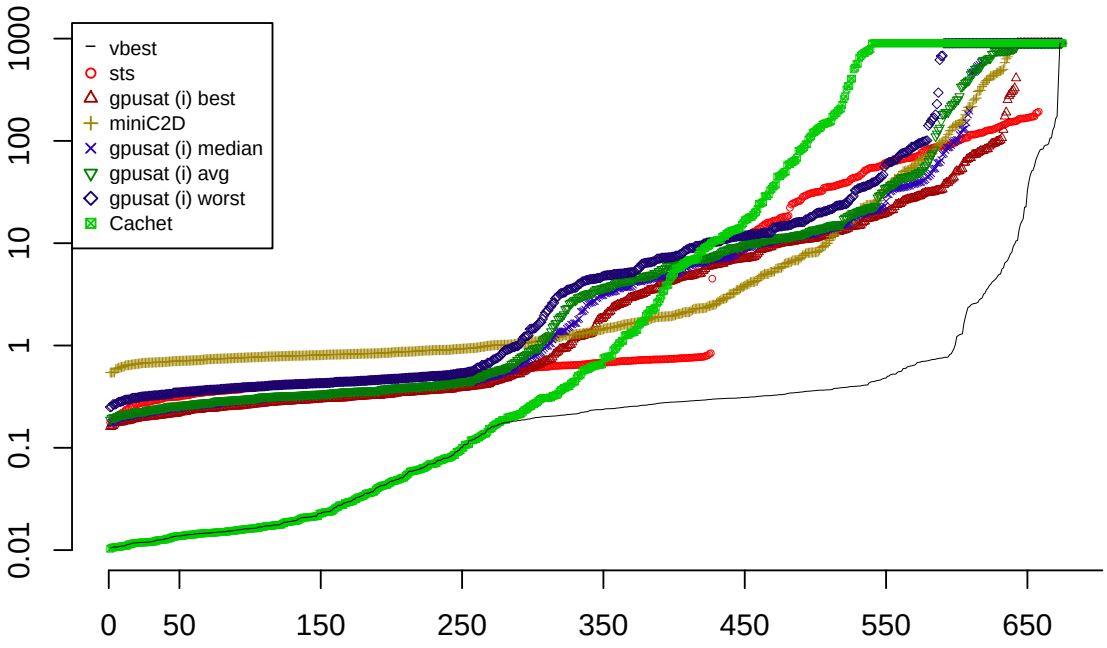


Figure 5.10: Cactus plot containing the min, max, and median of the solving time of the incidence algorithm for the WMC problem.

solver	mean	median	min	max
miniC2D	$4.4 \cdot 10^{-1}$	$7 \cdot 10^{-2}$	0	$1 \cdot 10^{00}$
sts	$1.03 \cdot 10^{-3}$	$4.11 \cdot 10^{00}$	0	$6.06 \cdot 10^{05}$
gpusat (p)	$7.83 \cdot 10^{-7}$	$4.89 \cdot 10^{-7}$	0	$4.39 \cdot 10^{-6}$
gpusat (i)	$7.94 \cdot 10^{-7}$	$5.01 \cdot 10^{-7}$	0	$4.39 \cdot 10^{-6}$
gpusat (p4)	$8.11 \cdot 10^{-7}$	$5.23 \cdot 10^{-7}$	0	$4.39 \cdot 10^{-6}$
gpusat (i4)	$8.20 \cdot 10^{-7}$	$5.29 \cdot 10^{-7}$	0	$4.39 \cdot 10^{-6}$

Table 5.21: The mean, median, minimal and maximal relative error for each WMC solver.

rank	solver	#
1	gpusat (p) best	670
2	gpusat (p) avg	670
3	gpusat (p) median	667
4	sts	658
5	gpusat (i) best	642
6	gpusat (i) avg	642
7	miniC2D	640
8	gpusat (p) worst	630
9	gpusat (i) median	618
10	gpusat (i) worst	590
11	cachet	540

Table 5.22: The ranking for the solver in the variations overview for WMC with the number of solved instances.

