Parallel Weighted Model Counting with Tensor Networks

Jeffrey M. Dudek and Moshe Y. Vardi

Rice University
{jmd11, vardi}@rice.edu

Abstract

A promising new algebraic approach to weighted model counting makes use of tensor networks, following a reduction from weighted model counting to tensor-network contraction. Prior work has focused on analyzing the single-core performance of this approach, and demonstrated that it is an effective addition to the current portfolio of weighted-model-counting algorithms.

In this work, we explore the impact of multi-core and GPU use on tensor-network contraction for weighted model counting. To leverage multiple cores, we implement a parallel portfolio of tree-decomposition solvers to find an order to contract tensors. To leverage a GPU, we use `TensorFlow` to perform the contractions. We compare the resulting weighted model counter on 1914 standard weighted model counting benchmarks and show that it significantly improves the virtual best solver.

1 Introduction

In *weighted model counting*, the task is to count the total weight, subject to a given weight function, of the set of solutions of input constraints. This fundamental task has applications in probabilistic reasoning, planning, inexact computing, engineering reliability, and statistical physics [5, 18, 28]. The development of model counters that can successfully compute the total weight on large industrial formulas is an area of active research [50, 65]. Although most model counters focus on single-core performance, there have been several parallel model counters, notably the multi-core (unweighted) model counter `countAntom` [11] and the GPU-based weighted model counter `gpuSAT2` [25, 26].

The parallelization of neural network training and inference has seen massive research across the machine learning and high-performance computing communities [1, 34, 51]. Consequently, GPUs give orders of magnitude of speedup over a single core for neural-network algorithms [36, 47]. In this work, we aim to directly leverage advances in multi-core and GPU performance designed for neural-network algorithms in the service of weighted model counting.

Tensor networks provide a natural bridge between high-performance computing and weighted model counting. Tensor networks are a tool used across computer science for reasoning about big-data processing, quantum systems, and more [9, 15, 49]. A tensor network describes a complex tensor as a computation on many simpler tensors, and the problem of tensor-network *contraction* is to perform this computation. Contraction is a key operation in neural network training and inference [7, 32, 37, 67], and, as such, many of the optimizations for neural networks also apply to tensor-network contraction [36, 47, 54].

Moreover, recent work [10, 19] has shown that tensor-network contraction can be used to perform weighted model counting using a 3-stage algorithm. First, in the *reduction* stage, a counting instance is reduced to a tensor-network problem. Second, in the *planning* stage, an order to contract tensors in the network is determined. Finally, in the *execution* stage, tensors in the network are contracted according to the best discovered order. The resulting weighted model counter was shown in [19] to be useful as part of a portfolio of existing model counters

when evaluated on a single core. In this work, we explore the impact of multiple-core and GPU use on tensor network contraction for weighted model counting.

The planning stage in [19] was done using a choice of several single-core heuristic tree-decomposition solvers [3, 30, 64]. There is little recent work on parallelizing heuristic tree-decomposition solvers. Instead, we implement a parallel portfolio of single-core tree-decomposition solvers and find that this portfolio significantly improves planning on multiple cores. Similar portfolio approaches have been well-studied and shown to be beneficial in the context of SAT solvers [8, 71]. As a theoretical contribution, we prove that branch-decomposition solvers can also be included in this portfolio. Unfortunately, we find that a current branch-decomposition solver does not significantly improve the portfolio.

The execution stage in [19] was done using `numpy` [48] and evaluated on a single core. We add an implementation of the execution stage that uses `TensorFlow` [1] to leverage a GPU for large contractions. Since GPU memory is significantly more limited than CPU memory, we add an implementation of *index slicing*. Index slicing is a recent technique from the tensor-network community [14, 29, 68], analogous to the classic technique of conditioning in Bayesian Network reasoning [16, 17, 52, 62], that allows memory to be significantly reduced at the cost of additional time. We find that, while multiple cores do not significantly improve the contraction stage, a GPU provides significant speedup, especially when augmented with index slicing.

We implement our techniques in `TensorOrder2`, a new parallel weighted model counter. We compare `TensorOrder2` to a variety of state-of-the-art counters. We show that the improved `TensorOrder2` is the fastest counter on 11% of benchmarks after preprocessing [42], outperforming the GPU-based counter `gpuSAT2` [26]. Thus `TensorOrder2` is useful as part of a portfolio of counters. All code is available at https://github.com/vardigroup/TensorOrder.

The rest of the paper is organized as follows: in Section 2, we give background information on weighted model counting, graph decompositions, and tensor networks. In Section 3, we discuss the algorithm for performing weighted model counting using tensor networks as outlined by [19]. In Section 4, we describe parallelization of this algorithm and two related theoretical improvements: planning with branch decompositions, and index slicing. In Section 5, we implement this parallelization in `TensorOrder2` and analyze its performance experimentally.

2 Preliminaries

In this section, we give background information on the three concepts we combine in this paper: weighted model counting, graph decompositions, and tensor-network contraction.

2.1 Literal-Weighted Model Counting

The task in weighted model counting is to count the total weight, subject to a given weight function, of the set of solutions of input constraints (typically given in CNF). We focus on so-called *literal-weight functions*, where the weight of a solution can be expressed as the product of weights associated with all satisfied literals. Formally:

Definition 2.1 (Weighted Model Count)**.** Let φ be a formula over Boolean variables X and let $W : X \times \{0,1\} \to \mathbb{R}$ be a function. The *(literal-)weighted model count* of φ w.r.t. W is $W(\varphi) \equiv \sum_{\tau \in [X]} \varphi(\tau) \cdot \prod_{x \in X} W(x, \tau(x))$, where $[X]$ is the set of all functions from X to $\{0,1\}$.

We focus in this work on weighted model counting, as opposed to *unweighted model counting* where the weight function W is constant. There are a variety of counters [13, 24, 65] that can perform only unweighted model counting and so we do not compare against them. Of particular

note here is `countAntom` [12], a multi-core unweighted model counter. An interesting direction for future work is to explore the integration of weights into `countAntom` and compare with tensor-network-based approaches to weighted model counting.

Existing approaches to weighted model counting can be split broadly into three categories: *direct reasoning*, *knowledge compilation*, and *dynamic programming*. In counters based on direct reasoning (e.g., `cachet` [60]), the idea is to reason directly about the CNF representation of φ. In counters based on knowledge compilation (e.g. `miniC2D` [50] and `d4` [43]), the idea is to compile φ into an alternative representation on which counting is easy. In counters based on dynamic programming (e.g. `ADDMC` [20] and `gpuSAT2` [25, 26]), the idea is to traverse the clause structure of φ. Tensor-network approaches to counting (e.g. `TensorOrder` [19] and this work) are also based on dynamic programming. Dynamic programming approaches often utilize graph decompositions, which we define in the next section.

2.2 Graphs and Graph Decompositions

A *graph* G has a nonempty set of vertices $\mathcal{V}(G)$, a set of (undirected) edges $\mathcal{E}(G)$, a function $\delta_G : \mathcal{V}(G) \to 2^{\mathcal{E}(G)}$ that gives the set of edges incident to each vertex, and a function $\epsilon_G : \mathcal{E}(G) \to 2^{\mathcal{V}(G)}$ that gives the set of vertices incident to each edge. . Each edge must be incident to exactly two vertices, but multiple edges can exist between two vertices. If $E \subset \mathcal{E}(G)$, let $\epsilon_G[E] = \bigcup_{e \in E} \epsilon_G(e)$.

A *tree* is a simple, connected, and acyclic graph. A *leaf* of a tree T is a vertex of degree one, and we use $\mathcal{L}(T)$ to denote the set of leaves of T. For every edge a of T, deleting a from T yields exactly two trees, whose leaves define a partition of $\mathcal{L}(T)$. Let $C_a \subseteq \mathcal{L}(T)$ denote an arbitrary element of this partition. A *rooted binary tree* is a tree T where either $|\mathcal{V}(T)| = 1$ or every vertex of T has degree one or three except a single vertex of degree two (called the *root*). If $|\mathcal{V}(T)| > 1$, the *immediate subtrees of* T are the two rooted binary trees that are the connected components of T after the root is removed.

In this work, we use three decompositions of a graph as a tree: tree decompositions [55], branch decompositions [55], and carving decompositions [61]. All decompose the graph into an *unrooted binary tree*, which is a tree where every vertex has degree one or three. First, we define tree decompositions [55]:

Definition 2.2. A *tree decomposition* for a graph G is an unrooted binary tree T together with a labeling function $\chi : \mathcal{V}(T) \to 2^{\mathcal{V}(G)}$ such that: (1) $\bigcup_{n \in \mathcal{V}(T)} \chi(n) = \mathcal{V}(G)$, (2) for all $e \in \mathcal{E}(G)$, there exists $n \in \mathcal{V}(T)$ s.t. $\epsilon_G(e) \subseteq \chi(n)$, and (3) for all $n, o, p \in \mathcal{V}(T)$, if p is on the path from n to o then $\chi(n) \cap \chi(o) \subseteq \chi(p)$.

