Sparse Matrix Belief Propagation

Reid M. Bixler

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science & Application

April 9, 2018
Blacksburg, Virginia

Keywords: Belief Propagation, Inference, GPU, Sparse Matrix

Sparse Matrix Belief Propagation

Reid M. Bixler

(ABSTRACT)

We propose sparse-matrix belief propagation, which executes loopy belief propagation in Markov random fields by replacing indexing over graph neighborhoods with sparse-matrix operations. This abstraction allows for seamless integration with optimized sparse linear algebra libraries, including those that perform matrix and tensor operations on modern hardware such as graphical processing units (GPUs). The sparse-matrix abstraction allows the implementation of belief propagation in a high-level language (e.g., Python) that is also able to leverage the power of GPU parallelization. We demonstrate sparse-matrix belief propagation by implementing it in a modern deep learning framework (PyTorch), measuring the resulting massive improvement in running time, and facilitating future integration into deep learning models.

Sparse Matrix Belief Propagation

Reid M. Bixler

(ABSTRACT - GENERAL)

We propose sparse-matrix belief propagation, a modified form of loopy belief propagation that encodes the structure of a graph with sparse matrices. Our modifications replace a potentially complicated design of indexing over graph neighborhoods with more optimized and easily interpretable sparse-matrix operations. These operations, available in sparse linear algebra libraries, can also be performed on modern hardware such as graphical processing units (GPUs). By abstracting away the original index-based design with sparse-matrices it is possible to implement belief propagation in a high-level language such as Python that can also use the power of GPU parallelization, rather than rely on abstruse low-level language implementations. We show that sparse-matrix belief propagation, when implemented in a modern deep learning framework (PyTorch), results in massive improvements irunning time when compared against the original index-based version. Additionally this implementation facilitates future integration into deep learning models for wider adoption and use by data scientists.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Belief propagation is a canonical inference algorithm for graphical models such as Markov random fields (MRFs) or Bayesian networks (Pearl, 2014; Wainwright et al., 2008). In graphs with cycles, loopy belief propagation performs approximate inference. Loopy belief propagation passes messages from the variable nodes to their neighbors along the graph structure. These messages are fused to estimate marginal probabilities, also referred to as beliefs. After enough iterations of the algorithm, these beliefs tend to represent a good approximate solution to the actual marginal probabilities. In this paper, we consider pairwise MRFs, which only have unary and pairwise factors.

Pairwise loopy belief propagation is useful for a number of applications: image processing and segmentation (Felzenszwalb and Huttenlocher, 2006), stereo matching (Sun et al., 2003), image interpolation and extrapolation of detail (Freeman et al., 2002), motion analysis (Poppe, 2007), and more (Yuan, 2004). In general, it can be seen that belief propagation is used in a variety of low-level computer vision problems. That aside, belief propagation is also used in modeling social networks (Zhang et al., 2007) and similar style problems with large graph-based networks (Pearl, 1986), while these will often have structural differences in the graphs or networks compared to computer vision problems.

One drawback of loopy belief propagation is that, though the algorithm is relatively simple, its implementation requires management of often irregular graph structures. This fact usually results in tedious indexing in software. The algorithm's message-passing routines can be compiled to be rather efficient, but when implemented in a high-level language, such as those used by data scientists, they can be prohibitively slow. Experts typically resort to writing external software in lower-level, compiled languages such as C++. The implementation of belief propagation (and its variants) as separate, compiled libraries creates a barrier for its integration into high-level data science workflows.

We instead derive loopy belief propagation as a sequence of matrix operations, resulting in sparse-matrix belief propagation. In particular, we use sparse-matrix products to represent the message-passing indexing. The resulting algorithm can then be implemented in a high-level language, and it can be executed using highly optimized sparse and dense matrix operations. Since matrix operations are much more general than loopy belief propagation, they are often built in as primitives in high-level mathematical languages. Moreover, their generality provides access to interfaces that implement matrix operations on modern hardware, such as graphical processing units (GPUs).

In this thesis, we describe sparse-matrix belief propagation and analyze its running time, showing that its running time is asymptotically equivalent to loopy belief propagation. We also describe how the abstraction can be used to implement other variations of belief propagation. We then demonstrate its performance on a variety of tests. We compare loopy belief propagation implemented in Python and in C++ against sparse-matrix belief propagation using scipy.sparse and PyTorch on CPUs and PyTorch on GPUs. The results illustrate the advantages of the sparse-matrix abstraction, and represent a first step toward full integration of belief propagation into modern machine learning and deep learning workflows.

1.1 Literature Review

Belief propagation is one of the canonical variational inference methods for probabilistic graphical models (Pearl, 2014; Wainwright et al., 2008). Loopy belief propagation is naturally amenable to fine-grained parallelism, as it involves sending messages across edges in a graph in parallel. The algorithm is classical and well studied, but because it involves tight loops and intricate indexing, it cannot be efficiently implemented in high-level or mathematical programming languages. Instead, practitioners rely on implementations in low-level languages such as C++ (Schmidt, 2007; Andres et al., 2012). Specific variations for parallel computing have been proposed (Schwing et al., 2011), and other variations have been implemented in graph-based parallel-computing frameworks (Low et al., 2014; Malewicz et al., 2010). Specialized implementations have also been created for belief propagation on GPUs (Zheng et al., 2012).

There are many works on probabilistic graphical models (Koller and Friedman, 2009). From the basic representations such as the Bayesian Network representation and undirected graphical models, there also exist more complicated models such as Gaussian network models and temporal models. Within each set of models exists a number of methods for inference: variable elimination, clique trees, propagation-based approximation, particle-based approximate inference, MAP inference, and more. Similarly, with inference there is also learning: structured learning, learning on undirected models, parameter estimation, and more.

Put simply, belief propagation is a method by which an inference problem can be solved. In our case, we focus on pairwise markov random fields which provide a good groundwork for computer-vision problems (Yedidia et al., 2003). Computer-vision problems require trying to learn the relation of pixels of an image to another, which belief propagation is perfect for. In these cases, we wish to infer the expected pixels by the data given. It is fairly straightforward to convert between similar graph structures (pairwise MRFs and Bayesian networks) into equivalent factor graphs, and the belief propagation algorithms for each are all mathematically analagous.

One of the often used belief propagation inference algorithms is loopy belief propagation. At the core, loopy belief propagation approximates the posterior marginals of each node within the Markov networks, which is known to be an NP-hard problem (Murphy et al., 1999). The basic loopy belief propagation algorithm is the well-known Pearl polytree algorithm on a Bayesian network with loops (Pearl, 2014). This algorithm iteratively updates the beliefs within the network toward the correct marginals when the graph is tree-structured, and is otherwise an approximation.

While loopy belief propagation is the canonical message-passing inference algorithm, many variations have been created to address some of its shortcomings. Some variations modify the inference objective to make belief propagation a convex optimization, such as tree-reweighted belief propagation (Wainwright et al., 2003) and convexified belief propagation (Meshi et al., 2009). Other variations compute the most likely variable state rather than marginal probabilities, such as max-product belief propagation (Wainwright et al., 2008) and max-product linear programming (Globerson and Jaakkola, 2008).

Linearized belief propagation approximates the message update formulas with linear operations (Gatterbauer et al., 2015), which, like our approach, can benefit from highly optimized linear algebra libraries and specialized hardware. However, our approach aims to retain the exact non-linear formulas of belief propagation (and variants), while linearized belief propagation is an approximation.

Belief propagation, while initially only intended for tree-like structures, eventually was used on loop-based graphs and structures. By generalizing belief propagation, it is possible to obtain more precise and accurate results than that of normal belief propagation (Yedidia et al., 2001). In these cases, the belief propagation must optimize for a single critical point when approximating free energy, in this case it is the Bethe free energy.

Our approach of using sparse matrices as an abstraction layer for implementing belief propagation relies on sparse matrix operations being implemented in efficient, optimized, compiled libraries. Special algorithms have been developed to parallelize these operations on GPUs, enabling sparse-matrix computations to use the thousands of cores typically available in such hardware (Bell and Garland, 2008). Other libraries for sparse-matrix computation, such as those built into MATLAB (Gilbert et al., 1992) and `scipy.sparse` often seamlessly provide multi-core parallelism. Frameworks for large-scale, distributed matrix computation, such as Apache Spark (Bosagh Zadeh et al., 2016), can also be used as backends for our approach. Finally, one of the more important recent advances in computing hardware, field-programmable gate arrays (FPGAs), also support sparse-matrix operations (Zhuo and Prasanna, 2005).

Similarly, there do exist some belief propagation algorithms that speedup the running time of belief propagation by modifying the algorithm for a subset of problems (McAuley and Caetano, 2010), (Huang and Jebara, 2011). For example, the class of problems related to variants of maximum weight matching have a set of efficient belief propagation algorithms (Bayati et al., 2006), (Huang and Jebara, 2007). Similarly, a number of low-level machine vision problems related to stereo, optical flow and image restoration have efficient alternatives (Felzenszwalb and Huttenlocher, 2006). Some efficient optimizations rely on the structure of the model, such as if it is conditionally factorizable or has higher order potentials (McAuley and Caetano, 2011; **?**). As opposed to our approach, these efficient optimizations rely inherently on the assumptions from their problem domain and cannot be applied to just any belief propagation problems. Though these efficient optimizations could prove useful paired with our research, these were not within our problem domain. However by combining it with our abstraction, we could perhaps produce even better speedups on some problems than we already have achieved.