5.2.2.2 Width 0 to 30

In Figure 5.11 we can see a cactus plot comparing the different WMC solvers for the width range 0 to 30. In this width range sts solved the most instances with a total of 658, closely followed by gpusat (p) with 657 instances. On the third place is miniC2D with 640 solved instances. gpusat (i) is on the fourth place with 615 instances. On the fifth place is gpusat p4 which solved 604 instances. On the sixth place is gpusat i4 with 559 solved instances and on the last place is Cachet with 540 instances.

5.2.2.3 Width 11 to 20

In Figure 5.12 we can see a cactus plot comparing the different WMC solvers for the width range 11 to 20. In this width range gpusat (p), gpusat p4 and gpusat (i) solved the most instances with 128, followed by gpusat i4 with 127 instances. On the fifth place is miniC2D with 126 solved instances. On the sixth place is sts with 121 solved instances and on the last place is Cachet with 92 instances.

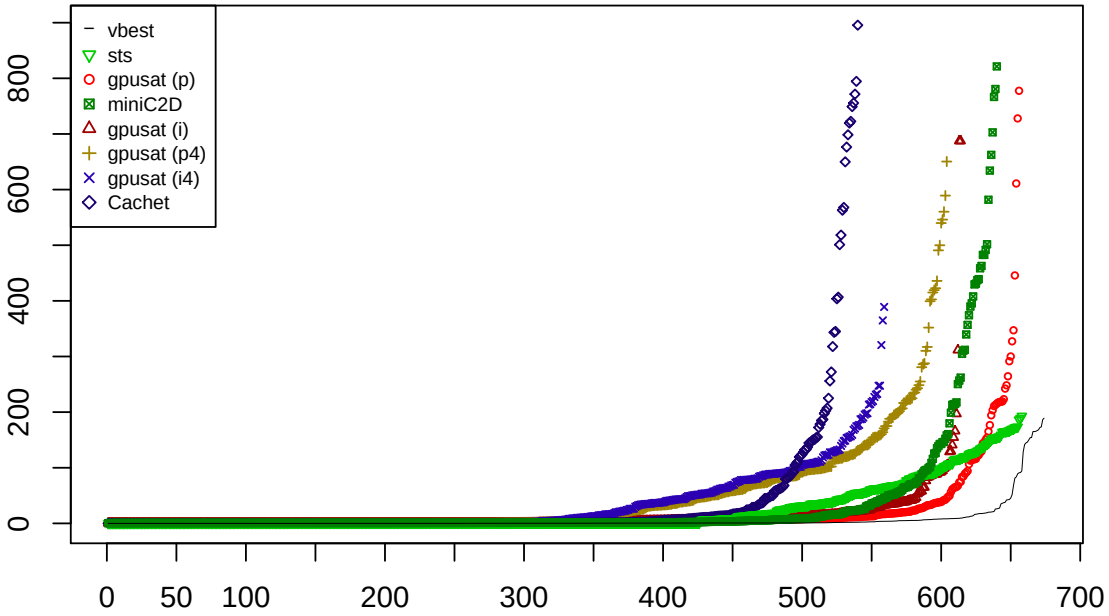


Figure 5.11: Cactus plot for the runtime of the WMC solvers in the width range 0-30.

