Development of a GPU-Accelerated Computational Platform for Parallel Inference of Bayesian Networks

Jacinta Roberts, Dr Paul Wu and D. Prof. Kerrie Mengersen

QUT Vacation Research Experience Scheme (VRES)

Submitted: February 25th, 2021

Abstract. Bayesian analysis has been used extensively in statistical decision theory to define custom models that incorporate data and expert knowledge to deliver insights into a myriad of scientific fields. Bayesian networks (BNs) are a useful tool to visualize probabilistic models, quantify interrelationships and make inferences to support decisions and understanding of real-world applications. Inference with Bayesian networks resides in the NP-hard complexity class which can render data analysis extremely time-consuming, or even prohibitive. In some cases, highly specialised optimisations can mitigate these issues, however, they are difficult to implement and generalise to all models. The purpose of this project is to design and implement a massively parallel computational platform which uses a Graphics Processing Unit (GPU) to accelerate the resolution of an approximate inference-based algorithm (MCMC) for Bayesian networks. To harness the high-computing performance of GPUs, NVIDIA's parallel programming model called "CUDA" will be used.

This paper demonstrates that GPUs present significant advantages in terms of the algorithm runtime and scalability compared to general-purpose central processing units (CPUs). By using CUDA, the execution time was reduced which resulted in 10-100 orders of magnitude in speedup for the parallelized MCMC algorithm. This was shown to vary depending upon the sample size, the number of dimensions and GPU architecture used. Further work is required to integrate the GPU-based solution into R for it to be used on existing Bayesian networks. By utilising this accelerated approach, Bayesian network learning and subsequent inferences could be performed in a timescale that enables real-time applications and inference in increasingly complex models and modelling scenarios.

Keywords: Bayesian networks, inference, MCMC, GPU, CUDA, parallel computing

Table of Contents

1.0 Introduction

1.1 Significance of Work

A Bayesian Network is a probabilistic graphical model that represents a set of variables and their conditional dependencies using data and expert knowledge (Pearl, 1988). They can be used for inference to estimate the probabilities for causal or subsequent events to support real-world decisions in a variety of fields including healthcare, energy, finance, sports, and sustainability (Korb & Nicholson, 2010). However, one of the major obstacles to using Bayesian methods for pattern recognition has been its computational expense. For most problems of realistic size, the computational power required for the joint grows exponentially (and the BN framework reduces it down to NP-hard) which can make analysis extremely time-consuming or prohibitive (Korb & Nicholson, 2010). The focus of this project was to implement an accelerated approach that computed Bayesian Inference in a shorter time and enabled real-time analysis on increasingly complex models.

The R statistical environment does not make use of parallelism by default, however with the amount of data continuing to grow, researchers and data scientists may resort to expensive solutions such as cluster hardware for large analysis tasks. Graphics processing units (GPUs) provide an inexpensive and computationally powerful alternative (Silva, 2010). Using R and the CUDA toolkit from Nvidia, this project focuses on implementing Bayesian Inference for GPU-equipped computers using a MCMC algorithm.

1.2 Context

This project was conducted as part of the Vacational Research Experience Scheme at QUT over the summer of 2021 which provides undergraduate students an opportunity to conduct research which has the potential to inform their perspectives on higher research degree pathways.

1.3 Project Objectives

The initial objectives of the research project were as follows:

- HLO1 Review existing approaches to parallel computation for Bayesian Network (BN) inference.
    - HLO1.1 Research and document key considerations and limitations in interfacing between R and C code on a GPU.
    - HLO1.2 Investigate what BNs are, how they are used, how inference is performed and the hidden Markov model (HMM) special case.
    - HLO1.3 Review the literature on parallel or High-Performance Computing (HPC) implementations of BNs.
- HLO2 Develop and pilot code to interface between R and simple C code on the GPU.
- HLO3 Develop and pilot a simple BN inference and learning algorithm for C code on the GPU.
- HLO4 Case study: develop and test using this code a pilot model of energy states and power output for triathlon cycling. (Note: that this objective was not able to be fulfilled due to time restrictions).

2.0 Literature Review

This literature review discusses a variety of topics including interfacing between R and C code on a GPU (2.1), an overview of BNs, inference algorithms and the HMM special case (2.2) and parallel and HPC implementations of BNs (2.3).

2.1 Interfacing between R and C code on a GPU

2.1.1 Comparison of Computing Platforms - CPUs and GPUs

Two major computing platforms are deemed suitable for processing data in a timely manner. The first is the CPU (central processing unit), which can run numerous types of applications and has recently provided multiple cores to support parallel processing. The second is the GPU (graphics processing unit), which was originally designed to accelerate graphics rendering with many small processing units. Over time, the GPU has evolved into a processor with unprecedented floating-point performance and programmability and today's GPUs greatly outperform CPUs in arithmetic throughput and memory bandwidth (Lee et al., 2010, p. 451).

