August, 2013

Optimizing Parallel Belief Propagation in Junction Trees using Regression

Lu Zheng, *Carnegie Mellon University*
Ole J Mengshoel, *Carnegie Mellon University*

Optimizing Parallel Belief Propagation in Junction Trees using Regression

Lu Zheng
ECE Department
Carnegie Mellon University
luzheng@andrew.cmu.com

Ole Mengshoel
ECE Department
Carnegie Mellon University
ole.mengshoel@sv.cmu.edu

ABSTRACT

The junction tree approach, with applications in artificial intelligence, computer vision, machine learning, and statistics, is often used for computing posterior distributions in probabilistic graphical models. One of the key challenges associated with junction trees is computational, and several parallel computing technologies - including many-core processors - have been investigated to meet this challenge. Many-core processors (including GPUs) are now programmable, unfortunately their complexities make it hard to manually tune their parameters in order to optimize software performance. In this paper, we investigate a machine learning approach to minimize the execution time of parallel junction tree algorithms implemented on a GPU. By carefully allocating a GPU's threads to different parallel computing opportunities in a junction tree, and treating this thread allocation problem as a machine learning problem, we find in experiments that regression - specifically support vector regression - can substantially outperform manual optimization.

Categories and Subject Descriptors

I.2.m [**Computing Methodologies**]: ARTIFICIAL INTELLIGENCEMiscellaneous

General Terms

Algorithm

Keywords

belief propagation, parallel computing, regression

1. INTRODUCTION

Parallel processing is becoming increasingly important in all areas of computing, including in knowledge discovery and machine learning. This is to a large extent due to recent developments in hardware, and in particular a key difference between Moore's law and clock frequency of CPUs. Moore's law, which states that the number of transistors that can be placed on an integrated circuit will increase exponentially, is as of 2013 still going strong. However, the clock frequency of CPUs has stalled, due to physical limits on heat dissipation in integrated circuits. As a consequence, computers are now multi-core (CPUs) or many-core (GPUs), and algorithms that take advantage of this fact will be at an advantage. The importance of parallel, and also distributed, computing in data mining is further increased due to Big Data; the size of the data sets available for analytical processing has recently been increasing drastically.

In this paper, we discuss parallel computing for belief propagation in junction trees. Belief propagation (BP) over junction tree can be used to compute posterior marginals in Bayesian networks (BNs) [9]. However, belief propagation is computationally hard, and the computational difficulty increases dramatically with the density of the BN, the number of states of each network node and the BN treewidth, which is upper bounded by the generated junction tree [12].

This computational issue may hinder the application of BNs in cases where real-time inference is required. Parallelization of Bayesian network computation is a feasible way of addressing this computational issue [8, 13, 17, 16, 7, 10, 6, 11, 2]. These parallel BP algorithms are implemented on various state of the art parallel computing platforms. However, due to the complexity of these modern platforms, the junction tree algorithm, and the way they interact with each other, it is not trivial to make parallel BP algorithms work efficiently on these platforms.

Many-core computers including GPUs, which are built around an array of processors running many threads of execution in parallel, are among the most popular platforms. However, it is non-trivial to optimize GPU programs. The challenges with GPU optimization for parallel BP includes:

- Junction tree clique and separator sizes vary significantly, resulting in unbalanced workload.
- The junction tree algorithm and a GPU platform have distinctive sets of parameters. Mismatch between algorithm and platform parameters can result in poor performance.
- The relationship between the input workload and the output performance metrics is generally unknown.
- In the two-dimensional parallel BP algorithm (see Section 3), allocation of threads to each parallel dimension requires great care.

KDD'13, August 11–14, 2013, Chicago, Illinois, USA.

ACM 978-1-4503-2174-7/13/08 ...$15.00.

In this paper, we focus on minimizing compute time of node level parallel BP on many core system, in particular GPUs, using system performance modeling. We expand on research using GPU to implement node level parallelism of belief propagation [18][6]. Our work is done on top of a parallel BP algorithm [18]. Our contribution in this work includes

- We investigate another dimension of parallelism, arithmetic parallelism, and integrate it with element-wise parallelism [18].
- We use statistical models for GPU parameter optimization, resulting in an average cross platform speedup of 10.70x (arithmetic average) or 8.68x (geometric average) as opposed to that of 3.43x (arithmetic average) or 2.44x (geometric average) obtained previously [18]
- We propose new metrics "squared deviance" and "miss rate" to measure the quality of statistical models for our problem.

This work is relevant to data mining and machine learning in two distinct ways. First, we investigate parallel computation using probabilistic graphical models, in particular junction trees [1], which are applied in many data ming contexts, including Expectation Maximization. Second, in order to solve the central problem of thread allocation for parallel junction tree computation, we take a machine learning approach.

Our paper is organized as follows: In Section 2, we briefly review previous research and formulate the problem we are going to solve. In Section 3, we describe the two-dimensional parallel belief propagation algorithm, where GPU parameter optimization dicussed in Section 4 is key to good performance. In Section 5, we describe our approach of using statistical models for system performance prediction and use it for GPU parameter optimization. Experimental results are discussed in Section 6. We conclude in Section 7.

2. BACKGROUND

2.1 Belief Propagation in Junction Tree

A BN is a compact representation of a joint distribution over a set of random variables $\mathcal{X}$. A BN is structured as a directed acyclic graph (DAG) whose vertices are the random variables and the directed edges represent dependency relationship among the random variables. The evidence in a Bayesian network consists of instantiated variables.

The junction tree algorithm propagates beliefs (or posteriors) over a derived graph called a junction tree. A junction tree is generated from a BN by means of moralization and triangulation [9]. Each vertex C_i of the junction tree contains a subset of the random variables that forms a clique in the moralized and triangulated BN, denoted by $\mathcal{X}_i \subseteq \mathcal{X}$. Associated with each vertex of the junction tree there is a potential table $\phi_{\mathcal{X}_i}$. With the above notations, a junction tree can be defined as $J = (\mathbb{T}, \Phi)$, where $\mathbb{T}$ represents a tree and Φ represents all the potential tables associated with this tree. Assuming C_i and C_j are adjacent, a separator S_{ij} is induced on a connecting edge. The variables contained in S_{ij} are defined to be $\mathcal{X}_i \cap \mathcal{X}_j$.