By formulating belief propagation as a series of matrix and tensor operations, we make it fit into modern deep learning software frameworks, such as PyTorch (Paszke et al., 2017) and TensorFlow (Abadi et al., 2016). Since these frameworks are designed to easily switch between CPU and GPU computation, we use PyTorch for one of our belief propagation implementations in our experiments. The added advantage is that, once these frameworks fully support back-propagation through sparse matrix products, we will be able to immediately back-propagate through belief propagation. Back-propagating through inference has been shown to allow more robust training of graphical model parameters (Domke, 2013). There do already exist some approaches for combining the benefits of MRFs and belief propagation into deep learning such as by formulating MAP inference for conditional random fields (CRFs) as a recurrent neural network (RNN) (Zheng et al., 2015).

Chapter 2

Background

In this section, we review belief propagation and some common—but not often documented—simplifications. Define a Markov random field (MRF) as a factorized probability distribution such that the probability of variable $x \in \mathbb{X}$ is

$$\Pr(X = \boldsymbol{x}) = \frac{1}{Z} \exp\left(\sum_{i \in V} \phi_i(x_i) + \sum_{(i,j) \in E} \phi(x_i, x_j) \right), \tag{2.1}$$

where the normalizing constant Z is

$$Z = \sum_{X \in \mathbb{X}} \exp\left(\sum_{i \in V} \phi_i(x_i) + \sum_{(i,j) \in E} \phi(x_i, x_j) \right), \tag{2.2}$$

and $\boldsymbol{x}$ represents the full state vector of all variables, x_i represents the state of the ith variable, and $G = \{V, E\}$ is a graph defining the structure of the MRF.

The goal of marginal inference is to compute or approximate the marginal probabilities of the variables

$$\{\Pr(x_1), \ldots, \Pr(x_n)\} \tag{2.3}$$

and of other subsets of variables. In pairwise MRFs, the marginal inference is often limited to the unary marginals and the pairwise marginals along the edges.

2.1 Loopy Belief Propagation

Belief propagation is a dynamic programming algorithm for computing marginal inference in tree-structured MRFs that is often applied in practice on non-tree, i.e., loopy, graphs. Loopy belief propagation is therefore a heuristic approximation that works well in many practical scenarios. The algorithm operates by passing messages among variables along the edges of the MRF structure.

The message sent from variable x_i to variable x_j is

$$m_{i \to j}[x_j] = \log \left(\sum_{x_i} \exp \left(a_{x_i} \right) \right), \tag{2.4}$$

where (as shorthand to fit the page)

$$a_{x_i} = \phi_{ij}(x_i, x_j) + \phi_i(x_i) + \sum_{k \in N_i \setminus j} m_{k \to i}[x_i] - d_{ij}; \tag{2.5}$$

N_i is the set of neighbors of variable i, i.e., $N_i = \{k | (i,k) \in E\}$; and d_{ij} is any constant value that is eventually cancelled out by normalization (see Eq. (2.7)). The message itself is a function of the receiving variable's state, which in a discrete setting can be represented by a vector (hence the bracket notation $m_{i \to j}[x_j]$ for indexing by the state of the variable).

The estimated unary marginals, i.e., the *beliefs*, are computed by fusing the incoming messages with the potential function and normalizing:

$$\begin{aligned} \Pr(x_j) \approx \exp(b_j[x_j]) = \\ \exp \left(\phi_j(x_j) + \sum_{i \in N_j} m_{i \to j}[x_j] - z_j \right), \end{aligned} \tag{2.6}$$

where z is a normalizing constant

$$z_j = \log \left(\sum_{x_j} \exp \left(\phi_j(x_j) + \sum_{i \in N_j} m_{i \to j}[x_j] \right) \right). \tag{2.7}$$

For convenience and numerical stability, we will consider the log-beliefs

$$b_j[x_j] = \phi_j(x_j) + \sum_{i \in N_j} m_{i \to j}[x_j] - z_j. \tag{2.8}$$

We again use bracket notation for indexing into the beliefs because, for discrete variables, the beliefs can most naturally be stored in a lookup table, which can be viewed as a vector.

2.2 Simplifications

The message update in Eq. (2.4) contains an expression that is nearly identical to the belief definition in Eq. (2.8). In the message update, the exponent is

$$a_{x_i} = \phi_{ij}(x_i, x_j) + \phi_i(x_i) + \sum_{k \in N_i \setminus j} m_{k \to i}[x_i] - d_{ij}. \tag{2.9}$$

The only difference between this expression and the belief update is that the belief uses the constant z_j to ensure normalization and the message update omits the message from the receiving variable (j). For finite message values, we can use the equivalence

$$\begin{aligned} \phi_i(x_i) + \sum_{k \in N_i \setminus j} m_{k \to i}[x_i] - d_{ij} = \\ \phi_i(x_i) + \sum_{k \in N_i} m_{j \to i}[x_i] - m_{j \to i}[x_i] - d_{ij} = \\ b_i[x_i] - m_{j \to i}[x_i], \end{aligned} \tag{2.10}$$

where the last equality sets $d_{ij} = z_i$. The message and belief updates can then be written as

$$\begin{aligned} b_j[x_j] &= \phi_j(x_j) + \sum_{i \in N_j} m_{i \to j}[x_j] - z_j, \\ m_{i \to j}[x_j] &= \log \left(\sum_{x_i} \exp\left(\phi_{ij}(x_i, x_j) + b_i[x_i] - m_{j \to i}[x_i]\right) \right). \end{aligned} \tag{2.11}$$

These two update equations repeat for all variables and edges, and when implemented in low-level languages optimized by pipelining compilers, yield the fastest computer programs that can run belief propagation. However, the indexing requires reasoning about the graph structure, making any code that implements such updates cumbersome to maintain, prone to errors, and difficult to adapt to new computing environments such as parallel computing settings.

Chapter 3

Sparse-Matrix Belief Propagation

In this section, we describe sparse-matrix belief propagation and analyze its complexity. Instead of directly implementing the updates, sparse-matrix belief propagation uses an abstraction layer of matrices and tensors. This abstraction allows belief propagation to be written in high-level languages such as Python and MATLAB, delegating the majority of computation into highly optimized linear algebra libraries.

3.1 Tensor Representations of the MRF, Beliefs, and Messages

We store a c by n belief matrix $\mathbf{B}$, where n is the number of variables and c is the maximum number of states of any variable. For simplicity, assume all variables have the cardinality c. This belief matrix is simply the stacked belief vectors, such that $B_{ij} = b_j[i]$. Similarly, we rearrange this matrix into an analogously shaped and ordered unary potential function matrix $\mathbf{\Phi}$, where $\Phi_{ij} = \phi_j(x_j = i)$.

We store the pairwise potentials in a three-dimensional tensor $\mathbf{\Gamma}$ of size $c \times c \times |E|$. Each kth slice of the tensor is a matrix representing the kth edge potential function as a table, i.e., $\mathbf{\Gamma}_{ijk} = \phi_{s_k t_k}(i, j)$.

Let the edges be defined redundantly such that each edge appears in forward and reverse order, i.e., $(i, j) \in E$ if and only if $(j, i) \in E$. Let the edges also be indexed in an arbitrary but fixed order

$$E = \{(s_1, t_1), (s_2, t_2), \dots, (s_{|E|}, t_{|E|})\}, \tag{3.1}$$

and define vectors $\boldsymbol{s} = [s_1, \dots, s_{|E|}]^\top$ and $\boldsymbol{t} = [t_1, \dots, t_{|E|}]^\top$. These variables encode a fixed ordering for the messages. The particular order does not matter, but to convert message passing into matrix operations, we need the order to be fixed.

In addition to the fixed order, we also define a *message reversal* order vector $\boldsymbol{r}$ where

$$E[r_i] = (j, i) \text{ if } E[i] = (i, j). \tag{3.2}$$

We can represent this reversal vector as a sparse, $|E|\times|E|$ permutation matrix $\mathbf{R}$ where $R_{ij} = 1$ iff $r_i = j$.

Define a c by $|E|$ message matrix $\mathbf{M}$ whose ith column is the ith message. In other words, $M_{ij} = m_{s_j \to t_j}[i]$, or equivalently, $\mathbf{M}$ is a horizontal stack of all messages:

$$\mathbf{M} = \left[m_{s_1 \to t_1}, \quad m_{s_2 \to t_2}, \quad \dots \quad , m_{s_{|E|} \to t_{|E|}} \right]. \tag{3.3}$$

Finally, we define a binary *sparse* matrix $\mathbf{T}$ (for *to*) with $|E|$ rows and n columns whose nonzero entries are as follows:

$$T_{ij} = \begin{cases} 1.0 & \text{if } t_i = j \\ 0 & \text{otherwise.} \end{cases} \tag{3.4}$$

This matrix $\mathbf{T}$ maps the ordered messages to the variables that *receive* the messages.

