Node Level Primitives for Exact Inference using GPGPU

Hyeran Jeon, Yinglong Xia, Viktor K. Prasanna
University of Southern California, Los Angeles, CA 90089
Email: {hyeranje, yinglonx, prasanna}@usc.edu

***Abstract*—Exact inference is a key problem in exploring probabilistic graphical models in a variety of multimedia applications. In performing exact inference, a series of computations known as node level primitives are performed between the potential tables in cliques and separators of a given junction tree. The computation complexity increases dramatically with the clique width and the number of states of random variables. In this paper, we propose a conflict-free data layout for potential tables on GPU. We map the algorithms for the primitives to the GPU architecture based on the proposed data layout. Several optimization techniques are presented to improve the performance. We implemented the proposed method on NVIDIA Tesla C870. Experimental results exhibit scalability over a wide range and show superior performance compared with state-of-the-art multicore CPUs such as Intel Xeon and AMD Opteron.**

Keywords-Node level primitives,GPGPU,Exact inference

I. Introduction

A full joint probability distribution for any real-world system can be used for inference. However, such a distribution increases dramatically with the number of variables used to model the system. It is known that independence and conditional independence relationships can greatly reduce the size of the joint probability distributions. This property is utilized by *Bayesian networks* [1]. Bayesian networks have been widely used in multimedia applications especially in speech recognition [2][3][4].

Given updated distribution of a set of variables in the Bayesian network, exact inference is the computation of updating all the remaining variables in the network. Exact inference is NP hard [5]. The most popular exact inference algorithm, proposed in [1], converts a Bayesian network into a *junction tree*, and then performs a series of computations, known as *node level primitives*, in the junction tree. The complexity of the primitives increases dramatically with the number of variables of the cliques, the number of states of the random variables in the cliques, and the number of children of each clique. In many cases exact inference must be performed in real time.

General-Purpose computation on Graphics Processing Units (GPGPU) is a promising computing method using GPU's high performance parallel computing power to accelerate a wide range of applications [6]. Compared with other existing multicore processors, GPU provides significantly more processing units and achieves higher throughput. However, due to its distinctive architecture, it remains a fundamental challenge in parallel computing to efficiently map algorithms onto GPU.

Our contributions in this paper include: (1) Mapping node level primitives onto GPU architecture using CUDA, (2) Design of optimization techniques, (3) Implementing the node level primitives and experimentally evaluating them on a state-of-the-art GPGPU platform.

The paper is organized as follows: Section II discusses the background of exact inference and GPGPU. We propose node level primitives and optimizations on GPU in Section III and show experimental results in Section IV. We conclude the paper in Section V.

This research was partially supported by the National Science Foundation under grant number CNS-0613376. NSF equipment grant CNS-0454407 is gratefully acknowledged.

II. Background

A. Node Level Primitives

As discussed in Section I, the most popular exact inference algorithm converts Bayesian network into a junction tree. A junction tree is defined as $\mathrm{J} = (\mathbb{T}, \hat{\mathbb{P}})$, where $\mathbb{T}$ represents a tree and $\hat{\mathbb{P}}$ denotes the parameter of the tree. Each vertex $\mathcal{C}_i$, known as a *clique* of J, is a set of random variables. Assuming $\mathcal{C}_i$ and $\mathcal{C}_j$ are adjacent, the *separator* between them is defined as $\mathcal{C}_i \cap \mathcal{C}_j$. $\hat{\mathbb{P}}$ is a set of *potential tables*(POT). The POT of $\mathcal{C}_i$, denoted $\psi_{\mathcal{C}_i}$, can be viewed as the joint distribution of the random variables in $\mathcal{C}_i$. For a clique with w variables, each having r states, the number of entries in $\mathcal{C}_i$ is r^w.

Exact inference in a junction tree requires propagating evidence at an arbitrary clique to all the other cliques. Mathematically, evidence propagation can be represented as [1]:

$$\psi^*_{\mathcal{S}} = \sum_{\mathcal{Y}\setminus\mathcal{S}} \psi^*_{\mathcal{Y}}, \quad \psi^*_{\mathcal{X}} = \psi_{\mathcal{X}} \frac{\psi^*_{\mathcal{S}}}{\psi_{\mathcal{S}}} \tag{1}$$

where $\mathcal{S}$ is a separator between cliques $\mathcal{X}$ and $\mathcal{Y}$; $\psi_{\mathcal{S}}$ ($\psi^*_{\mathcal{S}}$) denotes the original (updated) POT of $\mathcal{S}$; $\psi^*_{\mathcal{X}}$ is the updated POT of $\mathcal{C}_{\mathcal{X}}$.