The *width* of a tree decomposition is $width_t(T, \chi) \equiv \max_{n \in \mathcal{V}(T)} |\chi(n)| - 1$.

The treewidth of a graph G is the lowest width among all tree decompositions. Next, we define branch decompositions [55]:

Definition 2.3. A *branch decomposition* for a graph G with $\mathcal{E}(G) \neq \emptyset$ is an unrooted binary tree T whose leaves are the edges of G, i.e. $\mathcal{L}(T) = \mathcal{E}(G)$.

The *width* of T, denoted $width_b(T)$, is the maximum number of vertices in G that are endpoints of edges in both C_a and $\mathcal{E}(G) \setminus C_a$ for all $a \in \mathcal{E}(T)$, i.e., $width_b(T) \equiv \max_{a \in \mathcal{E}(T)} |\epsilon_G[C_a] \cap \epsilon_G[\mathcal{E}(G) \setminus C_a]|$.

The branchwidth of a graph G is the lowest width among all branch decompositions. Carving decompositions are the dual of branch decompositions and hence can be defined by swapping the role of $\mathcal{V}(G)$ and $\mathcal{E}(G)$ in Definition 2.3.

The treewidth (plus 1) of graph is no smaller than the branchwidth and is bounded from above by 3/2 times the branchwidth [55].

Given a CNF formula φ, a variety of associated graphs have been considered. The *incidence graph* of φ is the bipartite graph where both variables and clauses are vertices and edges indicate that the variable appears in the connected clause. The *primal graph* of φ is the graph where variables are vertices and edges indicate that two variables appear together in a clause. There are fixed-parameter tractable model counting algorithms with respect to the treewidth of the incidence graph and the primal graph [58]. If the treewidth of the primal graph of a formula φ is k, the treewidth of the incidence graph of φ is at most $k+1$ [38].

2.3 Tensors, Tensor Networks, and Tensor-Network Contraction

Tensors are a generalization of vectors and matrices to higher dimensions– a tensor with r dimensions is a table of values each labeled by r indices.

Fix a set **Ind** and define an *index* to be an element of **Ind**. For each index i fix a finite set $[i]$ called the *domain* of i. An index is analogous to a variable in constraint satisfaction. An *assignment* to $I \subseteq \mathbf{Ind}$ is a function τ that maps each index $i \in I$ to an element of $[i]$. Let $[I]$ denote the set of assignments to I, i.e., $[I] = \{\tau : I \to \bigcup_{i \in I}[i] \text{ s.t. } \tau(i) \in [i] \text{ for all } i \in I\}$.

We now formally define tensors as multidimensional arrays of values, indexed by assignments:

Definition 2.4 (Tensor)**.** A *tensor* A over a finite set of indices (denoted $\mathcal{I}(A)$) is a function $A : [\mathcal{I}(A)] \to \mathbb{R}$.

The *rank* of a tensor A is the cardinality of $\mathcal{I}(A)$. The memory to store a tensor (in a dense way) is exponential in the rank. For example, a scalar is a rank 0 tensor, a vector is a rank 1 tensor, and a matrix is a rank 2 tensor. Some other works generalize Definition 2.4 by replacing $\mathbb{R}$ with an arbitrary semiring.

A *tensor network* defines a complex tensor by combining a set of simpler tensors in a principled way. This is analogous to how a database query defines a resulting table in terms of a computation across many tables.

Definition 2.5 (Tensor Network)**.** A *tensor network* N is a nonempty set of tensors across which no index appears more than twice.

The set of indices of N that appear once (called *free indices*) is denoted by $\mathcal{F}(N)$. The set of indices of N that appear twice (called *bond indices*) is denoted by $\mathcal{B}(N)$.

The problem of *tensor-network contraction*, given an input tensor network N, is to compute the *contraction* of N by marginalizing all bond indices:

Definition 2.6 (Tensor-Network Contraction)**.** The *contraction* of a tensor network N is a tensor $\mathcal{T}(N)$ with indices $\mathcal{F}(N)$ (the set of free indices of N), i.e. a function $\mathcal{T}(N) : [\mathcal{F}(N)] \to \mathbb{R}$, that is defined for all $\tau \in [\mathcal{F}(N)]$ by

$$\mathcal{T}(N)(\tau) \equiv \sum_{\rho \in [\mathcal{B}(N)]} \prod_{A \in N} A((\rho \cup \tau)\big|_{\mathcal{I}(A)}). \tag{1}$$

A tensor network N' is a *partial contraction* of a tensor network N if there is a surjective function $f : N \to N'$ s.t. for every $A \in N'$ we have $\mathcal{T}(f^{-1}(A)) = A$; that is, if every tensor in N' is the contraction of some tensors of N. If N' is a partial contraction of N, then $\mathcal{T}(N') = \mathcal{T}(N)$.

Let A and B be tensors. Their *contraction* $A \cdot B$ is the contraction of the tensor network $\{A, B\}$. If $\mathcal{I}(A) = \mathcal{I}(B)$, their *sum* $A + B$ is the tensor with indices $\mathcal{I}(A)$ whose entries are given by the sum of the corresponding entries in A and B.

Algorithm 1 Weighted Model Counting with Tensor Networks

Input: A CNF formula φ, weight function W, and performance factor $\alpha \in \mathbb{R}$.
Output: $W(\varphi)$, the weighted model count of φ w.r.t. W.

```
N ← Reduce(φ, W)
repeat
    M, T ← Plan(N)
until α · TimeCost(M, T) < elapsed time in seconds
return Execute(M, T)(∅)
```

A tensor network can also be seen as a variant of a factor graph [40] with the additional restriction that no variable appears more than twice. The contraction of a tensor network corresponds to the marginalization of a factor graph [56], which is a a special case of the sum-of-products problem [6, 17] and the FAQ problem [2]. The restriction on variable appearance is heavily exploited in tools for tensor-network contraction and in this work, since it allows tensor contraction to be implemented as matrix multiplication and so leverage significant work in high-performance computing on matrix multiplication on CPUs [44] and GPUs [23].

3 Weighted Model Counting with Tensor Networks

In this section, we discuss the algorithm for literal-weighted model counting using tensor networks as outlined by [19]. This algorithm is presented as Algorithm 1 and has three stages.

First, in the *reduction* stage the input formula φ and weight function W is transformed into a tensor network N. We discuss in more detail in Section 3.1.

Second, in the *planning* stage a plan for contracting the tensor network N is determined. This plan takes the form of a *contraction tree* [22]:

Definition 3.1 (Contraction Tree)**.** Let N be a tensor network. A *contraction tree* for N is a rooted binary tree T whose leaves are the tensors of N.

The planning stage is allowed to modify the input tensor network N as long as the new tensor network M contracts to an identical tensor. We discuss various heuristics for contraction trees in Section 3.3. Planning in Algorithm 1 is an anytime process: we heuristically generate better contraction trees until one is "good enough" to use. The trade-off between planning and executing is governed by a parameter $\alpha \in \mathbb{R}$, which we determine empirically in Section 5.

Third, in the *execution* stage the chosen contraction tree is used to contract the tensor network. We discuss this algorithm in more detail in Section 3.2.

We assert the correctness of Algorithm 1 in the following theorem.

Theorem 3.1. *Let φ be a CNF formula and let W be a weight function. Assume: (1)* $\textsc{Reduce}(\varphi, W)$ *returns a tensor network N s.t. $\mathcal{T}(N)(\emptyset) = W(\varphi)$, (2)* $\textsc{Plan}(N)$ *returns a tensor network M and a contraction tree T for M s.t. $\mathcal{T}(M) = \mathcal{T}(N)$, and (3)* $\textsc{Execute}(M, T)$ *returns $\mathcal{T}(M)$ for all tensor networks M and contraction trees T for M. Then Algorithm 1 returns $W(\varphi)$.*

Proof. By Assumption 3, Algorithm 1 returns $\mathcal{T}(M)(\emptyset)$. By Assumption 2, this is equal to $\mathcal{T}(N)(\emptyset)$, which by Assumption 1 is exactly $W(\varphi)$. □

Organizationally, note that we discuss the execution stage in Section 3.2 before we discuss the planning stage. This is because we must understand how plans are used before we can evaluate various planning algorithms.

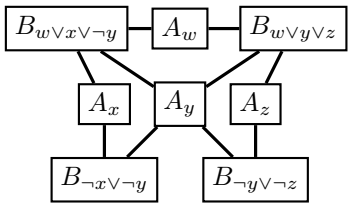

Figure 1: The tensor network produced by Theorem 3.2 on $\varphi = (w \vee x \vee \neg y) \wedge (w \vee y \vee z) \wedge (\neg x \vee \neg y) \wedge (\neg y \vee \neg z)$ has 8 tensors (indicated by vertices) and 10 indices (indicated by edges). The A_* tensors correspond to variables and have entries computed from the weight function. The B_* tensors correspond to clauses and have entries computed from the clause negations.