We define an analogous matrix $\mathbf{F}$ (for *from*), also binary and sparse with $|E|$ rows and n columns, whose nonzero entries are as follows:

$$F_{ij} = \begin{cases} 1.0 & \text{if } s_i = j \\ 0 & \text{otherwise.} \end{cases} \tag{3.5}$$

This matrix $\mathbf{F}$ maps the ordered messages to the variables that *send* the messages.

3.1.1 The logsumexp Operation

A common operation that occurs in various steps of computing log-space belief propagation is the $\mathrm{logsumexp}$ operation, which is defined as follows:

$$\mathrm{logsumexp}(\mathbf{A}) = \log\left(\exp\left(\mathbf{A}\right) \cdot \mathbf{1}\right), \tag{3.6}$$

where the $\log$ and $\exp$ operations are performed element-wise, and we use the matrix-vector product with the ones vector ($\mathbf{1}$) as a compact notation for summing across the rows of the exponentiated input matrix.[1] The resulting output is a column vector with the same number of rows as the input matrix $\mathbf{A}$.

3.1.2 Slice Indexing

To describe the message updates in a compact notation, we use slice indexing, which is common in matrix and tensor software because it can be implemented efficiently in linear time and easy to

[1]It is especially useful to form this abstraction because this operation is notoriously unstable in real floating-point arithmetic. Numerically stable implementations that adjust the exponent by subtracting a constant and adding the constant back to the output of the logarithm are possible. These stable implementations add a linear-time overhead to the otherwise linear-time operation, so they maintain the same asymptotic running time of the original $\mathrm{logsumexp}$ operation.

parallelize. We borrow syntax from `numpy` that is also reminiscent of the famous MATLAB syntax, where

$$\mathbf{A}[:, \boldsymbol{i}] = [\mathbf{A}[:, i_1], \mathbf{A}[:, i_2], \ldots] \,. \tag{3.7}$$

This slice indexing allows the reordering, selection, or repetition of the rows or columns of matrices or tensors.

3.2 Belief Propagation as Tensor Operations

Using these constructed matrices, the belief matrix is updated with the operations

$$\begin{aligned} \tilde{\mathbf{B}} &\leftarrow \mathbf{\Phi} + \mathbf{M}^\top \mathbf{T} \\ \mathbf{B} &\leftarrow \tilde{\mathbf{B}} - \mathbf{1} \, \mathrm{logsumexp}\left(\tilde{\mathbf{B}}\right), \end{aligned} \tag{3.8}$$

where the last operation uses the logsumexp operation to compute the normalizing constants of each belief column vector and multiplies by the ones vector ($\mathbf{1}$) to broadcast it across all rows.[2]

The message matrix $\mathbf{M}$ is updated with the following formula:

$$\mathbf{M} \leftarrow \mathrm{logsumexp}(\mathbf{\Gamma} + \mathbf{B}[:, \boldsymbol{s}] - \mathbf{M}[:, \boldsymbol{r}]). \tag{3.9}$$

This expression uses two forms of shorthand that require further explanation. First, the addition of the tensor $\mathbf{\Gamma}$ and the matrix $(\mathbf{B}[:, \boldsymbol{s}] - \mathbf{M}[:, \boldsymbol{r}])$ requires broadcasting. The tensor $\mathbf{\Gamma}$ is of size $c \times c \times |E|$, and the matrix $(\mathbf{B}[:, \boldsymbol{s}] - \mathbf{M}[:, \boldsymbol{r}])$ is of size $c \times |E|$. The broadcasting copies the matrix c times and stacks them as rows to form the same shape as $\mathbf{\Gamma}$. Second, the logsumexp operation sums across the columns of the summed tensor, outputting a tensor of shape $c \times 1 \times |E|$, which is then squeezed into a matrix of size $c \times |E|$.

The message matrix can equivalently be updated with this formula:

$$\mathbf{M} \leftarrow \mathrm{logsumexp}(\mathbf{\Gamma} + \mathbf{B}\mathbf{F}^\top - \mathbf{M}\mathbf{R}). \tag{3.10}$$

Here $\mathbf{B}\mathbf{F}^\top - \mathbf{M}\mathbf{R}$ is once again $c \times |E|$, and it is equivalent to the slice-notation form above.

Belief propagation is then implemented by iteratively running Eq. (3.8) and then either of the equivalent Eqs. (3.9) and (3.10).

3.3 Variations of Belief Propagation

Many variations of belief propagation can similarly be converted into a sparse-matrix format. We describe some of these variations here.

[2] In `numpy`, this broadcasting is automatically inferred from the size of the matrices being subtracted.

3.3.1 Tree-Reweighted Belief Propagation

The tree-reweighted variation of belief propagation (TRBP) computes messages corresponding to a convex combination of spanning trees over the input graph. The result is a procedure that optimizes a convex inference objective (Wainwright et al., 2003; Wainwright et al., 2008). The belief and message updates for TRBP are adjusted according to edge-appearance probabilities in a distribution of spanning trees over the MRF graph. These updates can be implemented in matrix form by using a length $|E|$ vector $\boldsymbol{\rho}$ containing the appearance probabilities ordered according to edge set E. The matrix-form updates for the beliefs and messages are then

$$\begin{aligned} \tilde{\mathbf{B}} &\leftarrow \mathbf{\Phi} + (\boldsymbol{\rho} \circ \mathbf{M})^\top \mathbf{T} \\ \mathbf{B} &\leftarrow \tilde{\mathbf{B}} - \mathbf{1} \operatorname{logsumexp}\left(\tilde{\mathbf{B}}\right), \\ \mathbf{M} &\leftarrow \operatorname{logsumexp}(\mathbf{\Gamma}/\boldsymbol{\rho} + \mathbf{B}\mathbf{F}^\top - \mathbf{M}\mathbf{R}), \end{aligned} \tag{3.11}$$

where element-wise product $\circ$ and element-wise division / are applied with appropriate broadcasting.

3.3.2 Convexified Belief Propagation

Another important variation of loopy belief propagation uses counting numbers to adjust the weighting of terms in the factorized entropy approximation. The resulting message update formulas weight each marginal by these counting numbers. Under certain conditions, such as when all counting numbers are non-negative, the inference objective can be shown to be concave, so this method is often referred to as *convexified Bethe belief propagation* (Meshi et al., 2009). We can exactly mimic the message and belief update formulas for convexified belief propagation by instantiating a vector $\boldsymbol{c}$ containing the counting numbers of each edge factor, resulting in the updates

$$\begin{aligned} \tilde{\mathbf{B}} &\leftarrow \mathbf{\Phi} + \mathbf{M}^\top \mathbf{T} \\ \mathbf{B} &\leftarrow \tilde{\mathbf{B}} - \mathbf{1} \operatorname{logsumexp}\left(\tilde{\mathbf{B}}\right), \\ \mathbf{M} &\leftarrow \operatorname{logsumexp}(\mathbf{\Gamma} + (\mathbf{B}\mathbf{F}^\top - \mathbf{M}\mathbf{R})/\boldsymbol{c}) \circ \boldsymbol{c}. \end{aligned} \tag{3.12}$$

The counting numbers for unary factors can be used to compute the inference objective, but they do not appear in the message-passing updates.

3.3.3 Max-Product Belief Propagation

Finally, we illustrate that sparse tensor operations can be used to conduct approximate *maximum a posteriori* (MAP) inference. The max-product belief propagation algorithm (Wainwright et al., 2008) is one method for approximating MAP inference, and it can be implemented with the following updates:

$$\begin{aligned} \mathbf{B} &\leftarrow \mathrm{onehotmax}(\boldsymbol{\Phi} + \mathbf{M}^\top \mathbf{T}) \\ \mathbf{M} &\leftarrow \mathrm{logsumexp}(\boldsymbol{\Gamma} + \mathbf{B}\mathbf{F}^\top - \mathbf{M}\mathbf{R}), \end{aligned} \tag{3.13}$$

where $\mathrm{onehotmax}$ is a function that returns an indicator vector with 1 for entries that are the maximum of each column and zeros everywhere else, e.g., the "one-hot" encoding. Similar conversions are also possible for variations of max-product, such as max-product linear programming (Globerson and Jaakkola, 2008).

3.4 Time-Complexity Analysis

To analyze our sparse-matrix formulation of loopy belief propagation, and to show that it requires an asymptotically equivalent running time to normal loopy belief propagation, we first revisit the sizes of all matrices involved in the update equations. The first step is the belief update operation, Eq. (3.8), which updates the c by n belief matrix $\mathbf{B}$. The potential matrix $\boldsymbol{\Phi}$ is also c by n; the message matrix $\mathbf{M}^\top$ is c by $|E|$; and the *sparse* message-recipient indexing matrix $\mathbf{T}$ is $|E|$ by n.

