A Parallel Irregular Wavefront Algorithm for Importance Sampling of Probabilistic Networks on GPU

Haohai Yu, Robert van Engelen
Department of Computer Science, Florida State University
Tallahassee, FL32306, USA
hy06cs.fsu.edu, rvanengelen@fsu.edu

***Abstract*—Importance sampling is a widely-used method for probabilistic inference with Bayesian probabilistic networks. Importance sampling is relatively easy to parallelize and parallel GPU implementations yield significant speedups over single-CPU implementations. However, because of physical limitations of GPU memory size and bandwidth, the maximum speedups that can be achieved are bounded by the high data transfer requirements of these algorithms. In this paper, we propose and evaluate a new parallel irregular wavefront algorithm for importance sampling of probabilistic networks on GPU. Performance results show that the proposed parallel algorithm achieves greater speedups due to the optimal local memory access compared to simple parallel GPU implementations.**

***Keywords*-Probabilistic graph models; Bayesian inference; Importance sampling; GPGPU parallelization.**

I. Introduction

Probabilistic networks are probabilistic graph models for probabilistic inference with uncertain data [13]. A probabilistic graph model efficiently models a joint probability distribution over a set of random variables and the probabilistic conditional (in)dependence relationships between them. Real-world applications range from computational biology and medicine to classification problems in information retrieval [1], [2], [5], [11]. Because probabilistic networks can solve complex decision problems under uncertainty, they form the basis of many decisions support systems.

More specifically, Bayesian probabilistic networks [10] provide a mathematically sound formalism by generalizing Bayes' theorem to answer probabilistic queries on complex problems by probabilistic inference. Probabilistic inference amounts to the computation of the posterior distribution of the random variables given some evidence. So-called exact methods compute the posterior distribution exactly. Approximate inference methods, such as stochastic sampling, rely on convergence of the approximation to the exact solution given sufficient time. Stochastic methods such as importance sampling are among the most widely-used approximate inference methods [13]. Examples are SIS [14], AIS-BN [4], DIS [9], RIS [16] and EPIS-BN [17].

Importance sampling offers several benefits for real-time probabilistic inference [6], including parallel execution on multicore and GPU machines to obtain significant speedups. However, because of physical limitations of the GPU memory size and bandwidth, the maximum speedups that can be achieved are strictly bounded by the high amount of data transferred in the parallel reduction step in these algorithms, especially for larger networks with hundreds of nodes.

In this paper we propose a new parallel wavefront algorithm for importance sampling of probabilistic networks on GPU. Inspired by the classic wavefront method [15], the algorithm uses multiple parallel threads that follow the dependence structure of the network as a wavefront, starting at the root of the network. By contrast to [15], the parallel access pattern is irregular and follows the probabilistic network branching patterns. To obtain speed improvements data has to be organized to ensure optimal data access in local memory, which is challenging given the randomness of the stochastic importance sampling method.

The remainder of this paper is organized as follows. Section II introduces Bayesian networks and importance sampling. Our parallel irregular wavefront sampling method is discussed in detail in section III. Section IV presents the experimental results on real-world networks compared to other implementations. We conclude with a summary and discussion of future work in section V.

II. Overview

This section introduces Bayesian networks, probabilistic inference, and parallel importance sampling.

A. Bayesian Networks

A *Bayesian network* is a graphical model of a *joint probability distribution* (JPD) over a set of *random variables* $\mathbf{V}$ represented by a *directed acyclic graph* (DAG) G.

Definition 1: A *Bayesian network* $BN = (G, \Pr)$ is a DAG $G = (\mathbf{V}, \mathbf{A})$ with vertices $\mathbf{V}$ and arcs $\mathbf{A}$, $\mathbf{A} \subseteq \mathbf{V} \times \mathbf{V}$. The JPD over the discrete random variables (vertices) $\mathbf{V}$ defined by

$$\Pr(\mathbf{V}) = \prod_{V \in \mathbf{V}} \Pr(V \mid \pi(V)) \quad ,$$

where the set of parents of a vertex V is $\pi(V)$. The *conditional probability tables* (CPT) of the BN assign values to $\Pr(V \mid \pi(V))$ for all $V \in \mathbf{V}$.

Published in the proceedings of IEEE HPCC 2011.

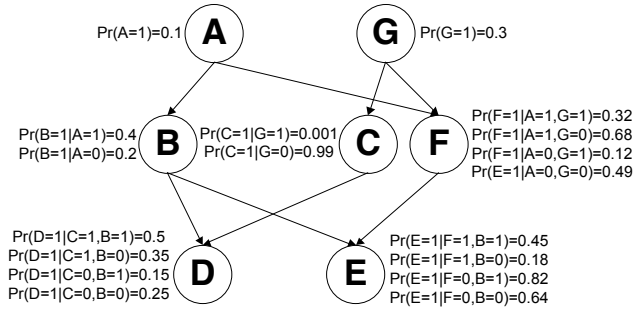

Figure 1. Example Bayesian Network

Figure 1 illustrates a Bayesian network[1] with $(0, 1)$-valued variables A to G. Each vertex represents a variable with arcs denoting (conditional) probabilistic influences. The set of observed variables is $\mathbf{E} \subseteq \mathbf{V}$ and a configuration $\mathbf{e}$ of $\mathbf{E}$ is called *evidence*. *Probabilistic inference* is the calculation of the *posterior marginalization* $\Pr(\mathbf{x} \mid \mathbf{e})$ given evidence $\mathbf{e}$ for a select configuration $\mathbf{x}$ of variables $\mathbf{X} \subseteq \mathbf{V}$.