The computation of belief propagation can be measured by *treewidth*, which is defined to be the minimal size of the largest set in junction tree minus one. Considering a junction tree with a treewidth t_w, the amount of computation is lowered bounded by $O(\exp(c * t_w))$ where c is a constant.

Belief propagation is invoked when we get new evidence $\boldsymbol{e}$ for a set of variables $\mathcal{E} \subseteq \mathcal{X}$. We need to update the potential tables Φ to reflect this new information. To do this, belief propagation over the junction tree is used, this is a two-phase procedure: evidence collection and evidence distribution. For the evidence collection phase, messages are collected from the leaf vertices all the way up to a designated root vertex. For the evidence distribution phase, messages are distributed from the root vertex to the leaf vertices. A recursive algorithm of collecting and distributing evidence is shown in Algorithm 1 and 2. In the following sections, we are going to focus on parallelizing each message passing in Algorithm 1 and 2.

Algorithm 1 Collect_Evidence($J, \mathcal{C}_i$)

```
for each child of C_i do
  Message_Passing(C_i, Collect_Evidence(J,child))
end for
return(C_i)
```

Algorithm 2 Distribute_Evidence($J, \mathcal{C}_i$)

```
for each child of C_i do
  Message_Passing(C_i, child)
  Distribute_Evidence(J, child)
end for
```

2.2 Graphics Processing Units (GPU)

GPUs are designed for compute-intensive, highly parallel computations. In GPUs, more transistors are devoted to data processing rather than data caching and flow control. GPUs are especially well-suited to problems that can be expressed as data-parallel computations where data elements are mapped to parallel processing threads. GPUs are mainly used as accelerators for compute-intensive parts of an application, and therefore attached to a host CPU that performs control-dominant computations.

The GPU is programmed using the CUDA programming framework [14]. An application is organized into a sequential host program that is run on a CPU, and one or more parallel GPU kernels that are run on a GPU.

In GPUs, threads launched are partitioned into thread blocks. There is a limit on the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads [14]. For a GPU to run efficiently and effectively, the concurrency provided by the platform should match the parallel opportunities in the application. The challenge in our work, detailed in Section 3, is that we have two dimensions of parallelism and therefore we need to allocate threads in each block to the two dimensions of parallel opportunities. A poor split may result in waste of computing resources and low efficiency.

3. TWO-DIMENSIONAL PARALLEL BELIEF PROPAGATION

We parallelize the atomic operation of belief propagation– message passing for junction trees. The advantage of doing so is that this node level parallelism can be embedded in different belief propagation algorithms unobtrusively, without any change of those algorithms.

Associated with each junction tree vertex $\mathcal{C}_i$ and the contained set of variables $\mathcal{X}_i$, there is a potential table $\phi_{\mathcal{X}_i}$ containing non-negative real numbers that are proportional to the joint distribution of $\mathcal{X}_i$. If each variable can take s_j states, the size of the potential table is $|\phi_{\mathcal{X}_i}| = \prod_{j=1}^{|\mathcal{X}_i|} s_j$, where $|\mathcal{X}_i|$ is the cardinality of $\mathcal{X}_i$.

Message passing from vertex $\mathcal{C}_i$ to an adjacent vertex $\mathcal{C}_k$, with separator $\mathcal{S}_{ik}$, involves two steps:

1. **Reduction**. The potential table $\phi_{\mathcal{S}_{ik}}$ of the separator is updated to $\phi^*_{\mathcal{S}_{ik}}$ by reducing the potential table $\phi_{\mathcal{X}_i}$:

$$\phi^*_{\mathcal{S}_{ik}} = \sum_{\mathcal{X}_i/\mathcal{S}_{ik}} \phi_{\mathcal{X}_i}. \quad (1)$$

2. **Scattering**. The potential table of $\mathcal{C}_k$ is updated using both the old and new table of $\mathcal{S}_{ik}$:

$$\phi^*_{\mathcal{X}_k} = \phi_{\mathcal{X}_k} \frac{\phi^*_{\mathcal{S}_{ik}}}{\phi_{\mathcal{S}_{ik}}}. \quad (2)$$

We define $\frac{0}{0} = 0$ in this case, that is, if the denominator in (2) is zero, then we simply set the corresponding $\phi^*_{\mathcal{X}_k}$ to zeros.

A close look at Equation (1) and (2) reveals two dimensions of parallelism opportunity in a message passing. The first dimension of parallelism is the separator potential table (SPT) *element-wise parallelism*. The second dimension of parallelism is the *arithmetic parallelism*.

Element-Wise Parallelism: An *index mapping table* $\mu_{\mathcal{X},\mathcal{S}}$ stores the index mappings from $\phi_{\mathcal{X}}$ to $\phi_{\mathcal{S}}$ [5]. We create $|\phi_{\mathcal{S}_{ik}}|$ mapping tables. In each mapping table $\mu_{\mathcal{X}_i,\phi_{\mathcal{S}_{ik}}(j)}$ we store the indices of the elements of $\phi_{\mathcal{X}_i}$ mapping to the j-th separator table element. Mathematically, $\mu_{\mathcal{X}_i,\phi_{\mathcal{S}_{ik}}(j)} = \{r \in [0, |\phi_{\mathcal{X}_i}| - 1] : \ \phi_{\mathcal{X}_i}(r)$ is mapped to $\phi_{\mathcal{S}_{ik}}(j)\}$.

With the index mapping table, element-wise parallelism can be obtained by assigning a specific group of threads to handle the computation related to a specific separator potential table.

Arithmetic Parallelism: Arithmetic parallelism needs to be explored in different ways for reduction and scattering, and also integrated with element-wise parallelism, as we will discuss now.

For reduction, given a certain fixed element j, Equation (1) is essentially a summation over all the clique potential table (CPT) $\phi_{\mathcal{X}_i}$ elements indicated by the corresponding mapping table $\mu_{\mathcal{X}_i,\phi_{\mathcal{S}_{ik}}(j)}$. The number of sums is $|\mu_{\mathcal{X}_i,\phi_{\mathcal{S}_{ik}}(j)}|$. We compute the summation in parallel by using an existing approach [4]. The summation is done in several iterations. In each iteration, the numbers are divided into two groups and the corresponding two numbers in each group are added in parallel.

For scattering, note that (2) updates the elements of $\phi_{\mathcal{X}_k}$ independently despite that $\phi_{\mathcal{S}_{ik}}$ and $\phi^*_{\mathcal{S}_{ik}}$ are re-used to update different elements. Therefore, we can compute each multiplication in (2) with a single thread.