The second step in Eq. (3.8) normalizes the beliefs. It subtracts from $\mathbf{B}$ the product of $\mathbf{1}$, which is a c by 1 vector, and $\mathrm{logsumexp}\left(\tilde{\mathbf{B}}\right)$, which is 1 by n. Explicitly writing the numerically stable $\mathrm{logsumexp}$ operation, the right side of this line can be expanded to

$$\tilde{\mathbf{B}} - \mathbf{1}\log\left(\mathrm{sum}\left(\exp\left(\mathbf{B} - \max\left(\mathbf{B}\right)\right)\right) + \max\left(\mathbf{B}\right)\right). \tag{3.14}$$

We next examine the message update Eq. (3.10), which updates $\mathbf{M}$. The three-dimensional tensor $\boldsymbol{\Gamma}$ is of size $c \times c \times |E|$; the *sparse* message-sender indexing matrix $\mathbf{F}^\top$ is c by $|E|$; and the *sparse* reversal permutation matrix R is $|E| \times |E|$. The message matrix $\mathbf{M}$ is c by $|E|$.

CPU computation These three update steps are the entirety of each belief propagation iteration. From the first line of Eq. (3.8), the main operation to analyze is the dense-sparse matrix multiplication $\mathbf{M}^\top \mathbf{T}$. Considering an $n \times m$ dense matrix A and a sparse matrix B of size $m \times p$ with s nonzero entries (i.e., $\|B\|_0 = s$), the sparse dot product has time complexity $O(ms)$ in sequential computation, as on a CPU. The time complexity of the sparse dot product depends upon the number of rows m and the number of sparse elements in the sparse matrix B. Every other computation in Eq. (3.8) involves element-wise operations on c by n matrices. Thus, Eq. (3.8) requires $O(nc + \|\mathbf{T}\|_0 c)$ time. Since the sparse indexing matrix $\mathbf{T}$ is defined to have a single nonzero entry per column, corresponding to edges, the time complexity of this step is $O(nc + |E|c)$.

In the message-update step, Eq. (3.10), the outer $\mathrm{logsumexp}$ operation and the additions involve element-wise operations over $c \times c \times |E|$ tensors. The matrix multiplications are all dense-sparse dot products, so the total cost of Eq. (3.10) is $O(|E|c^2 + \|\mathbf{F}\|_0 c + \|\mathbf{R}\| c)$ (Gilbert et al., 1992).

Since both indexing matrices **F** and **R** have one entry per edge, the total time complexity of the message update is $O(|E|c^2 + |E|c)$.

The combined computational cost of both steps is $O(nc + |E|c + |E|c^2)$. This iteration cost is the same as traditional belief propagation.

GPU computation Since the matrix operations in our method are simple, they are easily parallelized on GPUs. Ignoring the overhead of transferring data to GPU memory, we focus on the time complexity of the message passing. First consider the dense-sparse matrix multiplication mentioned previously, with a dense n by m matrix A and sparse m by p matrix B with s nonzero entries. GPU algorithms for sparse dot products use all available cores to run threads of matrix operations (Bell and Garland, 2008). In this case, each thread can run the multiplication operation of a single column in the sparse matrix B.

Given k cores/threads, we assume that there will be two cases: (1) when the number of sparse columns m is *less than or equal to* the number of cores k and (2) when the number of sparse columns m is *more than* the number of cores k. For either case, let s_i be the number of nonzero entries in column i. The time complexity of case (1) is $O(\max_i s_i)$, which is the time needed to process whichever column requires the most multiplications. For case (2), the complexity is $O(\lceil \frac{m}{k} \rceil \max_i s_i)$, which is the time for each set of cores to process k columns. In case (2), we are limited by our largest columns, as the rest of our smaller columns will be processed much sooner. Overall, the GPU time complexity of this operation is $O\left(\max\left(\max_i s_i, \lceil \frac{m}{k} \rceil \max_i s_i\right)\right)$. In our sparse indexing matrices, the to and from matrices (**T** and **F** respectively) may contain multiple entries per column so we define σ as the maximum number of entries, where $\sigma = \max_i s_i$. This quantity can be seen as the maximum number of neighbors across the whole graph. The reversal matrix **R** has only one entry per column.

For the element-wise dense matrix operations, which have time complexity $O(nm)$ on the CPU, we can again multi-thread each entry over the number of cores k in our GPU such that the time complexity is $O\left(\lceil \frac{nm}{k} \rceil\right)$.

The running time of the belief propagation steps is then $O\left(\left\lceil \frac{n}{k} \right\rceil c + \left\lceil \frac{|E|}{k} \right\rceil c\sigma + \left\lceil \frac{|E|c^2}{k} \right\rceil\right)$.

Based on our parallelism analysis, we expect significantly faster running times when running on the GPU, especially in cases where we have a large number of cores k. While we expect some time loss due to data-transfer overhead to the GPU, this overhead may be negligible when considering the overall time cost for every iteration of the message-passing matrix operations.

Finally, this analysis also applies to other shared-memory, multi-core, multithreaded environments (e.g., on CPUs), since in both CPU and GPU settings, matrix rows can be independently processed.

Chapter 4

Implementation Specifics

We first implemented the sparse matrix form of loopy belief propagation in a high-level language, in this case, Python due to the vast number of packages offering matrix operations and the wide usability within the machine learning community. We use the sparse matrix library `scipy.sparse` to execute the required sparse matrix functions for our algorithms. Using our previous definitions in Chapter 3, we created the necessary models for mimicking a Markov network, which was then used within a number of belief propagation techniques that took advantage of our sparse matrix form. This MarkovNet class contains all of the variables and functions for storing potential functions and structure specififcally for a pairwise MarkovNet. Note that we made the decision to focus on the pairwise version of Markov networks because we would have to heavily modify our design to be able to use graphs that contain more than unary or pairwise factors.

With the Markov network class defined, we designed and created a number of variations of belief propagation, all of which we have implemented within our Python library. These variations include: normal belief propagation, convexified belief propagation, matrix belief propagation, tree-reweighted belief propagation, and max-product belief propagation. While the normal belief propagation uses native Python loops to compute message passing, the other variations use the spare-matrix format to optimize for sparse and dense matrix operations wherever possible.

In addition to the belief propagators, we also implemented log-linear models for the use of discriminative learning which can convert from log linear features to a pairwise Markov random field as well as an approximate maximum likelihood learner for the purpose of doing generative learning on markov random field paramaters. In general, we intended to have our library have not only forms of inference using our sparse-matrix design but also forms of learning.

To ensure that our designs and implentations produce the correct results as we expect from the original loopy belief propagation (or other variations), we also implemented the original algorithms in our Python library to compare the results against our new designs. From the results of all of our tests (see Section 5), we are able to deduce that our new design using the sparse-matrix format produces the same exact results as the previous designs.

4.1 Conversion to PyTorch

While we focus on converting our library to PyTorch for our use case, our design can also be implemented in any other library that offers matrix operations on GPUs (or CPUs). We implement our design within PyTorch to show that bridging the gap between our design and any sparse-matrix package is straightforward and easy to do. As we had a large number of variations of belief propagation, we decided to focus only on converting the matrix belief propagation classes to PyTorch. The conversions between `scipy.sparse` and `pytorch.sparse` were fairly easy to implement, with the only problems arising with language differences between the two packages (e.g., the *cat* function in PyTorch is the same as the *hstack* function in scipy). Overall, it was fairly straightforward, and we were able to produce the same results from our Python implementation of matrix belief propagation as we had from the PyTorch (CPU) implementation.

With the PyTorch implementation completed it was quite simple to port a majority of the matrix calculations over to the GPU by using the CUDA package included within PyTorch. Any instance of a Tensor object within PyTorch needs to be converted to a CUDA Tensor with the *.cuda()* function call, which can easily be called in an if statement when checking for whether CUDA has been installed onto the host's PC. It was quite easy to convert between the CPU and GPU because PyTorch's error statements let the user know about an invalid conversion or operation from a non-CUDA tensor to a CUDA tensor. After modifying for CUDA, we were able to, once again, produce the same results that we were obtaining from both the matrix belief propagation as well as the PyTorch (CPU) matrix belief propagation.

With inference on PyTorch completed, we aimed to also implement learning with back-propagation and automatic differentiation (also available within the PyTorch library). However, we found out that the PyTorch authors have not yet implemented all of the required operations for sparse-matrix backwards propagation. PyTorch is still quite a young machine learning library, having been released October 2016, so there is still much room for improvement and modifications. Looking through the forums and issues on their GitHub repository, we were able to locate Issue #2389, which states that sparse operations for autograd have not yet been implemented but have been placed on medium priority. As such, we have implemented the embeddings for autogradients and backwards propagation over our Markov networks so that they will be usable for learning once the PyTorch community implements sparse matrix operations for autogradients.

PyTorch learning problems aside, we have been able to not only implement our sparse-matrix design in both the high-level language of Python but also we were also able to implement the same design within PyTorch for usage on the GPU via the CUDA library both of which produce the same results of previous designs (also included within our library).

4.2 Tests for Correctness

For evaluation purposes, we go defined a wide variety of tests for both our Python implementation and our PyTorch implementation. Note that in any case of randomization, we always use the same random seed to ensure that our randomly generated models are not the reason that our test cases fail. These tests for our belief propagation are as follows:

Exactness Test This test generates a Markov network represented as a chain model of random factors which are then passed into both our matrix belief propagator and brute force classes. This tests that the belief propagator produces the true marginals in the chain model as expected form our brute force inference. To do so, we run inference on both propagators and then compare the unary marginals, pairwise marginals, and the matrix BP's Bethe energy functional to the brute force log partition function

Consistency Test This test uses a randomized loop model with our matrix belief propagator which tests that the loopy matrix belief propagator infers marginals that are locally consistent. To do so, we compare all of the pairwise beliefs with the unary beliefs between the neighbors of all of the variables.

