Published in final edited form as:

ICS. 2010 June ; 2010: 95–104. doi:10.1145/1810085.1810101.

High-throughput Bayesian Network Learning using Heterogeneous Multicore Computers

Michael D. Linderman,

Computer Systems Laboratory, Stanford University

Vivek Athalye,

Computer Systems Laboratory, Stanford University

Teresa H. Meng,

Computer Systems Laboratory, Stanford University

Narges Bani Asadi,

Computer Systems Laboratory, Stanford University

Robert Bruggner, and

Biomedical Informatics, Stanford University

Garry P. Nolan

Microbiology and Immunology, Stanford University

Abstract

Aberrant intracellular signaling plays an important role in many diseases. The causal structure of signal transduction networks can be modeled as Bayesian Networks (BNs), and computationally learned from experimental data. However, learning the structure of Bayesian Networks (BNs) is an NP-hard problem that, even with fast heuristics, is too time consuming for large, clinically important networks (20–50 nodes). In this paper, we present a novel graphics processing unit (GPU)-accelerated implementation of a Monte Carlo Markov Chain-based algorithm for learning BNs that is up to 7.5-fold faster than current general-purpose processor (GPP)-based implementations.

The GPU-based implementation is just one of several implementations within the larger application, each optimized for a different input or machine configuration. We describe the methodology we use to build an extensible application, assembled from these variants, that can target a broad range of heterogeneous systems, e.g., GPUs, multicore GPPs. Specifically we show how we use the Merge programming model to efficiently integrate, test and intelligently select among the different potential implementations.

Categories and Subject Descriptors

G.3 [Mathematics of Computing]: Probabilistic Algorithms

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Keywords

GPU; MCMC; Bayesian Networks

General Terms

Algorithms; Performance

1. INTRODUCTION

Bayesian Networks are a class of probabilistic models that can be used to learn causal relationships from experimental data [16]. The motivating application for this work is the discovery of the causal structure of intracellular signal transduction networks (STN). Note that although this paper focuses on STN structure learning, the work presented here is equally applicable to other applications in systems biology, bioinformatics, data mining and other fields.

The interior of the cell is a complex environment in which multiple different dynamical systems interact to produce (un)desired outcomes. Chemical stimuli activate a cascade of intracellular signaling molecules that effect changes in cell activity. Recent studies have shown that small differences in the structure of these STNs across individuals are correlated with different therapeutic effectiveness and clinical outcomes [10]. High-throughput learning of the network structure on a per-individual basis could provide enhanced clinical diagnostics and insight into novel pharmaceutical targets.

STNs can be modeled as Bayesian Networks (BNs) [17]. BNs are directed acyclic graphs whose structure can encode the causal relationships between nodes. However, despite the development of new algorithms, learning the structure of BNs from experimental data is still too slow for large (20–50 node) clinically important networks [8]. In this paper, we present a graphics processing unit (GPU)-accelerated implementation of BN learning that is up to 7.5-fold faster than current general-purpose processor (GPP)-based implementations. With this speedup scientists can apply BN learning techniques to larger networks and turn a currently batch-oriented workflow into an interactive process.

The BN learning application has abundant coarse and fine-grain parallelism, making it well-suited for a wide range of heterogeneous architectures. However, the ratio and extent of the different types of parallelism depends on the particular inputs, which can vary greatly between experiments. No one implementation is the best choice across this parameter space. Delivering the best possible performance requires a suite of implementations, each optimized for different input and machine configurations. We use the Merge programming model [12] to integrate, test, and intelligently select among the different implementations available for any particular computation. Using Merge we can readily extend the BN-learning application with new implementations that exploit different hardware platforms, or are optimized for different input scenarios, with the Merge-based application tracking the Pareto optimal performance for the available implementations across these different input and hardware configurations.

This paper presents both a specific implementation for learning BN structure from experimental data and a pragmatic methodology for developing this and other applications for heterogeneous multicore computers. We make the following contributions:

- We describe BN learning, a computationally demanding and clinically important application well-suited for heterogeneous architectures.
- We present an aggressively optimized GPP implementation for BN learning, and a GPU-based implementation that is up to a further $7.5\times$ faster than the GPP version.
- We describe a pragmatic methodology, using the Merge programming model, for building a scalable, extensible application targeting different heterogeneous systems.

The rest of the paper is organized as follows: Section 2 provides relevant background on STNs and BNs; Section 3 describes the GPP and GPU implementations; Section 4 details how we used the Merge programming model; Section 5 presents performance results and Section 6 concludes with a discussion of related and future work.

2. BACKGROUND

2.1 Signal Transduction Networks

Signal transduction networks (STNs) are the “programs” that govern cellular behavior. For example, alterations in certain STN structures can result in increased survival and proliferation of cancer cells. Using flow cytometry, it is possible to track the activity of signaling molecules in individual cells. This data can be used to both learn the structure of the STNs *and* link specific STN structures, and alterations, with particular clinical outcomes [10].

Such links can become the basis for new clinical diagnostics and therapies. However, we can only find these small differences in STNs if we can efficiently learn the network structure on a per-individual basis for many individuals. Recent advances in flow cytometry technology can provide the per-individual data needed to analyze large (~ 50 node) networks. The challenge for us is to deliver the computational tools to efficiently learn STN structure from that data.

Figure 1 shows an example of how different network structures can be learned from flow cytometry data, and the relationship between those structures and clinical outcomes. The upper panels show cell population histograms of STAT3 and STAT5 signaling molecule activation. Samples with right-shifted histograms have “high” levels of STAT3 and STAT5 activation. Each histogram includes many thousands of cells. The large sample size permits statistically meaningful estimation of the node joint probability distributions, e.g., $P(\text{STAT5} = \text{high}, \text{STAT3} = \text{high} | \text{G-CSF})$, that can in turn be used to drive network structure learning. The corresponding signaling networks are shown in the bottom panels. Based on patient outcome, we observe that joint activation of STAT3 and STAT5 is correlated with worse clinical outcomes. Tests for such a structure could be an effective diagnostic for this cancer.