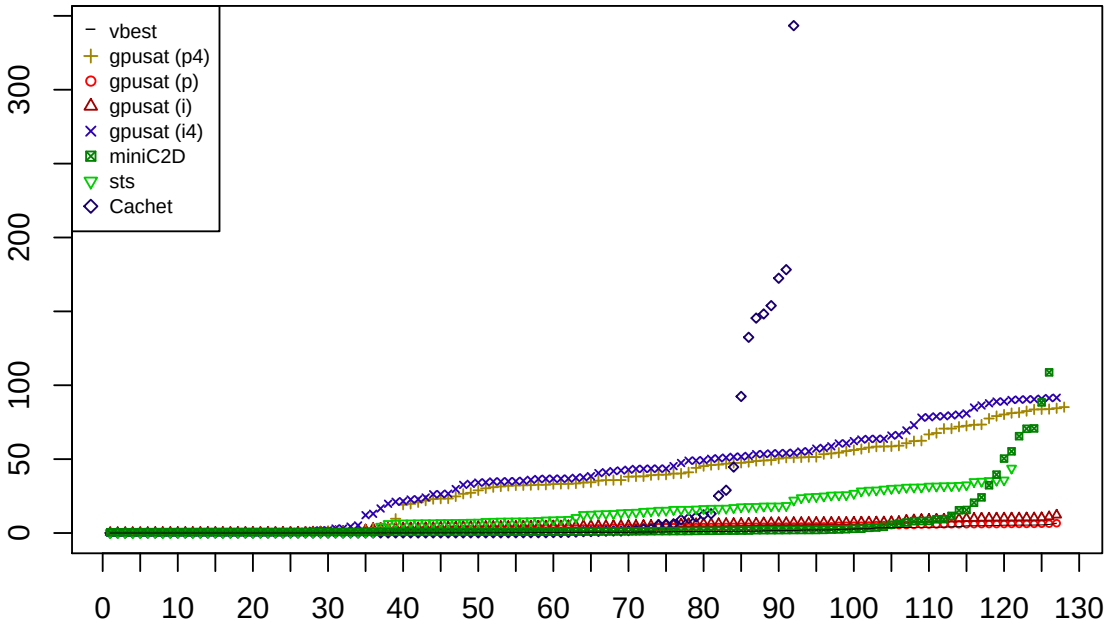


Figure 5.12: Cactus plot for the runtime of the WMC solvers in the width range 11-20.

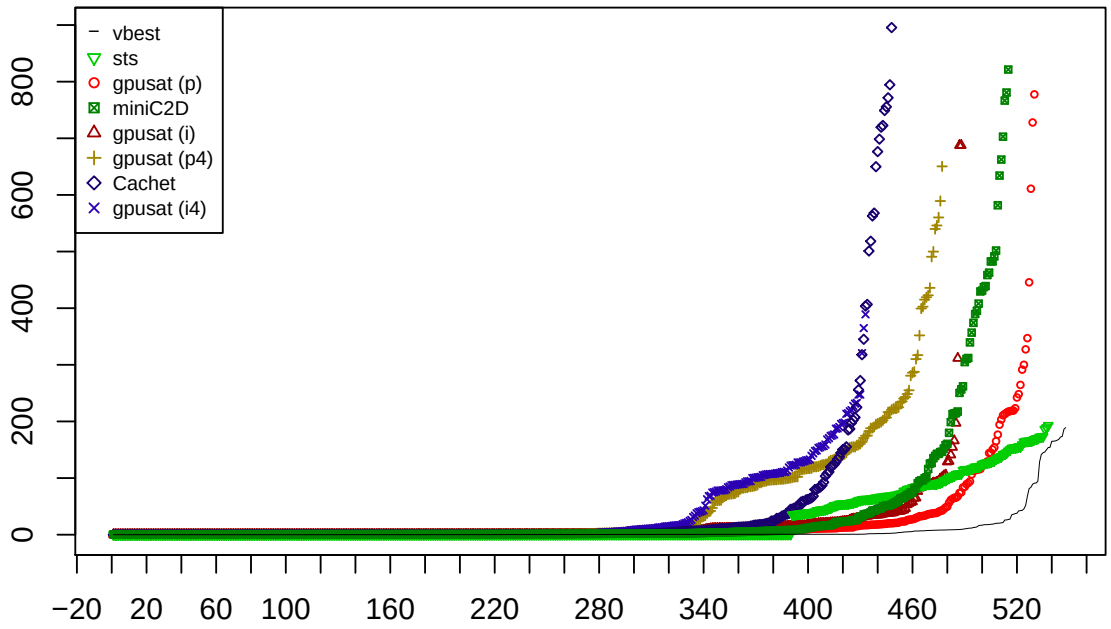


Figure 5.13: Cactus plot for the runtime of the WMC solvers in the width range 21-30.