For example, suppose $\mathsf{B}{=}1$ is evidence. Then, $\Pr(\mathsf{A}{=}1|\mathsf{B}{=}1) = \Pr(\mathsf{B}{=}1|\mathsf{A}{=}1)\Pr(\mathsf{A}{=}1)/\Pr(\mathsf{B}{=}1) = 0.18$ where $\Pr(\mathsf{B}{=}1) = \Pr(\mathsf{B}{=}1|\mathsf{A}{=}0)\Pr(\mathsf{A}{=}0) + \Pr(\mathsf{B}{=}1|\mathsf{A}{=}1)\Pr(\mathsf{A}{=}1) = 0.22$ by marginalization. This is a simple case. In general, the goal is to efficiently compute the posterior marginalization for any subset of variables given some evidence. For more details, we refer to [13].

A Directed Acyclic Graph G can be levelized [3], thereby defining the level of each variable $V \in \mathbf{V}$ as:

$$\mathcal{L}(V) \equiv \begin{cases} 0 & \text{if } \pi(V) = \varnothing \\ \max\{\mathcal{L}(V') \mid V' \in \pi(V)\} + 1 & \text{otherwise} \end{cases}$$

We define $\mathcal{L}_k(G) = \{V \in \mathbf{V} \mid \mathcal{L}(V) = k\}$ the set of variables at level k in G. For example, in Figure 1, $\mathcal{L}_0(G){=}\{\mathsf{A}, \mathsf{G}\}$, $\mathcal{L}_1(G){=}\{\mathsf{B}, \mathsf{C}, \mathsf{F}\}$, and $\mathcal{L}_2(G){=}\{\mathsf{D}, \mathsf{E}\}$.

B. Importance Sampling

Importance sampling is an efficient approximation and an alternative to exact numerical integration. It is the *de-facto* choice to approximate complex problems, including probabilistic networks. Importance sampling estimates the prior $\Pr$ and posterior $\Pr_{\mathbf{e}}$ from a Bayesian network by sampling its JPD.

Consider the problem of approximating the integral

$$\mathbf{E}[g(X) \mid p] = \int_{\Omega} g(x)\, p(x)\, dx \quad , \tag{1}$$

where $g(X)$ is an integrable function of X over domain Ω and $p(X)$ is a probability density of X over Ω.

[1]Random variables are written in upper case, with lower case for values or configurations of variables. Boldface is used for sets (e.g. variables $\mathbf{V}$ and a configuration $\mathbf{v}$). The set of all configurations of variables $\mathbf{X} \subseteq \mathbf{V}$ is denoted by $\mathit{Cfg}(\mathbf{X})$.

In real applications, it may be difficult to sample from p. The basic idea of importance sampling is to draw from a distribution other than p, say f, in certain way to reduce the variance of the sampling. Importance sampling [12] approaches this problem by rewriting Eq. (1) into

$$\mathbf{E}[g(X) \mid p] = \int_{\Omega} g(x) \frac{p(x)}{f(x)} f(x)\, dx \tag{2}$$

with $f(X)$ a probability distribution of X over Ω, called the *importance function*. It should be easy to sample from $f(X)$ so that we can generate samples $x_1, \ldots, x_N$ from $f(X)$ and use the following sample-mean formula

$$\hat{g}_N = \frac{1}{N} \sum_{i=1}^{N} g(x_i)\, w(x_i) \quad , \tag{3}$$

with importance weights $w(x_i) = \frac{p(x_i)}{f(x_i)}$.

For probabilistic inference on Bayesian networks it turns out to be practically impossible to define an importance function that works well in all cases [18]. Rather, an importance function family $\mathcal{F}$ is used where functions are typically updated during sampling to improve accuracy. Sampling algorithms differ in the choice of $\mathcal{F}$. Basically, forward sampling traverses the graph from the roots down instantiating variables V with values based on a random draw from the probability distribution over the variable's values as defined by the importance function $\mathcal{F}(V \mid \pi(V) \setminus \mathbf{E})$ over V and its parent configuration $\pi(V) \setminus \mathbf{E}$ (that excludes the set evidence variables $\mathbf{E}$). The order of vertices visited is the sampling order δ, which is consistent with the topological order of the Bayesian network.

Algorithm 1 defines the importance sampling process. Samples are produced in loop 1. For each sample $\mathbf{x}$ that is generated in loop 2, the posterior sum estimate p is updated at lines 3 and 4, which approximates the exact probability $p[V] \approx \Pr(V \mid \mathbf{e})$.

Advanced importance sampling algorithms such as SIS and AIS-BN use *CPT learning* to update the importance function $\mathcal{F}$ at line 5. The details of CPT learning methods are beyond the scope of this paper. The interested reader is referred to [4], [14] and [16].

Figure 2 illustrates *likelihood weighting* sampling applied to the Bayesian network of Figure 1 using Algorithm 1. Likelihood weighting is an importance sampling algorithm that uses the Bayesian network's prior JPD for the importance functions $\mathcal{F}$ and the update step at line 5 is not performed. The weight of a likelihood sample $\mathbf{x}$ is $1/\Pr(\mathbf{e} \mid \mathbf{x})$. Suppose for example that $\mathsf{E}{=}1$ is observed. Each step in Figure 2 shows the variable being sampled in sampling order $\delta = [\mathsf{A}, \mathsf{G}, \mathsf{B}, \mathsf{C}, \mathsf{F}, \mathsf{D}]$ (evidence E is skipped). Dashed arcs represent the network's graph dependencies that induce the sampling order δ. The weight of the sample $\mathbf{x} = [\mathsf{A}{=}0, \mathsf{B}{=}1, \mathsf{C}{=}1, \mathsf{D}{=}0, \mathsf{F}{=}0, \mathsf{G}{=}1]$ is $1/\Pr(\mathsf{E}{=}1 \mid \mathsf{B}{=}1, \mathsf{F}{=}0) = 0.82^{-1} \approx 1.22$.