Existing flow cytometry technology effectively limited scientists to investigating networks with less than 15 nodes. The recently developed CyToF mass cytometer [5] can increase that limit to 50 nodes or more. The number of potential graphs grows super-exponentially with the network size. As a result, practical BN learning algorithms utilize polynomial-time heuristics. A five-fold increase in network size translates to a large, but still tractable, $\sim 1000\times$ increase in computational complexity. Our current learning tools can just keep pace with currently $\sim 10\text{--}15$ node data sets. Now in the same few minutes or less, we need to learn the structures of clinically important 30–50 node networks with in-degrees (or number of parents) of 4–5, from datasets with tens to hundreds-of-thousands of cellular observations.

2.2 Learning Bayesian Networks

STNs can be modeled as Bayesian Networks (BNs) [17]. A BN is a directed acyclic graph (DAG). The joint distribution of the variables (a synonym for nodes) $V = V_1, \dots, V_n$ in the graph $\mathcal{G}$ is specified by the decomposition

$$P(V) = \prod_{i=1}^n P(V_i | \Pi_i^{\mathcal{G}}),$$

where $\Pi_i^{\mathcal{G}}$ represents the parent set of V_i and is a subset of $V \setminus V_i$.

In this work we restrict ourselves to the discrete case where each V_i is a categorical variable taking values in a finite set. The local probabilities $P(V_i | \Pi_i)$ are assumed to have a multinomial distribution and can be estimated from simple counting in the input data table.

The goal is to find the graph structure that best explains the data $\mathbb{X}$. To determine causality, $\mathbb{X}$ must include interventional data, not just observational data.

The number of potential graph structures grows super-exponentially with the size of the network; for $n=5, 10$, and 40 there are 29281, 4.17×10^{18} and 1.12×10^{276} graphs, respectively. Thus most BN learning algorithms utilize heuristic search methods such as hill climbing or simulated annealing. An alternate approach are sampling methods, such as Monte Carlo Markov Chains (MCMC), that perform a random walk to explore the posterior space of graphs $P(\mathcal{G} | \mathbb{X})$.

In this paper, we implement the order-based MCMC sampling algorithm described in [3]. Every DAG has at least one total ordering $\sqsubset$ such that $V_i \prec V_j$ if $V_i \in \Pi_j$. Order-based MCMC methods perform a random walk in the space of possible orders, at each step accepting or rejecting the proposed order based on the proposed and current order's posterior probability and the Metropolis-Hastings rule. The posterior probability of an order $P(\sqsubset | \mathbb{X})$ is the sum of the probabilities of all the graphs compatible with order, and is calculated as

$$P(\sqsubset | \mathbb{X}) \propto \sum_{\mathcal{G} \in \sqsubset} \prod_{i=1}^n P(V_i, \Pi_i; \mathbb{X}) = \prod_{i=1}^n \sum_{\Pi_i \in \Pi_{\sqsubset}} P(V_i, \Pi_i; \mathbb{X}) \quad (1)$$

where $P(V_i, \Pi_i; \mathbb{X})$ is the probability of a particular node-parent set combination. Highly probable graphs are then sampled from highly probable orders.

Order-based sampling has several desirable properties that motivate its use here; order methods: 1) are less prone to getting trapped in local minima than search methods, particularly when the search space is “peaky”; 2) perform sampling in the smaller order space and thus have reduced overall computational complexity; and 3) use dense data structures (unlike sparse graphs) that are more amenable to implementation on specialized accelerators like GPUs.

The complete algorithm, summarized in Figure 2, has three main components, the score generator, order sampler and graph sampler.

The score generator computes the local probabilities or local scores, $P(V_i, \Pi_i; \mathbb{X})$ for each node and parent set as follows:

$$P(V_i, \Pi_i; \mathbb{X}) = \gamma^{|\Pi_i|} \prod_{k=1}^{r_i} \frac{\Gamma(\alpha_{ik})}{\Gamma(\alpha_{ik} + N_{ik})} \prod_{j=1}^{|V_i|} \frac{\Gamma(N_{ik} + \alpha_{ijk})}{\Gamma(\alpha_{ijk})} \quad (2)$$

where γ and α are hyperparameters used to tune score generation, $r_i = \prod_{V_j \in \Pi_i} |V_j|$ or the number of different states of the parents, $\alpha_{ijk} = \frac{\alpha}{r_i |V_i|}$ and N_{ik} , N_{ijk} are sufficient statistics (counts) calculated from the experimental data. Effectively we are counting the occurrences of all different combinations of child and parent states. The space of parent sets is all possible combinations of parents for a node up to some limit, k , on the in-degree to a node.

The order sampler performs the random walk in order space. In each iteration, a candidate order, Π' , is proposed by swapping nodes in the current order, scored as shown in Equation 1, and accepted as the new current order according to the Metropolis-Hastings rule. To improve the mixing of MCMC, we implement Parallel Tempering (PT) [9], an enhanced version of MCMC. We run multiple MCMC chains in parallel at exponentially increasing “temperatures”. The higher temperature chains are more likely to accept the proposed order and thus less likely to get trapped in a local minima. High-scoring orders found by high-temperature chains are used to seed lower-temperature chains to improve mixing (the exchange operation). Further search coverage is obtained through (optional) “random restarts”, which reinitiate the search with a new random seed.

The graph sampler samples graphs from set of network structures compatible with high-scoring orders according to the conditional probability distribution. This is implemented by: 1) finding the highest-scoring parent set for each node compatible with an order to determine the highest-scoring graph for that order; and then 2) computing the cumulative distribution function (CDF) for the parent sets for each node compatible with the order, and sampling parent sets according to those CDFs to sample additional graphs.