CPUs and GPUs have different architectures and are built for different purposes. The CPU is suited to a wide variety of workloads, especially those for in which latency or per-core performance are highly important. The CPU focuses its smaller number of cores (typically 4 to 8) on individual tasks and on getting things done quickly which makes it uniquely well equipped for jobs ranging from serial computing to running databases (Lounis et al., 2015).

To compare, modern GPUs implement the single instruction, multiple threads (SIMT) execution model in parallel computing where single instruction, multiple data (SIMD) is combined with multithreading. This means that all shader processors (or CUDA cores) are scalar. They can carry out one float and one integer instruction per clock cycle on one data component (note, though, that the instruction itself might take multiple clock cycles to be processed), but the scheduling units organise them into groups so that vector operations can be performed.

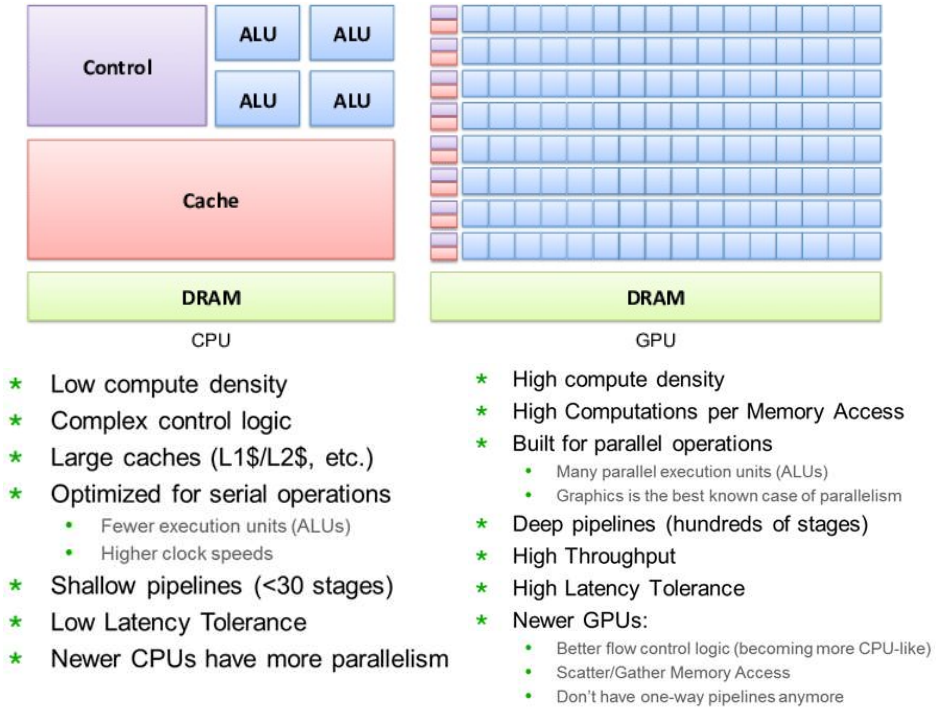

Figure 1 *– High-level differences between CPUs and GPUs (Lounis et al., 2015).*

2.1.2 Comparison of Programming Models – OpenCL and CUDA

Modern graphics processing units (GPUs) have evolved into general purpose computing devices, with the advent of general-purpose parallel programming models, such as CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language). These models have seen significant adoption in deep learning and in high performance computing due to the GPU's high compute power.

The primary difference between CUDA and OpenCL is that CUDA is a proprietary set of language extensions that works only on NVIDIA's GPUs whereas OpenCL is an open standard that can be used to program a variety of devices from different vendors (such as AMD). Although OpenCL promises a portable language for GPU programming, its generality may reduce performance. In research from Fang et al. (2011), 16 benchmarks ranging from synthetic applications to real-world ones and the results showed that for most applications, CUDA performed 30% better than OpenCL. However, this difference was mostly due to unfair comparisons such as programming model differences and compiler differences. For example, a programming model difference is that NDRange in OpenCL represents the number of work-items (or threads) in the whole problem domain, whereas GridDim in CUDA is the number of blocks of threads. Another difference were native kernel optimisations which could indicate whether to use shared memory, employ vectorization, unroll loops, reduce bank-conflicts, use texture memory, or access global memory in a coalesced way. OpenCL was shown to achieve similar performance to CUDA by implementing systematic code changes to tune the compiler and execution configuration parameters. It was concluded that OpenCL's portability did not fundamentally affect its performance (Fang et al., 2011).