Given the two dimensions of parallelism, our parallel message passing approach is illustrated in Figure 1. Denote $\mu_{\mathcal{X}_i[m]}$ to be the m-th element in table $\mu_{\mathcal{X}_i}$ and $\phi_{\mathcal{S}_{ik}}[n]$ the n-th element in table $\phi_{\mathcal{S}_{ik}}$. If the size of mapping table $\mu_{\mathcal{X}_i,\mathcal{S}_{ik}}$ is integer power of 2 (assuming $|\mu_{\mathcal{X}_i,\mathcal{S}_{ik}}| = 2^d$), the parallel reduction algorithm can be written as in Algorithm 3. If $\mu_{\mathcal{X}_i,\mathcal{S}_{ik}}$ is not integer power of 2, we can use techniques such as zero-padding to make it integer power of 2. Algorithm 3 integrates the two dimensions of parallelism for the reduction step - the element-wise parallelism determined by $|\phi_{\mathcal{S}_{ik}}|$ and the arithmetic parallelism determined by the size of mapping table $|\mu_{\mathcal{X}_i,\mathcal{S}_{ik}}|$. The scattering step is shown in Algorithm 4.

Algorithm 3 Reduction($\phi_{\mathcal{X}_i}, \phi_{\mathcal{S}_{ik}}, \mu_{\mathcal{X}_i,\mathcal{S}_{ik}}$)

Require: $\mu_{\mathcal{X}_i,\mathcal{S}_{ik}} = 2^d$
Ensure: $\phi_{\mathcal{X}_i}$
  for $m = 1$ *to* $|\phi_{\mathcal{S}_{ik}}|$ **in parallel do**
    for $n = 1$ *to* d **do**
      for $k = 0$ *to* $2^{d-n} - 1$ **in parallel do**
        $\phi_{\mathcal{X}_i}[\mu_{\mathcal{X}_i,\mathcal{S}_{ik}(m)}[k]] = \phi_{\mathcal{X}_i}[\mu_{\mathcal{X}_i,\mathcal{S}_{ik}(m)}[k]] + \phi_{\mathcal{X}_i}[\mu_{\mathcal{X}_i,\mathcal{S}_{ik}(m)}[k + 2^{d-n}]]$
      end for
    end for
    $\phi_{\mathcal{S}_{ik}}[m] = \phi_{\mathcal{X}_i}[\mu_{\mathcal{X}_i,\mathcal{S}_{ik}(m)}[0]]$
  end for

Algorithm 4 Scattering($\phi_{\mathcal{X}_k}, \phi_{\mathcal{S}_{ik}}, \mu_{\mathcal{X}_k,\mathcal{S}_{ik}}$)

  for $m = 1$ *to* $|\phi_{\mathcal{X}_k}|$ **in parallel do**
    $\phi_{\mathcal{X}_k}[m] = \frac{\phi^*_{\mathcal{S}_{ik}}}{\phi_{\mathcal{S}_{ik}}} \phi_{\mathcal{X}_k}[m]$
  end for

4. PARAMETER OPTIMIZATION FOR PARALLEL MESSAGE PASSING

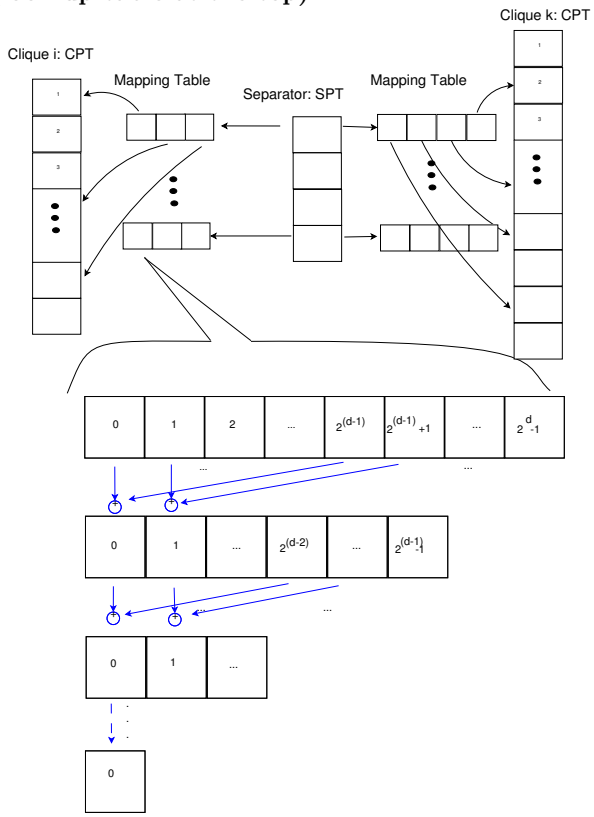
Belief propagation is essentially a sequence of message passings $\{m_i\}_{i=1}^{2M-2}$ over the edges of a junction tree, where M is the number of nodes in the junction tree. Each message passing has a reduction step and a scattering step. The parallel algorithms for reduction (Algorithm 3) and scattering (Algorithm 4) assume infinite threads available for parallel computing. However, a parallel computing platform such as a GPU has only limited number of threads available.

GPU message passing is repeatly called by the CPU. Each time the GPU is called, the thread block size as well as the thread allocation can be set from the CPU side. Therefore, when implementing Algorithm 3 and Algorithm 4 on a GPU, a programmer faces the problem of how to set thread block size and allocate the GPU parallel threads to the two dimensions of parallelism, i.e., we need to find a sequence of GPU run-time parameters for each message passing.

4.1 Junction Tree and GPU Parameters

Intrinsic Junction Tree Parameters: Consider a message passing from $\mathcal{C}_i$ to $\mathcal{C}_k$ through a separator $\mathcal{S}$. From the computational perspective, the input cliques and separators can be characterized by a set of *Intrinsic Junction Tree Parameter* $\mathbb{P}_{intr} = \{|\phi_{\mathcal{X}_i}|, |\phi_{\mathcal{X}_k}|, |\phi_{\mathcal{S}}|\}$, where $|\phi_{\mathcal{X}_i}|, |\phi_{\mathcal{X}_k}|, |\phi_{\mathcal{S}}|$ represent the size of potential tables of $\mathcal{C}_i$, $\mathcal{C}_k$, and $\mathcal{S}$ respectively.