3.1 The Reduction Stage

There is a reduction from weighted model counting to tensor-network contraction, as described by the following theorem of [19]:

Theorem 3.2. *Let φ be a CNF formula over Boolean variables X and let W be a weight function. One can construct in polynomial time a tensor network $N_{\varphi,W}$ such that $\mathcal{F}(N_{\varphi,W}) = \emptyset$ and the contraction of $N_{\varphi,W}$ is $W(\varphi)$.*

Sketch of Construction. Let $I = \{(x, C) \in X \times \varphi : x \text{ appears in } C\}$ be a set of indices, each with domain $\{0, 1\}$. The key idea is create a tensor A_x for each variable $x \in X$ and a tensor B_C for each clause $C \in \varphi$ so that each index $(x, C) \in I$ is an index of A_x and B_C.

For each $x \in X$, let A_x be a tensor with indices $I \cap (\{x\} \times \varphi)$. For each $\tau \in [I \cap (\{x\} \times \varphi)]$, define $A_x(\tau)$ to be $W(x, 0)$ if τ is always 0, $W(x, 1)$ if τ is always 1, and 0 otherwise.

For each $C \in \varphi$, let B_C be a tensor with indices $I \cap (X \times \{C\})$. For each $\tau \in [I \cap (X \times \{C\})]$, define $B_C(\tau)$ to be 1 if $\{x \in X : x \text{ appears in } C \text{ and } \tau((x, C)) = 1\}$ satisfies C and 0 otherwise.

Then $N_{\varphi,W} = \{A_x : x \in X\} \cup \{B_C : C \in \varphi\}$ is the desired tensor network. □

Theorem 3.2 proves that $\textsc{Reduce}(\varphi, W) = N_{\varphi,W}$ satisfies assumption 1 in Theorem 3.1. See Figure 1 for an example of the reduction. This reduction is closely related to the formulation of model counting as the marginalization of a factor graph representing the constraints (but assigns tensors to variables as well, not just clauses). This reduction can also be extended to other types of constraints, e.g. parity or cardinality constraints.

3.2 The Execution Stage

For formulas φ with hundreds of clauses, the corresponding tensor network produced by Theorem 3.2 has hundreds of bond indices. Directly following Equation 1 in this case is infeasible, since it sums over an exponential number of terms.

Instead, Algorithm 2 shows how to compute $\mathcal{T}(N)$ for a tensor network N using a contraction tree T as a guide. The key idea is to repeatedly choose two tensors $A_1, A_2 \in N$ (according to the structure of T) and contract them. One can prove inductively that Algorithm 2 satisfies assumption 3 of Theorem 3.1.

Each contraction in Algorithm 2 contains exactly two tensors and so can be implemented as a matrix multiplication. In [19], `numpy` [48] was used to perform each of these contractions. Although the choice of contraction tree does not affect the correctness of Algorithm 2, it may have a dramatic impact on the running-time and memory usage. We explore this further in the following section.

Algorithm 2 Recursively contracting a tensor network

Input: A tensor network N and a contraction tree T for N.
Output: $\mathcal{T}(N)$, the contraction of N.

```
1: procedure EXECUTE(N, T)
2:     if |N| = 1 then
3:         return the tensor contained in N
4:     else
5:         T_1, T_2 ← immediate subtrees of T
6:         return EXECUTE(L(T_1), T_1) · EXECUTE(L(T_2), T_2)
```

3.3 The Planning Stage

The task in the planning stage is, given a tensor network, to find a contraction tree that minimizes the computational cost of Algorithm 2.

Several *cost-based* approaches [32, 53] aim for minimizing the total number of floating point operations to perform Algorithm 2. Most cost-based approaches are designed for tensor networks with very few, large tensors, whereas weighted model counting benchmarks produce tensor networks with many, small tensors. These approaches are thus inappropriate for counting.

Instead, we focus here on *structure-based* approaches, which analyze the rank of intermediate tensors that appear during Algorithm 2. The ranks indicate the amount of memory and computation required at each recursive stage. The goal is then to find a contraction tree with small *max-rank*, which is the maximum rank of all tensors that appear during Algorithm 2.

This is done through analysis of the *structure graph*, a representation of a tensor network as a graph where tensors are vertices and indices indicate the edges between tensors [19, 46, 72]:

Definition 3.2 (Structure Graph)**.** Let N be a tensor network. The *structure graph* of N is the graph, denoted $G = \text{struct}(N)$, where $\mathcal{V}(G) = N \cup \{z\}$ (z is a fresh vertex called the *free vertex*), $\mathcal{E}(G) = \mathcal{B}(N) \cup \mathcal{F}(N)$, $\delta_G(z) = \mathcal{F}(N)$, and, for all $A \in N$, $\delta_G(A) = \mathcal{I}(A)$.

For example, if φ is a formula and $N_{\varphi,W}$ is the tensor network resulting from Theorem 3.2, then $\text{struct}(N_{\varphi,W})$ is the incidence graph of φ (and is independent of the weight function).

One structure-based approach for finding contraction trees [21, 46] utilizes low-width tree decompositions of the line graph of $\text{struct}(N)$. This approach is analogous to *variable elimination* on factor graphs [6, 35], which uses tree decompositions of the primal graph of a factor graph.

Unfortunately, for many tensor network all possible contraction trees have large max-rank. [19] observed that this is the case for many standard counting benchmarks. This is because, for each variable x, the number of times x appears in φ is a lower bound on the max-rank of all contraction trees of $\text{REDUCE}(\varphi, \cdot)$. Thus traditional tensor network planning algorithms (e.g., [21, 39, 46]) fail on counting benchmarks that have variables which appear more than 30 times.

Instead, [19] relaxed the planning stage by allowing modifications to the input tensor network N. In particular, [19] utilized tree decompositions of $\text{struct}(N)$ to factor high-rank, highly-structured tensors as a preprocessing step, generalizing a prior method of splitting large CNF clauses [59]. In this case, a tensor is highly-structured if it is *tree-factorable*:

Definition 3.3. A tensor A is *tree factorable* if, for every tree T whose leaves are $\mathcal{I}(A)$, there is a tensor network N_A and a bijection $g_A : \mathcal{V}(T) \to N_A$ s.t. A is the contraction of N_A and:

1. g_A is an isomorphism between T and $\text{struct}(N_A)$ with the free vertex removed,
2. for every index i of A, i is an index of $g_A(i)$, and

3. for some index i of A, every index $j \in \mathcal{B}(N_A)$ satisfies $|[j]| \leq |[i]|$.

Informally, a tensor is tree-factorable if it can be replaced by arbitrary trees of low-rank tensors. All tensors produced by Theorem 3.2 are tree factorable. The preprocessing method of [19] is then formalized as follows:

Theorem 3.3. *Let N be a tensor network of tree-factorable tensors such that $|\mathcal{F}(N)| \leq 3$ and struct(N) has a tree decomposition of width $w \geq 1$. Then we can construct in polynomial time a tensor network M and a contraction tree T for M s.t. max-rank$(T) \leq \lceil 4(w+1)/3 \rceil$ and N is a partial contraction of M.*

This gives us the FactorTree(N) planning method, which computes a tree decomposition of struct(N) and returns the M and T that result from Theorem 3.3. Because N is a partial contraction of M, $\mathcal{T}(N) = \mathcal{T}(M)$. Since all tensors produced by Thereom 3.2 are tree-factorable, FactorTree(N) satisfies assumption 2 in Theorem 3.1.

While both FactorTree(N) and variable elimination [6, 35] use tree decompositions, they consider different graphs. For example, consider computing the model count of $\psi = (\vee_{i=1}^{n} x_i) \wedge (\vee_{i=1}^{n} \neg x_i)$. FactorTree$(N)$ uses tree decompositions of the incidence graph of ψ, which has treewidth 2. Variable elimination uses tree decompositions of the primal graph of ψ, which has treewidth $n-1$. Thus FactorTree(N) can exhibit significantly better behavior than variable elimination on some formulas. On the other hand, the behavior of FactorTree(N) is at most slightly worse than variable elimination: as noted above, if the treewidth of the primal graph of a formula φ is k, the treewidth of the incidence graph of φ is at most $k+1$ [38].

4 Parallelizing Tensor-Network Contraction

In this section, we discuss opportunities for parallelization of Algorithm 1. Since the reduction stage was not a significant source of runtime in [19], we focus on opportunities in the planning and execution stages.