Procedure `ImportanceSampling` $(\mathbf{e}, \delta, N)$
Input: evidence $\mathbf{e}$ for vertices $\mathbf{E}$, sampling order δ, number of samples N
Output: posterior sum estimate $p : V \rightarrow [0, 1]$
Initialize $\mathcal{F}$
$\mathbf{x} \leftarrow \mathbf{e}$
foreach $V \in \mathbf{V}$ **do** $p[V] \leftarrow 0$
1 **for** $i = 1$ **to** N **do**
2 **foreach** $V \in \mathbf{V} \backslash \mathbf{E}$ *in order* δ **do**
  `Sample` $(V, \mathbf{x}, \mathcal{F})$
  end
3 $w \leftarrow \frac{\prod_{V \in \mathbf{V}} \Pr(\mathbf{x}[V] \mid \mathbf{x}[\pi(V)])}{\prod_{V \in \mathbf{V}} \mathcal{F}(\mathbf{x}[V] \mid \mathbf{x}[\pi(V) \backslash \mathbf{E}])}$
4 **foreach** $V \in \mathbf{V} \backslash \mathbf{E}$ **do** $p[V] \leftarrow p[V] + w$
5 **if** $\mathcal{F}$ *needs updates* **then** Update($\mathcal{F}$)
end
Normalize p

Procedure `Sample` $(V, \mathbf{x}, \mathcal{F})$
Input: the variable V being sampled, the partial sample $\mathbf{x}$, importance function $\mathcal{F}$
Output: sample $\mathbf{x}[V] = v$ is a value v for V
$s \leftarrow 0$
pick $r \in (0, 1]$ uniform random
6 **foreach** $v \in Cfg(V)$ **do**
7 $s \leftarrow s + \mathcal{F}(v \mid \mathbf{x}[\pi(V) \backslash \mathbf{E}])$
  if $s \geq r$ **then**
  $\mathbf{x}[V] \leftarrow v$
  break
  end
end

Algorithm 1: Importance Sampling Procedure

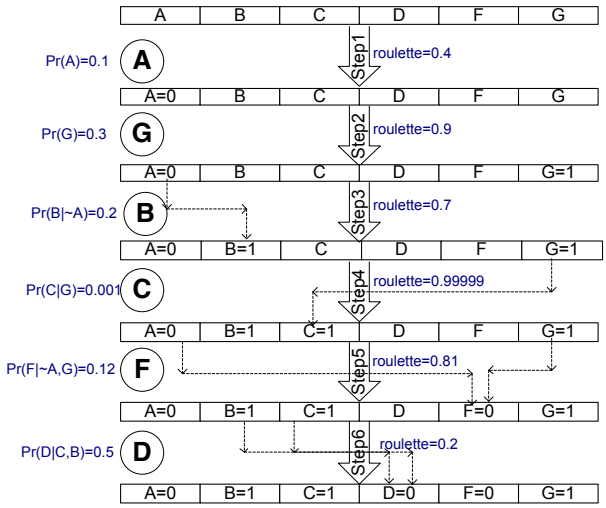

Figure 2. Sequential Likelihood Sampling

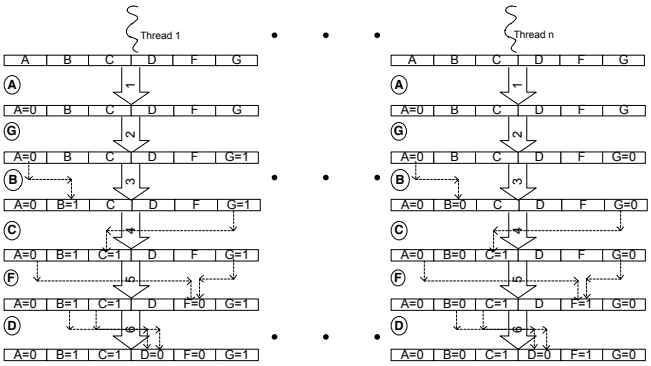

Figure 3. Simple Parallel Sampling

A straight-forward approach to speed up importance sampling is to parallelize the outer loop (line 1 in Algorithm 1). Each processor or parallel thread samples independently a privatized sample $\mathbf{x}$ to update a privatized sum estimate p, see Figure 3. The final results for p are reduced and normalized over all threads at the end Algorithm 1 line 4. This process repeats as long as the number of samples N to take exceeds the number of threads available. Hundreds of thousands of samples are typical in real-world applications. Independent random number sequences have to be used [7] to ensure threads generate truly independent samples.

For reasonably large Bayesian networks with hundreds of random variables the memory bandwidth requirements quickly become excessive when running many threads, causing memory contention in shared memory systems. That is, the parallel speedup is memory bound. The basic processing unit in GPU is the processing element (PE). Each PE has its own exclusive private memory. Physically adjacent PEs are grouped together as a compute unit (CU). PEs in the same CU share local memories, memory channels and a threading management unit. Global memory, constant memory, and texture memory are shared by all PEs. Global memory and constant memory may or may not be cached. For GPU, maintaining cache consistency for global memory is not practical due to the large number of PEs. Only constant memory (read-only), texture memory, and instructions are cached. Local memory (also known as scratch memory or local data share) works as software-controlled cache. For most GPUs its speed is only secondary to registers. The size of local memory is limited. For example, ATI $Radeon^{TM}$ HD 5870 only has 32KB local memory for each CU.