Normalization Test This test uses a randomized loop model with our matrix belief propagator which tests that the unary and pairwise beliefs properly sum to 1.0. To do so, we simply check that all of the unary beliefs sum up to 1 for the variables and that the pairwise beliefs sum to 1 for the neighbors of those variables.

Speedup Test This test uses a randomized grid model with both of our propagators (matrix BP and normal BP) in order to test that the matrix belief propagator is faster than the native Python loop-based belief propagator. For this test we set the maximum number of possible iterations for both propagators to be 30,000 (although they often finish in no more than 30 iterations for most cases) and time how long inference takes for both. For timing, we use Python's native *time* package. After confirming that matrix BP is faster, we then assert that the unary and pairwise beliefs agree between both models (as we want to ensure they arrived at the same approximation).

Conditioning Test This test uses a randomized loop model on the matrix belief propagator to test that conditioning on variable properly sets variables to conditioned state. To do this, we simply check whether the variable beliefs are not only initially set to the correct state but also that a variable changes beliefs based off of the conditions of another variable.

Overflow Test This test uses a randomized chain model on the matrix belief propagator for the purposes of testing that it does not fail when given very large, poorly scaled factors. This is simply done by setting very large unary factors (i.e. 1000, 2000, 3000, 4000) and then running inference with the intention of having no overflow errors being raised.

Grid Consistency Test This test uses a randomized grid model with our matrix belief propagator to test that it infers consistent marginals on a grid Markov random field. This test is exactly similar to the previous Consistency Test except we substitute the randomized loop model with a randomized grid model.

Belief Propagation Message Test This test uses a randomized grid model with simple edges on both of our propagators (matrix BP and normal BP) for the purposes of testing that matrix belief propagation and loop-based belief propagation calculate the same messages and beliefs at each iteration of inference. To do this, we check every iteration of inference on our model ensuring that the messages, unary beliefs, and pairwise beliefs all agree for each iteration.

Chapter 5

Empirical Evaluation

In this section, we describe our empirical evaluation of sparse-matrix belief propagation. We measure the running time for belief propagation on a variety of MRFs using different software implementations and hardware, including optimized and compiled code for CPU-based computation and sparse-matrix belief propagation in a high-level language for both CPU- and GPU-based computation.

5.1 Experimental Setup

We generate grid MRFs of varying sizes. We randomly generate potential functions for each MRF such that the log potentials are independently normally distributed with variance 1.0. We use MRFs with different variable cardinalities c from the set $\{8, 16, 32, 64\}$. We run experiments with MRFs structured as square, two-dimensional grids, where the number of rows and columns in the grids are $\{8, 16, 32, 64, 128, 256, 512\}$. In other words, the number of variables in models with these grid sizes are, respectively, 64, 256, 1024, 4,096, 16,384, 64,536, and 262,144. We run all implementations of belief propagation until the total absolute change in the messages is less than 10^{-8}.

We run our experiments on different hardware setups. We use two different multi-core CPUs: a 2.4 Ghz Intel i7 with 4 cores and a 4 Ghz Intel i7 with 4 cores.

We also run sparse-matrix belief propagation on various GPUs. We run on an Nvidia GTX 780M (1,536 cores, 4 GB memory), an Nvidia GTX 1080 (2,560 cores, 8 GB), an Nvidia GTX 1080Ti (3,584 cores, 11 GB), and an Nvidia Tesla P40 (3840 cores, 24 GB).

5.2 Variations of Belief Propagation

We compare four different implementations of belief propagation. First, we use the compiled and optimized C++ implementation of belief propagation in the OpenGM library (Andres et al., 2012). This software represents the low-level implementation. Second, we use a direct implementation of simplified belief propagation (see Section 2.2) in Python and `numpy` using Python loops and dictionaries (hash maps) to manage indexing over graph structure. Third, we use an implementation of sparse-matrix belief propagation in Python using `scipy.sparse`. Fourth, we use an implementation of sparse-matrix belief propagation in Python using the deep learning library PyTorch, which enables easily switching between CPU and GPU computation. More in-depth results and running times can be found in Appendix A for more specific comparisons between different implementations and hardwares.

5.3 Results and Discussion

Considering the results for CPU-based belief propagation in Fig. 5.1, the sparse-matrix belief propagation is faster than any other CPU-based belief propagation for all MRF sizes and all variable cardinalities. Similarly, the curves show a clear linearity, with a slope suggesting that all the CPU-based belief propagation algorithms increase in time complexity at a linear rate. It is also evident that the PyTorch implementation is consistently slower than the `scipy.sparse` implementation, which is to be expected because PyTorch operations incur an additional overhead for their ability to integrate with deep learning procedures (e.g., back-propagation and related tasks).

Notably, we can see that the direct Python loop-based implementation is by far the slowest of these options. However, when the variable cardinality increases to large values, the Python implementation nearly matches the speed of sparse-matrix belief propagation with PyTorch on the CPU. While OpenGM's belief propagation does offer some benefits compared to Python initially at a lower values of c, it actually results in the slowest running times at $c \geq 32$. We can conclude that despite the compiled and optimized C++ code, the number of variables and states can eventually overshadow any speedups initially seen at lower values.

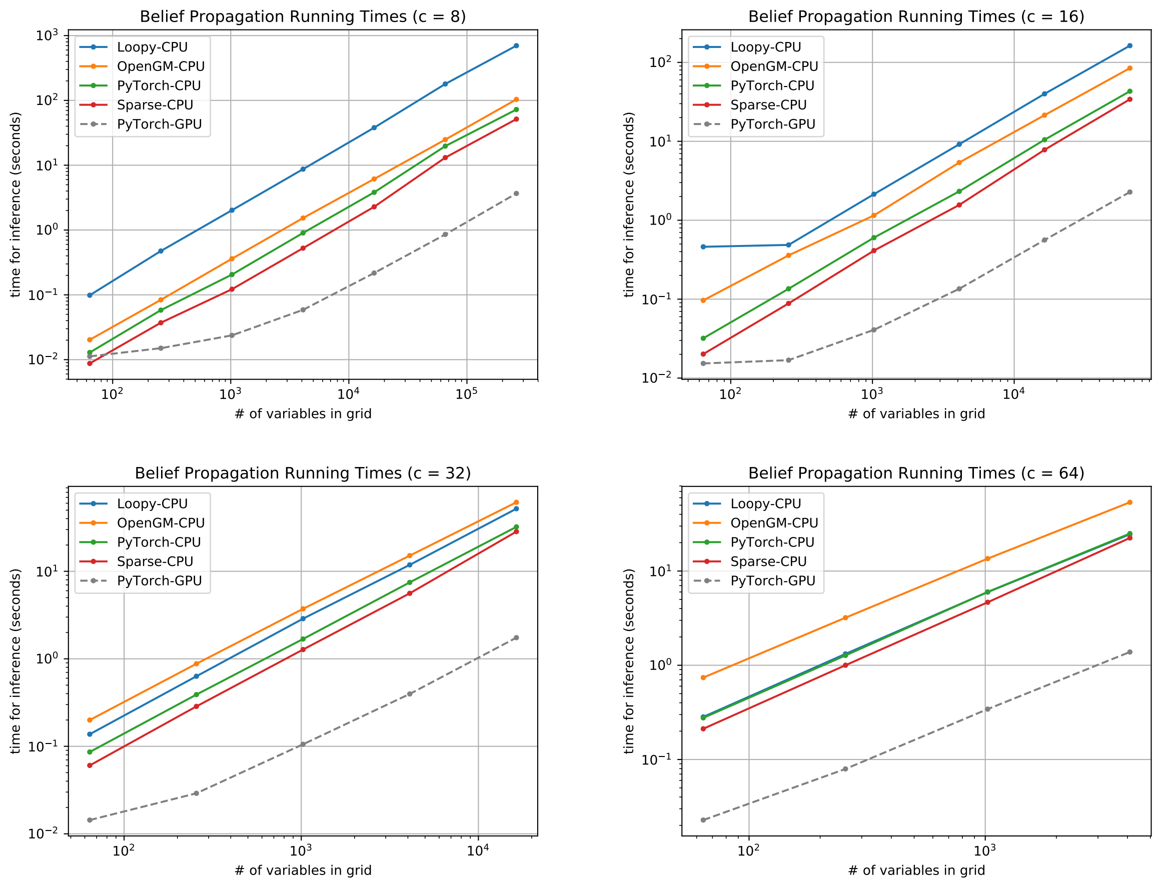

Figure 5.1: Log-log plots of belief propagation running times for four different implementations on the CPU. Each plot shows the results for different variable cardinalities c. OpenGM refers to the compiled C++ library, Loopy refers to the direct Python implementation. Sparse uses implements sparse-matrix belief propagation with `scipy.sparse`. And PyTorch implements it with the PyTorch library. We also plot the running time using PyTorch on the least powerful GPU we tested (Nvidia GTX780M) for comparison. The CPU runs plotted here use a 2.4 Ghz Intel i7.

