

# Parallel Weighted Model Counting with Tensor Networks

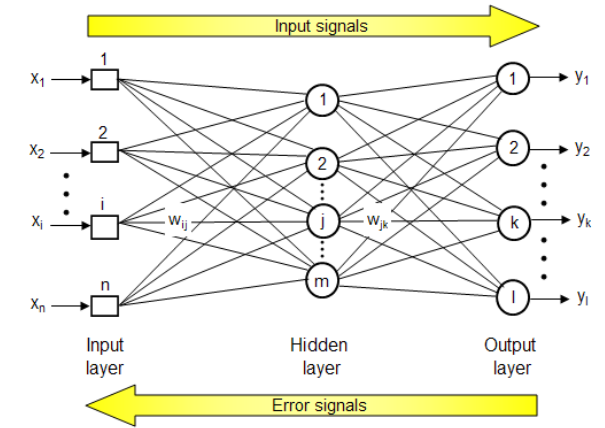
**Jeffrey M. Dudek**, Moshe Y. Vardi

{jmd11, vardi}@rice.edu



# Motivation

Neural networks are a parallelization “success story”



Hundreds of thousands of research hours in optimizing compilers

Libraries (numpy, TensorFlow, pyTorch, ...)

Hardware (CPU, GPU, TPU)


Can we leverage all this work for weighted model counting?

**Yes!** Using tensor network contraction

# Overview

Can we leverage neural network parallelization for weighted model counting?

**Yes!** Using tensor network contraction (TNC)

- [Biamonte et. al., 10] Algorithm for exact, unweighted model counting with TNC
- [**Dudek** et al., 19] Algorithm & tool for exact, literal-weighted model counting with TNC
- [  ] This algorithm can be parallelized on multiple CPUs and GPU

**Key Idea:** Perform model counting via. a sequence of matrix multiplications

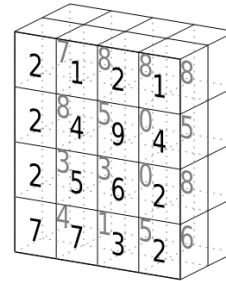
- Matrix multiplications are core operation in neural network algorithms

# Background: Tensors and Tensor Contraction

A *tensor* is a multi-dimensional array, generalizing a matrix.

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 4 \\ 6 & 8 \end{bmatrix}$$



2	1	8	8
2	4	5	0
2	5	6	0
7	7	3	2

*Tensor contraction* is a generalization of matrix multiplication to tensors.

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

$$D_i = \sum_y \sum_x A_{xy} B_{yx}$$

$$E_{xyzw} = A_z B_{xyw}$$

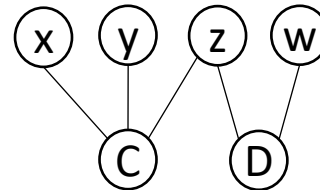
Idea: “Sum out over shared dimensions”

Can be implemented with transpose and matrix multiplication

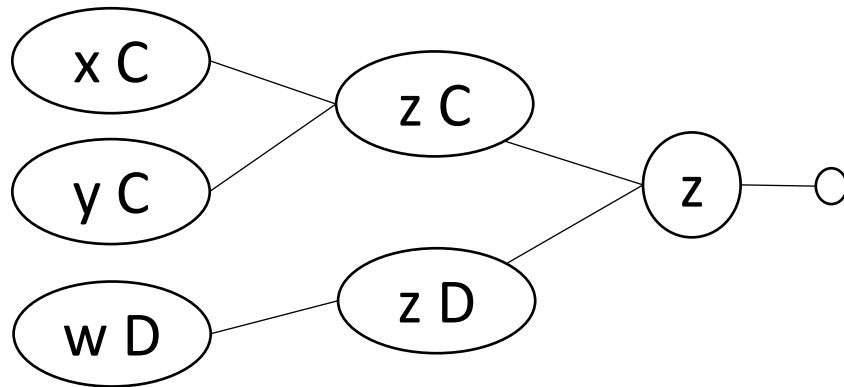
# Tensor Contraction for WMC [Dudek et al., 19]

