Graphical Model Inference using GPUs

Nadathur Satish Narayanan Sundaram
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley, CA, USA
{nrsatish, narayans}@eecs.berkeley.edu

Abstract

Graphical modeling is a method that is increasingly being used to model real-life applications such as medical diagnosis, security and emergency response systems, and computer vision. A graphical model represents random variables and dependencies between them by means of a graph, in which some variables may not be observable in practice. An important problem in graphical models is that of inference, the process of finding the probability distributions of hidden variables given those of the observed variables. A general technique for inference in graphical models uses junction trees. Inference on junction trees is computationally intensive and has a lot of data-level parallelism. However, the large number and lack of locality of memory accesses limits exploitable parallelism on CPU platforms. Graphical Processing Units(GPUs) are well suited to exploit data-level parallelism, and are being increasingly used for scientific computations. In this work, we exploit the support for data-level parallelism on GPUs to speedup exact inference on junction trees and rely on multi-threading to hide memory latency. We achieve two orders of magnitude speedup on a CPU/GPU system using an NVIDIA GeForce 8800 part over a standalone CPU system.

1 Introduction

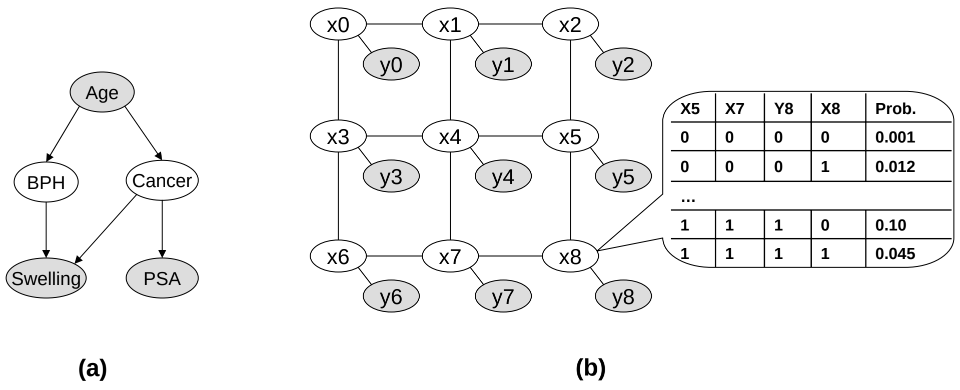
Statistical / Graphical Modeling is a technique used increasingly in many practical applications such as medical diagnosis, security and emergency response systems, and computer vision. A graphical model is a family of probability distributions defined in terms of a directed or undirected graph. The nodes in the graph are identified with random variables, and joint probability distributions are defined by taking products over functions defined on connected subsets of nodes [1]. The edges in the graph represent dependence between the variables in the model. The meaning of the edges changes depending on the type of model considered.

There are two main types of graphical models: directed and undirected. A directed model is a directed acyclic graph (DAG) where each node is a random variable and the absence of an edge indicates that the nodes concerned are independent of each other. The presence of an edge intuitively indicates a causal relationship between the two nodes. An example of this is shown in Fig 1(a). In this example, high blood pressure and cancer can be explained by a common cause, old age. However, given the age of a person, the two diseases are independent of each other. Both these diseases could produce swelling as a symptom, which only cancer causes prostate-specific antigen tests to come back positive.

Undirected graphical models represent independence more naturally: two (sets of) nodes A and B are conditionally independent given a third set, C, if all paths between the nodes in A and B are separated by a node in C [2]. However, the notion of causality is not present in undirected models. Fig. 1(b) shows an example of an undirected model. In particular, it is a two dimensional grid (an example of a Hidden Markov Random Field). Hidden Markov Random Fields are used in image processing, filtering, speech recognition and other applications [3].

In addition to the graph structure, we also need to provide the parameters of the model. For a directed model, we must specify the Conditional Probability Distribution (CPD) at each node. If the variables are discrete, this can be represented as a table (CPT), which lists the probability that the child node takes on each of its different values for each combination of values of its parents [2]. In the case of an undirected graph, we use probabilities of cliques in the graph [4].