Thus, the suitability of importance sampling for GPU parallelization is limited. At least the samples $\mathbf{x}$ should be privatized and mapped to local memories since these arrays are constantly read and written. Because the size of local memory is limited and is shared by the entire thread group in the CU of a GPU, implementations are either forced to limit the number of threads per group or constantly move the samples $\mathbf{x}$ between global and local memory. Obviously, none of these choices is attractive.

III. Parallel Irregular Wavefront Sampling

The proposed parallel irregular wavefront algorithm parallelizes the sampling algorithm over irregular data to benefit local memory access.

A. Wavefront Parallelization

The classic wavefront method [15] parallelizes a loop nest by propagating computations in parallel across a hyperplane, i.e. with parallelism across the front of the wave. More specifically, a *unimodular loop transformation*, known as *loop skewing*, can be applied to parallelize loop nests over regular array data structures with linear access patterns. However, the importance sampling procedure does not admit this type of transformation directly to Algorithm 1 loops 1 and 2, because of the irregular data access patterns and the sampling order δ that forces loop 2 to be serialized. The goal is to parallelize Algorithm 1 at line 2 by splitting the loop into a series of parallel wavefronts, where the size of each parallel wavefront is determined by the irregular network structure's dependences. The result is that multiple GPU threads co-operate to produce a sample stored in local memory (managed cache).

We define the following requirement to remove the sample order restriction:

Definition 2: Given a Bayesian network $BN = (G, \Pr)$ with graph $G = (\mathbf{V}, \mathbf{A})$, the set of random variables that must be sampled ahead of a variable $V \in \mathbf{V}$ are its ancestors modulo the evidence vertices $\mathbf{E}$:

$$D(V) = (\pi(V) \backslash \mathbf{E}) \cup \left(\bigcup_{V' \in \pi(V), V' \notin \mathbf{E}} D(V') \right) \quad .$$

Now, we define the *spontaneously independent sample set* (SISS) S such that $\forall V \in S$, $D(V) \cap S = \varnothing$.

A SISS expresses the maximum amount of parallelism available given the network connectivity under evidence $\mathbf{E}$, since all variables in a SISS can be independently sampled. Since evidence $\mathbf{E}$ can "break" the dependence chain, SISS is not identical to level sets. The level set $\mathcal{L}_k(G)$ forms a subset of SISS given evidence $\mathbf{E}$. That is, for all $V \in \mathcal{L}_k(G)$ we have that $D(V) \cap \mathcal{L}_k(G) = \varnothing$.

It remains to factor in the details of importance sampling algorithms with respect to the choice of importance function $\mathcal{F}$ used in the `Sample` function in Algorithm 1 and consider the complexity of evaluating Algorithm 1 line 7. For a common importance function family $\mathcal{F}$ that represents a sampling distribution it is preferable to avoid unnecessary synchronization and this means to sample in parallel across the levels sets rather than the less restrictive SISS that exposes more parallelism.

Algorithm 2 defines the parallel irregular wavefront algorithm iterating over levels sets in loop 1 and executing the sampling in parallel at line 2. The barrier synchronization at line 3 is needed to ensure GPU memory consistency after each iteration. Line 4 is a parallel version of the weight computation by parallel reductions of the product operations in the numerator and denominator of the fraction. CPT learning to update F is done optionally at line 5.

Procedure `WavefrontSampling` $(\mathbf{e}, \mathcal{L}, N)$
Input: evidence $\mathbf{e}$ for vertices $\mathbf{E}$, level sets $\mathcal{L}_k(G)$, number of samples N
Output: posterior sum estimate $p : V \rightarrow [0, 1]$
$\mathbf{x} \leftarrow \mathbf{e}$
forall $V \in \mathbf{V}$ **do** $p[V] \leftarrow 0$
for $i = 1$ **to** N **do**
1 **for** $k = 0$ **to** $|\mathcal{L}(G)| - 1$ **do**
2 **forall** $V \in \mathcal{L}_k(G)$ **do** `Sample` $(V, \mathbf{x}, \mathcal{F})$
3 barrier()
end
4 Parallel reductions for products:
$w \leftarrow \frac{\prod_{V \in \mathbf{V}} \Pr(\mathbf{x}[V] \mid \mathbf{x}[\pi(V)])}{\prod_{V \in \mathbf{V}} \mathcal{F}(\mathbf{x}[V] \mid \mathbf{x}[\pi(V) \backslash \mathbf{E}])}$
barrier()
forall $V \in \mathbf{V} \backslash \mathbf{E}$ **do** $p[V] \leftarrow p[V] + w$
5 **if** $\mathcal{F}$ *needs updates* **then** Update($\mathcal{F}$)
end
6 Parallel reduction to normalize p

Algorithm 2: Parallel Wavefront Sampling

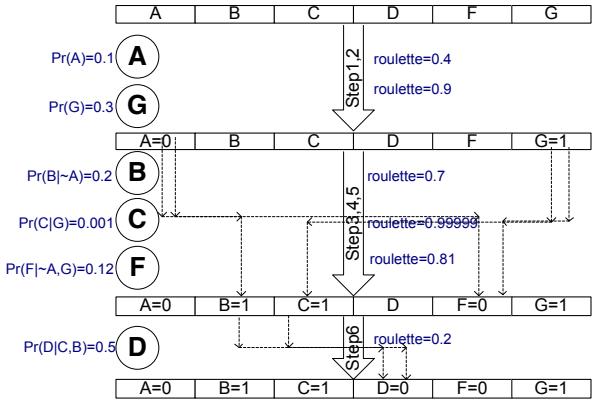

Figure 4. Parallel Wavefront Likelihood Sampling

It is important to point out that the modifications made in Algorithm 2 do not affect the sampling distribution. In other words, the sampling distribution generated at line 5 of Algorithm 2 is the same as that of Algorithm 1.