5.2.2.4 Width 21 to 30

In Figure 5.13 we can see a cactus plot comparing the different WMC solvers for the width range 21 to 30. In this width range sts solved the most instances with a total of 538, followed by gpusat (p) with 530 instances. On the third place is miniC2D with 515 solved instances. gpusat (i) is on the fourth place with 488 instances. On the fifth place is gpusat i4 which solved 433 instances. On the sixth place is Cachet with 448 solved instances and on the last place is gpusat i4 with 433 instances.

5.2.2.5 Width 31 to 45

In Figure 5.14 we can see a cactus plot comparing the different WMC solvers for the width range 31 to 45. In this width range sts solved the most instances with a total of 350, followed by miniC2D with 254 instances. On the third place is Cachet with 218 solved instances. gpusat (p) is on the fourth place with 199 instances. On the fifth place is gpusat with the incidence algorithm which solved 186 instances. On the sixth place is gpusat p4 with 177 solved instances and on the last place is gpusat i4 with 167 instances.

5.2.2.6 Width Total

In Figure 5.15 we can see a cactus plot comparing the different WMC solvers for all benchmark instances. Overall sts solved the most instances with a total of 1013, followed by miniC2D with 903 instances. On the third place is gpusat (p) with 846 solved instances. gpusat (i) is on the fourth place with 799 instances. On the fifth place is gpusat p4 which

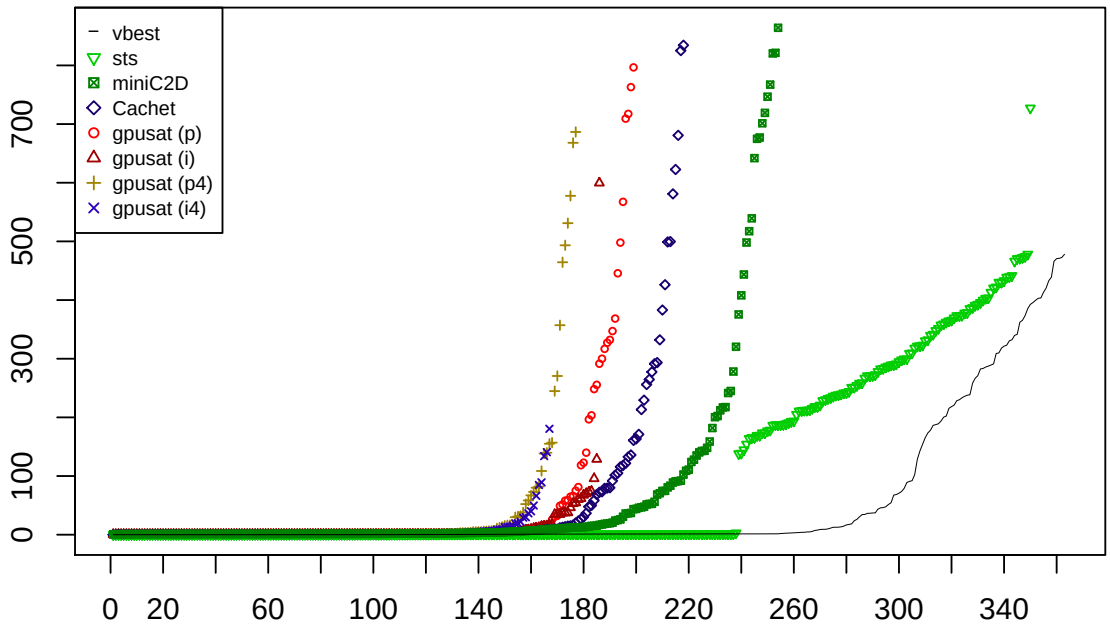


Figure 5.14: Cactus plot for the runtime of the WMC solvers in the width range 31-45.

solved 779 instances. On the sixth place is Cachet with 764 solved instances and on the last place is gpusat i4 with 724 instances.