4.1 Parallelizing the Planning Stage with a Portfolio

As discussed in Section 3.3, FactorTree(N) first computes a tree decomposition of struct(N) and then applies Theorem 3.3. In [19], most time in the planning stage was spent finding low-width tree decompositions with a choice of single-core heuristic tree-decomposition solvers [3, 30, 64]. Finding low-width tree decompositions is thus a valuable target for parallelization.

We were unable to find state-of-the-art parallel heuristic tree-decomposition solvers. There are several classic algorithms for parallel tree-decomposition construction [41, 63], but no recent implementations. There is also an exact tree-decomposition solver that leverages a GPU [66], but exact tools are not able to scale to handle our benchmarks. Existing single-core heuristic tree-decomposition solvers are highly optimized and so parallelizing them is nontrivial.

Instead, we take inspiration from the SAT solving community. One competitive approach to building a parallel SAT solver is to run a variety of SAT solvers in parallel across multiple cores [8, 45, 71]. The portfolio SAT solver `CSHCpar` [45] won the open parallel track in the 2013 SAT Competition with this technique, and such portfolio solvers are now banned from most tracks due to their effectiveness relative to their ease of implementation. Similar parallel portfolios are thus promising for integration into the planning stage.

We apply this technique to heuristic tree-decomposition solvers by running a variety of single-core heuristic tree-decomposition solvers in parallel on multiple cores and collating their outputs into a single stream. We analyze the performance of this technique in Section 5.

While Theorem 3.3 lets us integrate tree-decomposition solvers into the portfolio, the portfolio might also be improved by integrating solvers of other graph decompositions. The following theorem shows that branch-decomposition solvers may also be used for FactorTree(N):

Theorem 4.1. *Let N be a tensor network of tree-factorable tensors such that $|\mathcal{F}(N)| \leq 3$ and $G = \textit{struct}(N)$ has a branch decomposition T of width $w \geq 1$. Then we can construct in polynomial time a tensor network M and a contraction tree T for M s.t. max-rank$(T) \leq \lceil 4w/3 \rceil$ and N is a partial contraction of M.*

Sketch of Construction. For simplicity, we sketch here only the case when $\mathcal{F}(N) = \emptyset$, as occurs in counting. For each $A \in N$, $\mathcal{I}(A) = \delta_G(A)$ is a subset of the leaves of T and so the smallest connected component of T containing $\mathcal{I}(A)$ is a dimension tree T_A of A. Factor A with T_A using Definition 3.3 to get N_A and g_A.

We now construct the contraction tree for $M = \cup_{A \in N} N_A$. For each $n \in \mathcal{V}(T)$, let $M_n = \{B \ : \ B \in N_A, g_A(B) = n\}$. At each leaf $\ell \in \mathcal{L}(T)$, attach an arbitrary contraction tree of M_ℓ. At each non-leaf $n \in \mathcal{V}(T)$, partition M_n into three equally-sized sets and attach an arbitrary contraction tree for each to the three edges incident to n. These attachments create a carving decomposition T' from T for struct(M), of width no larger than $\lceil 4w/3 \rceil$. Finally, apply Theorem 2 of [19] to construct a contraction tree S for M from T' s.t. max-rank$(S) \leq \lceil 4w/3 \rceil$. □

The full proof is an extension of the proof of Theorem 3.3 in [19]. Theorem 4.1 subsumes Theorem 3.3, since given a graph G and a tree decomposition for G of width $w+1$ one can construct in polynomial time a branch decomposition for G of width w [55]. Moreover, on many graphs there are branch decompositions whose width is smaller than all tree decompositions. We explore this potential in practice in Section 5.

It was previously known that branch decompositions can also be used in variable elimination [6], but Theorem 4.1 considers branch decompositions of a different graph. For example, consider again computing the model count of $\psi = (\vee_{i=1}^{n} x_i) \wedge (\vee_{i=1}^{n} \neg x_i)$. Theorem 4.1 uses branch decompositions of the incidence graph of ψ, which has branchwidth 2. Variable elimination uses branch decompositions of the primal graph of ψ, which has branchwidth $n-1$.

4.2 Parallelizing the Execution Stage with TensorFlow and Slicing

As discussed in Section 3.2, each contraction in Algorithm 2 can be implemented as a matrix multiplication. This was done in [19] using `numpy` [48], and it is straightforward to adjust the implementation to leverage multiple cores with `numpy` and a GPU with `TensorFlow` [1].

The primary challenge that emerges when using a GPU is dealing with the small onboard memory. For example, the `NVIDIA Tesla-V100` (which we use in Section 5) has just 16GB of onboard memory. This limits the size of tensors that can be easily manipulated. A single contraction of two tensors can be performed across multiple GPU kernel calls [57], and similar techniques were implemented in `gpusat2` [26]. These techniques, however, require parts of the input tensors to be copied into and out of GPU memory, which incurs significant slowdown.

Instead, we use *index slicing* [14, 29, 68] of a tensor network N. This technique is analogous to *conditioning* on Bayesian networks [16, 17, 52, 62]. The idea is to represent $\mathcal{T}(N)$ as the sum of contractions of smaller tensor networks. Each smaller network contains tensor slices:

Definition 4.1. Let A be a tensor, $I \subseteq$ **Ind**, and $\eta \in [I]$. Then the *η-slice* of A is the tensor $A[\eta]$ with indices $\mathcal{I}(A) \setminus I$ defined for all $\tau \in [\mathcal{I}(A) \setminus I]$ by $A[\eta](\tau) \equiv A((\tau \cup \eta)\big|_{\mathcal{I}(A)})$.

These are called slices because every value in A appears in exactly one tensor in $\{A[\eta] : \eta \in [I]\}$. We now define and prove the correctness of index slicing:

Algorithm 3 Sliced contraction of a tensor network

Input: A tensor network N, a contraction tree T for N, and a memory bound m.
Output: $\mathcal{T}(N)$, the contraction of N, performed using at most m memory.

1: $I \leftarrow \emptyset$
2: **while** $\textsc{MemCost}(N, T, I) > m$ **do**
3: $\quad I \leftarrow I \cup \{\textsc{ChooseSliceIndex}(N, T, I)\}$
4: **return** $\sum_{\eta \in [E]} \textsc{Execute}(N[\eta], T[\eta])$

Theorem 4.2. *Let N be a tensor network and let $I \subseteq \mathcal{B}(N)$. For each $\eta \in [I]$, let $N[\eta] = \{A[\eta] : A \in N\}$. Then $\mathcal{T}(N) = \sum_{\eta \in [I]} \mathcal{T}(N[\eta])$.*

Proof. Move the summation over $\eta \in [I]$ to the outside of Equation 1, then apply the definition of tensor slices and recombine terms. □

By choosing I carefully, computing each $\mathcal{T}(N[\eta])$ uses less intermediate memory (compared to computing $\mathcal{T}(N)$) while using the same contraction tree. In exchange, the number of floating point operations to compute all $\mathcal{T}(N[\eta])$ terms increases.

Choosing I is itself a difficult problem. Our goal is to choose the smallest I so that contracting each network slice $N[\eta]$ can be done in onboard memory. We first consider adapting Bayesian network conditioning heuristics to the context of tensor networks. Two popular conditioning heuristics are (1) *cutset conditioning* [52], which chooses I so that each network slice is a tree, and (2) *recursive conditioning* [16], which chooses I so that each network slice is disconnected[1]. Both of these heuristics result in a choice of I far larger than our goal requires.

Instead, in this work as a first step we use a heuristic from [14, 29]: choose I incrementally, greedily minimizing the memory cost of contracting $N[\eta]$ until the memory usage fits in onboard memory. Unlike cutset and recursive conditioning, the resulting networks $N[\eta]$ are typically still highly connected. One direction for future work is to compare other heuristics for choosing I (e.g., see the discussion in Section 10 of [17]).

This gives us Algorithm 3, which performs the execution stage with limited memory at the cost of additional time. $T[\eta]$ is the contraction tree obtained by computing the η-slice of every tensor in T. $\textsc{MemCost}(N, T, I)$ computes the memory for one $\textsc{Execute}(N[\eta], T[\eta])$ call. $\textsc{ChooseSliceIndex}(N, T, I)$ chooses the next slice index greedily to minimize memory cost.

5 Implementation and Evaluation

We aim to answer the following research questions:

(RQ1) Is the planning stage improved by a parallel portfolio of decomposition tools?

(RQ2) Is the planning stage improved by adding a branch-decomposition tool?

(RQ3) When should Algorithm 1 transition from the planning stage to the execution stage (i.e., what should be the value of the performance factor α)?

(RQ4) Is the execution stage improved by leveraging multiple cores and a GPU?

(RQ5) Do parallel tensor network approaches improve a portfolio of state-of-the-art weighted model counters (`cachet`, `miniC2D`, `d4`, `ADDMC`, and `gpuSAT2`)?