Figure 1: Two parallelism opportunities in a junction tree: element-wise and arithmetic parallelism. Arithmetic parallelism (tree structure at the bottom) is added on top of the element-wise parallelism (look up table at the top).



GPU Parameters: Before invoking a GPU kernel function for message passing, two question should be resolved on the CPU side: (1) What should thread-block size be? (2) In each thread block, how should threads be divided between element-wise and arithmetic parallelism? To efficiently compute the message passing, we need to carefully choose the set of *GPU parameters:* $\mathbb{P}_{gpu} = \{K_r,\ K_s,\ p_a^r,\ p_e^r,\ p_a^s,\ p_e^s\}$, where K_r and K_s are the total threads of one thread block for reduction and scattering respectively; p_a^r and p_a^s are the number of threads used for arithmetic parallelism in reduction and scattering respectively; and p_e^r and p_e^s are the number of threads used for element-wise parallelism used by each thread block in reduction and scattering.

4.2 GPU Optimization Examples

Thread allocation optimization is very important for parallel message passing in junction tree. Due to the variability of cliques and separators involved in message passing, the performance surfaces have greatly varying shapes.

Figure 1 suggests if a mapping table is large, it is better to assign more threads to the arithmetic parallelism; if a separator table is large, it would be wise to assign more threads to the element-wise parallelism.

Figure 2 shows three examples of different performance surfaces with respect to GPU parameters p_e^r and K_r. For these examples, we get several intuitions about choosing GPU parameters: 1) The search space is so diverse that using the same set of parameters for all message passings can seldom achieve optimal performance. Good parameters for one message passing could work inefficiently for another. 2) For a given message passing, optimal GPU parameters can result in huge improvement over a poor choice of GPU parameters (in some cases, more than 20x difference).

4.3 GPU Optimization

The metrics for measuring system performance vary. In our work, we use the execution time for belief propagation to all the cliques in junction tree as our metric. Since belief propagation over a junction tree is a sequence of message passings, in our node level parallel BP algorithm, minimizing the total BP execution time can be broken down to a sequence of tasks, minimizing the execution time for each message passing.

It requires $N = 2(M-1)$ message passings to complete belief propagation over a junction tree J with M cliques. Let $f_n : \mathbb{P}_{intr} \times \mathbb{P}_{gpu} \rightarrow \mathbb{R}$ be the execution time for one message passing. The total BP time is

$$f(J, \mathbb{P}_{gpu}^1, \ldots, \mathbb{P}_{gpu}^N) = \sum_{n=1}^{N} f_n(\mathbb{P}_{intr}^n, \mathbb{P}_{gpu}^n), \tag{3}$$

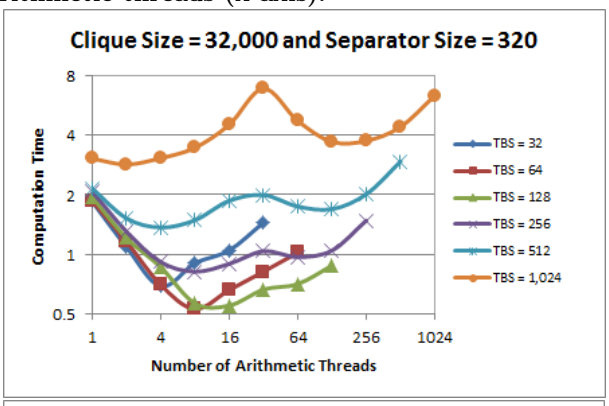
where $\mathbb{P}_{intr}^n$ and $\mathbb{P}_{gpu}^n$ are the parameters of the n-th message passing.

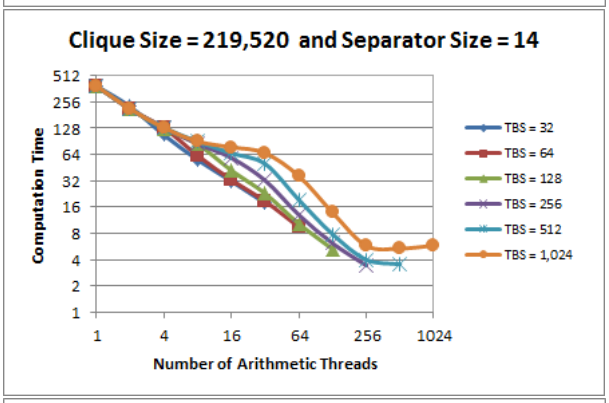
Thus, the GPU optimization problem can be modeled as:

$$\begin{aligned} \min_{\mathbb{P}_{gpu}^1, \ldots, \mathbb{P}_{gpu}^N} & \sum_{n=1}^{N} f_n(\mathbb{P}_{intr}^n, \mathbb{P}_{gpu}^n), \\ \text{s.t. } : & p_e^{rn} * p_a^{rn} \leq K_r^n, \quad \forall\ n \\ & p_e^{sn} * p_a^{sn} \leq K_s^n, \quad \forall\ n \end{aligned} \tag{4}$$

Unfortunately, traditional optimization techniques can not be applied to this optimization problem since an analytical form of $f_n(\cdot)$ is usually not available due to the complexity

Figure 2: Examples of how GPU execution time (y-axis) varies with different GPU parameters, specifically the thread block size (TBS) and number of arithmetic threads (x-axis).





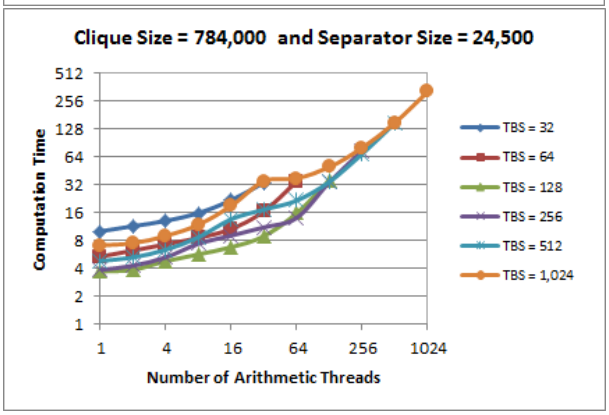
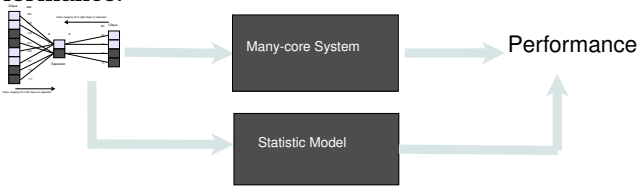


Figure 3: Statistical model of many-core system performance.



of the hardware platform. Fortunately, statistical learning can provide approximations for $f_n(\cdot)$, as we discuss next.