Input:  $F = (x \vee y \vee \bar{z}) \wedge (z \vee w)$ , Literal-Weight function

1. **Graph:** Construct incidence graph  $G = (V, E)$



2. **Planning:** Find a tree-decomposition of G



## Tree Decomposition of G

Nodes are labelled by variables and clauses

Requirements on labels based on clause structure

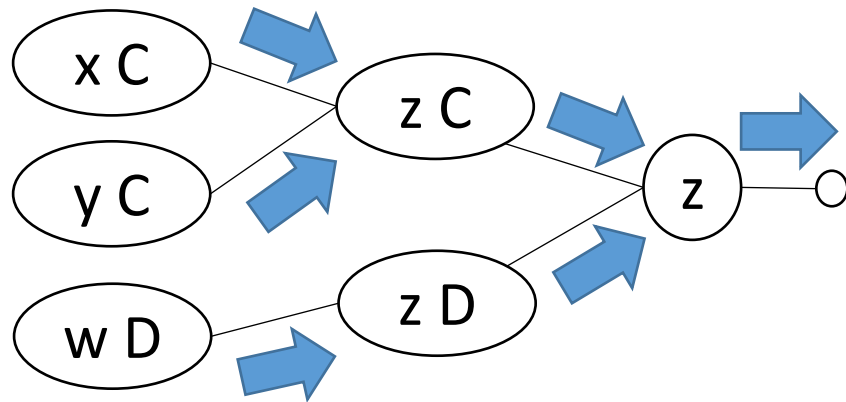
“Width” is maximum size of labels

Use existing tools: Tamaki [Tamaki, 17], FlowCutter [Haman and Strasser, 18], htd [Abseher et al., 17]

- Anytime algorithms: find better and better plans given more time.
- Ultimately spend ~50% of total runtime on planning.

# Tensor Contraction for WMC [Dudek et al., 19]

## 3. Execution: Process tree-decomposition from leaves to root to compute WMC



Pass *tensors* from children to parent

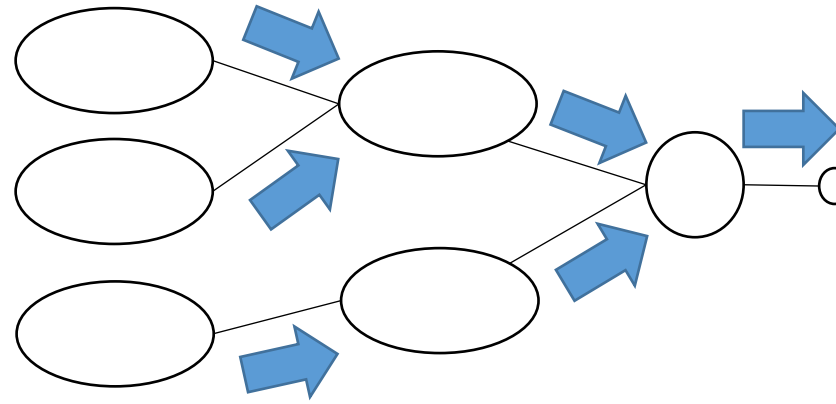
- **Theorem:** Treewidth  $k \implies$  tensors have  $\leq \left\lceil \frac{4}{3}(k + 1) \right\rceil$  dimensions

Parent processes results from children with a single *tensor contraction*

Can be viewed as preprocessing based on treewidth of incidence graph

- Generalizing [Samer and Szeider, 06] on SAT preprocessing to limit variable appearances.