A typical use case of graphical models is when some variables in the model are hidden (cannot be observed). The standard convention is to shade the nodes that are observable. For instance, in Fig 1(a), the age of patients and their symptoms are observable, and the diseases are hidden. In this case, an important problem to be solved is *inference*: Given a set of observed variables and probability distributions relating the variables, we are asked to find the probability distributions of the hidden variables. Inference on graphical models is a well-studied problem and many tech-



X5	X7	Y8	X8	Prob.
0	0	0	0	0.001
0	0	0	1	0.012
...				
1	1	1	0	0.10
1	1	1	1	0.045

Figure 1: Types of graphical models:(a) A directed graphical model and (b) an undirected model (in particular, a Markov Random Field).

niques have been proposed to solve it. A general technique that works for both directed and undirected graphs is to use a junction tree algorithm [4]. A junction tree is a tree where each node represents a *clique* in the original model. The process of conversion of a graphical model to a junction tree is not discussed here. The interested reader is referred to [4] for details.

Solving the inference problem exactly on junction trees involves a set of reduction operations over different elements of the CPT tables corresponding to nodes of the junction tree (which are cliques of the original graph). These tables can get very large, especially if the original graph was not very sparse, and had large cliques. Thus these reduction operations get very expensive, leading to a problem with high computational requirement.

There have been efforts to parallelize the junction tree algorithm to increase execution speed [5] [6] [7]. Such approaches rely on the topology of the junction tree for parallelism, which may not always be available. For instance, Markov Random Fields only give limited benefits in parallelism based on the junction tree topology. Parallelizing the reduction operations inside each node of the tree is another option for speedup. This requires extensive support for data level parallelism.

GPUs are being increasingly used to perform scientific computations. GPUs offer the ability to process a large number of floating point operations in parallel, and are especially efficient for data-parallel applications. The traditional limitation to harnessing the enormous GPU processing power has been the GPU programming model, which required all applications to work in terms of vertices and pixels. However, efforts have been made recently to facilitate the programming of general purpose algorithms on GPUs, both from academia: the BROOK project from Stanford [8] and industry: the CUDA programming model by NVIDIA [9]. As a result of such efforts, GPUs have been successfully used to accelerate various algorithms (sparse-matrix computations [10], pattern recognition etc.).

In this project, we explore mapping the junction tree inference algorithm to an NVIDIA GPU using the CUDA infrastructure to exploit the processing power of GPUs. We proceed to explain the algorithm in detail in Section 2, the GPU architecture in Section 3 and the mapping in Section 4. We then give our results in Section 5 and conclude in Section 6.

2 Graphical Model Inference Using Junction trees

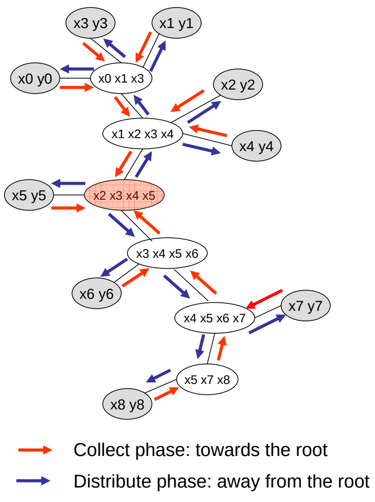


Figure 2: Junction tree for the Markov Random Field in Fig. 1(b).

The junction tree is a structure that makes explicit the relationship between graph-centric locality and efficient probabilistic inference. It is a general technique that works not just on probability inference problems but on a more general class of problems that involve factorized potentials on graphs. The algorithm that we discuss is called the "Hugin algorithm".

The junction tree is a *clique tree* i.e. its nodes are cliques of the underlying graph. Between each linked pair of cliques is a *separator set* - the intersection of the two cliques. Junction trees satisfy the property that all cliques containing a common variable form a connected sub-tree. It must be noted that this property is not satisfied by all clique trees. We do not deal with the generation of the junction trees in this project. We assume that the junction tree is available as an input and we only focus on solving the problem of inference on the junction tree. We generate junction trees from graphical models using BNT [11]. BNT uses a greedy heuristic to obtain a close-to-optimal junction tree. Fig. 2 gives the junction tree for the Markov Random Field in Fig. 1(b).

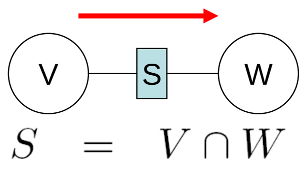


Figure 3: Update step for an edge S from V to W.