Figure 4 illustrates the parallel irregular wavefront algorithm Algorithm 2 executed on the example Bayesian network of Figure 1 with evidence E=1. Here we use the level sets in the parallel loop 1, which are $\mathcal{L}_0(G)$={A, G}, $\mathcal{L}_1(G)$={B, C, F}, and $\mathcal{L}_2(G)$={D, E}. The wavefront approach exploits parallelism in each level set which is high for larger Bayesian networks with many variables that can be sampled independently.

B. GPU Memory Access Optimizations

The importance functions $\mathcal{F}$ are tables that are typically stored in memory in the form of arrays. The size of these arrays combined is in the range of mega-bytes for real-world Bayesian networks, which means that the data cannot be mapped to constant memory or to local memory of a GPU. Therefore, repeatedly reading the $\mathcal{F}$ tables by multiple threads in a thread group is expensive since the data is not cached and has to be fetched from global memory.

The CPTs of the Bayesian network are stored as arrays in global memory. An alternative storage is the read-only texture memory which is cached in GPU and in fact faster to access. However, texture memory requires image addressing (every read/write must be aligned to a base-4 address). In our experiments we found that the cost of alignment was greater than the benefits, mainly because reading the CPT and $\mathcal{F}$ tables from memory is rather random and unpredictable.

To reduce the memory requirements for loading the $\mathcal{F}$ tables, we exploit properties of the probability distribution from which we sample. This gives us a probabilistic "oracle" to predict the number of memory reads to fetch the $\mathcal{F}$ tables. This idea is best illustrated by an example. Consider variable C in Figure 1 and suppose $\mathsf{G}{=}0$ is observed as evidence. The posterior marginal probability of C is given by $\Pr(\mathsf{C}{=}0 \mid \mathsf{G}{=}0) = 0.01$ and $\Pr(\mathsf{C}{=}1 \mid \mathsf{G}{=}0) = 0.99$. From procedure `Sample` in Algorithm 1 we observe that the expected number of memory read operations to fetch $\mathcal{F}(\mathsf{C}{=}0 \mid \pi(V)\backslash\mathbf{E})$ first is $0.01 \times 1 + 0.99 \times 2 = 1.99$ times. The expected number of memory read operations to fetch $\mathcal{F}(\mathsf{C}{=}1 \mid \pi(V)\backslash\mathbf{E})$ first is $0.99 \times 1 + 0.01 \times 2 = 1.01$ times. Hence, we can predict the memory read operation frequency, which is probabilistically dependent on the order of the variable's values considered in the loop 6 of Algorithm 1. Clearly, this strategy works best in Bayesian networks with more extreme probability distributions.

We define the mapping $\beta_V : \mathit{Cfg}(V) \rightarrow \mathit{Cfg}(V)$ as the incrementally-sorted values $\mathit{Cfg}(V)$ according to V's posterior marginal probabilities. However, we do not need to perform a complex computation for V's posterior marginal probabilities to determine this order, because the approximation from the accumulated p values suffices. The mapping β_V is stored in local memory, otherwise the cost of accessing β_V may overcome the benefits this prediction provides.

With respect to the storing the graph G of the network, it is best represented as an adjacency matrix to avoid pointer chasing and indirect addressing on GPU. G can be stored in local memory or in constant memory of the GPU.

IV. Results

We implemented the proposed parallel irregular wavefront algorithm with OpenCL on several GPU machines and we used AIS-BN and SIS CPT learning methods, which are advanced techniques that are frequently used for probabilistic inference.

Table I
Bayesian Networks Used in Experiments

	ANDES	CPCS360B	CPCS422B	CPCS179
$\lvert\mathbf{V}\rvert$	223	360	422	179
$\lvert\mathbf{A}\rvert$	338	729	867	239
$\lvert\mathbf{E}\rvert$	25	20	50	10

A. Test Cases and Machine Configurations

Four real-world Bayesian Networks were used in the experiments, ANDES [5] and three versions of CPCS [11]. For each Bayesian network, 50 test cases were generated by selecting evidence vertices and values randomly. Table I lists the characteristics of each Bayesian network used for testing.

Table II lists the hardware and software configurations of the experimental setup. The *Cache* column represents the cache between CPU and global (main) memory. ATI HD 5800 is GPU has 20 CUs (SIMD engines), each CU has 16 Stream Cores and each Stream Core has 5 PEs. 4 PEs are for arithmetic operations and 1 PE is for ultra-arithmetic operations (functions like exp, log etc). Each CU supports from 64 threads to 256 threads. The *Stream SDK2.2* is AMD/ATI's standard development kit for GPGPU. It supports OpenCL1.1 and has a performance profiling tool called *Stream Profiling*.