5. MATHEMATICAL MODELS FOR PARALLEL MESSAGE PASSING

5.1 Theoretical Model for Message Passing

Before we proceed to the black-box modeling approach with pre-execution parameters and post-execution performance, as our first attempt to characterizes the relationship between the message passing workload, GPU parameters and the output performance (execution time), we develop a simple mathematical model.

For simplicity, assume that the GPU takes a constant time τ_a and τ_m for the add and multiplication operations respectively. We ignore memory access time, device set up time, etc. Suppose the GPU can accommodate N_b thread blocks to run simultaneously. Consider a message passing from $\mathcal{C}_i$ to $\mathcal{C}_k$ using GPU parameter p_a^s, p_e^s, p_a^r and p_a^r.

Define $g(\phi, p) = \left\lceil \frac{|\phi|}{p} \right\rceil$. The time for reduction is

$$T_r = \left\lceil \frac{1}{N_b} g(\phi_{\mathcal{S}}, p_e^r) \right\rceil g(\phi_{\mathcal{X}_i}, p_a^r) \lfloor (\log_2 p_a^r + 1) \rfloor \tau_a \quad (5)$$

the time for scattering is

$$T_s = \left\lceil \frac{1}{N_b} g(\phi_{\mathcal{S}}, p_e^s) \right\rceil g(\phi_{\mathcal{X}_j}, p_a^s) \tau_a \quad (6)$$

The total message passing time between $\mathcal{C}_i$ and $\mathcal{C}_k$ is given by $T = T_r + T_s$. Due to this decomposition of message passing time into reduction time and scattering time, we can optimize the GPU parameters related to reduction and scattering separately. For reduction, we have:

$$\begin{aligned} &\min_{p_e^r, p_a^r} T_r \\ &\text{s.t.} \quad : p_e^r * p_a^r \leq K_r \end{aligned} \quad (7)$$

and for scattering we get:

$$\begin{aligned} &\min_{p_e^r, p_a^r} T_s \\ &\text{s.t.} \quad : p_e^s * p_a^s \leq K_s \end{aligned} \quad (8)$$

It is analytically and numerically hard to optimize (7) and (8) due to the irregular form of both the objective and constraint functions. In addition, the overly simplified assumptions make them not practically useful for the purpose of parameter selection. Consequently, we turn to a regression approach as discussed below.

5.2 Regression Models for Message Passing

In this subsection, we develop statistical models for message passing. The models used are *Polynomial Regression* and *Support Vector Regression (SVR)*. Essentially, we want

to establish a statistical relationship between the GPU configuration parameters and the performance as illustrated in Figure 3.

Polynomial Regression: For the polynomial model, in order to get better insight about how the thread allocation affects the GPU execution time, we use the Lasso method to shrink the model and compare the resulting model with (7) and (8). A polynomial Lasso has the form

$$\hat{\beta}^{Lasso} = \arg\min_{\beta} \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} Poly_j(\mathbf{x}_i) \right)^2$$
$$\text{s.t.} \sum_{j=1}^{p} |\beta_j| \leq t,$$

where $Poly_j(\mathbf{x})$ is a polynomial function of the feature vector $\mathbf{x}$. The Lasso in the *equivalent Lagrangian* form is

$$\hat{\beta}^{Lasso} = \arg\min_{\beta} \sum_{i=1}^{N} \left(y_i - \beta_0 - \sum_{j=1}^{p} Poly_j(\mathbf{x}_i) \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|, \tag{9}$$

where λ is the Lagrangian multiplier.

Support Vector Regression (SVR): A second regression model we use is support vector regression [3]. In SVR, the input is first mapped onto a high-dimensional feature space using some fixed (nonlinear) mapping, and then a linear model is constructed in this feature space. The linear model (in the feature space) $f(\mathbf{x}, \omega)$ is given by

$$f(\mathbf{x}, \omega) = \sum_{j=1}^{m} \omega_j g_j(\mathbf{x}) + b, \tag{10}$$

where $g_j(\mathbf{x}),\ j = 1, \ldots, m$, denotes a set of nonlinear transformations, and b is the bias term. The loss function of SVR is called ϵ-insensitive loss function defined as

$$L_\omega(y, f(\mathbf{x}, \omega)) = \begin{cases} 0 & \text{if } |y - f(\mathbf{x}, \omega)| \leq \epsilon \\ |y - f(\mathbf{x}, \omega)| - \epsilon & otherwise \end{cases} \tag{11}$$

SVR performs linear regression in the high-dimensional feature space using ϵ-insensitive loss and, at the same time, tries to reduce model complexity by minimizing $\|\omega\|^2$. Thus SVR is formulated as minimization of the following function:

$$\min \frac{1}{2}\|\omega\|^2 + C \sum_{i=1}^{n} (\xi_i + \xi_i^*)$$
$$s.t. \begin{cases} y_i - f(\mathbf{x}, \omega) \leq \epsilon + \xi_i^* \\ f(\mathbf{x}, \omega) - y_i \leq \epsilon + \xi_i^* \\ \xi_i, \xi_i^* \geq 0, i = 1, \ldots, n \end{cases}$$

5.3 Features for Regression Models

The features we collected for regression model training are shown in Table 1. There are two possible ways that statistical modeling can help us with run-time GPU parameter selection: 1) We can directly approximate the function

$$(K^*, p_a^*) = \arg\min_{K, p_a} T(K, p_a, |\phi_{\mathcal{S}}|, |\phi_{\mathcal{X}}|), \tag{12}$$

where $T : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ is the GPU execution time as a function of $K, p_a, |\phi_{\mathcal{S}}|$ and $|\phi_{\mathcal{X}}|$. Practically, the number of possible values K and p_a is finite, therefore, we can model (12) as a classification problem. Or 2) we can alternatively take an indirect approach by first training a regression model for GPU execution time and then search the regression model to obtain the best run-time GPU parameters. In this paper, we use the regression method.

Feature	Description
$\|\phi_{\mathcal{S}}\|$	Size of separator potential table
$\|\phi_{\mathcal{X}}\|$	Size of clique potential table
K	Number of threads of thread block
p_a	Threads for arithmetic parallelism

Table 1: Features used in regression models.