[1] The full recursive conditioning procedure then recurses on each connected component. While recursive conditioning is an any-space algorithm, the partial caching required for this is difficult to implement on a GPU.

We implement our changes on top of `TensorOrder` [19] (which implements Algorithm 1) to produce `TensorOrder2`, a new parallel weighted model counter. Implementation details are described in Section 5.1. All code is available at https://github.com/vardigroup/TensorOrder.

We use a standard set[2] of 1914 weighted model counting benchmarks [20]. Of these, 1091 benchmarks[3] are from Bayesian inference problems [60] and 823 benchmarks[4] are unweighted benchmarks from various domains that were artificially weighted by [20]. For weighted model counters that cannot handle real weights larger than 1 (`cachet` and `gpusat2`), we rescale the weights of benchmarks with larger weights. In Experiment 3, we also consider preprocessing these 1914 benchmarks by applying `pmc-eq` [42] (which preserves weighted model count). We evaluate the performance of each tool using the PAR-2 score, which is the sum of of the wall-clock times for each completed benchmark, plus twice the timeout for each uncompleted benchmark.

All counters are run in the Docker images (one for each counter) with Docker 19.03.5. All experiments are run on Google Cloud `n1-standard-8` machines with 8 cores (Intel Haswell, 2.3 GHz) and 30 GB RAM. GPU-based counters are provided an `NVIDIA Tesla V100` GPU (16 GB of onboard RAM) using NVIDIA driver 418.67 and CUDA 10.1.243.

5.1 Implementation Details of `TensorOrder2`

`TensorOrder2` is primarily implemented in Python 3 as a modified version of `TensorOrder`. We replace portions of the Python code with C++ (`g++` v7.4.0) using Cython 0.29.15 for general speedup, especially in FactorTree(·).

Planning. `TensorOrder` contains an implementation of the planning stage using a choice of three single-core tree-decomposition solvers: `Tamaki` [64], `FlowCutter` [30], and `htd` [3]. We add to `TensorOrder2` an implementation of Theorem 4.1 and use it to add a branch-decomposition solver `Hicks` [31]. We implement a parallel portfolio of graph-decomposition solvers in C++ and give `TensorOrder2` access to two portfolios, each with access to all cores: `P3` (which combines `Tamaki`, `FlowCutter`, and `htd`) and `P4` (which includes `Hicks` as well).

Execution. `TensorOrder` is able to perform the execution stage on a single core and on multiple cores using `numpy` v1.18.1 and `OpenBLAS` v0.2.20. We add to `TensorOrder2` the ability to contract tensors on a GPU with `TensorFlow` v2.1.0 [1]. To avoid GPU kernel calls for small contractions, `TensorOrder2` uses a GPU only for contractions where one of the tensors involved has rank ≥ 20, and reverts back to using multi-core `numpy` otherwise. We also add an implementation of Algorithm 3. Overall, `TensorOrder2` runs the execution stage on three hardware configurations: `CPU1` (restricted to a single CPU core), `CPU8` (allowed to use all 8 CPU cores), and `GPU` (allowed to use all 8 CPU cores and use a GPU).

5.2 Experiment 1: The Planning Stage (RQ1 and RQ2)

We run each planning implementation (`FlowCutter`, `htd`, `Tamaki`, `Hicks`, `P3`, and `P4`) once on each of our 1914 benchmarks and save all contraction trees found within 1000 seconds (without executing the contractions). Results are summarized in Figure 2.

We observe that the parallel portfolio planners outperform all four single-core planners after 5 seconds. In fact, after 20 seconds both portfolios perform almost as well as the virtual best solver. We conclude that portfolio solvers significantly speed up the planning stage.

[2] https://github.com/vardigroup/ADDMC/releases/tag/v1.0.0
[3] https://www.cs.rochester.edu/u/kautz/Cachet/
[4] http://www.cril.univ-artois.fr/KC/benchmarks.html

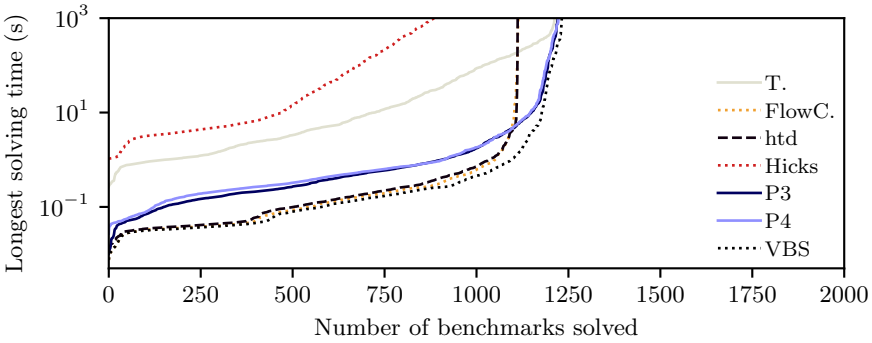

Figure 2: A cactus plot of the performance of various planners. A planner "solves" a benchmark when it finds a contraction tree of max rank 30 or smaller.

We also observe that `P3` and `P4` perform almost identically in Figure 2. Although after 1000 seconds `P4` has found better contraction trees than `P3` on 407 benchmarks, most improvements are small (reducing the max-rank by 1 or 2) or still do not result in good-enough contraction trees. We conclude that adding `Hicks` improves the portfolio slightly, but not significantly.

5.3 Experiment 2: Determining the Performance Factor (RQ3)

We take each contraction tree discovered in Experiment 1 (with max-rank below 36) and use `TensorOrder2` to execute the tree with a timeout of 1000 seconds on each of three hardware configurations (`CPU1`, `CPU8`, and `GPU`). We observe that the max-rank of almost all solved contraction trees is 30 or smaller.

Given a performance factor, a benchmark, and a planner, we use the planning times from Experiment 1 to determine which contraction tree would have been chosen in step 4 of Algorithm 1. We then add the execution time of the relevant contraction tree on each hardware. In this way, we simulate Algorithm 1 for a given planner and hardware with many performance factors.

For each planner and hardware, Table 2 shows the performance factor α that minimizes the simulated PAR-2 score. We observe that the performance factor for `CPU8` is lower than for `CPU1`, but not necessarily higher or lower than for `GPU`. We conclude that different combinations of planners and hardware are optimized by different performance factors.

5.4 Experiment 3: End-to-End Performance (RQ4 and RQ5)

Finally, we compare `TensorOrder2` with state-of-the-art weighted model counters `cachet`, `miniC2D`, `d4`, `ADDMC`, and `gpuSAT2`. We consider `TensorOrder2` using `P4` combined with each hardware configuration (`CPU1`, `CPU8`, and `GPU`), along with `Tamaki` + `CPU1` as the best non-

Table 1: The performance factor for each combination of planner and hardware that minimizes the simulated PAR-2 score.

	`Tamaki`	`FlowCutter`	`htd`	`Hicks`	`P3`	`P4`
`CPU1`	$3.8 \cdot 10^{-11}$	$4.8 \cdot 10^{-12}$	$1.6 \cdot 10^{-12}$	$1.0 \cdot 10^{-21}$	$1.4 \cdot 10^{-11}$	$1.6 \cdot 10^{-11}$
`CPU8`	$7.8 \cdot 10^{-12}$	$1.8 \cdot 10^{-12}$	$1.3 \cdot 10^{-12}$	$1.0 \cdot 10^{-21}$	$5.5 \cdot 10^{-12}$	$6.2 \cdot 10^{-12}$
`GPU`	$2.1 \cdot 10^{-12}$	$5.5 \cdot 10^{-13}$	$1.3 \cdot 10^{-12}$	$1.0 \cdot 10^{-21}$	$3.0 \cdot 10^{-12}$	$3.8 \cdot 10^{-12}$