For each GPU and CPU implementation also an "exponential version" that uses exp and log replacements to reduce the round-off error is also implemented and tested. This mitigates the problem using single floating pointer precision in GPU. Basically, the right-hand side of the assignment line 3 in Algorithm 1 can simply be replaced by:

$$\exp\Big(\sum_{V\in\mathbf{V}} \log(\Pr(\mathbf{x}[V] \mid \mathbf{x}[\pi(V)])) - \log(\mathcal{F}(\mathbf{x}[V], \mathbf{x}[\pi(V)\backslash\mathbf{E}]))\Big) \quad .$$

We use AIS-BN and SIS with four versions of each represented as AISCPU, AISEXPCPU, AISGPU, AISEXPGPU, SISCPU, SISEXPCPU, SISGPU and SISEXPGPU. All CPU programs compared against are compiled with "-O3" option. Because the ATI HD 5800 does not support double floating point, single precision is used in GPU versions and double precision for CPU. In each test case, 120,000 samples were sampled and half of them are used for training the importance function of AIS-BN and SIS.

Importance sampling requires samples to be independently drawn with respect to each other. Each thread's $rand()$ should generate a independent random number sequence. We used the 48bits parallel Linear Congruential Generator of SPRNG [8] for the GPU. SPRNG parallelizes LCG through parameterization [7]. It supports more than 7,000 different random number sequences and each sequence's period is 2^{48}. In the experiments the maximum number of threads is 5120 and the number of samples for

Table II
SOFTWARE AND HARDWARE CONFIGURATIONS

Target	Compiler	OS	Clock	Local Mem	Global Mem	Constant Mem	Cache
ATI HD 5800	Stream SDK2.2, VS9	Windows 7	765MHz	32KB	512 MB	64KB	NA
AMD Athlon, 4200+	g++ 4.4.3	Linux 2.6.32-27	1GHz	NA	2GB	NA	512KB
Intel Core 2, 6700	g++ 4.4.3	Linux 2.6.32-24	2.66GHz	NA	2GB	NA	4MB

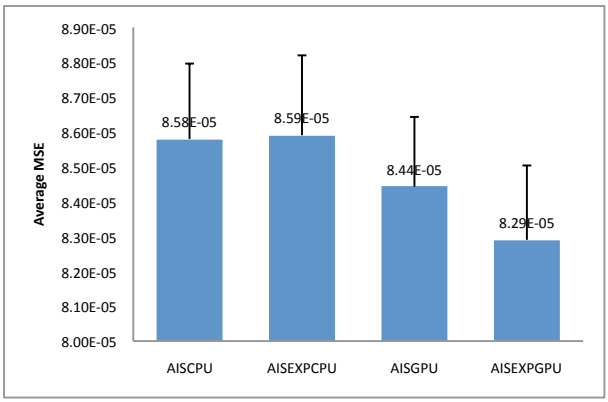

Figure 5. Average MSE of AIS-BN Test Cases

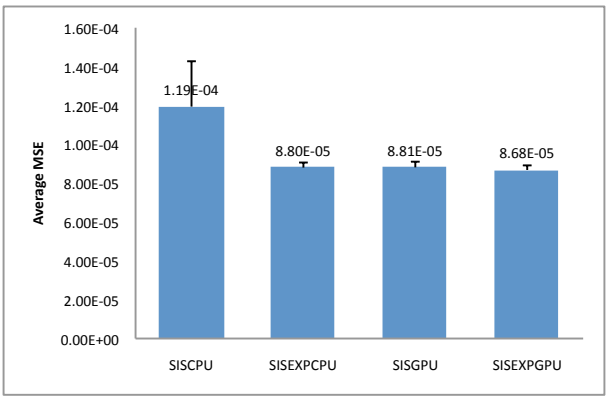

Figure 6. Average MSE of SIS Test Cases

each test case is 120,000. These LCG generators satisfy our requirements.

B. Accuracy Results

The *MSE* (mean squared error) metric was used to measure the error of the importance sampling results compared to the exact solution:

$$MSE = \sqrt{\frac{1}{\sum_{X_i \in \mathbf{X}} \|X_i\|} \sum_{X_i \in \mathbf{X}} \sum_{j=1}^{\|X_i\|} (\Pr'_{\mathbf{e}}(x_{ij}) - \Pr_{\mathbf{e}}(x_{ij}))^2}$$

Fig 5 shows the average MSE results for four versions based on AIS-BN. and Fig 6 shows the average MSE results for four versions based on SIS. Clearly the accuracies of GPU versions is quite close to those of CPU ones.

C. Speedup Results

Table III lists the performance of the parallel irregular wavefront algorithm on GPU (Wave-GPU), the simple parallel method on GPU (Simple-GPU), and CPU-based sampling programs. The third, fourth, fifth and sixth columns show the average sampling times of 50 test cases for each version on AMD Athlon 4200+, Intel Core2 6700 and ATI HD 5800 (both for simple parallel and wavefront), and the measurement is second. The last three columns demonstrate the speedup for wavefront over the CPU versions, and the wavefront over the simple parallel method.

In the best case, the wavefront algorithm achieves a 60-fold speedup compared to the Intel CPU and a 103 fold speedup compared to the AMD CPU. Even in the worst case, wavefront is still 18 times faster than Intel CPU and 27 times faster than AMD CPU. The wavefront algorithm outperforms the simple parallel method by 2.77 times in best cases and 1.38 times in the worst case.

The main reason for differences in the speedup for ANDES is that ANDES is a Bayesian network with a high level of determinisms and consequently many samples are inconsistent. Sequential algorithms sample variables one by one and can stop at the first node whose value is inconsistent. Threads in GPU detect the inconsistent samples not earlier than at the point of synchronization, so more sampling time is lost.

It may be surprising to find that the GPU's exponential algorithm's sampling time is quite close to the normal sampling time. One of the five PEs of the ATI HD 5800 is dedicated to ultra-arithmetic operations. Thus, exp() and log() functions are implemented by hardware on the ATI HD 5800.