Equation 1 implies three types of node level primitives: *Multiplication*(i.e. $\psi_{\mathcal{X}} \frac{\psi^*_{\mathcal{S}}}{\psi_{\mathcal{S}}}$ in Eq 1) and *Division*(i.e. $\frac{\psi^*_{\mathcal{S}}}{\psi_{\mathcal{S}}}$ in Eq 1) between two POTs and *Marginalization*(i.e. $\sum_{\mathcal{Y}\setminus\mathcal{S}} \psi^*_{\mathcal{Y}}$ in Eq 1) that obtain the POT for separators using a clique POT [7]. The details are discussed in Section III.

B. GPGPU

Tera Flop/s of peak performance of recent GPUs and GPGPU infrastructures supported by GPU vendors to utilize GPUs as general purpose computing architectures makes

GPUs to be one of the most promising high performance architectures. NVIDIA Tesla architecture and CUDA programming model [8], the representative GPGPU programming environment, have been stimulating various studies of parallel processing. Tesla consists of a scalable number of streaming multiprocessors(SMs), each comprising of eight streaming processor(SP) cores and 16 KB shared memory. In C870, one of the latest models of Tesla, up to 768 hardware threads are supported by each SM and 12288 threads can be supported as a whole. A bunch of threads(32 in C870), named *warp* is scheduled concurrently. This massive thread level parallelism achieves 512 GFlop/s, which is much higher peak performance than that of any other multicore CPU [9]. Tesla, which is an accelerator cooperate with Host CPU, provides a large, but slow off-chip global memory that can be accessed by all GPU thread blocks and Host CPU, while providing small but fast on-chip shared memories that are individually shared among threads in the same thread block. During the past few years, matrix computations on GPU have been studied. For example, Sengupta et.al. [10] proposed segmentation technique to fit scan primitive into GPU architecture. Williams et.al. [11] showed several optimization techniques such as thread mapping and data reuse to reduce memory access latency and minimize global memory access for sparse matrix-vector multiplication on GPU. However, to the best of our knowledge, there have not been any studies on exact inference using GPGPU.

III. NODE LEVEL PRIMITIVES USING GPGPU

A. Table Multiplication and Table Division

In exact inference, Table Multiplication occurs between a clique POT and its separators. For each entry in a separator, Table Multiplication multiplies the data in the entry with data in another entry of the clique POT, where the random variables shared by the separator and the clique have identical states.

Table Multiplication requires the identification of the relationship between entries in the separator and those in the clique POT [7]. We resolve this issue by using *mapping vectors*. A mapping vector has as many entries as the number of variables of the corresponding separator and indicates each variable's index in the clique POT. For each entry of a separator POT, we can find the associated entries in the clique POT using mapping vector in O(W_s) time, where W_s is the number of variables in a separator POT. The computation of Table Multiplication itself can be executed in O($\frac{d_c \times |\psi_C| \times W_s}{\#hardware\_threads}$) time, where d_c is the number of children of a clique and $|\psi_C|$ is the size of clique POT.

However, if the code is composed of a loop which deals with one child's separator table in each iteration, the control branch overhead at every iteration will significantly impact the execution time. To reduce this overhead, we unrolled the loop by the number of children of a node. Detailed explanation is in Section III-D2.

Table Division occurs between two separator POTs which always have the same size. In this case, the implementation is straightforward if we use as many threads as the separator POT size. By letting each of the threads divide one entry of a separator POT by the corresponding entry of the other separator POT, Table Division can be performed in O($\frac{|\psi_S|}{\#hardware\_threads}$) time.

B. Table Marginalization

Table Marginalization is used to obtain separator POTs from a given clique POT [7]. In Marginalization, we resolve index calculation by using the same technique in the Table Multiplication.

The clique POTs are typically much larger than separator POTs as discussed in the previous section. Hence, the mapping vector maps multiple entries in the clique POT to a single entry in the separator POT. If as many threads as the clique POT size are used for Marginalization like the other primitives, *write conflict* would occur. Since CUDA does not support synchronization APIs except a barrier, *__syncthread()*, the only way to solve the write conflict issue is to use as many threads as the separator POT size.