2.1 Hugin Algorithm

We will focus on one algorithm for junction tree inference called the "Hugin algorithm". The basic philosophy behind the junction tree inference algorithms is *maintaining consistency* i.e. the potentials for all cliques must give the same marginals for the variables they have in common. This notion of consistency is required for the correctness of any inference algorithm. Junction trees have the special property that local consistency implies global consistency and vice versa. So, it is sufficient if consistency is maintained between cliques that are connected by an edge. This ensures consistency for all variables. For any two linked cliques V and W as shown in Fig. 3, consistency implies that

$$\sum_{V \setminus S} \psi_V = \sum_{W \setminus S} \psi_W = \phi_S \tag{1}$$

$$\psi_{V \cup W} = \frac{\psi_V \psi_W}{\phi_S} \tag{2}$$

where ψ_V and ψ_W are the CPT(or table of potentials, in general) of the cliques V and W
and ϕ_S is the CPT(or table of potentials) of the separator set S
Formally, the pseudo code for the procedure can be given as a two phase algorithm:

```
Collect_Evidence(root);
Distribute_Evidence(root);
```

where the two functions can be defined recursively as

```
Collect_Evidence(node)
    begin
        for each child of node
            begin
                Update(node, Collect_Evidence(child))
            end
        return (node)
    end

Distribute_Evidence(node)
    begin
        for each child of node
            begin
                Update(child, node)
                Distribute_Evidence(child)
            end
    end
```

The operations to be performed for a message from V to W [Update(V,W)] in figure 3 are :

$$\phi^*_S = \sum_{V \setminus S} \psi_V \tag{3}$$

$$\psi^*_W = \frac{\phi^*_S}{\phi_S} \psi_W \tag{4}$$

Here, equation (3) denotes the first phase of the marginalization, referred to as **S update** henceforth. Equation (4), the second phase of the marginalization is referred to as **W update** in the remainder of the paper.

The algorithm described earlier does a graph traversal in DFS order (this is not necessary - BFS traversal would work as well). The choice of the root can be arbitrary i.e. the correctness of the algorithm does not depend on it, but its performance can. We chose the root as the "center" of the graph i.e the clique with the least maximum distance from any other clique in the junction tree. Graph traversal was done in BFS order - from the leaves to the root during the collect phase and from the root to the leaves during the distribute phase.

The interesting thing about the marginalization is the amount of computations involved and the parallelism that is inherent in it. For instance, if we take two moderately sized cliques(V and W) of 12 variables and their separator set (S) of size 4, then the number of sums required to calculate each entry of S during the S update phase is $2^{12-4} = 2^8 = 256$. Similarly, during the W update phase, each S table entry is used to update 256 entries in the W table independently. Thus, there is large amount of data-level parallelism to be exploited, especially for large cliques.

2.2 Practical Concerns

There are a few practical issues that have to be taken care of for the algorithm to be efficient and correct.

2.2.1 Table sizes

There is a CPT associated with each clique and edge. The size of this table is exponential in the size of the corresponding clique. Each table has an intrinsic ordering of the variables. This is the reason for non-consecutive memory accesses when marginalizing some variables. This is shown clearly in Fig. 6

2.2.2 Precision Issues

The W update phase of the marginalization involves multiplication and division on small values. This, in turn implies that the table values must be stores as log-values to maintain precision. However, this creates a problem during the S update phase, which involves only additions. The update phase is then implemented as follows:

$$\phi_S^* = log\left(\sum_{V\setminus S} exp(\psi_V)\right) \quad (5)$$

$$\psi_W^* = \phi_S^* - \phi_S + \psi_W \quad (6)$$

3 CPU/GPU platforms

3.1 GPU Architecture

GPU architectures are specialized for compute-intensive, highly-parallel computation, and therefore designed such that more transistors are devoted to data processing than data caching or flow control. They are especially well suited to data-parallel computations. GPUs are mainly used as accelerators for specific parts of an application, and as such is attached to a host CPU that performs most of the control-dominant computation.