5.4 Metrics for Regression Model Quality

Since the trained model is used to optimize GPU parameters, we want to find a regression model whose minimum point is the same as or close to that of the real GPU performance surface. Residual squared sum (RSS) is often used to measure the quality of a regression model's fit to the training data. However, RSS is not a direct metric for the quality of a regression model in our thread allocation optimization problem. In other words, small RSS does not necessarily guarantee a good model.

Our goal is to find optimal estimated parameters K^* and p_a^* for reduction and scattering with given $|\phi_{\mathcal{S}}|$ and $|\phi_{\mathcal{X}}|$. Thus, we propose to use the squared deviance (SD) from the real optimal value as a metric for model training quality, e.g.,

$$SD_r = \sum_{|\phi_{\mathcal{S}}|, |\phi_{\mathcal{X}}|} (T_r(|\phi_{\mathcal{S}}|, |\phi_{\mathcal{X}}|, \hat{K}^*, \hat{p}_a^*) - T_r^*)^2,$$

$$SD_s = \sum_{|\phi_{\mathcal{S}}|, |\phi_{\mathcal{X}}|} (T_s(|\phi_{\mathcal{S}}|, |\phi_{\mathcal{X}}|, \hat{K}^*, \hat{p}_a^*) - T_s^*)^2,$$

where $T_r(\cdot)/T_s(\cdot)$ is the measured GPU reduction/scattering time with respect to junction tree parameters $|\phi_{\mathcal{S}}|$, $|\phi_{\mathcal{X}}|$ and GPU parameters K and p and

$$T^* = \min_{K,p} T(|\phi_{\mathcal{S}}|, |\phi_{\mathcal{X}}|, K, p). \tag{13}$$

$\hat{K}^*$ and $\hat{p}_a^*$ are the optimal parameters obtained from the statistical model with given $|\phi_{\mathcal{S}}|$ and $|\phi_{\mathcal{X}}|$.

Aside from the squared deviance from the optimal value, we also use the *miss rate* (MR) as a measurement of model quality. The miss rate is defined as

$$MR = \frac{\sum_{|\phi_{\mathcal{S}}|, |\phi_{\mathcal{X}}|} \mathbf{1}(\hat{K}^* \neq K^* \text{ or } \hat{p}_a^* \neq p_a^*)}{N}, \tag{14}$$

where $\mathbf{1}(\cdot)$ is the indicator function, K^* and p_a^* are the real optimal GPU parameters for a given $|\phi_{\mathcal{S}}|$ and $|\phi_{\mathcal{X}}|$, and N is the total number of message passings in the training set. Practically, we exhaustively try all the small number of possible GPU configurations on GPUs to find K^* and p_a^*.

6. EXPERIMENTS

In this section, we address the following questions:

- How accurately can a statistical model emulate GPU performance?
- How much GPU execution time can be saved as a result of using statistical model-based parameter optimization compared to manual parameter optimization?

NVIDIA Geforce GTX 460	
# of Processing cores	336
Shared Memory	48K per block
Global Memory	785MB
Memory Bandwidth	90 GB/sec peak
Intel Core 2 Quad CPU	
# of cores	4
Processor Clock	2.5GHz
Cache	8MB
Memory	9 GB

Table 2: Experimental Platforms: GPU and CPU

Step	Method	RSS	SD	MR
Reduce	Lasso($\lambda = 0$)	12.3e5	20.84	12.73%
	Lasso($\lambda = 1se$)	14.9e5	11.3e3	23.62%
	SVR	13.6e5	1.78	18.58%
Scatter	Lasso($\lambda = 0$)	1.31e3	0.92	1.84%
	Lasso($\lambda = 1se$)	1.57e3	0.46	2.01%
	SVR	2.69e3	0.72	2.18%

Table 3: Residual Squared Sum (RSS), Squared Deviance (SD) and Miss Rate (MR) of polynomial and SVR model for GPU reduction and scattering execution time

6.1 Experimental Data and Platforms

Our implementation is tested on a number of BNs[1] from different problem domains, with varying structures and state spaces. In our experiments, we compile a BN into a junction tree offline and then run belief propagation over the junction tree.

As a baseline, we implement a sequential junction tree program on an Intel CPU, whose execution time is comparable to that of the SMILE [15], a widely used C++ software package for BNs inference. As a second baseline, we use the SMILE junction tree algorithm. Detailed information for the CPU and GPU platforms is in Table 2. We have performed sanity checks on the parallel junction tree algorithm to ensure the correctness of our implementation.

6.2 Regression Results

We use both polynomial-lasso regression and support vector regression to fit the data. For the polynomial model, the terms we include in the model are $|\phi_{\mathcal{S}}|$, $|\phi_{\mathcal{X}}|$, K, K^2, p_e, p_e^2, and all the interaction terms of the above-mentioned terms. We set the Lagrangian multiplier λ in (9) to be λ_{min}, which is the value of λ that gives minimum mean cross-validated error, and λ_{1se}, which is the largest value of λ that gives the mean cross-validated error within 1 standard error of minimum.

For SVR, we use a radial basis kernel

$$g(x) = e^{-\gamma\|\mathbf{x}-\mathbf{v}\|^2}, \tag{15}$$

where the kernel bandwidth γ is chosen to be

$$\gamma = \frac{2N}{\sum_{i=1}^{N}(\mathbf{x}_i - \bar{\mathbf{x}})^2}. \tag{16}$$

Other parameters for SVR training are

[1] http://bndg.cs.aau.dk/html/bayesian_networks.html

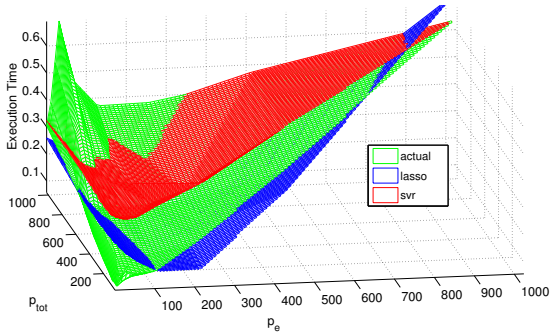


Figure 4: Search space for statistical models (lasso and SVR) versus real GPU execution time with $|\phi_{\mathcal{X}}| = 219,520$ and $|\phi_{\mathcal{S}}| = 14$ and varying GPU parameters, degree of arithmetic parallelism p_a and thread block size K.