# Related Work



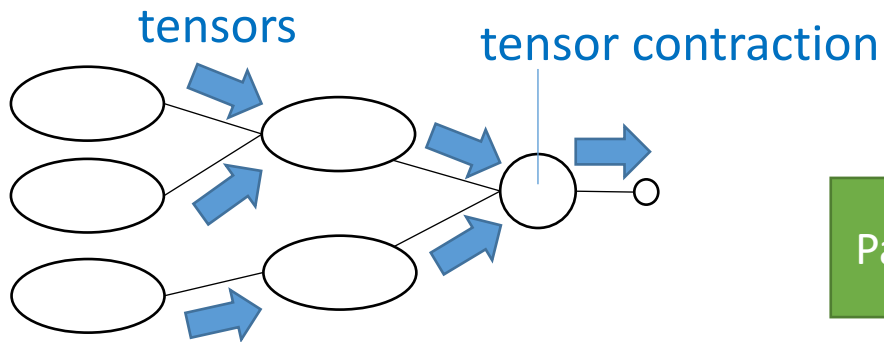
Model counting with tree decompositions [Samer and Szeider, 07]

	What is passed from child to parent?	What does each internal node do?
TensorOrder [Dudek et al., 19]	Tensor	Tensor Contraction
gpuSAT [Fichte et al., 18]	Array	Custom GPU Kernel
gpuSAT2 [Fichte et al., 19]	Binary Search Tree	Custom GPU Kernel
ADDMC [Dudek et al., 20]	Algebraic Decision Diagram (ADD)	ADD Multiplication

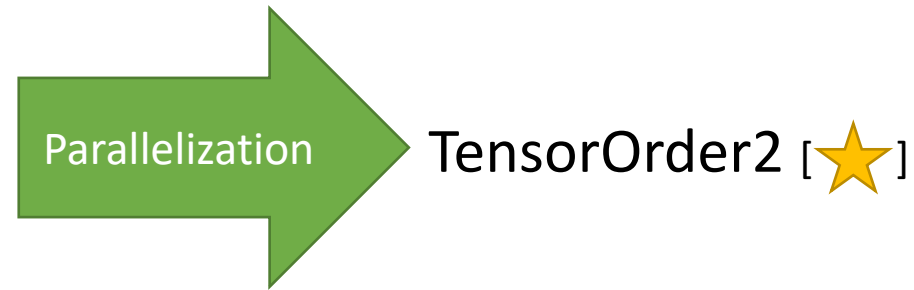
Single CPU

GPU

# Parallelizing Tensor Contraction for WMC



TensorOrder [Dudek et al., 19]



## Key Insight:

- On large benchmarks, runtime is dominated by just a few difficult tensor contractions
- Tensor contractions can each be individually parallelized using existing libraries (TensorFlow)



# Challenge 1: High-Memory Plans

On-board GPU memory is limited

- Moving data in and out of on-board memory is expensive

**Solution: Conditioning** [Darwiche 01, Dechter 99]

$$\#F = \#F[x \mapsto 0] + \#F[x \mapsto 1]$$

Run algorithm separately for  $F[x \mapsto 0]$  and  $F[x \mapsto 1]$

- Planning: Can use the same tree-decomposition for  $\#F[x \mapsto 0]$  and  $\#F[x \mapsto 1]$
- Execution: Doubles execution time, but reduces the size of the largest tensor (by choosing  $x$ )