In this project, we consider the NVIDIA GeForce 8800 GTX GPU. The device is implemented as a set of 16 multiprocessors as shown in Fig 4. Each multiprocessor has a Single Instruction, Multiple Data architecture (SIMD): at

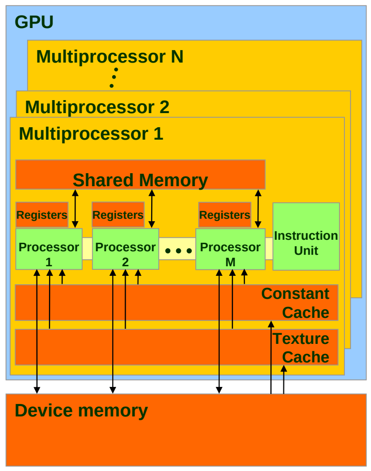


Figure 4: NVIDIA GeForce 8800 GPU Architecture.

any given clock cycle, each processor of the multiprocessor executes the same instruction on different data. There are 8 processors in each multiprocessor, and thus a total of 128 floating point units active each cycle. Each multiprocessor has a set of registers and a shared memory block that is shared by all processors. There is a global memory shared by all multiprocessors (768 MB). The bandwidth between the global and all shared memories is about a total of 90 Gbps, and is usually not a constraint. The host CPU and GPU are connected by PCI Express, which gives a typical bandwidth of 800 Mbps, with a peak of 3.2 Gbps. We refer the reader to [9] for more details.

3.2 Programming Model

The traditional problem with leveraging GPU performance for non-graphics applications has been that the GPU could only be programmed through graphics API, imposing a high learning curve for the novice programmer. There has been a recent effort by NVIDIA to provide a Compute Unified Device Architecture (CUDA) that provides a programming environment close to C. This has support for exploiting the data-parallel nature of non-graphics applications through multi-threading. It also has support for memory gather and scatter operations from non-contiguous portions of memory.

When programmed through CUDA, the GPU is viewed as a compute device capable of executing a very high number of threads in parallel. It operates as a coprocessor to the main host CPU. A portion of the application that is executed many times, but independently on different data can be isolated into a function that is executed on the device as different threads. This function is called a kernel. If there are many function, they form different kernels. When a kernel is called by the CPU, it blocks till the GPU completes the kernel execution. Thus kernel executions are sequential and non-overlapping.

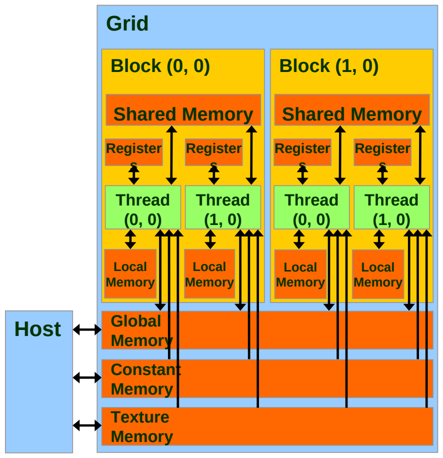


Figure 5: Logical organization of the GeForce 8800.

A kernel is organized as a set of thread blocks (shown in Fig 5). A thread block is a batch of threads that all execute on one of the 16 multiprocessors. As a result, they can cooperate together by efficiently sharing data through shared memory, and can synchronize their execution to coordinate memory access. There is a maximum limit on the number of threads per block. However, blocks of the same size can execute the same kernel batched together as a grid of blocks, so that the total number of threads that can be launched in parallel is much higher. This comes at the expense of reduced thread cooperation: threads in different thread blocks cannot communicate and synchronize with each other. In practice, we need a large number of thread blocks to ensure that the compute power of the GPU is efficiently utilized.

3.3 Advantages of GPU style architecture

The main advantage of a GPU style architecture is that they have a large number of floating point units that can operate in parallel. Applications that require a large number of computations using the same function on a large data set should see considerable speedup on the GPU. GPU architectures also have specialized vector hardware for math operations like exponentiation and logarithms. This is also a source of speedup for CPU/GPU systems, since there are many floating point units in parallel.

A GPU uses multi-threading to hide memory latencies: a thread that is blocked waiting for memory gives up control of the floating point execution unit to a thread that has received its required memory values and can compute. This technique is different from the way CPU systems hide latency: by using caches to exploit memory locality. In fact, there is no cache for global read/write memory on the GPU. The area of the caches is used for more computation units. This technique can be useful if data locality is low for an application. However, the technique requires the presence of a sufficient number of threads that are free to start at any given point of time: this is why having a large number of independent thread blocks is important for performance.

4 Parallelization of inference on CPU/GPU platforms

Since the major portion of the running time of the inference algorithm is in the marginalizations, it is efficient to exploit the data-level parallelism inherent in the operation. The marginalization operation, as explained in section 2, involves a reduction operation on non-consecutive memory locations in the probability table.