5.3 Discussion

Our tree width overview shows that about half of our instances have a width below 30 and about two third of our instances have width below 40. At the moment, we were able to compete with other solvers up to a width of 30. As there are still some possible future improvements for our solver to save memory and improve performance, a width of 40 should be in our possible reach. For our hardware we needed to split tables at a width of 24 when we use the double4 type and at a width of 26 when we use the normal double type. In the width interval from 0 to 30 we were not able to solve 22 of 670 instances in the case of WMC and 23 of 721 #SAT instances. gpusat solved more instances with the primal algorithm than with the incidence algorithm, which could be due to the simpler solving algorithm used for the primal graph, and the widths of both graph representations are close together.

The usage of WMC for #SAT instances with an unified weight gave almost no precision loss, but enabled us to solve instances with higher model counts. The higher precision type was slower than the double type, but it solved the same number of instances for small width. The higher precision types did not pay off as the gain was rather small and the memory usage was increased by 4 times. gpusat was not able to solve as many instances as sts for WMC, but the weighted model count of sts was in some cases far

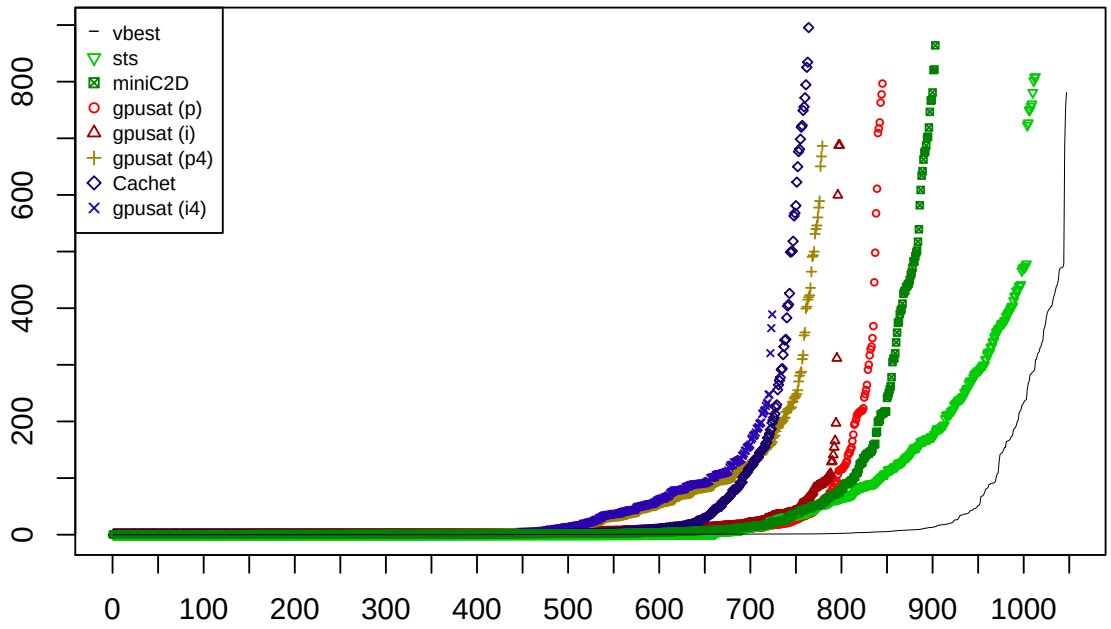


Figure 5.15: Cactus plot for the runtime of the WMC solver in the whole width range.

away from the correct model count, in the worst case the failure was about 606421 times larger than the correct result.

CHAPTER 6

Conclusion

6.1 Summary

In this thesis we took a closer look on how to parallelize dynamic programming algorithms on the GPU. For this purpose we implemented gpusat, which is able to solve the #SAT and WMC problem on the GPU with OpenCL. gpusat is based on dynamic programming on tree decompositions. OpenCL enables us to easily parallelize tasks on the GPU, CPU, and other devices.

We pointed out challenges, one is faced when implementing algorithms on GPUs and techniques we used to overcome these challenges and optimize the performance of our algorithms.

We collected benchmark sets, for the #SAT and WMC problems and found that almost half of the problem instances had a tree width of 30 or below and about two third of the instances had a width of 40 or below for the incidence and primal graphs. For these instances we were able to generate tree decompositions in under a second with htd.