In Fig. 5.1, we also include the running times for sparse-matrix belief propagation with PyTorch on the GPU—shown with the dotted gray line. A direct comparison is not exactly fair, since we are comparing across different computing hardware, though these curves were measured on the same physical computer. The comparison makes clear that the GPU offers significant speed increases over any CPU-based belief propagation, even that of the faster `scipy.sparse` implementation. Interestingly, there is a trend with the GPU runtimes that are sub-linear in the log-log plots, representing the cases where the number of sparse columns is not yet more than the number of cores of the GPU.

Examining the results for GPU-based belief propagation in Fig. 5.2, a majority of the GPUs are fairly close in running time between the three powerful Nvidia units: the 1080, the 1080Ti, and the

P40. The 780M understandably lags behind. As seen previously, until the cores are saturated with operations, there appears to be a pseudo-constant time cost. And once the cores are saturated, the running times grow linearly. This trend is best seen at $c = 16$ for the first two or three points. We also include the fastest CPU running time, using `scipy.sparse` on an Intel 4 Ghz i7, to illustrate the drastic difference in time between sparse-matrix belief propagation on the CPU and GPU.

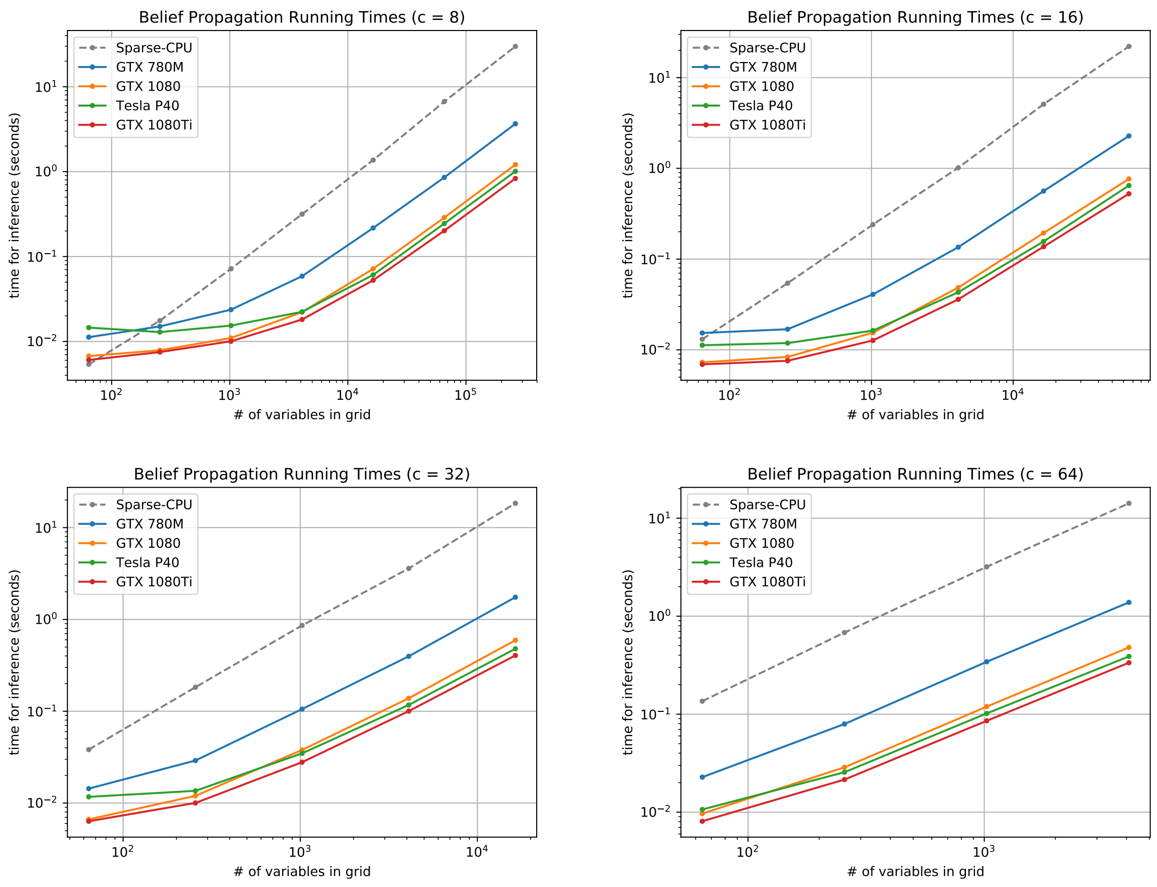

Figure 5.2: Log-log plots of PyTorch-GPU belief propagation running times for four different GPUs (780M, 1080, Tesla P40, 1080Ti) and the fastest CPU method (Sparse-CPU) with different variable cardinalities c. The CPU is the 4Ghz Intel i7.

These results demonstrate that sparse-matrix belief propagation enables the fastest running times for inference in these grid MRFs on the CPU. And using different software backends (`scipy.sparse` or PyTorch) for the sparse-matrix operations leads to different behavior, with PyTorch incurring some overhead resulting in slower computation. Once ported to the GPU, the speedups are even more drastic, resulting in running times that are many factors faster than those seen on the CPU, easily outweighing the overhead cost of using software backends like PyTorch that support seamless switching from CPU to GPU computation.

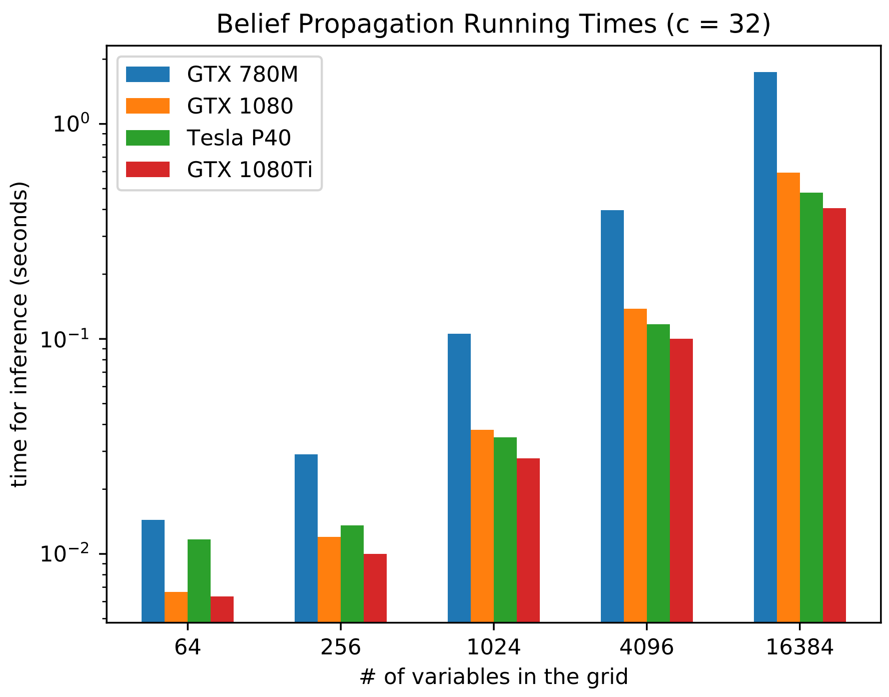

Figure 5.3: Log plot of PyTorch-GPU belief propagation running times for four different GPUs (780M, 1080, Tesla P40, 1080Ti) with 1536, 2560, 3840, and 3584 cores respectively, given a cardinality c of 32. This shows the benefits of increasing the number of cores to parallelize more matrix operations to decrease runtime.

Modifying the graph of results for GPU-based belief propagation in Fig. 5.3, it is possible to see the benefits of increasing the number of cores in the GPUs. As the number of variables in the grid increase (with the first instance guaranteed to fill up the GTX 780M's 1536 cores), the cores in all of the GPUs will be saturated with operations. We can therefore hypothesize that, in general, more cores will result in faster inference times. However, there are some noticeable differences between the Tesla P40 and GTX 1080Ti which can most likely be attributed to the quality of the hardware as well as the power of each of the cores.