4.1 S - Update

The S update phase of the marginalization is given by equation (3). There are two options for parallelizing the S update operation :

1. Divide the elements of V into contiguous chunks and allocate each chunk to a block of threads to form partial sum tables of S. These tables would then have to be summed up to get the required S table.

2. For each element of S, allocate a block of threads to read all the required elements of V and sum them up.

The restrictions that the CUDA programming model impose imply that option 1 is infeasible because of the impossibility of synchronizing different thread blocks (needed for adding the partial sum tables generated by different blocks). The second option is feasible and is implemented.

Each thread in the implementation reads in two values of the V table that correspond to a single element in S. Each thread performs the exponentiation and sum of these values. Once these operations are done, the threads perform a reduction(sum) to get the new entry in the S table. This implementation does not require any locks or synchronization between different blocks.

4.2 W - Update

The W update phase of the marginalization is given by equation (4). This phase could be parallelized in two ways as mentioned before - Dividing the W table among blocks or dividing S table among thread blocks. We chose to divide

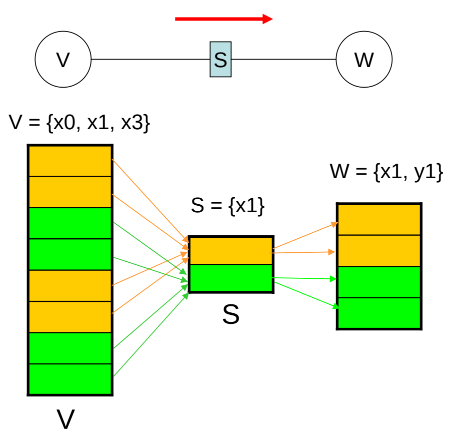


Figure 6: Irregular memory accesses are required in the update step from the node $\{x0,x1,x3\}$ to $\{x1,y1\}$.

it by the S table i.e. each thread block updates all the entries in the W table that require that particular element of S table. Dividing the W table among blocks is worse because of memory contention(A single entry of the S table has to read by many blocks). This problem can be mitigated in the latter case by reading the S table entry once and putting it in the shared memory(which can be read by all the threads in the block).

4.3 Graph traversal

It might be noted that there is some amount of parallelism in the graph traversal phase of the algorithm too. Edges that do not share any nodes and are ready to execute can execute in parallel. However, the amount of parallelism is very small compared to the amount of data parallelism involved in the marginalization. For Markov random fields, most of the graph level parallelism is from the leaves of the junction tree. However, these nodes are very small and updating these edges involve very few floating point operations. From Fig. 9, we find that most of the time is taken by edge updates that require more than 10^7 floating point operations. It should also be noted that not all leaves can be executed concurrently. In Fig. 2, nodes x_0y_0 and x_1y_1 cannot be executed simultaneously because both of them have to update the node $x_0x_1x_3$. Thus exploiting this parallelism requires a lot of modifications to the existing code and gives only marginal benefits. Hence, this was not explored any further.

4.4 Expected performance

We expect the performance of the CPU/GPU system to be better than that of the standalone CPU system because of the following reasons:

Increased parallelism in the GPU leads to more floating point operations per second, thus speeding up the marginalization.Since the reductions can be done independently with little synchronization, parallelization is very effective.

Better memory latency hiding due to multi-threading. The GPU executes thread blocks in a multiprocessor in a time sliced fashion, preempting a set of threads when they wait for memory accesses and executing another set of threads. This hides memory latency very efficiently when the memory accesses are irregular and the number of threads is large.

Better support for mathematical functions in the GPU can also give a constant speedup to the application. In particular, the GPU can perform exponential and logarithmic operations efficiently. These take up a significant amount of the processing required for an edge update in Hugin's algorithm.

5 Experimental Results

We compared the performance of the CPU/GPU system on the junction tree inference algorithm to that of the standalone CPU system. The GPU used was the NVIDIA GeForce 8800 GTX running at 1.35 GHz having a total of 768 MB device memory. The CPU had an Intel Pentium 4 processor at 2.4 GHz with 1 GB RAM. This comparison measures the speedup obtained by performing the reductions corresponding to junction tree nodes on the GPU. The rest of the algorithm including the graph traversal and initial setup, occurs on the CPU in both systems.