To offset the limited degree of parallelism, we let Marginalization between a clique and all its children be executed concurrently, by assigning as many threads as the sum of all children separators POT. Furthermore, by using the proposed mapping vector, we only need to use $\frac{|\psi_C|}{|\psi_S|}$ entries of the clique POT to update an entry of a separator POT in Table Marginalization. Consequently, Marginalization for a clique can be performed in O($\frac{d_c \times \frac{|\psi_C|}{|\psi_S|}}{MIN(d_c \times |\psi_S|, \#hardware\_threads)}$) time.

C. Computation kernels of exact inference

For each clique in a junction tree, we will execute node level primitives in certain order: 1) marginalizing the clique POT using the given separator POT, 2) dividing the separator POT by the marginalized clique POT, and 3) multiplying the result of the division with the original clique POT. We call this composition of primitives as *computation kernel* [12]. During the execution of a computation kernel at a node, the results of primitive computation do not have to be moved to Host CPU because they are used by the following primitives.

D. Optimization techniques

1) Conflict-free POT organization: All table computations are assumed to be done by using shared memories. Since a shared memory consists of 16 banks, *bank conflict* occurs whenever more than two threads in the same warp try to access the same bank. To avoid bank conflicts, we assign data to threads column-major order. As Figure 1 shows, row-major data structure brings about bank conflicts unless all threads in the same warp always access offsets different from each other's. The conventional method to resolve the bank conflict is to use padding [13]. However, as padding insertion requires complex calculations, we resolved the bank conflict issue by simply converting each thread's POT from row-major to column-major order. As the threads in the same warp never access the same bank with the column-major POT, the bank conflict problem can be solved without any further index

calculations. We represent all the input data of primitives in column-major order.

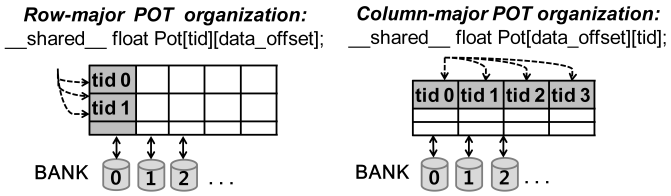

Figure 1. column-major POT vs. row-major POT

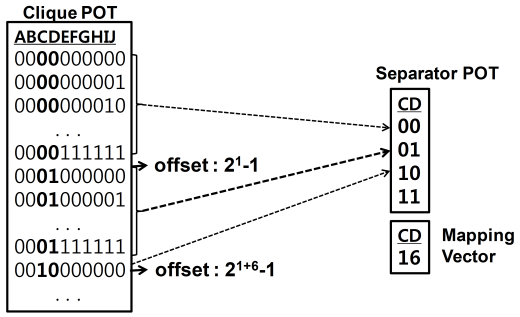

Figure 2. Mapping vector and the possible loop unrolling degree

2) Loop unrolling: To account for the limited parallelism as in Table Marginalization, we use loop unrolling to exploit additional parallelism. Loop unrolling is applied to the following three cases: 1) when the input data is copied from global memory to shared memory, 2) when the same table computation is conducted across the children nodes, and 3) when multiple adjacent source addresses are mapped to the same destination address like in Table Marginalization. In case 1), the unrolling depth is limited by the share memory size. As the thread blocks on the same SM have to share a 16 KB shared memory, each thread can have data array up to $\frac{16KB}{\#thread\_blocks \times \#threads\_in\_block \times float\_type\_in\_byte}$. We spread as many data copy operations as each thread's data array size. The unrolling depth of case 2) is the number of children of a clique. In case 3), which is applied to Table Marginalization, if the least significant variable of a separator is the Nth variable of the corresponding clique and r is the number of states of variables, we can unroll at least r^N iterations because every r^N entries of the clique POT are mapped to the same element of the separator POT as shown in Figure 2.