To compare gpusat with other approaches we did extensive benchmarking. Our benchmark experiments showed, that gpusat is competitive with other #SAT solvers for a tree width of up to 30. We were even able to solve some instances for a width of up to 45.

6.2 Future Work

There are still some open points for further improvement for gpusat.

The main point for future improvement will be to reduce the memory consumption of gpusat with a map type. At the moment we save the positive and negative solutions

in an array. With a map type we would only need to save the positive solutions and therefore reducing the memory consumption and the number of table splits.

Another point for improvement will be a better preprocessing of the SAT formulas. In our experiments we have seen, that we can reduce the width with a better preprocessing.

List of Figures

List of Tables

Listings

List of Algorithms

Bibliography

[ACP87] 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.

[AMW17] Michael Abseher, Nysret Musliu, and Stefan Woltran. htd - A free, open-source framework for (customized) tree decompositions and beyond. In *Proceedings of the 14th International Conference on Integration of AI and OR Techniques in Constraint Programming, CPAIOR 2017, Padua, Italy, June 5-8, 2017*, pages 376–386, 2017.

[BB06] Emgad H. Bachoore and Hans L. Bodlaender. A branch and bound algorithm for exact, upper, and lower bounds on treewidth. In *Proceedings of the Second International Conference on Algorithmic Aspects in Information and Management, AAIM 2006, Hong Kong, China, June 20-22, 2006, Proceedings*, pages 255–266, 2006.

[BK10] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.

[BK11] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations II. lower bounds. *Inf. Comput.*, 209(7):1103–1119, 2011.

[Bod05] Hans L. Bodlaender. Discovering treewidth. In *Proceedings of the 31st Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2005, Liptovský Ján, Slovakia, January 22-28, 2005, Proceedings*, pages 1–16, 2005.

[BSB15] Jan Burchard, Tobias Schubert, and Bernd Becker. Laissez-faire caching for parallel #SAT solving. In *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing - SAT 2015, Austin, TX, USA, September 24-27, 2015*, pages 46–61, 2015.

[BSS+12] Sander Beckers, Gorik De Samblanx, Floris De Smedt, Toon Goedemé, Lars Struyf, and Joost Vennekens. Parallel hybrid sat solving using opencl. In *Proceedings of the 24th Benelux Conference on Artificial Intelligence, BNAIC 2012*, pages 11–18. Maastricht University, 2012.

[CdBD15] Arthur Choi, Guy Van den Broeck, and Adnan Darwiche. Tractable learning for structured probability spaces: A case study in learning preference distributions. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 2861–2868, 2015.

[CFM+14] 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 *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 1722–1730, 2014.

[Cos13] Carlos F. Costa. Parallelization of sat algorithms on gpus. Technical report, Technical report, INESC-ID, Technical University of Lisbon, 2013.

[Cro06] Douglas Crockford. The application/json media type for javascript object notation (JSON). *RFC*, 4627:1–10, 2006.

[CW16] Günther Charwat and Stefan Woltran. Dynamic programming-based QBF solving. In *Proceedings of the 4th International Workshop on Quantified Boolean Formulas (QBF 2016) co-located with 19th International Conference on Theory and Applications of Satisfiability Testing (SAT 2016), Bordeaux, France, July 4, 2016.*, pages 27–40, 2016.

[Dar04] Adnan Darwiche. New advances in compiling CNF into decomposable negation normal form. In *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 328–332, 2004.

[Dar11] Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 819–826, 2011.

[DJK08] Hervé Deleau, Christophe Jaillet, and Michaël Krajecki. Gpu4sat: solving the sat problem on gpu. In *Proceedings of the 9th International Workshop on State–of–the–Art in Scientific and Parallel Computing, PARA 2008 , Trondheim, Norway*, 2008.

[DKTW17] Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The PACE 2017 parameterized algorithms and computational experiments challenge: The second iteration. In *Proceedings of the 12th International Symposium on Parameterized and Exact Computation, IPEC 2017, September 6-8, 2017, Vienna, Austria*, pages 30:1–30:12, 2017.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DMPV17] Leonardo Dueñas-Osorio, Kuldeep S. Meel, Roger Paredes, and Moshe Y. Vardi. Counting-based reliability estimation for power-transmission grids. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 4488–4494, 2017.