We ran our experiments on a set of synthetic benchmarks obtained by forming the junction trees of Markov Random Fields of differing sizes. The CPD tables for the nodes in the Markov Random Fields were chosen randomly. We list the sizes of the Markov Random Fields, along with their key characteristics viz. number of floating point operations and number of memory accesses required for the inference algorithm in Table 1. Since our algorithm has both exponentiation/logarithmic and addition/subtraction operations, we normalized the performance by counting each exponentiation/logarithmic operation to be worth 8 flops and each addition/subtraction to be worth 1 flop. Also, note that since a heuristic is used for forming the junction tree, smaller grids sometimes produce more expensive junction trees.

5.1 Results

We obtained the results shown in Fig. 7 on running our benchmarks on both the CPU/GPU as well as the standalone

Grid Size	# flops	# mem accesses
8 x 8	7.58E+05	3.28E+04
10 x 10	6.93E+06	2.11E+05
9 x 9	9.44E+06	2.48E+05
11 x 11	2.47E+07	6.79E+05
11 x 13	8.63E+07	2.02E+06
11 x 16	2.53E+08	5.45E+06
12 x 12	8.05E+08	1.53E+07
14 x 14	8.95E+08	1.85E+07
13 x 13	1.24E+09	2.31E+07
14 x 16	2.55E+09	4.56E+07
15 x 15	2.71E+09	5.36E+07
16 x 16	1.65E+10	2.85E+08

Table 1: Characteristics of benchmarks used for the inference algorithm. Inference was performed on junction trees obtained from Markov Random Fields with specified grid sizes.

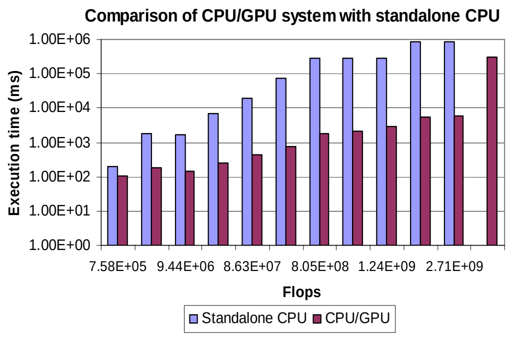


Figure 7: Speedup of the CPU/GPU system compared to the standalone CPU.

CPU systems. On the whole, we obtain from 10x - 150x speedup except for the smallest example. On small examples, we only obtain around a 10x speedup because there are an insufficient number of threads that can run at any point of time to either utilize the full parallelism in the GPU platform or to hide memory access times by multi-threading. The speedup increases with medium sized examples with increasing number of threads, building up to about 150x. For the largest example, the CPU did not finish computation within 1000s, at which point we cut off our run.

Furthermore, we conducted an experiment to study the effect of irregular memory access on execution times. In this experiment, we ran a modified (incorrect) form of junction tree inference where each system only accessed consecutive regions of memory, but performed the same number of memory operations as the original algorithm. This was purely a performance study, since the algorithm was no longer functionally correct. We observed a speedup of 3.4x for the CPU and 1.5x for the CPU/GPU system. This shows that the CPU/GPU system is less dependent on memory data locality for obtaining speedup.

We performed an experiment to compute the extent to which the exponentiation and logarithmic operations influence the speedup. This was also a performance study, and the resulting algorithm was not functionally correct. We found that removing the exponentiation and logarithmic operators resulted in a speedup of about 4 for the CPU system, and only 1.1 for the CPU/GPU system. This shows that the CPU/GPU system did benefit from parallel exponentiation operations.

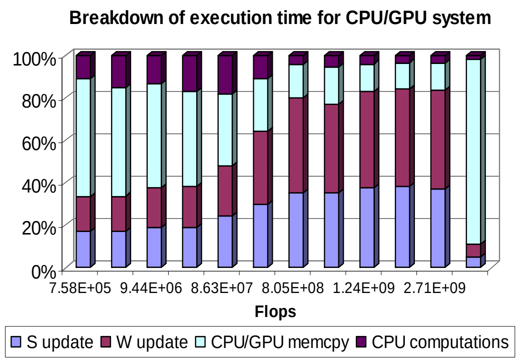


Figure 8: Breakdown of execution times for different stages of the inference algorithm for a CPU/GPU system.

The breakdown of computation time in the CPU/GPU system is shown in Fig. 8. For small instances, we can see that the sequential component of the code (consisting of the CPU graph traversal and data transfer times) takes over 60% of the total time. This is because small examples do not have many operations that can be done in parallel. For medium sized examples, the parallel S-update and W-update steps take up 70-80% of the total execution time. In the case of the largest examples corresponding to the 16x16 grid, we have an anomalous behavior: there is insufficient memory to store the tables, leading to CPU disk accesses and inflated memory transfer times.