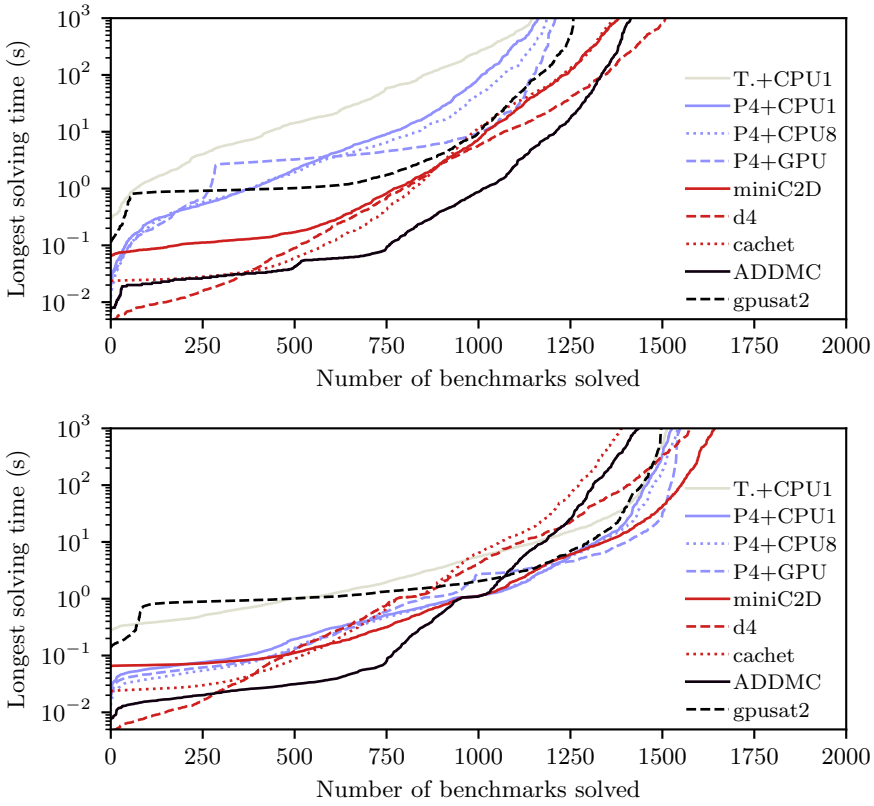

Figure 3: A cactus plot of the number of benchmarks solved by various counters, without (above) and with (below) the `pmc-eq` [42] preprocessor.

parallel configuration from [19]. Note that `P4+CPU1` still leverages multiple cores in the planning stage. The performance factor from Experiment 2 is used for each `TensorOrder2` configuration.

We run each counter once on each benchmark (both with and without `pmc-eq` preprocessing) with a timeout of 1000 seconds and record the wall-clock time taken. When preprocessing is used, both the timeout and the recorded time include preprocessing time. For `TensorOrder2`, recorded times include all of Algorithm 1. Results are summarized in Figure 3 and Table 2.

We observe that the performance of `TensorOrder2` is improved by the portfolio planner and, on hard benchmarks, by executing on a multi-core CPU and on a GPU. The flat line at 3 seconds for `P4 + GPU` is caused by overhead from initializing the GPU.

Comparing `TensorOrder2` with the other counters, `TensorOrder2` is competitive without preprocessing but solves fewer benchmarks than all other counters, although `TensorOrder2` (with some configuration) is faster than all other counters on 158 benchmarks before preprocessing. We observe that preprocessing significantly boosts `TensorOrder2` relative to other counters, similar to prior observations with `gpusat2` [26]. `TensorOrder2` solves the third-most preprocessed benchmarks of any solver and has the second-lowest PAR-2 score (notably, outperforming `gpusat2` in both measures). `TensorOrder2` (with some configuration) is faster than all other counters on 200 benchmarks with preprocessing. Since `TensorOrder2` improves the virtual best solver on 158 benchmarks without preprocessing and on 200 benchmarks with preprocessing, we conclude that `TensorOrder2` is useful as part of a portfolio of counters.

Table 2: The numbers of benchmarks solved by each counter fastest and in total after 1000 seconds, and the PAR-2 score.

	Without preprocessing			With `pmc-eq` preprocessing		
	# Fastest	# Solved	PAR-2 Score	# Fastest	# Solved	PAR-2 Score
`T.+CPU1`	0	1151	1640803.	0	1514	834301.
`P4+CPU1`	45	1164	1562474.	83	1526	805547.
`P4+CPU8`	50	1185	1500968.	67	1542	771821.
`P4+GPU`	63	1210	1436949.	50	1549	745659.
`miniC2D`	50	1381	1131457.	221	1643	585908.
`d4`	615	1508	883829.	550	1575	747318.
`cachet`	264	1363	1156309.	221	1391	1099003.
`ADDMC`	640	1415	1032903.	491	1436	1008326.
`gpusat2`	37	1258	1342646.	25	1497	854828.

6 Discussion

In this work, we explored the impact of multiple-core and GPU use on tensor network contraction for weighted model counting. We implemented our techniques in `TensorOrder2`, a new parallel counter, and showed that `TensorOrder2` is useful as part of a portfolio of counters.

In the planning stage, we showed that a parallel portfolio of graph-decomposition solvers is significantly faster than single-core approaches. We proved that branch decomposition solvers can also be included in this portfolio, but concluded that a state-of-the-art branch decomposition solver only slightly improves the portfolio. For future work, it would be interesting to consider leveraging other width parameters (e.g. [4] or [27]) in the portfolio as well. It may also be possible to improve the portfolio through more advanced algorithm-selection techniques [33, 70]. Finally, one could develop parallel heuristic decomposition solvers directly, e.g., by adapting the exact GPU-based tree-decomposition solver [66] into a heuristic solver.

In the execution stage, we showed that tensor contractions can be performed with `TensorFlow` on a GPU. When combined with index slicing, we concluded that a GPU speeds up the execution stage for many hard benchmarks. For easier benchmarks, the overhead of a GPU may outweigh any contraction speedups. We focused here on parallelism within a single tensor contraction, but there are opportunities in future work to exploit higher-level parallelism, e.g. by running each slice computation in Algorithm 3 on a separate GPU.

`TensorFlow` also supports performing tensor contractions on TPUs (tensor processing unit [34]), which are specialized hardware designed for neural network training and inference. Tensor networks therefore provide a natural framework to leverage TPUs for weighted model counting as well. There are additional challenges in the TPU setting: floating-point precision is limited, and there is a (100+ second) compilation stage from a contraction tree into XLA [69]. We plan to explore these challenges further in future work.

Acknowledgments

This work was supported in part by the NSF (grants IIS-1527668, CCF-1704883, IIS-1830549, and DMS-1547433) and by Google Cloud.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proc. of OSDI*, pages 265–283, 2016.

[2] Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. FAQ: questions asked frequently. In *Proc. of PODS*, pages 13–28. ACM, 2016.

[3] Michael Abseher, Nysret Musliu, and Stefan Woltran. htd- a free, open-source framework for (customized) tree decompositions and beyond. In *Proc. of CPAIOR*, pages 376–386, 2017.

[4] Isolde Adler, Georg Gottlob, and Martin Grohe. Hypertree width and related hypergraph invariants. *European Journal of Combinatorics*, 28(8):2167–2181, 2007.

[5] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *Proc. of FOCS*, pages 340–351, 2003.

[6] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Solving #SAT and bayesian inference with backtracking search. *Journal of Artificial Intelligence Research*, 34:391–442, 2009.

[7] Brett W Bader and Tamara G Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2007.

[8] Tomáš Balyo, Peter Sanders, and Carsten Sinz. HordeSat: A massively parallel portfolio SAT solver. In *Proc. of SAT*, pages 156–172. Springer, 2015.

[9] Jacob Biamonte and Ville Bergholm. Tensor networks in a nutshell. *arXiv preprint arXiv:1708.00006*, 2017.

[10] Jacob D Biamonte, Jason Morton, and Jacob Turner. Tensor network contractions for #SAT. *Journal of Statistical Physics*, 160(5):1389–1404, 2015.

[11] Jan Burchard, Tobias Schubert, and Bernd Becker. Laissez-faire caching for parallel# sat solving. In *Proc. of SAT*, pages 46–61. Springer, 2015.

[12] Jan Burchard, Tobias Schubert, and Bernd Becker. Laissez-faire caching for parallel #SAT solving. In *Proc. of SAT*, pages 46–61. Springer, 2015.

[13] Günther Charwat and Stefan Woltran. Dynamic programming-based QBF solving. In *Proc. of SAT*, pages 27–40, 2016.

[14] Jianxin Chen, Fang Zhang, Cupjin Huang, Michael Newman, and Yaoyun Shi. Classical simulation of intermediate-size quantum circuits. *arXiv preprint arXiv:1805.01450*, 2018.

[15] Andrzej Cichocki. Era of big data processing: A new approach via tensor networks and tensor decompositions. *arXiv preprint arXiv:1403.2048*, 2014.

[16] Adnan Darwiche. Recursive conditioning. *Artificial Intelligence*, 126(1-2):5–41, 2001.

[17] Rina Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

[18] Carmel Domshlak and Jörg Hoffmann. Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research*, 30(1):565–620, 2007.

[19] Jeffrey M Dudek, Leonardo Dueñas-Osorio, and Moshe Y Vardi. Efficient contraction of large tensor networks for weighted model counting through graph decompositions. *arXiv preprint arXiv:1908.04381*, 2019.

[20] Jeffrey M Dudek, Vu H N Phan, and Moshe Y Vardi. ADDMC: Weighted model counting with algebraic decision diagrams. In *Proc. of AAAI*, 2020.

[21] Eugene F Dumitrescu, Allison L Fisher, Timothy D Goodrich, Travis S Humble, Blair D Sullivan, and Andrew L Wright. Benchmarking treewidth as a practical component of tensor network simulations. *PloS one*, 13(12), 2018.