2.3 Implications of the Learning Workflow

Section 2.2 gives a hint of the large number of parameters the user can adjust to improve learning accuracy. Much of this parameter tweaking occurs in the score generation phase, in which the parent set probabilities are extracted from the cytometry data. There is not one accepted methodology or set of parameters for this operation; as a result our users, the domain experts, often try many different combinations of parameters in the course of their analysis. Supporting this kind of interactive workflow requires quick feedback to the user, in the form of learned graphs, on the effects of their choices. Thus our goal is to learn the structure of (even large) networks in a few seconds or less.

Although there is no canonical set of parameters, to orient the reader, typical parameters might be: input datasets of 1,000–10,000 observations, an in-degree (or k) limit of 4, 10,000 MCMC iterations with 10 tempering chains (or 100,000 iterations of a single chain), no restarts, sampling the best graph only for synthetic networks, and similar parameters, but with 10,000–100,000+ observations and perhaps up to 50–100 restarts for experimental data. The number of parent sets grows as n choose k , and thus is $\sim 74,000$ for $k = 4$ for a 37-node network. The number of orders sampled depends on if and when the MCMC sampler converges, the number of tempering chains, and other parameters, and typically ranges from just tens of orders to one order per every two MCMC iterations.

We need to deliver (near) interactive performance for the above parameters in many different venues, e.g., at the instrument itself, on the scientist's personal desktop/laptop, on remote compute clusters. Each setting will have different hardware available, ranging from older single-core GPPs to modern multicore GPPs supplemented with GPUs or FPGAs. And the parameters chosen by the user will on a per-run basis affect the amount and kind of parallelism available to be mapped to those very different resources. Using (more) restarts or (more) tempering chains increase the amount of coarse-grain thread level parallelism, while increasing the maximum in-degree, and thus the number of parent sets, results in more fine-grain parallelism.

Delivering the best possible performance across this range of input and machine configurations requires assembling our application from, and selecting at runtime among, a collection of specialized software modules each optimized for different input or machine configurations.

3. BN LEARNING IMPLEMENTATION

Execution time is typically, although not exclusively, dominated by the order sampler. Thus our goal, and the focus of this paper, is to accelerate the order sampler, particularly for large networks and many parent sets. The performance evaluation in Section 5 presents results, though, for all components of the application.

Unless otherwise noted, such as for the GPU, all three components described in Figure 2, were implemented in C99 with extensive use of the glib library.

3.1 GPP Order Sampler Implementation

Pseudo-code for the order sampler is shown in Figure 3. The algorithm is implemented with five nested loops: 1) “restarts” on line 1; 2) “MCMC” on line 3; 3) “chains” on line 4; 4) “nodes” on line 7; and 5) “parents” on line 9. All of which, except the MCMC loop, are potentially parallelized.

The two inner-most loops implement Equation 1, the accumulation of the order score. Many of the probabilities $P(V_i, \Pi_i; \mathbb{X})$ are very small, so computations are performed in log-space. In log-space, the outer multiplication across nodes becomes a sum, and the inner sum is accumulated as shown in Figure 3 line 10.

Numerous sequential, i.e., algorithm-based, and parallel optimizations have been developed for the order-sampler (and when relevant, applied to the graph sampler, which has a similar structure).

3.1.1 Sequential Optimizations—Execution time of the reference order-sampler implementation is dominated by the inner-most “parent” loop. The computations within the “parent” loop are only performed when the parent set is compatible with the order. Depending on the node and order, only a small fraction of the parent sets will be compatible with the order. Thus the OP/byte ratio is $\ll 0$, and the execution time is dominated by the time to load the parent set and perform the compatibility check. The “sequential” optimizations directly improve the performance of these inner-most loops, or memoize around them all together.

The score of a particular node is a deterministic function of the set of nodes preceding it in the order. On average, for each MCMC iteration the preceding set of half of the nodes in the order will be unaffected by the node swap and thus have the same score. Thus we memoize per-node scores with a hash-table keyed by the compatibility set. Node memoization can be very effective; for some inputs we observe hit rates up to 97% and speedups better than $3\times$. Note that you can also memoize the full order, however, we found that for real data the hit rate was often less than 1%, resulting in minimal, if any, benefits. This and other optimizations are described in further detail in Section 5.

Parent sets are stored as compact bit vectors (as described in [3]) to minimize the data footprint and simplify the compatibility check. The order being scored is converted to a set of compatibility bit vectors that are “1” for all variables that may be parents for a given node and order. The bit vector representations minimize the amount of data movement needed to load the parent sets and enable the compatibility check to be implemented using fast bitwise operations.

At load time, parent sets are sorted (if not already sorted) and an index is created for the most significant bit (MSB). Instead of testing all parent sets for compatibility, the index is used to skip those parent sets that, based on their MSB, are known to be incompatible. Depending on the node and order, only a fraction of the parent sets must be tested. In the extreme case, the first node in the order only tests the null parent set (the one with no

parents). Skipping known-incompatible nodes increases the critical OP/byte ratio by increasing the number of iterations that actually perform the score accumulation.

An alternate approach to improving the OP/byte is to re-order the loop nest to make the “chains” loop the inner-most loop, and thus reuse each parent set/score across all the chains. However, as will be described in Section 5 this approach increases the inner loop complexity, and is not readily compatible with optimizations that skip some or all of the parent sets, resulting in a net performance degradation.

The expression $\log(1+\exp(x))$ in the score accumulation is only non-linear in a small input range around zero; for $x \ll 0$ and $x \gg 0$ the expression simplifies to 0 and x respectively. The accumulation operation is implemented as:

```
1 diff = score - ns;
   if (diff > 16) ns = score;
   else if (diff < -16) ns = 0;
   else ns += log(1+ exp(diff));
```

Performance and accuracy is sensitive to the selection of the approximation boundary, $[-16,16]$ in this case. The closer the boundary is to zero, the fewer times the slower log and exp operations are invoked, but the more inaccurate the approximation is at the boundary. We developed a static precision analysis tool [13] to automatically determine the closest-to-zero boundary where the error in the computation and approximation are equal. We store the parent scores as 32-bit floats, at this precision the optimal boundary is at ~ 14.6 . Optimizing the approximation boundary improved performance by 10–15% over the original $[-30,30]$ boundary in an already heavily optimized application.