For the GPU wavefront experiments in Table III, both the number of concurrent samples and thread group size are set to the maximum. To verify the effects of these two factors, we varied the number of concurrent samples and thread group size on CPCS422B's AISEXPGPU test cases. Figure 7 shows the relation between the average sampling time and the number of concurrent samples, while thread group size is fixed to 256. The x-axis represents the number of concurrent samples ranging from 1 to 16 (16 is the maximum number of concurrent samples for CPCS422B). And the y-axis is the average sampling time measured in second.

Figure 8 shows the thread group size's effect on the performance of wavefront algorithm, while the number of concurrent samples is set to the maximum. The x-axis is

Table III
AVERAGE SAMPLING TIMES AND PARALLEL SPEEDUPS

Bayesian Network	Sampling Method	Sampling Time				Speedup Ratio		
		AMD Athlon	Intel Core 2	Simple-GPU ATI HD	Wave-GPU ATI HD	Wave-GPU vs. AMD	Wave-GPU vs. Intel	Wave-GPU vs. Simple-GPU
CPCS422B	AISEXP	203.38	120.15	6.53	2.78	73.06	43.16	2.35
	AIS	135.58	82.44	6.49	2.79	48.68	29.60	2.33
	SISEXP	219.04	128.95	6.54	2.79	78.54	46.24	2.34
	SIS	150.76	90.85	6.52	2.79	53.97	32.53	2.37
CPCS360B	AISEXP	170.74	100.66	4.96	1.79	95.39	56.24	**2.77**
	AIS	119.41	72.49	4.94	1.79	66.71	40.50	2.76
	SISEXP	185.09	108.27	4.95	1.79	**103.36**	**60.46**	**2.77**
	SIS	131.80	79.78	4.93	1.79	73.55	44.52	2.75
CPCS179	AISEXP	72.03	43.41	2.03	1.22	58.83	35.45	1.66
	AIS	47.02	29.96	2.01	1.22	38.42	24.48	1.65
	SISEXP	80.00	47.92	2.02	1.21	66.06	39.57	1.67
	SIS	54.49	34.15	2.00	1.21	45.02	28.21	1.65
ANDES	AISEXP	98.94	60.12	2.89	2.09	47.35	28.77	**1.38**
	AIS	55.59	36.90	2.90	2.04	**27.25**	**18.09**	1.42
	SISEXP	110.25	66.98	2.89	2.09	52.76	32.06	**1.38**
	SIS	65.31	42.35	2.90	2.04	31.98	20.74	1.42

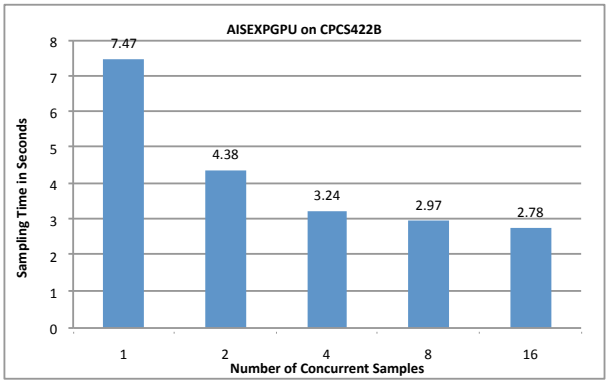

Figure 7. Average Sampling Time for AISEXP on CPCS422B with Varying Number of Concurrent Samples

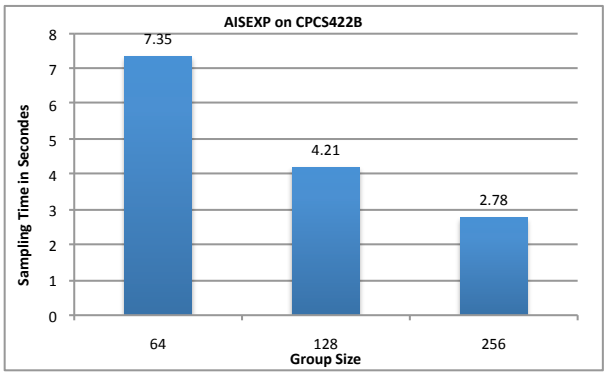

Figure 8. Average Sampling Time for AISEXP on CPCS422B with Varying Thread Group Sizes and 16 Concurrent Samples

the thread group size ranged from 64 to 256 (256 is the maximum thread group size for ATI HD 5800). And the y-axis is the average sampling time measured in second. It is interesting to notice that the speedups between cases of 128 (group size) and 64 (group size), and between cases of 256 (group size) and 128 (group size) are different (1.74 for 128 vs 64 and 1.51 for 256 vs 128). We believe the reason is that although more parallel jobs can help to increase the utilization rate of GPU and to hide the memory latency, those threads also increase the chance of memory read/write conflicts. So we should not expect a steady speedup when the number of threads increases.

To investigate the wavefront algorithm's internal behavior, we used the ATI Stream Profiling tool to profile GPU kernel's key characteristics. Table IV shows the profiling results. The column *IO* is the ratio between Host-GPU IO time and total sampling time. Because wavefront algorithm only transfers data (CPTs, results etc) between host and GPU at initialization and finalization stages, and the size of

Table IV
PERFORMANCE PROFILING RESULTS

BN	IO	ALUBusy	CacheHit	LDSStallALU
CPCS422B	0.47%	36.67%	27.61%	16.36%
CPCS360B	0.16%	35.89%	26.47%	15.10%
CPCS179	0.24%	30.35%	10.77%	12.52%
ANDES	0.13%	24.70%	17.60%	8.10%

data is only a few megabytes, so the Host-GPU IO time is trivial. *ALUBusy* is the percentage of GPU time that ALU instructions are processed. *CacheHit* is the percentage of cache hit. This value is low, because only constant memory fetches are cached. *LDSStallALU* is the percentage of ALU being stalled by local memory read/write operations. From table IV, it is easy to conclude that wavefront algorithm is still a memory bound algorithm.