We obtained a peak throughput of 6.31 Gflops/sec. This was only sustained for update steps with large cliques. The average throughput for the large test cases was about 205 Mflops/s. This is as opposed to the 343 Gflops/s peak throughput of the device. However, this is only achievable under ideal memory access conditions, and the assumption of an infinite number of threads with no dependencies. The CPU only achieves an average of 1.4 Mflops/s for the large instances.

We performed a final experiment to calculate how much performance increase was being ignored by not considering node-level parallelism. Fig. 9 shows the cumulative execution time of update steps corresponding to different edges in

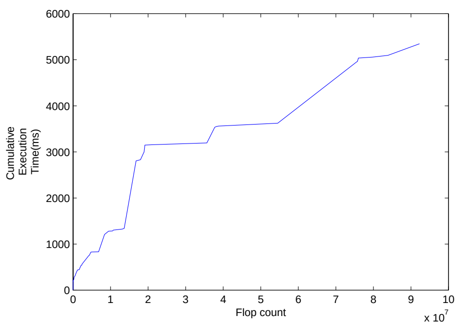


Figure 9: Cumulative execution time of different update steps versus the flop count of the update step for the 15x15 grid.

the graph with respect to their flop count for the 15x15 grid. The intent of the graph is to figure out which update steps contribute most to the execution time. We find that 85% of the execution time is taken up by steps with more than 10^7 floating point operations. These correspond to roughly 20% of the edges that are close to the root node. Most of the node-level parallelism is within the other 80% of the edge updates, which are near the leaves. Thus our speedup by parallelizing these edge updates is only around 15%. Furthermore, this is an optimistic estimate of increase in performance, since not all edge updates near the leaves can occur in parallel. Edges that share a common node cannot be updated in parallel due to read/write conflicts of the CPD tables. Thus we are justified in ignoring node-level parallelism in junction trees corresponding to Markov Random Fields.

6 Conclusion

This project demonstrates the feasibility of using a GPU as an accelerator for large parallel operations. Since the GPUs use multi-threading to hide memory latencies (as opposed to caches in CPUs), they provide significant benefit for algorithms that involve lot of parallelism but bad memory locality. The CPU/GPU system was found to benefit the most for moderately sized problems. The GPU is also very efficient at performing exponentiation and logarithm operations which have been exploited by the algorithm. For very large problems, the CPU memory becomes a limiting factor, leading to significant slowdown. Exact inference using junction trees is not an efficient method for large graphs. Approximate inference algorithms, that do not have such memory constraints, would be better suited for the purpose.

References

[1] M. I. Jordan, "Graphical Models," *Statistical Science (Special Issue on Bayesian Statistics)*, vol. 19, pp. 140–155, 2004.

[2] K. Murphy, "A Brief Introduction to Graphical Models and Bayesian Networks," 1998. `http://www.cs.ubc.ca/~murphyk/Bayes/bnintro.html`.

[3] H. Kuensch, S. Geman, and A. Kehagias, "Hidden Markov Random Fields," in *Annals of Applied Probability*, pp. 577 – 602, 1995.

[4] M. I. Jordan, "An Introduction to Probabilistic Graphical Models (Draft copy)."

[5] F. Diez and J. Mira, "Distributed inference in Bayesian Networks," in *Cybernetics and Systems*, vol. 25(1), pp. 39 – 61, Jan 1994.

[6] A. Kozlov, "Parallel Implementations of Probabilistic Inference," in *Computer*, vol. 29(12), pp. 33 – 40, Dec 1996.

[7] A. Kozlov and J.P.Singh, "A parallel lauritzenspiegelhalter algorithm for probabilistic inference," in *SC 94*, pp. 33 – 40, 1994.

[8] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," in *SIGGRAPH*, vol. 23(3), pp. 777 – 786, Aug 2004.

[9] "NVIDIA CUDA," 2007. `http://developer.nvidia.com/object/cuda.html`.

[10] J. Bolz, I. Farmer, E. Grinspun, and P. Schroder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," in *SIGGRAPH*, vol. 22(3), pp. 917 – 924, Jul 2003.

[11] K. Murphy, "The Bayes Net Toolbox for Matlab," in *Computing Science and Statistics*, vol. 33, pp. 331 – 350, Oct 2001.