[22] Glen Evenbly and Robert NC Pfeifer. Improving the efficiency of variational tensor network algorithms. *Physical Review B*, 89(24):245118, 2014.

[23] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. Understanding the efficiency of GPU

algorithms for matrix-matrix multiplication. In *Proc. of EUROGRAPHICS*, pages 133–137. ACM, 2004.

[24] Johannes K Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. Answer set solving with bounded treewidth revisited. In *Proc. of LPNMR*, pages 132–145. Springer, 2017.

[25] Johannes K Fichte, Markus Hecher, Stefan Woltran, and Markus Zisser. Weighted model counting on the GPU by exploiting small treewidth. In *Proc. of ESA*, pages 28:1–28:16, 2018.

[26] Johannes K Fichte, Markus Hecher, and Markus Zisser. gpusat2–an improved GPU model counter. In *Proc. of CP*, 2019.

[27] Robert Ganian and Stefan Szeider. New width parameters for model counting. In *Proc. of SAT*, pages 38–52. Springer, 2017.

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

[29] Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *arXiv preprint arXiv:2002.01935*, 2020.

[30] Michael Hamann and Ben Strasser. Graph bisection with pareto optimization. *Journal of Experimental Algorithmics (JEA)*, 23(1):1–2, 2018.

[31] Illya V Hicks. Branchwidth heuristics. *Congressus Numerantium*, pages 31–50, 2002.

[32] So Hirata. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A*, 107(46):9887–9897, 2003.

[33] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[34] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proc. of ISCA*, pages 1–12, 2017.

[35] Kalev Kask, Rina Dechter, Javier Larrosa, and Avi Dechter. Unifying tree decompositions for reasoning in graphical models. *Artificial Intelligence*, 166(1-2):165–193, 2005.

[36] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and P Sadayappan. A code generator for high-performance tensor contractions on GPUs. In *Proc. of CGO*, pages 85–95. IEEE Press, 2019.

[37] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. of PACMPL*, page 77, 2017.

[38] Phokion G Kolaitis and Moshe Y Vardi. Conjunctive-query containment and constraint satisfaction. *Journal of Computer and System Sciences*, 61(2):302–332, 2000.

[39] Stefanos Kourtis, Claudio Chamon, Eduardo R Mucciolo, and Andrei E Ruckenstein. Fast counting with tensor networks. *arXiv preprint arXiv:1805.00475*, 2018.

[40] Frank R Kschischang, Brendan J Frey, and Hans-Andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory*, 47(2):498–519, 2001.

[41] Jens Lagergren. Efficient parallel algorithms for tree-decomposition and related problems. In *Proc. of FOCS*, pages 173–182. IEEE, 1990.

[42] Jean-Marie Lagniez and Pierre Marquis. Preprocessing for propositional model counting. In *Proc. of AAAI*, pages 2688–2694, 2014.

[43] Jean-Marie Lagniez and Pierre Marquis. An improved decision-DNNF compiler. In *Proc. of IJCAI*, pages 667–673, 2017.

[44] Chuck L Lawson, Richard J Hanson, David R Kincaid, and Fred T Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 1977.

[45] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Parallel lingeling, CCASat, and CSCH-based portfolios. *Proc. of SAT Competition*, pages 26–27, 2013.

[46] Igor L Markov and Yaoyun Shi. Simulating quantum computation by contracting tensor networks. *SIAM Journal on Computing*, 38(3):963–981, 2008.

[47] Thomas Nelson, Axel Rivera, Prasanna Balaprakash, Mary Hall, Paul D Hovland, Elizabeth Jessup, and Boyana Norris. Generating efficient tensor contractions for GPUs. In *Proc. of ICPP*, pages 969–978. IEEE, 2015.

[48] NumPy. http://www.numpy.org/. Accessed: 2020-02-20.

[49] Román Orús. A practical introduction to tensor networks: Matrix product states and projected entangled pair states. *Annals of Physics*, 349:117–158, 2014.

[50] Umut Oztok and Adnan Darwiche. A top-down compiler for sentential decision diagrams. In *Proc. of IJCAI*, pages 3141–3148, 2015.

[51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Proc. of NeurIPS*, pages 8024–8035, 2019.

[52] Judea Pearl. Fusion, propagation, and structuring in belief networks. *Artificial intelligence*, 29(3):241–288, 1986.

[53] Robert Pfeifer, Jutho Haegeman, and Frank Verstraete. Faster identification of optimal contraction sequences for tensor networks. *Physical Review E*, 90(3):033315, 2014.

[54] Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal, and Stefan Leichenauer. Tensornetwork: A library for physics and machine learning. *arXiv preprint arXiv:1905.01330*, 2019.

[55] Neil Robertson and Paul D Seymour. Graph minors. X. Obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52(2):153–190, 1991.

[56] Elina Robeva and Anna Seigal. Duality of graphical models and tensor networks. *arXiv preprint arXiv:1710.01437*, 2017.

[57] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. of PPoPP*, pages 73–82, 2008.

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

[59] Marko Samer and Stefan Szeider. Constraint satisfaction with bounded treewidth revisited. *Journal of Computer and System Sciences*, 76(2):103–114, 2010.

[60] Tian Sang, Paul Beame, and Henry Kautz. Heuristics for fast exact model counting. In *Proc. of SAT*, pages 226–240, 2005.

[61] Paul D Seymour and Robin Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.

[62] Ross D Shachter, Stig K Andersen, and Peter Szolovits. Global conditioning for probabilistic inference in belief networks. In *Proc. of UAI*, pages 514–522. Elsevier, 1994.

[63] Blair D Sullivan, Dinesh Weerapurage, and Chris Groër. Parallel algorithms for graph optimization using tree decompositions. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1838–1847. IEEE, 2013.

[64] Hisao Tamaki. Positive-instance driven dynamic programming for treewidth. In *Proc. of ESA*, pages 68:1–68:13, 2017.

[65] Marc Thurley. SharpSAT: counting models with advanced component caching and implicit BCP. In *Proc. of SAT*, pages 424–429, 2006.

[66] Tom C van der Zanden and Hans L Bodlaender. Computing Treewidth on the GPU. In *Proc. of IPEC*, volume 89, pages 29:1–29:13, 2017.

[67] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint*

arXiv:1802.04730, 2018.

[68] Benjamin Villalonga, Sergio Boixo, Bron Nelson, Christopher Henze, Eleanor Rieffel, Rupak Biswas, and Salvatore Mandrà. A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware. *NPJ Quantum Information*, 5:1–16, 2019.

[69] XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla. Accessed: 2020-02-20.

[70] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Evaluating component solver contributions to portfolio-based algorithm selectors. In *Proc. of SAT*, pages 228–241. Springer, 2012.

[71] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

[72] Lexing Ying. Tensor network skeletonization. *Multiscale Modeling & Simulation*, 15(4):1423–1447, 2017.

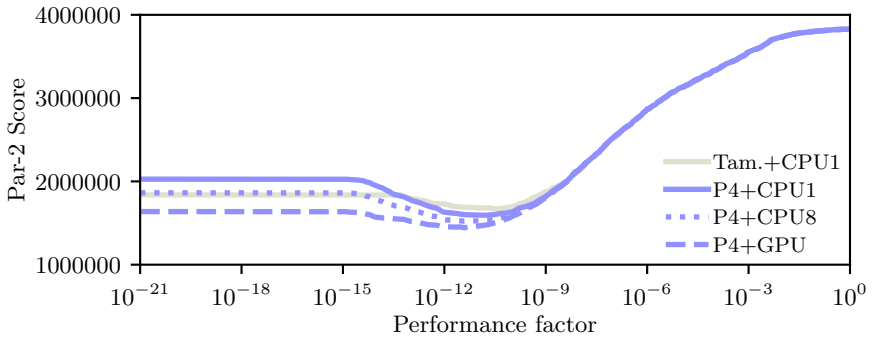

Figure 4: A graph of the simulated PAR-2 score for various combinations of planners and hardware as the performance factor varies.

A Additional Experimental Results

Figure 4 indicates how varying the performance factor affects the simulated PAR-2 score for various combinations of planners and hardware.

B A Proof of Theorem 4.1

In this section, we present a complete proof of Theorem 4.1. Note that the proof differs from the proof of Theorem 3.3 in [19] only in Part 1 and Part 4 (and in the definition of ρ in Part 3).

Proof. The proof proceeds in five steps: (1) compute the factored tensor network M, (2) construct a graph H that is a simplified version of the structure graph of M, (3) construct a carving decomposition S of H, (4) bound the width of S, and (5) use S to find a contraction tree for M. Working with H instead of directly working with the structure graph of M allows us to cleanly handle tensor networks with free indices.