To evaluate the effect of the memory optimizations discussed in Section III-B, Table V compares the size of global memory fetches (in KB) with and without the optimizations for four random selected test cases. The optimizations are

Table V
GLOBAL MEMORY FETCHES

BN	Optimized	Not Optimized	Reduced
CPCS422B	10906326	16531479	34.03%
CPCS360B	7282385	12292421	40.76%
CPCS179	3831576	3930213	2.51%
ANDES	4651554	4785843	2.81%

expected to perform better with extreme distributions of Bayesian networks, thus CPCS422B and CPCS360B benefit more from the optimization than CPCS179 and ANDES. For CPCS422B and CPCS360B we observed up to 15% performance improvement.

V. CONCLUSIONS

This paper proposed a parallel irregular wavefront algorithm for importance sampling on GPU to reduce the memory bandwidth requirements by leveraging the characteristics of Bayesian networks and the sampling procedure. The experimental results demonstrated the increased speedup of the wavefront approach on GPU compared to the simple parallel method on GPU and compared to CPU implementations. The results are verified with real-world Bayesian networks of reasonably large sizes, and the use of the advanced but common importance sampling methods AIS-BN and SIS. The approach is general and not dependent on the importance sampling method used. Hence other importance sampling methods will benefit from the parallel irregular wavefront algorithm.

The current implementation is based on OpenCL and ATI's GPU. Nvidia's CUDA is another important GPGPU platform. We plan to implement the parallel irregular wavefront algorithm on Nvidia's GPU with CUDA in the future.

REFERENCES

[1] S. Andreassen, R. Hovorka, J. Benn, K. G. Olesen, and E. R. Carson. A model-based approach to insulin adjustment. In M. Stefanelli, A. Hasman, M. Fieschi, and J. Talmon, editors, *Proceedings of the Third Conference on Artificial Intelligence in Medicine*, pages 239–248. Springer-Verlag, 1991.

[2] S. Andreassen, F. Jensen, S. Andersen, B. Falck, U. Kjærulff, M. Woldbye, A. Sørensen, A. Rosenfalck, and F. Jensen. MUNIN — an expert EMG assistant. In J. E. Desmedt, editor, *Computer-Aided Electromyography and Expert Systems*, chapter 21. Elsevier Science Publishers, Amsterdam, 1989.

[3] E. Castillo, J. M. Gutierrez, and A. S. Hadi. *Expert Systems and Probabilistic Network Models*. Springer Publishers, New York, 1997.

[4] J. Cheng and M. J. Druzdzel. AIS-BN: An adaptive importance sampling algorithm for evidential reasoning in large Bayesian networks. *Journal of Artificial Intelligence Research*, 13:155–188, 2000.

[5] C. Conati, A. S. Gertner, K. VanLehn, and M. J. Druzdzel. On-line student modeling for coached problem solving using Bayesian networks. In *Proceedings of the Sixth International Conference on User Modeling (UM–96)*, pages 231–242, Vienna, New York, 1997. Springer Verlag.

[6] H. Guo and W. Hsu. A survey of algorithms for real-time Bayesian network inference. In *In the joint AAAI-02/KDD-02/UAI-02 workshop on Real-Time Decision Support and Diagnosis Systems*, Edmonton, Alberta, Canada, 2002.

[7] M. Mascagni. Parallel linear congruential generators with prime moduli. *Parallel Computing*, 24(8):923–936, 1997.

[8] M. Mascagni. Scalable Parallel Random Number Generators Library (SPRNG) . *homepage:* `http://sprng.fsu.edu`, 1999-2007.

[9] S. Moral and A. Salmerón. Dynamic importance sampling in bayesian networks based on probability trees. *International Journal of Approximate Reasoning*, 38:245–261, 2005.

[10] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1988.

[11] M. Pradhan, G. Provan, B. Middleton, and M. Henrion. Knowledge engineering for large belief networks. In *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence*, pages 484–490. Morgan Kaufmann Publishers, 1994.

[12] R. Y. Rubinstein. *Simulation and Monte Carlo Method*. John Wiley and Sons, Hoboken, NJ, 1981.

[13] S. Russell and P. Norvig. Artificial intelligence: A modern approach. In *Prentice Hall Series in Artificial Intelligence*. Prentice Hall, 1995.

[14] R. D. Shachter and M. A. Peot. Simulation approaches to general probabilistic inference on belief networks. In *Proceedings of the 5th Conference on Uncertainty in Artificial Intelligence*, volume 5, 1990.

[15] M. Wolfe. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–293, 1986.

[16] H. Yu and R. van Engelen. Refractor importance sampling. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, pages 603–611, July 2008.

[17] C. Yuan and M. J. Druzdzel. Importance sampling in Bayesian networks an influence-based approximation strategy for importance functions. In *Proceedings of the 21th Conference on Uncertainty in Artificial Intelligence*, pages 650–657, July 2005.

[18] C. Yuan and M. J. Druzdzel. Theoretical analysis and practical insights on importance sampling in Bayesian networks. *Int. J. Approx. Reasoning*, 46(2):320–333, 2007.