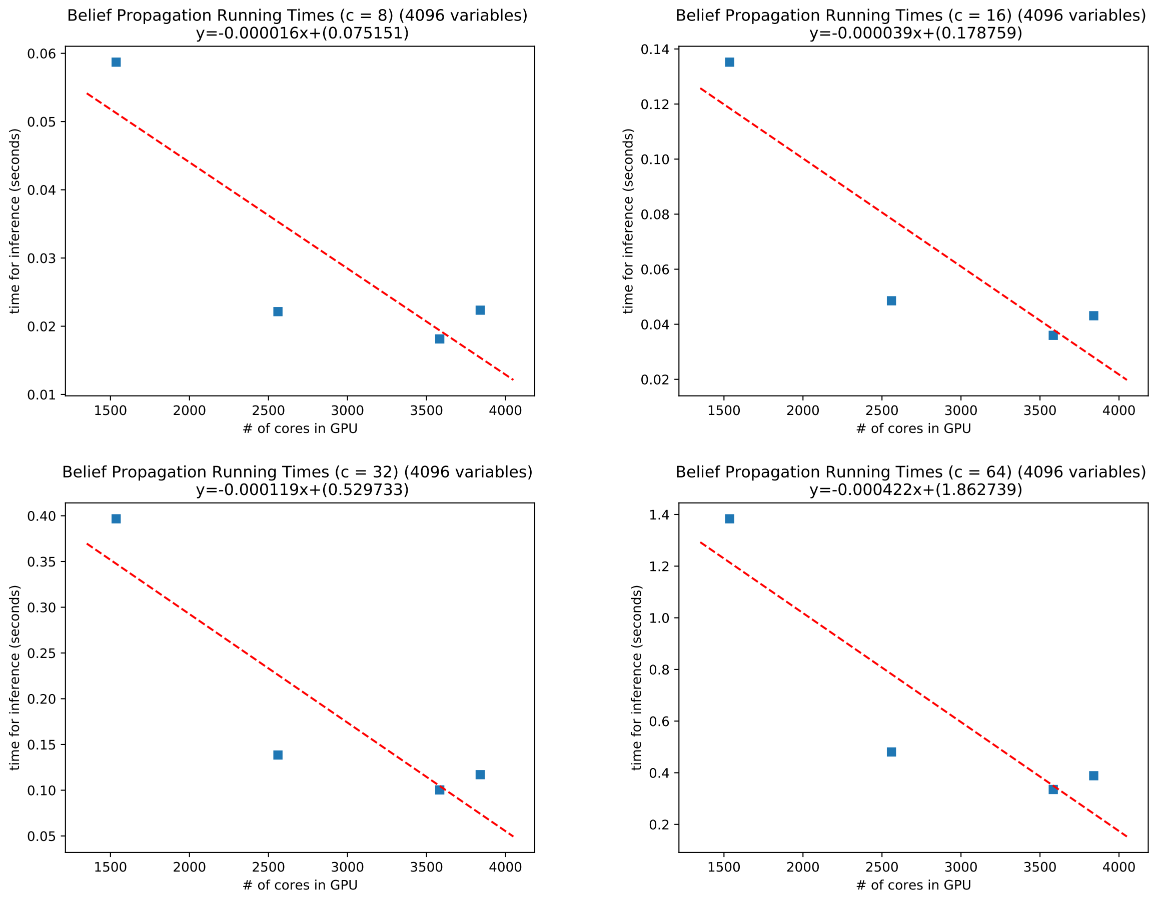

Figure 5.4: Plots of PyTorch-GPU belief propagation running times for four different GPUs (780M, 1080, Tesla P40, 1080Ti) with 1536, 2560, 3840, and 3584 cores respectively, for variable cardinalities c. The equations for each trendline are provided for each scatterplot to provide a general jist that more cores will tend to result in better inference times.

If look closer into Fig. 5.3, we can try to find the trendline for all possible cardinalities of this single grid size in order to generate Fig. 5.4 to hypothesize potential results for a variable number cores. As expected, as the number of states c is increased there will be more pronounced differences in the GPUs because the number of cores k will be doing more work. This can be seen as the slope of the trendlines increases approximately 33% when c doubles. It is interesting to note that the trendlines and scatterplots of all 4 values of c are extremely similar in shape, which could be attributed to a scalar increase based off of the number of states.

Chapter 6

Conclusion

We presented sparse-matrix belief propagation, which exactly reformulates loopy belief propagation (and its variants) as a series of matrix and tensor operations. This reformulation creates an abstract interface between belief propagation and a variety of highly optimized libraries for sparse-matrix and tensor operations. We demonstrated how sparse-matrix belief propagation scales as efficiently as low-level, compiled and optimized implementations, yet it can be implemented in high-level mathematical programming languages. We also demonstrated how the abstraction layer allows easy portability to advanced computing hardware by running sparse-matrix belief propagation on GPUs. The immediately resulting parallelization benefits required little effort once the sparse-matrix abstraction was in place. Our software library with these implementations are available as open-source software along with this thesis.

6.1 Open Questions and Next Steps

There are still a number of research directions that we would like to pursue. We mainly focused on analyzing the benefits of sparse-matrix belief propagation on grid-based MRFs, but there are many different structures of MRFs used in important applications (chain models, random graphs, graphs based on structures of real networks). Similarly, applying our approach to real-world examples would help confirm the utility of it in current problems within machine learning. Likewise, we focused on fairly sparse graphs that did not have many non-zero entries per column. It would be interesting to explore the difference between how sparse-matrix belief propagation behaves on the dense and sparse matrices on different hardware and whether fully dense matrices would still result in notable speed improvements, or if overhead from the sparse-matrix format would become a bottleneck.

We did extensively test our sparse-matrix belief propagation on a number of GPUs and CPUs, but it would be useful to see the benefit of getting a significant number of cores in the GPU to see whether the time complexity remains consistent until all of the cores are filled. Another research vector would be looking at graphs that rely on factors that are more than unary or pairwise, since we have only implemented the required algorithms up to pairwise belief propagation. Finally, we implemented learning on the CPU using sparse-matrices but we were hindered by PyTorch's lack of sparse-matrix operations for auto-gradients and backwards propagation for learning. Although we have implemented the embedding required for learning with sparse-matrix propagation on the GPU, we are unable to actually test the usability of it until the PyTorch developers implement the necessary functions required for actually running it.

Our sparse-matrix formulation of belief propagation is derived for pairwise MRFs, so it remains an open question what modifications are necessary for higher-order MRFs which may have arbitrarily large factors. Benefits similar to those we measured on GPUs can arise from the sparse-matrix abstraction for other computing backends, such as the use of compute-cluster parallelism through libraries such as Apache Spark, or the computation of belief propagation on FPGAs. Finally, the integration of belief propagation into deep learning models is straightforward with the matrix abstraction. Though many popular frameworks do not yet support back-propagation through sparse dot products, such support is forthcoming according to their respective developer communities.

Bibliography

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283.

Andres, B., Beier, T., and Kappes, J. H. (2012). Opengm: A C++ library for discrete graphical models. *CoRR*, abs/1206.0111.

Bayati, M., Shah, D., and Sharma, M. (2006). A simpler max-product maximum weight matching algorithm and the auction algorithm. In *Information Theory, 2006 IEEE International Symposium on*, pages 557–561. IEEE.

Bell, N. and Garland, M. (2008). Efficient sparse matrix-vector multiplication on CUDA. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation.

Bosagh Zadeh, R., Meng, X., Ulanov, A., Yavuz, B., Pu, L., Venkataraman, S., Sparks, E., Staple, A., and Zaharia, M. (2016). Matrix computations and optimization in Apache Spark. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 31–38. ACM.

Domke, J. (2013). Learning graphical model parameters with approximate marginal inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(10):2454–2467.

Felzenszwalb, P. F. and Huttenlocher, D. P. (2006). Efficient belief propagation for early vision. *International journal of computer vision*, 70(1):41–54.

Freeman, W. T., Jones, T. R., and Pasztor, E. C. (2002). Example-based super-resolution. *IEEE Computer graphics and Applications*, 22(2):56–65.

Gatterbauer, W., Günnemann, S., Koutra, D., and Faloutsos, C. (2015). Linearized and single-pass belief propagation. In *Proceedings of the VLDB Endowment*, volume 8, pages 581–592. VLDB Endowment.

Gilbert, J. R., Moler, C., and Schreiber, R. (1992). Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356.

Globerson, A. and Jaakkola, T. S. (2008). Fixing max-product: Convergent message passing algorithms for MAP LP-relaxations. In *Advances in Neural Information Processing Systems*, pages 553–560.

Huang, B. and Jebara, T. (2007). Loopy belief propagation for bipartite maximum weight b-matching. In *Artificial Intelligence and Statistics*, pages 195–202.

Huang, B. and Jebara, T. (2011). Fast b-matching via sufficient selection belief propagation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 361–369.

Koller, D. and Friedman, N. (2009). *Probabilistic graphical models: principles and techniques*. MIT press.

Low, Y., Gonzalez, J. E., Kyrola, A., Bickson, D., Guestrin, C. E., and Hellerstein, J. (2014). Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*.

Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM.

McAuley, J. J. and Caetano, T. S. (2010). Exploiting data-independence for fast belief-propagation. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 767–774.

McAuley, J. J. and Caetano, T. S. (2011). Faster algorithms for max-product message-passing. *Journal of Machine Learning Research*, 12(Apr):1349–1388.

Meshi, O., Jaimovich, A., Globerson, A., and Friedman, N. (2009). Convexifying the Bethe free energy. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 402–410. AUAI Press.

Murphy, K. P., Weiss, Y., and Jordan, M. I. (1999). Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 467–475. Morgan Kaufmann Publishers Inc.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in PyTorch. In *NIPS Workshop on Autodiff*.

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

Pearl, J. (2014). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

Poppe, R. (2007). Vision-based human motion analysis: An overview. *Computer vision and image understanding*, 108(1-2):4–18.

Schmidt, M. (2007). UGM: A Matlab toolbox for probabilistic undirected graphical models.

Schwing, A., Hazan, T., Pollefeys, M., and Urtasun, R. (2011). Distributed message passing for large scale graphical models. In *Computer Vision and Pattern Recognition*.

Sun, J., Zheng, N.-N., and Shum, H.-Y. (2003). Stereo matching using belief propagation. *IEEE Transactions on pattern analysis and machine intelligence*, 25(7):787–800.