3.1.2 Parallelization—The restarts, tempering chains and node loops offer coarse-grain parallelism, while the parent set loop offers fine-grain data level parallelism (DLP). We experimented with threaded GPP implementations for all three coarse-grain loops, using a work queue-based approach for the latter two. However as will be described in Section 5, only the parallel implementation of the outer-most restart loop yielded speedups of more than 30% on quad-core GPPs.

As a result of the caching, the amount of “work” to score even a whole chain can be as small as a few hash-table lookups. Such little work is often not enough to overcome the threading overhead. When node must be scored, the inner-most loop for 1–2 nodes can effectively saturate the CPU’s main memory bandwidth; thus parallelizing this loop within the same CPU socket results in only modest performance gains. As a result, we focused our GPP-based concurrency efforts on the outer-most restarts loop, which is trivially parallel, and when available, use the GPU to exploit the parallelism in the “node” and “parent” loops.

3.2 GPU Order Sampler Implementation

Modern GPU’s device memory bandwidth is an order-of-magnitude higher than GPP’s main memory bandwidth for applications, like the order sampler, with abundant fine-grain

parallelism. As described previously, order sampler performance is often bandwidth limited; motivating the development of a GPU-based implementation. Using NVIDIA's CUDA [15] programming model we implemented a GPU-accelerated version of the order scorer (Figure 3, lines 7–13) for BNs up to 64 nodes.

A block diagram of NVIDIA GPUs is shown in Figure 4. NVIDIA GPUs implement a single instruction multiple thread (SIMT) execution model (a hybrid of SIMD and SPMD execution models). CUDA kernels are implemented as a grid of concurrent thread blocks, where each block can itself contain hundreds of concurrent SIMT threads. The thread blocks are scheduled across the different multiprocessor units, while the threads within each block are time multiplexed in groups of 32, termed *warps*, onto the 8-wide SIMT lanes within each multiprocessor (shown in Figure 4). Each warp executes the same instruction across all 32 threads, using predicated execution to implement branching within threads.

Each SIMT thread has its own local register space and shares a low-latency, on-chip 16 KB local scratchpad memory with other threads in its thread block. All threads have access to the same global memory. There are additional read-only memory spaces accessible, but they are not used in this application and not described here. The device main memory space is distinct from the host GPP's memory space, and can only be accessed from a GPU-kernel or via the CUDA driver.

The NVIDIA GPUs used in this work (GTX 285, Tesla) provide 30 8-wide SIMT multiprocessor units capable of executing 240 threads simultaneously. To fully exploit the capabilities of these GPUs, and particularly to hide the DRAM access latency, the application must create tens-of-thousands of fine-grain threads. Fortunately, the inner two loops (lines 5–11 in Figure 3) in the order sampler offer abundant parallelism. For example, for a 37 node network with a maximum in-degree of 4, up to 2.7 million SIMT threads could be created to score a single order. However, in practice, such aggressive parallelism is unnecessary and actually is complexity inefficient. Instead we aim to generate approximately 15,000 threads – 60 thread blocks, each with 256 threads – in order to fully utilize the GPU.

On the GPP, prior to GPU kernel invocation, each node of each order is checked against the node score caches to eliminate previously scored nodes. The remaining un-scored nodes are divided to create at least 60 thread blocks, i.e. the scoring of each node might be split across multiple thread blocks. The appropriate pointers into the parent sets/score and the compatibility bit vectors for each thread block are copied to the GPU's memory space, and the GPU kernel invoked via the CUDA driver. The parent sets and scores were previously copied to the GPU once at the beginning of the program, and are reused by each kernel invocation. When the GPU kernel is complete, the computed scores, one per thread block, are copied back to the host GPP and combined as needed to produce the final score for each node in the order.

Each thread block has 256 threads; the t^{th} thread evaluates the $t, t + 256, t + 512, \dots$ parent set. Each thread maintains a private accumulator, which are combined at the end of the kernel using a parallel tree-based reduction. To maximize global memory bandwidth, the

parent sets and scores are carefully laid out and aligned in the GPU’s memory to ensure that accesses to adjacent memory locations by “adjacent” threads in each warp are coalesced into a single multi-word access. To maximize arithmetic throughput, all computations are performed with 32-bit floating point, and the transcendental operations in the score accumulation are implemented using the GPU’s fast hardware intrinsics (special function unit).

4. BUILDING HETEROGENEOUS APPLICATIONS WITH MERGE

Section 3 describes a few of the many ways the order sampler’s extensive parallelism can be exploited. No one of those implementations, however, will be the best, or even correct, choice across the whole range of input and machine configurations listed in Section 2.3. Consistently delivering the best possible performance requires a suite of implementations, each optimized for different execution scenarios. We use the Merge programming model [12, 11] to ease the integration, testing and selection from this suite of implementations.

In the interest of space, the interested reader is pointed to [12, 11] for more information about Merge. Our focus in this paper is to describe how and why we use Merge, and the extensions we made to improve programmer efficiency, enhance the interface between domain and compute-experts, and provide more control over which invariant is invoked.

Merge implements predicate dispatch [14], “bundling” different implementations for the same computation into generic functions that expose the common C/C++ function interface, while internally selecting an appropriate variant. With each variant, the programmer provides a set of annotations, e.g., restrictions on the inputs, available hardware, that describe the invariants for that particular implementation. Variants are ordered by annotation specificity, performance and programmer-supplied hints. Calls to the dispatch wrapper (generic function) are automatically specialized to a particular implementation, based on the above criteria.