2.1.3 An Overview of the GPU Architecture

Streaming Multiprocessors (SMs) perform the computation within a GPU, with each SM having its own control units, registers, execution pipelines and caches. There are many CUDA cores (shader processors) per SM, with the exact configuration depending upon the GPU microarchitecture implemented. CUDA cores are flexible mathematics pipelines which have CUDA threads mapped onto them in a synchronised manner to do continuous computation. A CUDA core is a single computational resource that accepts instructions (a shader program) and executes it. A CUDA core itself contains dispatchers, operand collectors, result queues and Arithmetic Logic Units (FP Unit and INT Units) (Figure 2). The Arithmetic Logic Unit (ALU) is the base unit that performs the mathematical operations in the shader program.

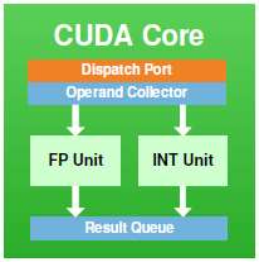

Figure 2 *– Fundamental components of a CUDA Core (NVIDIA, 2009).*

For this project, the Nvidia GeForce RTX 2060 will be used which has 1,920 shader processors (CUDA cores) and uses the Turing TU104 GPU microarchitecture for SMs (Figure 3). It has 30 SMs which each contain 64 CUDA cores per SM. This microarchitecture is partitioned into 4 processing blocks, with each block containing 16 FP32 ALUs, 16 INT32 ALUs, one warp scheduler and one dispatch unit. Each Turing SM supports parallel execution of FP32 and INT32 operations and independent thread scheduling (NVIDIA, 2018). There are various other GPU microarchitectures from Nvidia including the Fermi, Kepler, Pascal, Volta and more. The most significant change over the years, other than there simply being more units, involves how they are arranged and sectioned.

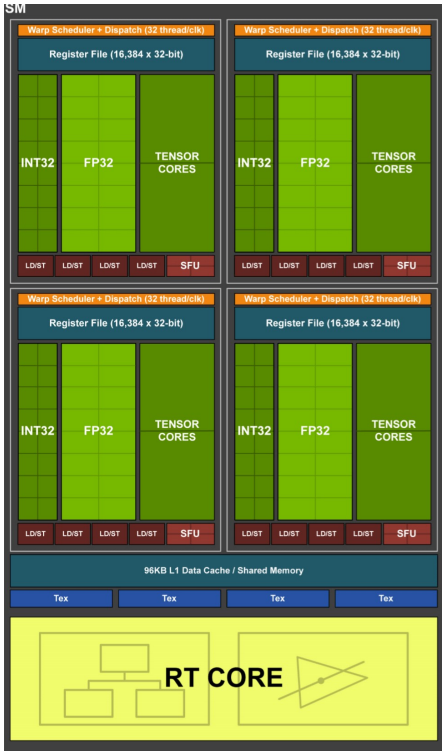

Figure 3 *– NVIDIA's Turing GPU Microarchitecture used in the GeForce RTX 2060 (NVIDIA, 2018).*

2.1.4 Scalable Parallel Programming with CUDA

The CUDA parallel programming model has three key abstractions at its core: a hierarchy of thread groups, shared memories, and barrier synchronisation. These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. A CUDA application will take advantage of heterogeneous computing architectures to execute serial code in a Host (CPU) thread and parallel code in many Device (GPU) threads (Figure 4).

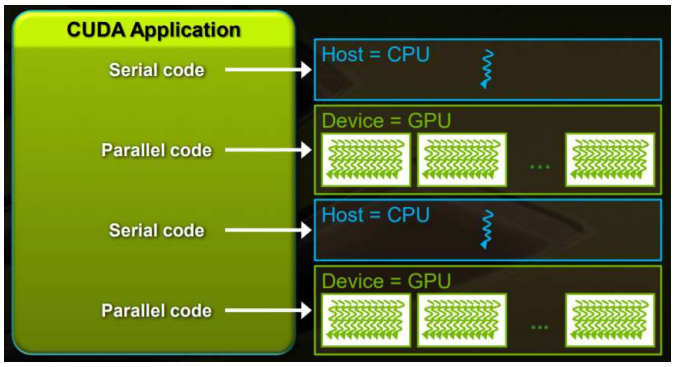

Figure 4 *– Anatomy of a CUDA Application (Messmer, 2013).*

This parallel portion of the application that executes on the GPU by an array of threads is known as the kernel. Within the kernel, all threads run the same code and each thread has an ID that it uses to compute memory addresses and make control decisions. Each thread is executed by a CUDA core and these threads are grouped into blocks. Each of these blocks is executed by one SM and does not migrate. A kernel is then executed as a grid of blocks of threads (Figure 5).