IV. EXPERIMENTS

To evaluate the optimized primitives, NVIDIA Tesla D870 Deskside system was used. D870 has two C870 GPUs. We used one of them in our experiments. Salient features of C870 is shown in Table I. Clique POTs were created by generating random numbers. The parameters are shown in Table I. The thread block size was determined empirically to be 96. The execution times were measured while increasing the number of thread blocks having the same size from 1 to 128 to evaluate scalability. Note that as C870 used in the experiments comprises of 16 SMs, where each has 768 hardware threads, 128 thread blocks each containing 96 threads achieve the maximum occupancy [14]. We measured the execution time for performing the primitive computations on the GPU, so the time taken for transferring data between CPU and GPU was not included.

TABLE I
EXPERIMENTAL ENVIRONMENT

NVIDIA Tesla C870	
# of Processing Cores	128 SPs(16 SMs)
Peak Performance	430 GFlops
Total Dedicated Memory	1.5 GB GDDR3 at 800MHz
Memory Interface	384 bit GDDR3
Memory Bandwidth	76.8 GB/sec peak

PARAMETER SETTINGS

Parameter	*Default*	*Range*	*Description*
W_c	15	15, 20, 25	Width of clique POT
r	2	2, 3	# of states of each variables
W_s	2	1, 2, 4	Width of separator POT
d	2	1, 2, 4	# of children of a clique

A. Performance improvement due to optimizations

Figure 3 shows the effect of the proposed optimization techniques by comparing performance of three versions of Table Division: 1) *Naive Code* which is the baseline code without any optimizations, 2) code with loop unrolling, and 3) code with loop unrolling and the proposed POT structure. The execution time improved by upto 88% and further by 18% when loop unrolling and the conflict-free POT are employed, respectively.

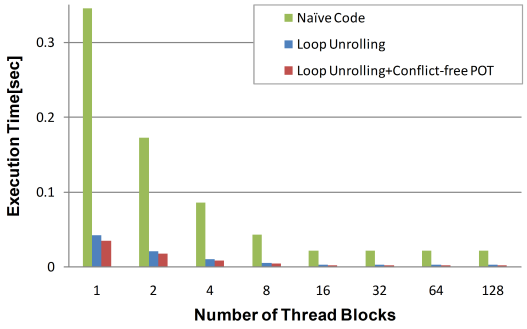

Figure 3. Performance improvement due to optimizations

B. Scalability of Primitives

Figure 4 shows the scalability of each primitive. As shown in Figure 4(c) and (d), the execution times were not affected by the clique degree(the number of children) and the separator width(the number of variables). This is because the loops traversing all the children were unrolled in Table Multiplication and Division and, the Table Marginalization for all children separator POTs were performed concurrently.

In Figure 4(a) and (b), the execution times increased in proportion to the number of states of variables and clique width and decreased as the number of thread blocks increases. The execution times tapered off when the thread block is greater than 16 due to the limited loop unrolling depth caused by the lack of shared memory available to each thread block.

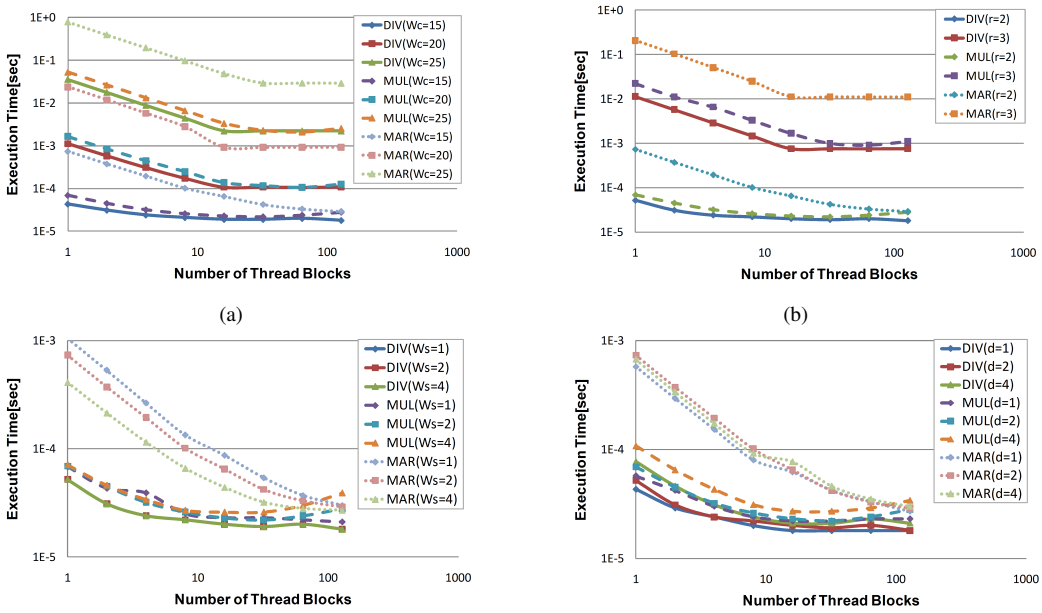

Figure 4. Scalability of Primitives with respect to various parameters: (a) clique width, (b) number of states, (c) separator width, and (d) clique degree