The mapping is encapsulated inside the generic function, i.e., it is not embedded in the larger application. This level of indirection serves as the interface between the domain-experts who use these kernels, but whome have only limited exposure to performance programming, and the compute-experts who implement the kernels, but whom are not familiar with the application¹.

The current order sampler has two generic functions, and four specialized variants alongside the original baseline implementation. The potential call graphs within the order sampler are shown in Figure 5. The key generic function is `sample_orders`, the primary entry point for the order sampler. Note that the specialized variants all share the GPP MCMC loop. One of the common trade-offs for Merge-style programming models is that in the pursuit of stand-alone implementations, and easy extensibility, code must be duplicated. We maximize code reuse by passing variant-specific state through shared code using closures, specifically using

¹These two groups correspond to the productivity and efficiency programmers described in “A View from Berkeley” [4].

the blocks extension in the clang C-family front-end [2] that is the basis of our re-implementation of the Merge compiler.

Prior to invoking the MCMC loop function, for example, the optimized implementations initialize variant-specific data structures, e.g., copying the read-only parent scores to the GPU; references to these data structures are captured in the closure, which is then invoked inside the MCMC loop. Using closures enabled us to eliminate more than 300 lines of duplicated code (in a library file with ~ 980 lines total).

All variants in a generic function are interchangeable (subject to their annotations); we leverage this constraint to facilitate testing. When the application is compiled in the new “test” mode, instead of selecting a particular variant, the dispatch wrappers will invoke all applicable variants concurrently in different processes. A special test variant describes how the states of these processes should be compared after the functions-under-test complete. With a test variant, we can define variant “equality” separately from any particular implementation, and quickly test each new implementation against the baseline version.

Separate test variants further define the interface between the domain and compute-expert. The compute-expert does not need to understand what constitutes a valid output at the application level; instead they can simply focus on satisfying the correctness criteria included alongside the baseline variant. This feature complements, but does not replace, basic validation. We still need to ensure that the baseline implementation is correct. Further, the programmer still needs to define what “equals” means in the context of a particular computation. But once that code is place, each new variant can easily be validated against that criteria.

Configuration annotations are use to specify the necessary hardware, e.g., NVIDIA GPU, for a particular variant. These predicates are translated into calls into the runtime environment to determine the availability of specific resources. The runtime environment is initialized using a set a Merge launcher programs, modeled on `mpirun` that allow the user (or resource manager middleware) to allocate resources on a per-run basis. With the launchers the user can control how the application runs on their platform.

5. PERFORMANCE EVALUATION

The BN learning application is in regular use on the platforms described in Table 1. Execution time is wall clock time measured end-to-end – inputs read from disk, output written to disk – with the time utility. Unless otherwise noted, all runs use a single (quad-core) processor and GPU, and the following parameters: 1000 data observations, $k\text{-limit}=4$, MCMC iterations=10000, PT chains=10. The score generator and graph sampler are 4-way multi-threaded, the order sampler is not, as described below.

Figure 6 shows the execution time breakdown on the “consumer” and “pro” workstations for the BN learning application (using parameters listed above) to accurately learn randomly generated graphs. The GPU-based implementation of the order sampler and the complete BN learning workflow is up to $7.5\times$ and $3.6\times$ faster, respectively, than the optimized GPP

implementation on the “consumer” workstation, and $4.3\times$ and $3.1\times$ faster on the “pro” workstation.

To place these results in context, we learned two well known networks: 1) an 11-node STN from human T-cells [17]; and 2) the 37-node ALARM network [1]. To facilitate comparison across implementations that use different parameters, we define a more portable throughput metric, node-parents/s (NP/s). This value is obtained by:

$$\frac{\text{PT Chains} * \text{MCMC iter.} * \text{Nodes} * \text{Parents/node}}{\text{Exec. Time}}. \quad (3)$$

Table 2 shows the throughput in NP/s for the different order sampler implementations for both networks, including the initial baseline (base) GPP implementation (base does not include the memoization and indexing optimizations). We achieve throughput of 4.76 giga-NP/s (GNP/s) and 14.5 GNP/s for the best T-cell and ALARM implementations (bold) respectively². For context, [3] reports 8.8 GNP/s and 9.9 GNP/s for the T-cell and ALARM networks, respectively, for the same algorithm custom implemented on 4 interconnected FPGAs; and 247 kilo-NP/s (KNP/s) and 73.3 KNP/s for a GPP implementation.

The optimized GPP sampler is $6\times$ faster than the baseline for the ALARM network, and represents a best-effort implementation for use in situations where a high-performance GPU might not be available or appropriate. Table 3 breaks down the contribution of the memoization, indexing and numerical optimizations described in Sec 3.1; the Base and +Prec columns correspond to the Base and GPP columns in Table 2. As described previously, node memoization can be particularly effective, contributing 50% or more of the speedup for both networks. The “hit rate” for the ALARM network order sampler run presented is 66%, and can grow to 97+% for longer runs (e.g., with multiple restarts). For the latter, the performance of the hash-table operations become particularly important; optimizing those operations is an area of ongoing work.

Performance for the above runs are largely bandwidth limited. Each iteration of the inner-most loop loads one parent set, and potentially one parent score, but does not reuse either until the next order is scored; by which point, for large networks, the previously loaded sets/scores have been evicted from the caches. The presented ALARM network order sampler run loads 278.3 GB of parent sets/scores, but only performs 35.6 double-precision GFLOPs and 28.9 64-bit bitwise GOPs of computational work. The OP/byte ratio is ~ 0.2 , and deep in the bandwidth limited region as predicted by the Roofline model [18]. As a result, and as shown in Figure 6, order sampler performance is very sensitive to differences in processor bandwidth. For example, running on the GTX 285 GPU, which has a 13% higher clock rate but 55% higher main memory bandwidth than the architecturally similar Tesla GPU, the order sampler GPU kernel is 25% faster for the ALARM network. Interestingly, by using a faster, gaming oriented GPU, the “consumer” workstation can achieve near performance parity with the “pro” workstation, despite its faster Nehalem-generation processors.