Part 1: Factoring the network. Next, for each $v \in \mathcal{V}(G)$, define T_v to be the smallest connected component of T containing $\delta_G(v) \subseteq \mathcal{L}(T)$. Consider each $A \in N$. If $\mathcal{F}(A) = \emptyset$, let $N_A = \{N_A\}$. Otherwise, observe that T_A is a dimension tree of A and so we can factor A with T_A using Definition 3.3 to get a tensor network N_A and a bijection $g_A : \mathcal{V}(T_A) \to N_A$. Define $M = \cup_{A \in N} N_A$ and let G' be the structure graph of M with free vertex z'. The remainder of the proof is devoted to bounding the carving width of G'. To do this, it is helpful to define $\rho : \mathcal{V}(T) \to \mathcal{V}(G)$ by $\rho(n) = \{v \in \mathcal{V}(G) : n \in \mathcal{V}(T_v), |\delta_{T_v}(n)| = 3\}$. Note that $|\rho(n)| \leq w$ for all $n \in \mathcal{V}(T)$.

Part 2: Constructing a simplified structure graph of M. In order to easily characterize G', we define a new, closely-related graph H by taking a copy of T_v for each $v \in \mathcal{V}(G)$ and connecting these copies where indicated by g. Formally, the vertices of H are $\{(v, n) : v \in \mathcal{V}(G), n \in \mathcal{V}(T_v)\}$. For every $v \in \mathcal{V}(G)$ and every arc in T with endpoints $n, m \in \mathcal{V}(T_v)$, we add an edge between (v, n) and (v, m). Moreover, for each $e \in \mathcal{E}(G)$ incident to $v, w \in \mathcal{V}(G)$, we add an edge between $(v, g(e))$ and $(w, g(e))$.

We will prove in Part 5 that the carving width of G' is bounded from above by the carving width of H. We therefore focus in Part 3 and Part 4 on bounding the carving width of H. It is

helpful for this to define the two projections $\pi_G : \mathcal{V}(H) \to \mathcal{V}(G)$ and $\pi_T : \mathcal{V}(H) \to \mathcal{V}(T)$ that indicate respectively the first or second component of a vertex of H.

Part 3. Constructing a carving decomposition S of H. The idea of the construction is, for each $n \in \mathcal{V}(T)$, to attach the elements of $\pi_T^{-1}(n)$ as leaves along the arcs incident to n. To do this, for every leaf node $\ell \in \mathcal{L}(T)$ with incident arc $a \in \delta_T(\ell)$ define $H_{\ell,a} = \pi_T^{-1}(\ell)$. For every non-leaf node $n \in \mathcal{V}(T) \setminus \mathcal{L}(T)$ partition $\pi_T^{-1}(n)$ into three sets $\{H_{n,a} : a \in \delta_T(n)\}$, ensuring that the degree 3 vertices are divided evenly (the degree 1 and 2 vertices can be placed arbitrarily). Observe that $\{H_{n,a} : n \in \mathcal{V}(T), a \in \delta_T(n)\}$ is a partition of $\mathcal{V}(H)$, and there are at most $\lceil |\rho(n)|/3 \rceil$ vertices of degree 3 in each $H_{n,a}$, since there are exactly $|\rho(n)|$ vertices of degree 3 in $\pi_T^{-1}(n)$.

We use this to construct a carving decomposition S from T by adding each element of $H_{n,a}$ as a leaf along the arc a. Formally, let x_v denote a fresh vertex for each $v \in \mathcal{V}(H)$, let y_n denote a fresh vertex for each $n \in \mathcal{V}(T)$, and let $z_{n,a}$ denote a fresh vertex for each $n \in \mathcal{V}(T)$ and $a \in \delta_T(n)$. Define $\mathcal{V}(S)$ to be the union of $\mathcal{V}(H)$ with the set of these free vertices.

We add an arc between v and x_v for every $v \in \mathcal{V}(H)$. Moreover, for every $a \in \mathcal{E}(T)$ with endpoints $o, p \in \epsilon_T(a)$ add an arc between $y_{o,a}$ and $y_{p,a}$. For every $n \in \mathcal{V}(T)$ and incident arc $a \in \delta_T(n)$, construct an arbitrary sequence $I_{n,a}$ from $\{x_v : v \in H_{n,a}\}$. If $H_{n,a} = \emptyset$ then add an arc between y_n and $z_{n,a}$. Otherwise, add arcs between y_n and the first element of I, between consecutive elements of $I_{n,a}$, and between the last element of $I_{n,a}$ and $z_{n,a}$.

Finally, remove the previous leaves of T from S. The resulting tree S is a carving decomposition of H, since we have added all vertices of H as leaves and removed the previous leaves of T.

Part 4: Computing the width of S. In this part, we separately bound the width of the partition induced by each of the three kinds of arcs in S.

First, consider an arc b between some $v \in \mathcal{V}(H)$ and x_v. Since all vertices of H are degree 3 or smaller, b defines a partition of width at most $3 \leq \lceil 4w/3 \rceil$.

Next, consider an arc c_a between $y_{o,a}$ and $y_{p,a}$ for some arc $a \in \mathcal{E}(T)$ with endpoints $o, p \in \epsilon_T(a)$. Observe that removing a from T defines a partition $\{B_o, B_p\}$ of $\mathcal{V}(T)$, denoted so that $o \in B_o$ and $p \in B_p$.

Then removing c_a from S defines the partition $\{\pi_T^{-1}(B_o), \pi_T^{-1}(B_p)\}$ of $\mathcal{L}(S)$. By construction of H, all edges between $\pi_T^{-1}(B_o)$ and $\pi_T^{-1}(B_p)$ are between $\pi_T^{-1}(o)$ and $\pi_T^{-1}(p)$. Observe that every edge $e \in \mathcal{E}(H)$ between $\pi_T^{-1}(o)$ and $\pi_T^{-1}(p)$ corresponds under g_v to a in T_v for some v. It follows that the number of edges between $\pi_T^{-1}(o)$ and $\pi_T^{-1}(o)$ is exactly the number of vertices in G that are endpoints of edges in both C_a and $\mathcal{E}(G) \setminus C_a$, which is bounded by w. Thus the partition defined by c_a has width no larger than w.

Finally, consider an arc d added as one of the sequence of $|H_{n,a}| + 1$ arcs between y_n, $I_{n,a}$, and $z_{n,a}$ for some $n \in \mathcal{V}(T)$ and $a \in \delta_T(n)$. Some elements of $H_{n,a}$ have changed blocks from the partition defined by c_a. Each vertex of degree 2 that changes blocks does not affect the width of the partition, but each vertex of degree 3 that changes blocks increases the width by 1. There are at most $\lceil w/3 \rceil$ elements of degree 3 added as leaves between y_n and $z_{n,a}$. Thus the partition defined by d has width at most $w + \lceil w/3 \rceil = \lceil 4w/3 \rceil$.

It follows that the width of S is at most $\lceil 4w/3 \rceil$.

Part 5: Bounding the max-rank of M. Let z be the free vertex of the structure graph of N. We first construct a new graph H' from H by, if $\mathcal{F}(N) \neq \emptyset$, contracting all vertices in $\pi_G^{-1}(z)$ to a single vertex z. If $\mathcal{F}(N) = \emptyset$, instead add z as a fresh degree 0 vertex to H'. Moreover, for all $A \in N$ with $\mathcal{I}(A) = \emptyset$ add A as a degree 0 vertex to H'.

Note that adding degree 0 vertices to a graph does not affect the carving width. Moreover, since $|\mathcal{F}(N)| \leq 3$ all vertices (except at most one) of $\pi_G^{-1}(z)$ are degree 2 or smaller. It follows

that contracting $\pi_G^{-1}(z)$ does not increase the carving width. Thus the carving width of H' is at most $\lceil 4w/3 \rceil$.

Moreover, H' and G' are isomorphic. To prove this, define an isomorphism $\phi : \mathcal{V}(H') \to \mathcal{V}(G')$ between H' and G' by, for all $v \in \mathcal{V}(H')$:

$$\phi(v) \equiv \begin{cases} v & \text{if } v \in N \text{ and } \mathcal{I}(v) = \emptyset \\ z' & v = z \\ g_{\pi_G(v)}(\pi_T(v)) & \text{if } v \in \mathcal{V}(H) \text{ and } \pi_G(v) \in N \end{cases}$$

ϕ is indeed an isomorphism between H' and G' because the functions g_A are all isomorphisms and because an edge exists between $\pi_G^{-1}(v)$ and $\pi_G^{-1}(w)$ for $v, w \in \mathcal{V}(G)$ if and only if there is an edge between v and w in G. Thus the carving width of G' is at most $\lceil 4w/3 \rceil$. By Theorem 2 of [19], then, M has a contraction tree of max rank no larger than $\lceil 4w/3 \rceil$. □ □