Greedily condition until all tensors fit entirely in on-board memory

# Challenge 2: GPU Overhead

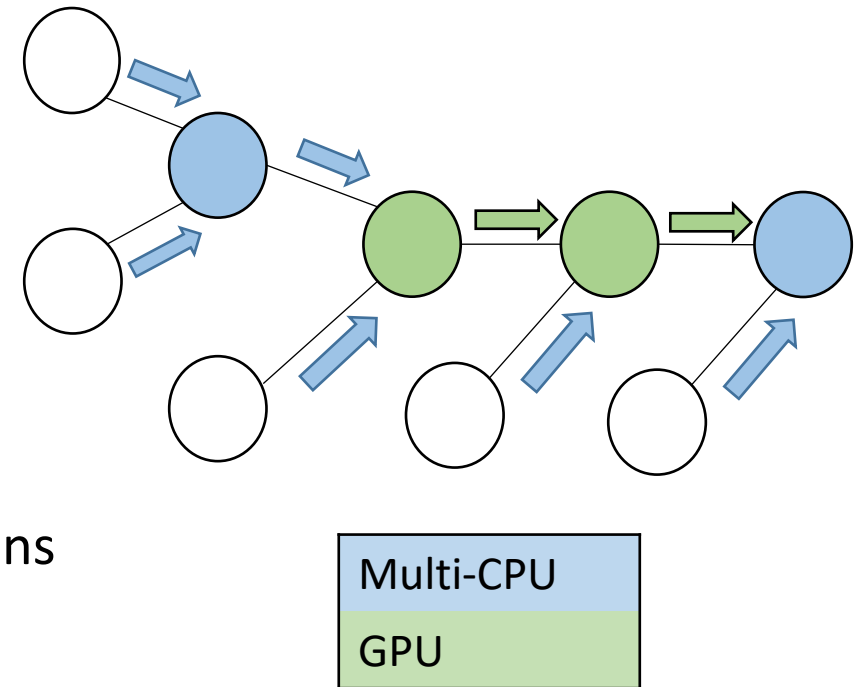
Overhead 1: Initialization of GPU (~1 second on startup)

Overhead 2: Per tensor contraction call

- Small tensor contractions are slower on the GPU than CPU

**Solution:** Hybrid Approach

- Use GPU for large contractions and CPU for small contractions
- Flexibility earned by using black-box matrix multiplication libraries



# Challenge 3: Runtime of Planning Stage

We observe ~50% of total time is spent in the planning stage

- No prior work attempts to parallelize planning
- Little work on parallel tree-decomposition tools

**Solution:** Algorithm Portfolio [Xu et al., 08]

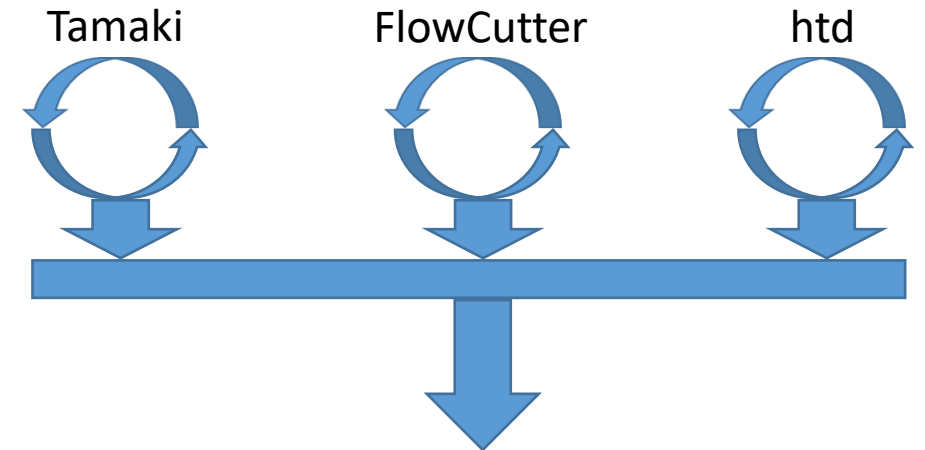
Developed a portfolio of tree-decomposition solvers

- Run multiple tree-decomposition tools in parallel
- Return best tree-decompositions

Experimental results:

- Portfolio can nearly match VBS of all tree-decomposition tools

Applications to other treewidth-based algorithms



# Experimental Evaluation

**TensorOrder2:** Parallel Weighted Model Counter

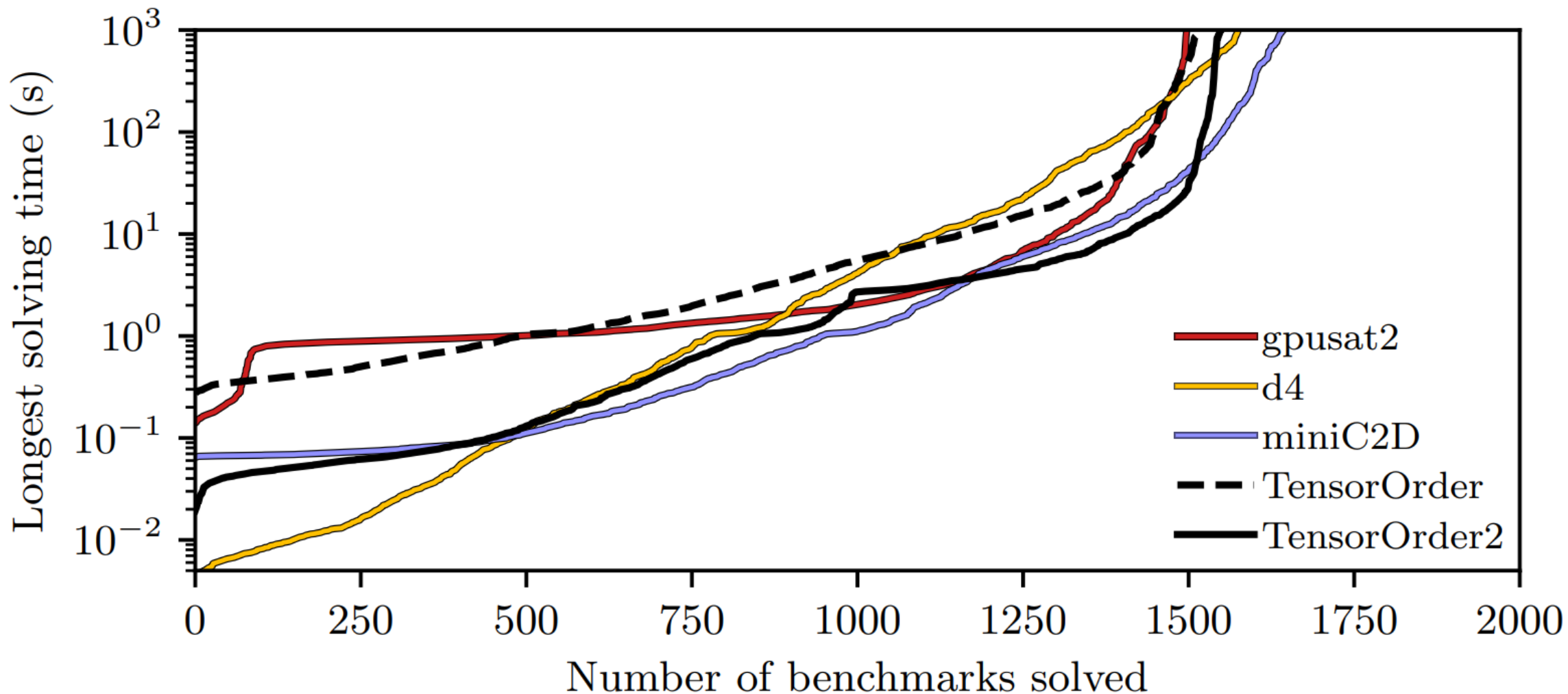
<https://github.com/vardigroup/TensorOrder>

1914 weighted, exact model counting benchmarks

- 1091 from Bayesian inference [Sang et al. 05] and 823 from various domains

Experiments run on Google Cloud

- Docker images
- 8x 2.3GHz cores, 32GB RAM
- NVIDIA Tesla V100 GPU



TensorOrder2 (GPU + portfolio planner) is 2<sup>nd</sup> best solver by PAR-2 score.  
**Overall, TensorOrder2 is the fastest solver on 200 benchmarks.**

\* With pmc-eq preprocessing used for all counters (time included in graph) [Lagniez and Marquis, 14]

# Overview and Conclusion

**Key Idea:** Tensor networks represent WMC as a sequence of matrix multiplications

## Contributions:

- Parallelized WMC using high-level APIs to run execution stage on CPU and GPU
- Parallelized planning stage with a portfolio of tree-decomposition solvers

TensorOrder2 is fastest solver on 200 benchmarks (out of 1914)

<https://github.com/vardigroup/TensorOrder>

## Future Work:

- Better leverage parallel hardware
- Improve planning portfolio