²Note that because of the memoization, throughput is non-deterministic and will be different for different numbers of observations or runtime parameters

As described previously, we experimented with other optimizations to improve the OP/byte ratio. We tried restructuring the loop nest to make the PT loop the inner-most loop, and thus reuse each parent set/score across all the chains. However, the increased complexity in the inner-most loop resulted in worse performance for both the GPP and GPU. Additionally some optimizations that are effective on the GPP do not translate to the GPU. On the GPP, using parent set indexing reduced the amount of parent sets/scores loaded for the ALARM network by $\sim 4\times$. However, the increased control flow required resulted in a performance degradation when implemented on the GPU.

The combination of node memoization and indexing results in work imbalances that can make it challenging to achieve meaningful speedups with multi-threaded implementations of the “chain” and “parent” loops. We experimented with parallelizing both loops, however, the best results we obtained were a 30% speedup on our quad-core processors, and only for larger networks. When a node hits in the node hash-tables, there is little work with which to amortize the threading overhead, and when the node must be scored, with such a low OP/byte ratio, little concurrency is needed to saturate the available memory bandwidth. That is not to say that there cannot be a performant multi-threaded implementation; however, such efforts are unlikely to outperform the GPU implementation of order scoring. Thus we focused our resources on the GPU implementation and parallelizing the outer-most “restarts” loop.

Multiple random restarts can (optionally) be used to build confidence that the algorithm has converged to a “good” solution. Each restart is completely independent and so readily parallelized on both shared and distributed memory systems. Table 4 shows the execution time for learning the ALARM network with 50 random restarts on different subsets of the cluster. Random restarts are equally distributed across all processes. Different processes do not share the same node memoization hash-tables (to minimize synchronization); as a result, order sampler (and graph sampler) performance will not scale linearly. As the application runs longer, i.e. for more iterations or restarts, the node hash-table “hit rate” and overall throughput increases. Thus execution time scales sub-linearly with the number of restarts. The best performance we could expect when using the whole cluster, for example, is equivalent to running $\lceil 50/2 \rceil = 2$ restarts on a single GPP core, ~ 164 sec., or 7 restarts using the GPU, ~ 118 sec.

The MPI-based distributed memory implementation is the newest of the order sampler variants, and is very much an area of ongoing work. However, the preliminary implementation has already yielded interesting observations. The user’s choice of order sampler parameters (strategies) not only affects the amount of parallelism, but also the amount of I/O, and the workload for downstream tools. For example, one could achieve a total of 100,000 MCMC iterations with a single chain, 10 PT chains running 10,000 iterations, or 10 restarts with 10,000 iterations. The multi-chain run, because it only saves orders from the lowest energy chain, has 10% of the I/O of the other two strategies, and $10\times$ less work for the graph sampler, yet produces comparable orders. Reducing the I/O can have a real impact on cluster performance, using 10 chains (vs. 1) can reduce execution time several-fold when using all 32 cores. Similarly, using multiple restarts exploits the available parallel implementations while the other strategies do not. Since the CyToF mass cytometer

is just coming online, we cannot yet quantify the performance accuracy-tradeoff of these different strategies. Building just such an understanding is an important area of ongoing work.

As shown in Figure 6 and Table 2 the GPU implementation only delivers a speedup for networks large enough to both cover the GPU-CPU overheads, e.g., data transfer and kernel launch latencies, and generate enough threads to fully utilize the GPU’s many SIMT units. Similarly, the GPP optimizations are not effective for very small networks, or few parents. To deliver the best possible performance across a wide range of inputs and machine configurations, we need to use all the above implementations. We use the Merge annotations to guide variant specialization.

A good parameter to guide selection is total parents, $\sum \Pi_n$. The compute expert supplies with each implementation a predicate, e.g., `total_par > 150000` that indicates the Pparameter range over which that variant will be most effective. Figure 7 shows the performance as a function of number of parents for a 40-node synthetic network. Guided by the annotations, the Merge application delivers near Pareto-optimal performance across the input parameter space. And since Merge generic functions are at the top of the call stack there is negligible performance penalty for the more complex dispatch. The above Merge annotations complement similar restrictions (i.e., `restarts > 1`) on SMP and MPI variants for the “restarts” loop.

6. DISCUSSION AND CONCLUSIONS

Small aberrations in intracellular signaling play an important role in many diseases. Identifying these aberrations can lead to improved clinical diagnostics and therapies. However, we can only find these small differences in the STNs if we can efficiently learn STN structures (modeled as BNs) on a per-individual basis for many individuals. Learning BN structure from experimental data is computationally demanding. In this paper, we present a GPU-accelerated implementation of a MCMC-based algorithm for learning BNs. Our implementation leverages the combined capabilities of the GPP-GPU heterogeneous system to deliver up to a $7.5\times$ speedup over the GPP-only implementation, and equivalent or better performance – at lower cost and with less design effort – than alternative heterogeneous architectures.

The order sampler presented in this work is an optimized reimplement of the algorithm proposed in [3]. In companion work [6], the same algorithm has also been reimplemented on the BEE3 4-FPGA system [7], permitting a meaningful comparison of these two learning systems.

One of the initial datasets produced with the CyToF instrument (and analyzed with both learning systems) is a multi-timepoint analysis of a 22-node STN in Jurkat’s T-cells (a well-known cell line). Table 5 shows execution times for learning networks at a single timepoint from this dataset using the different hardware platforms in Table 1.