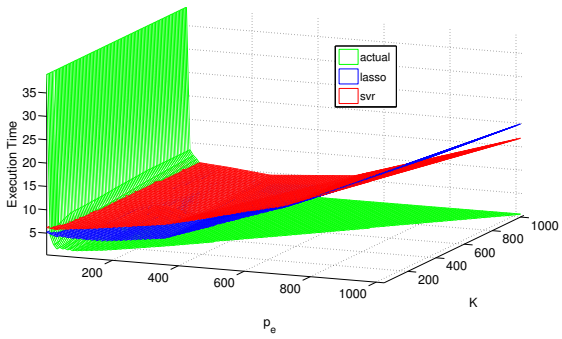


Figure 5: Search space for statistical models (lasso and SVR) versus real GPU execution time with $|\phi_{\mathcal{X}}| = 784,000$ and $|\phi_{\mathcal{S}}| = 24,500$ and varying GPU parameters, degree of arithmetic parallelism p_a and thread block size K.

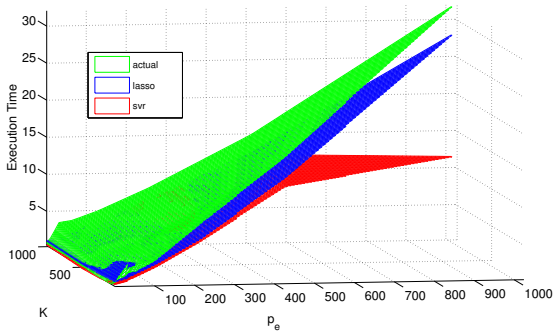
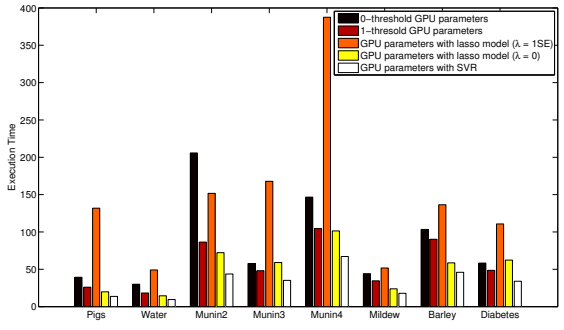


Figure 6: Search space for statistical models (lasso and SVR) versus real GPU execution time with $|\phi_{\mathcal{X}}| = 32,000$ and $|\phi_{\mathcal{S}}| = 320$ and varying GPU parameters, degree of arithmetic parallelism p_a and thread block size K.

		Pigs	Water	Munin2	Munin3	Munin4	Mildew	Barley	Diabetes
Data Statistics	# of JT nodes	368	20	860	904	872	28	36	337
	Ave. CPT size	1,972	173,297	5,653	3,443	16,444	341,651	512,044	32443
	Ave. SPT size	339	713	533	2,099	9,273	26,065	39,318	1845
Manual	0-threshold [ms]	39.29	29.97	205.8	57.65	146.64	44.16	103.38	58.30
	1-threshold [ms]	26.06	18.21	86.4	48.17	104.44	34.56	90.18	48.63
Regression	SVR [ms]	**13.70**	**9.47**	**43.73**	**35.18**	**67.15**	**17.84**	**45.96**	**33.98**
	Lasso ($\lambda = 0$) [ms]	19.8	14.48	72.28	59.01	101.41	23.85	58.63	62.32
	Lasso ($\lambda = 1se$)[ms]	131.8	49.13	151.71	167.8	387.54	51.72	136.41	110.72
Previous	GPU time [ms]	75	52	125	104	342	53	106	94
	CPU time [ms]	51	120	210	137	473	355	974	420
	SMILE [ms]	130	160	140	120	430	280	780	240
Speedup	SVR/CPU Speedup	3.72x	12.68x	4.80x	3.89x	7.04x	19.90x	21.19x	12.36x

Table 4: The GPU execution time (in milliseconds) for different GPU parameter optimization methods, using different junction trees (Pigs, Water, ...) with very different clique potential table (CPT) and separator potential table (SPT) characteristics. The table shows junction tree information (Data Statistics); varying GPU optimization methods (Manual Optimization: 0-threshold and 1-threshold versus Regression based Optimization: SVR and lasso); Previous results [18] and Speedup (current SVR versus previous CPU [18]).

Figure 7: GPU execution times for different parameter optimization methods both manual (0-threshold and 1-threshold) and regression (lasso and SVR). Optimization using SVR is best in all cases.



- The weight for slack terms: $C = 10$
- The ϵ in insensitive loss function: $\epsilon = 0.001$.

In Figure 4, 5 and 6, we plot three examples of statistical models emulating measured GPU execution time T_r for the reduction step. In each figure, $|\phi_{\mathcal{X}}|$ and $|\phi_S|$ are fixed and the GPU parameters K and p_a change. These three examples correspond to the three examples in Figure 2. In Figure 4, both SVR and lasso approximate the GPU time nicely. However, in Figure 5, neither SVR nor lasso approximate the GPU time well, because of an abrupt drop of GPU time when p_a increases. However, a closer look shows that the minimum points of both the SVR and lasso models are located not far from the minimum point of the measured GPU time. In Figure 6, lasso approximates the GPU time better than SVR, but both statistical models' minimum points are close to that of the real GPU execution time surface.

Table 3 shows the residual sum of squares (RSS), the squared deviance (SD) from the optimal value, and the miss rate (MR) of models. From the table, we see that GPU execution time for reduction is harder to emulate than scattering. This suggests that reduction is likely to be the bottleneck of parallel message passing. Also from the table we see that even though SVR does not have the lowest miss rate, its squared deviance is the smallest for reduction. That is to say, even though SVR might have missed some optimal parameters, its parameter choices are not much worse than the optimal. Thus we expect SVR to perform the best in thread allocation, which is illustrated experimentally in Table 4.

6.3 Manual versus Regression-Based GPU Parameter Optimization

In this section we compare different GPU parameter settings and show that regression-based GPU parameter optimization can achieve much better performance than (extensive) manual parameter optimization. We use 0-threshold and 1-threshold parameter selection scheme, suitable for manual optimization, as benchmarks. In the 0-threshold parameter setting, where there is no threshold on the mapping table size, we set $K = 256$ and $q_a = 16$ for all the message passings throughout belief propagation. The 0-threshold parameter setting is perhaps the most straightforward and widely used GPU parameter selection scheme. A programmer just sets a group of reasonable GPU parameters which do not change at run time.