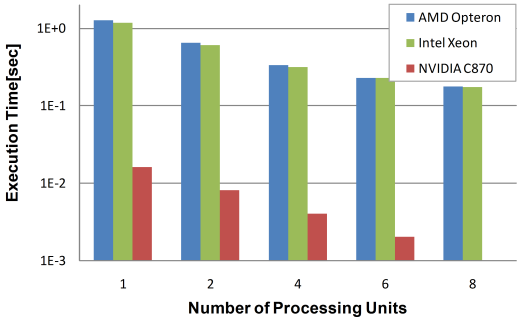

Figure 5. Comparison of execution time for performing a computation kernel on various platforms

C. Scalability of Computation Kernel

Figure 5 shows the performance comparison among C870 and two state-of-the-art multicore CPUs when a computation kernel for one clique(W_c=20, W_s=2, r=2, d=4) is executed. A Quad-Core AMD Opteron 2350 and a Quad-Core Intel Xeon 5335 were used in the experiments. The number of processing units is the number of hardware cores for Opteron and Xeon and the number of thread blocks for C870. The massive thread level parallelism allowed C870 to achieve almost 100× Speedup.

V. CONCLUSION

In this paper, we examined the performance improvements for node level primitives for exact inference using GPGPU. We optimized node level primitives (i.e. Table Multiplication, Division, and Marginalization) to take advantage of the massive thread level parallelism that the GPU provides. 88% and 18% of performance enhancements were achieved by using loop unrolling and conflict-free POT organization, respectively. Additionally, we showed that GPGPU outperforms state-of-the-art multicore CPUs for the computation kernels. Based on the results of this study, we are planning to explore exact inference in an entire junction tree using GPGPU using a scheduler running on the CPU.

REFERENCES

[1] S. L. Lauritzen and D. J. Spiegelhalter, "Local computation with probabilities and graphical structures and their application to expert systems," *J. Royal Statistical Society B*, vol. 50, pp. 157–224, 1988.
[2] A. V. Nefian, L. Liang, X. Pi, X. Liu, and K. Murphy, "Dynamic bayesian networks for audio-visual speech recognition," *EURASIP J. Appl. Signal Process.*, vol. 2002, no. 1, pp. 1274–1288, 2002.
[3] S. Wachsmuth and G. Sagerer, "Bayesian networks for speech and image integration," in *Eighteenth national conference on Artificial intelligence*. Menlo Park, CA, USA: American Association for Artificial Intelligence, 2002, pp. 300–306.
[4] L. Xie and H. Yang, "Dynamic bayesian network inversion for robust speech recognition," *IEICE - Trans. Inf. Syst.*, vol. E90-D, no. 7, pp. 1117–1120, 2007.
[5] D. Pennock, "Logarithmic time parallel Bayesian inference," in *Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence*, 1998, pp. 431–438.
[6] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.
[7] Y. Xia and V. K. Prasanna, "Node level primitives for parallel exact inference," in *Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing*, 2007, pp. 221–228.
[8] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.
[9] "NVIDIA Tesla C870." [Online]. Available: http://www.nvidia.co.in/page/tesla_gpu_processor.html
[10] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," in *Graphics Hardware 2007*. ACM, August 2007, pp. 97–106.
[11] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Parallel Comput.*, vol. 35, no. 3, pp. 178–194, 2009.
[12] Y. Xia and V. K. Prasanna, "Scalable node-level computation kernels for parallel exact inference," *IEEE Trans. Comput.*, vol. 59, no. 1, pp. 103–115, 2010.
[13] H. Nguyen, *GPU Gems 3*. Addison-Wesley Professional, August 2007.
[14] "CUDA programming guide, version2.1." [Online]. Available: http://developer.nvidia.com/object/cuda_2_1_downloads.html