The new FPGA implementation performs order and graph sampling with 50 restarts in 45.1, 30.0, and 26.6 seconds using 2, 3 and 4 FPGAs respectively for the above dataset. These results correspond to speedups of 1.4–2.6, 1.1–1.8, .76–1.3 vs. the GPP implementation presented here on the workstations, cluster node and cluster respectively (GPUs are not used in these runs). The retail cost for 4-FPGA platform used is ~\$65,000, greater than 10-fold more expensive than the workstations and 2-fold more expensive than the cluster³. GPP/GPU cost effectiveness will be even better for larger networks that can take advantage of the GPUs (22 nodes is too small to overcome GPU overhead) and that require many FPGAs to assemble sufficient BRAM capacity to store all the parent sets/scores (~40 FPGAs for 40 nodes, K-limit=4 when using the same model of FPGAs as [6]).

Node memoization plays a key role in helping the GPP/GPU system “keep pace” with the custom FPGA implementation. The FPGA’s aggregate BRAM bandwidth (the performance determiner) and the NVIDIA GTX 285’s peak main memory bandwidth are similar. By custom designing the FPGA implementation, though, one can actually leverage all of that bandwidth, and further improve throughput by optimizing the score accumulation. However, all of that BRAM bandwidth must be statically allocated to each node, and cannot be re-targeted away from previously scored nodes/orders. In contrast, the GPU and GPP’s bandwidth is only used for those nodes/orders that have not yet been scored. We leverage the best capabilities of both components of our tightly-coupled (relative to a GPP-FPGA system) heterogeneous system; we use the GPP’s efficiency at implementing large, but ill-structured, hash-tables to minimize the amount of computational work, while using the GPU to accelerate the bandwidth limited, but well-structured, node scoring computation. Note that caching could also be implemented on the FPGA, but has not yet, and so the performance implications are unknown by the authors.

One trade-off with aggressive caching is non-deterministic performance. Unlike the deterministic FPGA implementation, execution time for the GPP/GPUs is a function of how fast the algorithm converges. For example, if we increase the number of CyToF observations to 68,000, the execution time on the “pro” workstation for score generation increases to 13 sec., but the combined time for the order and graph samplers decreases to 26.5 sec. from the 63.4 sec. in Table 5. Thus we cannot provide any execution guarantees to the user. But can also deliver even higher throughput for well-behaved datasets.

This paper focused on a particular BN learning algorithm. However, BN learning is a constantly evolving field, with new algorithms and heuristics being developed regularly. The techniques we used to accelerate order scoring in both the GPP and GPU implementations can readily be used in other order sampling algorithms, such as the recently proposed Equi-Energy sampler [8], to improve execution performance and efficiency.

In this context, we believe that the novel GPU-accelerated BN learning algorithm is a compelling solution for the challenge of learning ever larger BNs from experimental data. We demonstrate up to an $7.5\times$ speedup over the heavily optimized GPP implementation. However, that speedup cannot be realized for all datasets. To deliver the best possible

³Note that the FPGA cost envelope will of course depend on the particular FPGAs boards used for which there are many choices

performance in a wide range of situations we use the Merge programming model to integrate and smartly select among several implementations, each optimized for a different input or machine configuration. Merge enables us to build a scalable and extensible application that can deliver near-Pareto-optimal performance across the available implementations.

Acknowledgments

The authors would like to thank Narges Bani Asadi and Wing Wong for sharing the source code from their previous BN learning publications, and Chris Fletcher for providing the FPGA performance results. Additionally we would like to thank Sean Bendall for the producing the CyToF data, Karen Sachs for discretizing that data, her and Byron Ellis' suggestions on validating the application, and Matthew Ho and David Dill for their help with precision analysis. We would like to thank James Balfour and the anonymous reviewers for their many helpful suggestions which greatly improved this paper. This work was partially supported by the NVIDIA Corporation and NIH grant R01 CA130826-01.

References

1. Bayesian networks repository. <http://compbio.cs.huji.ac.il/Repository/>
2. clang: A c language family frontend for llvm. <http://clang.llvm.org>
3. Asadi NB, Meng TH, Wong WH. Reconfigurable computing for learning bayesian networks. Proc of FPGA. 2008:203–211.
4. Asanovic, K., Bodik, R., Catanzaro, BC., Gebis, JJ., Husbands, P., Keutzer, K., Patterson, DA., Plishker, WL., Shalf, J., Williams, SW., Yelick, KA. Technical Report UCB/EECS-2006-183. University of California; Berkeley: 2006. The landscape of parallel computing research: A view from Berkeley.
5. Bandura D, Baranov V, Ornatsky O, Antonov A, Kinach R, Lou X, Pavlov S, Vorobiev S, Dick J, Tanner S. Mass cytometry: Technique for real time single cell multitarget immunoassay based on inductively coupled plasma time-of-flight mass spectrometry. Anal Chem. 2009
6. Bani Asadi N, Fletcher CW, Gibeling G, Sachs K, Glass EN, Zhou Z, Burke D, Wawrzynek J, Wong WH, Nolan GP. Parallelearn: A massively parallel scalable system for learning interaction networks. Proc of ICS. 2010
7. Chang C, Wawrzynek J, Broderon RW. BEE2: A high-end reconfigurable computing system. IEEE Design and Test of Computers. 2005; 22(2):114–125.
8. Ellis B, Wong WH. Learning causal bayesian network structures from experimental data. Journal of the American Statistical Association. Jun.2008 103:778–789.
9. Geyer CJ. Markov chain monte carlo maximum likelihood. Computing Science and Statistics: Proc of 23rd Symp Interface. 1991:156–163.
10. Irish JM, Kotecha N, Nolan GP. Mapping normal and cancer cell signalling networks: towards single-cell proteomics. Nat Rev Cancer. 2006; 6(2):146–155. [PubMed: 16491074]
11. Linderman MD, Balfour J, Meng TH, Dally WJ. Embracing heterogeneity – parallel programming for changing hardware. Proc of First USENIX Workshop on Hot Parallelism. 2009
12. Linderman MD, Collins JD, Wang H, Meng TH. Merge: A programming model for heterogeneous multi-core systems. Proc of ASPLOS. 2008:287–296.
13. Linderman MD, Ho M, Dill DL, Meng TH, Nolan GP. Towards program optimization through automated analysis of numerical precision. Proc of CGO. 2010 page to appear.
14. Millstein T. Practical predicate dispatch. Proc of OOPSLA. 2004:345–264.
15. NVIDIA. NVIDIA CUDA Compute Unified Device Architecture Programming Guide (2.0). 2008
16. Pearl J. Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann. 1988
17. Sachs K, Perez O, Pe'er D, Lauffenburger DA, Nolan GP. Causal protein-signaling networks derived from multiparameter single-cell data. Science. 2005; 308(5721):523–529. [PubMed: 15845847]

18. Williams SW, Waterman A, Patterson DA. Roofline: An insightful visual performance model for multicore architectures. *Comm of the ACM*. 2009; 52(4):65–76.