In the 1-threshold parameter selection with threshold $t_1 = 90$, we still keep $K = 256$, but we set p_a in the following way:

$$p_a = \begin{cases} 4 & \text{if} |\mu_{\mathcal{X}_i, \mathcal{S}_{ik}(n)}| \leq 90 \\ 128 & \text{if} |\mu_{\mathcal{X}_i, \mathcal{S}_{ik}(n)}| > 90 \end{cases}, \tag{17}$$

where $\mu_{\mathcal{X}}$ is the mapping table size. The rationale behind this 1-threshold parameter optimization is that in reduction/scattering cases where there is a long mapping table (see Figure 1), there are many opportunities for arithmetic parallelism and therefore we should assign many threads (in (17), $p_a = 128$). In cases where the mapping table is "short", many threads should be assigned to element-wise parallelism, leaving "few" to arithmetic parallelism (in 17, $p_a = 4$). In (17), $t_1 = 90$ is chosen as a reasonable threshold to differentiate short and long mapping tables.

In experiments, we used the SVR and polynomial-lasso models to select GPU parameter for each BP message passing. Results are summarized in Table 4 and Figure 7. On

average over all data sets, we get a speedup of 10.70x (arithmetic average) or 8.68x (geometric average) as compared to that of 3.43x (arithmetic average) or 2.44x (geometric average) achieved previously only using the element-wise parallelism [18]. We also compare the parallel junction tree algorithm with SMILE, the improvement is still significant as shown in Table 4.

We highlight several points in Table 4: 1) Across the columns, we see that parallelism opportunity determines the GPU performance. The GPU in general performs well for data sets that have big cliques (which means more arithmetic parallelism) or big separators (which means more element-wise parallelism). For the data-sets that have neither big cliques nor big separators, such as "Munin2" and "Munin3", the GPU speedup is smaller.

2)Across the different statistical models, we see that SVR is best for all the data sets we have, which coincide with our observation for squared deviation from optimal value and miss rate in Table 3. Lasso($\lambda = 0$) is comparable to the 1-threshold parameter optimization; Lasso($\lambda = 1se$), with a severe punishment on model complexity, performs the worst of all. The difference between the best and worst statistical model parameter settings can be as large as 5-7x.

7. CONCLUSION AND FUTURE WORK

In this paper, we discuss a two-dimensional parallel algorithm for belief propagation over junction trees, and implement the algorithm on a GPU. Due to the great variety in clique and separator sizes in junction trees from applications, the parallel opportunity for both dimensions of parallelism varies. Since the GPU performs best when the concurrency provided by the GPU matches the parallel opportunity in the algorithm, it is necessary to carefully optimize the thread allocation for both dimensions of parallelism as well as for thread blocks. Experiments show a large difference in GPU performance given different thread allocations. Therefore, we use statistical models to approximate the parameter space, for the purpose of searching for optimal parameters. Among the models we used, SVR performs best, and outperforms manual GPU optimization. We show that our approach is an effective way to improve the GPU performance when seeking for fast junction tree belief propagation.

8. REFERENCES

[1] A. Basak, I. Brinster, X. Ma, and O. J. Mengshoel. Accelerating bayesian network parameter learning using hadoop and mapreduce. In *Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, BigMine '12, pages 101–108. ACM, 2012.

[2] R. Bekkerman, M. Bilenko, and J. Langford, editors. *Scaling up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, 2011.

[3] H. Drucker, C. Burges, L. Kaufman, A. Smola, and V. Vapnik. Support vector regression machines. pages 155–161, 1996.

[4] W. D. Hillis and J. G. L. Steele. Data parallel algorithms. pages 1170–1183, 1986.

[5] C. Huang and A. Darwiche. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*, (3):225–263, 1994.

[6] H. Jeon, Y. Xia, and V. K. Prasanna. Parallel exact inference on a CPU-GPGPU heterogenous system. In *Proc. of the 39th International Conference on Parallel Processing*, pages 61–70, 2010.

[7] K. Kask, R. Dechter, and A. Gelfand. BEEM: bucket elimination with external memory. In *Proc. of the 26th Annual Conference on Uncertainty in Artificial Intelligence (UAI-10)*, pages 268–276, 2010.

[8] A. V. Kozlov and J. P. Singh. A parallel Lauritzen-Spiegelhalter algorithm for probabilistic inference. In *Proc. of the 1994 ACM/IEEE conference on Supercomputing*, pages 320–329, 1994.

[9] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, 50(2):157–224, 1988.

[10] M. D. Linderman, R. Bruggner, V. Athalye, T. H. Meng, N. B. Asadi, and G. P. Nolan. High-throughput Bayesian network learning using heterogeneous multicore computers. In *Proc. of the 24th ACM International Conference on Supercomputing*, pages 95–104, 2010.

[11] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. GraphLab: A new framework for parallel machine learning. In *Proc. of the 26th Annual Conference on Uncertainty in Artificial Intelligence (UAI-10)*, pages 340–349, 2010.

[12] O. J. Mengshoel. Understanding the scalability of Bayesian network inference using clique tree growth curves. *Artificial Intelligence*, 174:984–1006, 2010.

[13] V. K. Namasivayam and V. K. Prasanna. Scalable parallel implementation of exact inference in Bayesian networks. In *Proc. of the 12th International Conference on Parallel and Distributed System*, pages 143–150, 2006.

[14] NVIDIA. NVIDIA CUDA C programming guide. `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`.

[15] U. of Pittsburgh. GeNIe and SMILE documentation. `http://genie.sis.pitt.edu/`.

[16] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient computation of sum-products on GPUs through software-managed cache. In *Proc. of the 22nd ACM International Conference on Supercomputing*, pages 309–318, 2008.

[17] Y. Xia and V. K. Prasanna. Node level primitives for parallel exact inference. In *Proc. of the 19th International Symposium on Computer Architechture and High Performance Computing*, pages 221–228, 2007.

[18] L. Zheng, O. J. Mengshoel, and J. Chong. Belief propagation by message passing in junction trees: Computing each message faster using GPU parallelization. In *Proc. of the 27th Conference in Uncertainty in Artificial Intelligence (UAI-11)*, pages 822–830, Barcelona, Spain, 2011.