[EGS12] Stefano Ermon, Carla P. Gomes, and Bart Selman. Uniform solution sampling using a constraint solver as an oracle. In *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence, Catalina Island, CA, USA, August 14-18, 2012*, pages 255–264, 2012.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing, , SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003*, pages 502–518, 2003.

[FF12] Hironori Fujii and Noriyuki Fujimoto. Gpu acceleration of bcp procedure for sat algorithms. In Hamid R. Arabnia, Hiroshi Ishii, Minoru Ito Kazuki Joe, and Hiroaki Nishikawa, editors, *Proceedings of the 2012 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'12)*, pages 10–16. CSREA Press, 2012.

[FHMW17] Johannes Klaus Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. Answer set solving with bounded treewidth revisited. In *Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2017, Espoo, Finland, July 3-6, 2017*, pages 132–145, 2017.

[GD04] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th Conference in Uncertainty in Artificial Intelligence, UAI '04, Banff, Canada, July 7-11, 2004*, pages 201–208, 2004.

[GKS12] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012.

[GSS09] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In *Handbook of Satisfiability*, pages 633–654. 2009.

[HMS15] Thomas Hammerl, Nysret Musliu, and Werner Schafhauser. Metaheuristic algorithms and tree decomposition. In *Springer Handbook of Computational Intelligence*, pages 1255–1270. Springer, 2015.

[JYP+17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan,

Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 1–12, 2017.

[KLMT13] Frédéric Koriche, Jean-Marie Lagniez, Pierre Marquis, and Samuel Thomas. Knowledge compilation for model counting: Affine decision trees. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence, IJCAI 2013, Beijing, China, August 3-9, 2013*, pages 947–953, 2013.

[Klo94] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994.

[KMM13] Vladimir Klebanov, Norbert Manthey, and Christian J. Muise. Sat-based analysis and quantification of information flow in programs. In *Proceedings of the 10th International Conference on Quantitative Evaluation of Systems, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013.*, pages 177–192, 2013.

[KV00] Phokion G. Kolaitis and Moshe Y. Vardi. Conjunctive-query containment and constraint satisfaction. *J. Comput. Syst. Sci.*, 61(2):302–332, 2000.

[LM17] Jean-Marie Lagniez and Pierre Marquis. An improved decision-DNNF compiler. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 667–673, 2017.

[Man16] Norbert Manthey. Towards next generation sequential and parallel SAT solvers. *KI*, 30(3-4):339–342, 2016.

[MMBH12] Christian J. Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. Dsharp: Fast d-dnnf compilation with sharpsat. In *Proceedings of the 25th Canadian Conference on Advances in Artificial Intelligence, Canadian AI 2012, Toronto, ON, Canada, May 28-30, 2012.*, pages 356–361, 2012.

[Mun11] Aaftab Munshi. The opencl specification version: 1.2 document revision: 15, 2011.

[PDFP15] Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Cud@sat: SAT solving on gpus. *J. Exp. Theor. Artif. Intell.*, 27(3):293–316, 2015.

[Rot96] Dan Roth. On the hardness of approximate reasoning. *Artif. Intell.*, 82(1-2):273–302, 1996.

[RS84] Neil Robertson and Paul D. Seymour. Graph minors. III. planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

[SBB+04] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing, SAT 2004, 10-13 May 2004, Vancouver, BC, Canada*, 2004.

[SS10] Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *J. Discrete Algorithms*, 8(1):50–64, 2010.

[Thu06] Marc Thurley. sharpsat - counting models with advanced component caching and implicit BCP. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing - SAT 2006, , Seattle, WA, USA, August 12-15, 2006*, pages 424–429, 2006.

[TS16] Takahisa Toda and Takehide Soh. Implementing efficient all solutions SAT solvers. *ACM Journal of Experimental Algorithmics*, 21(1):1.12:1–1.12:44, 2016.