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

Figure 1.

Clinical outcome for different STN structures as determined from flow cytometry data (adapted from [10]). The upper panels show cell population histograms of STAT3 and STAT5 activation (phosphorylation) for different patients with and without stimulation. Differing activation, e.g., G-CSF increases STAT activity in patients B and C but not in Z, indicates different network structures (bottom) and is correlated with different clinical outcomes.

Figure 2.

Block diagram of BN learning algorithm using order-based MCMC sampling. Inputs are a table of data observations, and a list of potential parents to be considered for each node. The parent scores are the table of local probabilities of each parent set for each node as computed by Equation 2. The output of the order sampler is a list of orders and their scores (Equation 1), from which high-scoring graphs are sampled.

```

for all restarts do
  initialize all  $\square$ 
  for  $i = 0$  to  $\text{maxiter}$  do
    for all  $\square_c$  in chains do
5:       $\square'_c := \text{swap\_nodes}(\square_c)$ 
       $\square'_c.\text{score} := 0$ 
      for all nodes  $n$  in  $\square'_c$  do
         $ns := -\text{LARGE\_NUMBER}$ 
        for all parents  $p$  in  $\Pi_n$  compatible with  $\square'_c$  do
10:           $ns := ns + \log(1 + \exp(p.\text{score} - ns))$ 
        end for
         $\square'_c.\text{score} := \square'_c.\text{score} + ns$ 
      end for
      if  $(\square'_c.\text{score} - \square_c.\text{score})/T_c > \log(u)$  then
15:         $\text{save}(\square'_c)$ 
         $\square_c := \square'_c$ 
      end if
    end for
    for  $\square_c$  in chains do
20:      if  $((\frac{1}{T_c} - \frac{1}{T_{c+1}}) * (\square_{c+1}.\text{score} - \square_c.\text{score})) >$ 
         $\log(u)$  then
         $\text{exchange}(\square_c, \square_{c+1})$ 
      end if
    end for
  end for
25: end for

```

Figure 3.

Pseudo-code for parallel tempering order sampler. Symbols follow the conventions in the text, T_c is the temperature of chain c , u is a uniformly distributed random number in $(0,1]$.

Figure 4. NVIDIA GPU block diagram showing internal architecture and system connectivity

Figure 5.
Schematic of order sampler call graph showing potential paths through the application

Figure 6. Execution time for BN learning of randomly generated graphs for different order samplers.

Figure 7.

Order sampler execution time on the “pro” workstation for 40 node network with different number of parents for base, GPP and GPU variants and the Merge application that integrates all three.

Table 1

Evaluation platforms

Platform	Description
“Consumer”	2.66 GHz Intel Core 2 Quad with 8 GB RAM, 1.48 GHz NVIDIA GTX 285 GPU with 2 GB RAM, Centos 5.3 Linux
“Pro”	3.07 GHz Intel Core i7 with 12 GB RAM, 1.3 GHz NVIDIA Tesla c1060 GPUs with 4 GB RAM, Ubuntu 9.10 Linux
“Cluster”	Four nodes with dual 2.26 GHz Intel Xeon 5520 processors, 12 GB RAM and dual 2.3 GHz Tesla GPUs per node (in 2 s1070 units), GigE, Rocks 5.3 cluster Linux

Table 2

Order sampler execution time and throughput in node-parents/second (NP/s) for a 11-node human T-cell STN and the 37-node ALARM network on the “pro” workstation. 5000 and 5400 observations were used respectively.

	Base	GPP	GPU
T-cell (seconds)	1.21	0.13	2.9
T-cell (NP/s)	0.51e9	4.76e9	0.21e9
ALARM (seconds)	378.3	62.8	19.0
ALARM (NP/s)	0.73e9	4.39e9	14.5e9

Table 3

Order sampler GPP execution time for subsets of optimizations for 11-node T-cell and 37-node ALARM networks on the “pro” workstation.

Time (s)	Base	+Node memo.	+Index	+Prec.
T-cell	1.21	0.14	0.13	0.13
ALARM	378.3	125.4	71.1	62.8

Execution time (seconds) to learn the 37-node ALARM network with 50 random restarts using different subsets of the cluster. One process per core or GPU.

Table 4

	Score Gen.		Order Sampler			Graph Samp.	
	GPP	GPP	1 GPU	2 GPU	8 GPU	GPP	GPP
1 core	82	4184	771			114	
1 node	4 cores	25.2	1469		412	47.3	
	8 cores	13.9	795			31.3	
cluster MPI	32 cores	7.1	229		121	17.2	

Table 5

Execution time learning 22-node STN on different hardware platforms. Input: 10,000 observations. Parameters: k-limit=4, 100,000 MCMC iterations, no parallel tempering, 50 restarts (parameters chosen to permit direct comparison with [6]).

Time (seconds)	Cusmr	Pro	Clust.	Node	Clust.
Score Gen.	2.8	2.1		1.65	
Order Sampl.	49.12	44.6	31.5		18.0
Graph Sampl.	20.18	18.8	16.9		16.5