Tarlow, D., Givoni, I., and Zemel, R. (2010). Hop-map: Efficient message passing with high order potentials. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 812–819.

Wainwright, M. J., Jaakkola, T. S., and Willsky, A. S. (2003). Tree-reweighted belief propagation algorithms and approximate ml estimation by pseudo-moment matching. In *International Conference in Artificial Intelligence and Statistics*.

Wainwright, M. J., Jordan, M. I., et al. (2008). Graphical models, exponential families, and variational inference. *Foundations and Trends in Machine Learning*, 1(1–2):1–305.

Yedidia, J. S., Freeman, W. T., and Weiss, Y. (2001). Generalized belief propagation. In *Advances in Neural Information Processing Systems*, pages 689–695.

Yedidia, J. S., Freeman, W. T., and Weiss, Y. (2003). Understanding belief propagation and its generalizations. *Exploring Artificial Intelligence in the New Millennium*, 8:236–239.

Yuan, D. (2004). Understanding belief propagation and its applications.

Zhang, J., Tang, J., and Li, J. (2007). Expert finding in a social network. In *International Conference on Database Systems for Advanced Applications*, pages 1066–1069. Springer.

Zheng, L., Mengshoel, O., and Chong, J. (2012). Belief propagation by message passing in junction trees: Computing each message faster using GPU parallelization. *arXiv preprint arXiv:1202.3777*.

Zheng, S., Jayasumana, S., Romera-Paredes, B., Vineet, V., Su, Z., Du, D., Huang, C., and Torr, P. H. (2015). Conditional random fields as recurrent neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1529–1537.

Zhuo, L. and Prasanna, V. K. (2005). Sparse matrix-vector multiplication on fpgas. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, pages 63–74. ACM.

Appendix A

Belief Propagation Running Time Tables

A.1 Nvidia GTX780M

Table A.1: GTX780M: Belief Propagation Running Times (seconds), c = 8

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.020272	0.011237	0.012888	0.008742	0.098578
16	0.083722	0.015032	0.058068	0.037125	0.474721
32	0.360800	0.023691	0.205534	0.122105	2.022244
64	1.531692	0.058706	0.902016	0.525256	8.698764
128	6.132734	0.217165	3.817908	2.282667	37.969736
256	24.843525	0.855206	19.713295	13.101705	179.301628
512	103.600243	3.667111	72.331408	51.483916	699.274341

Table A.2: GTX780M: Belief Propagation Running Times (seconds), c = 16

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.096507	0.015329	0.031925	0.020133	0.460826
16	0.358772	0.016870	0.135298	0.088272	0.486754
32	1.150753	0.040840	0.601358	0.411977	2.137228
64	5.384442	0.135217	2.322249	1.562178	9.173213
128	21.453021	0.561791	10.471162	7.811223	39.972905
256	84.609197	2.277056	43.121254	34.089050	162.973760

Table A.3: GTX780M: Belief Propagation Running Times (seconds), c = 32

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.199723	0.014366	0.086149	0.060483	0.137518
16	0.877941	0.029045	0.390077	0.285896	0.631804
32	3.727052	0.105524	1.684750	1.276569	2.876917
64	15.109935	0.396690	7.473420	5.603009	11.846864
128	61.515682	1.745220	32.201635	28.427104	51.934883

Table A.4: GTX780M: Belief Propagation Running Times (seconds), c = 64

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.740198	0.022770	0.276710	0.211892	0.283248
16	3.198477	0.079424	1.272262	1.001056	1.320565
32	13.549847	0.342601	6.001980	4.663128	5.978763
64	53.679988	1.383379	24.984524	22.453724	24.563946

A.2 Nvidia GTX1080

Table A.5: GTX1080: Belief Propagation Running Times (seconds), c = 8

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.020272	0.006733	0.008534	0.005782	0.065790
16	0.083722	0.007871	0.032336	0.018857	0.296915
32	0.360800	0.010978	0.116780	0.077353	1.332910
64	1.531692	0.022142	0.497928	0.351014	5.985465
128	6.132734	0.071725	2.251999	1.593751	26.016482
256	24.843525	0.287736	9.182033	8.157151	115.385464
512	103.600243	1.209909	42.375265	52.849959	495.101471

Table A.6: GTX1080: Belief Propagation Running Times (seconds), c = 16

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.096507	0.007291	0.021194	0.014105	0.323346
16	0.358772	0.008348	0.063954	0.058162	0.331568
32	1.150753	0.015454	0.276859	0.270383	1.384073
64	5.384442	0.048541	1.198638	1.129079	6.219800
128	21.453021	0.193675	5.428982	5.629873	27.669391
256	84.609197	0.764596	22.204164	28.202716	112.273983

Table A.7: GTX1080: Belief Propagation Running Times (seconds), c = 32

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.199723	0.006640	0.038504	0.041042	0.091913
16	0.877941	0.011980	0.182865	0.209775	0.413384
32	3.727052	0.037853	0.810485	0.993179	1.935465
64	15.109935	0.138474	3.448630	3.886026	8.039828
128	61.515682	0.593276	15.246372	23.180357	35.220745

Table A.8: GTX1080: Belief Propagation Running Times (seconds), c = 64

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.740198	0.009673	0.124163	0.155785	0.179556
16	3.198477	0.028704	0.594520	0.786308	0.876142
32	13.549847	0.119439	3.193508	3.583863	3.861142
64	53.679988	0.480456	13.274430	16.237296	15.900084

A.3 Nvidia GTX1080Ti

Table A.9: GTX1080Ti: Belief Propagation Running Times (seconds), c = 8

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.020272	0.006051	0.007864	0.005374	0.059837
16	0.083722	0.007495	0.029971	0.017588	0.272105
32	0.360800	0.010052	0.098275	0.071773	1.196866
64	1.531692	0.018141	0.400870	0.316002	5.400465
128	6.132734	0.052596	1.761170	1.369214	23.792207
256	24.843525	0.201496	7.855352	6.691314	102.918219
512	103.600243	0.831604	34.830824	29.866451	455.517229

Table A.10: GTX1080Ti: Belief Propagation Running Times (seconds), c = 16

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.096507	0.006916	0.019813	0.013084	0.319430
16	0.358772	0.007572	0.058478	0.054238	0.298507
32	1.150753	0.012684	0.243220	0.240177	1.256672
64	5.384442	0.035976	1.029164	1.015032	5.692176
128	21.453021	0.137003	4.712802	5.091448	25.402714
256	84.609197	0.524517	19.212330	22.067404	103.313868

Table A.11: GTX1080Ti: Belief Propagation Running Times (seconds), c = 32

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.199723	0.006339	0.040258	0.038213	0.084747
16	0.877941	0.010025	0.160960	0.182602	0.384991
32	3.727052	0.027862	0.693443	0.860372	1.745555
64	15.109935	0.100321	3.062812	3.587058	7.406049
128	61.515682	0.405351	13.418391	18.403050	32.680911

Table A.12: GTX1080Ti: Belief Propagation Running Times (seconds), c = 64

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.740198	0.008083	0.109308	0.135710	0.168454
16	3.198477	0.021513	0.499943	0.680351	0.787509
32	13.549847	0.085563	2.448388	3.189963	3.585489
64	53.679988	0.335110	9.983072	14.196262	16.991835

A.4 Nvidia Tesla P40

Table A.13: Tesla P40: Belief Propagation Running Times (seconds), c = 8

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.020272	0.014558	0.137449	0.009174	0.100881
16	0.083722	0.012896	0.066122	0.028871	0.493219
32	0.360800	0.015374	0.162383	0.113912	2.402048
64	1.531692	0.022354	0.640148	0.478938	8.760202
128	6.132734	0.060849	2.649968	2.531628	39.524056
256	24.843525	0.245069	10.572738	12.411715	179.771758
512	103.600243	1.009314	44.768773	55.433869	829.963888

Table A.14: Tesla P40: Belief Propagation Running Times (seconds), c = 16

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.096507	0.011239	0.070383	0.032316	0.338633
16	0.358772	0.011868	0.095390	0.084592	0.511104
32	1.150753	0.016287	0.376829	0.357259	2.083594
64	5.384442	0.043109	1.508700	1.697729	9.355308
128	21.453021	0.156361	5.817397	7.168117	44.920947
256	84.609197	0.646450	22.975880	38.280244	197.858308

Table A.15: Tesla P40: Belief Propagation Running Times (seconds), c = 32

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.199723	0.011685	0.104925	0.070912	0.139143
16	0.877941	0.013591	0.254402	0.266805	0.635405
32	3.727052	0.034786	0.837890	1.512913	2.913891
64	15.109935	0.117012	3.775627	5.781395	12.233949
128	61.515682	0.479337	15.250191	30.758329	54.476234

Table A.16: Tesla P40: Belief Propagation Running Times (seconds), c = 64

# rows/cols	OpenGM	Torch-CUDA	Torch-Py	Sparse-Py	Loop-Py
8	0.740198	0.010656	0.148268	0.197692	0.274660
16	3.198477	0.025574	0.634041	1.074605	1.309389
32	13.549847	0.101743	3.159493	5.153469	5.952063
64	53.679988	0.388470	11.215913	22.243880	24.023032