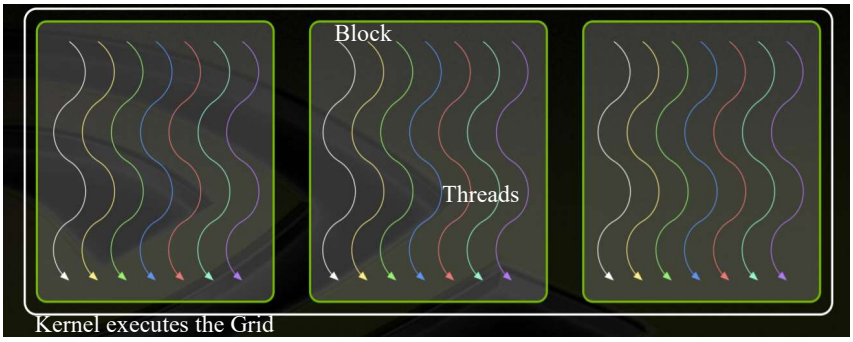

Figure 5 *– Fundamental components of a CUDA Core (Messmer, 2013).*

Thread blocks enable scalability as they can execute in any order, concurrently or sequentially which enables a kernel to scale across any number of SMs (Figure 6) (Messmer, 2013).

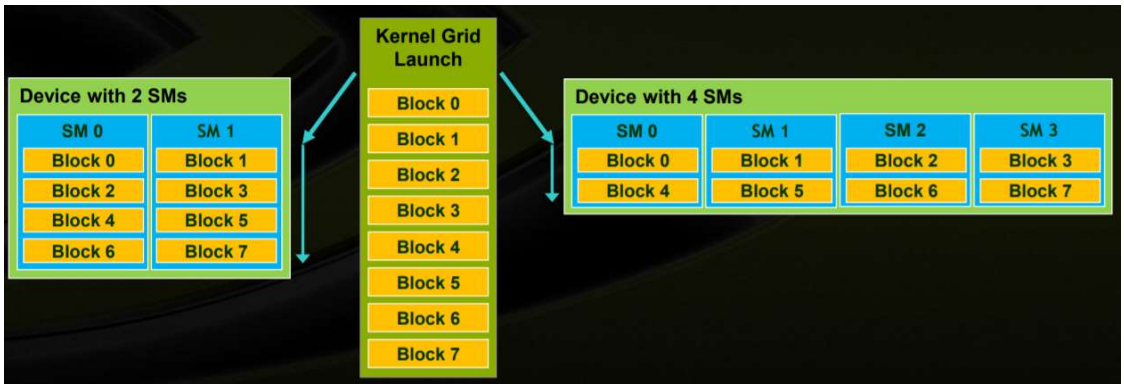

Figure 6 *– Kernel Grid with 8 blocks of threads scaling to devices with 2 or 4 SMs (Messmer, 2013).*

GPUs achieve high performance by executing thousands of threads concurrently. To simplify GPU design, the neighbour threads are bundled in grouped referred to as warps. Employing warp-level granularity simplifies the thread scheduler significantly as it facilitates using coarse-grained schedulable elements. In parallel computing, optimal performance is achieved when there is a balance between the two extremes of fine-grained (which improves speedup but can cause synchronisation overheads) and coarse-grained parallelism (which reduces communication overhead but can cause load imbalance). In addition, this approach keeps many threads at the same pace providing an opportunity to exploit common control flow and memory access patterns. Unlike grids and blocks, warps are an implementation detail that is not directly accessible by the programmer.

In an NVIDIA GPU, a warp is a collection of threads, 32 in current implementations, that are executed simultaneously by a Streaming Multiprocessor (SM) (Figure 7). Multiple warps can be executed on a SM at once. When a warp of CUDA threads is eligible for issuing, it takes 32 CUDA cores to run on and gets locked onto it. This makes 32 CUDA cores work as a team, to implicitly do a lock-step iteration of CUDA kernel instructions. The mapping between warps and thread blocks can affect the kernel's performance. Threads are grouped

into thread blocks to improve data mapping and communicate with each other more efficiently using shared memory that is only accessible by threads in the same thread block. Therefore, it is a good practice to keep the size of a thread block a multiple of 32 to optimise computing efficiency and facilitate memory coalescing. Memory coalescing is a technique which allows optimal usage of the global memory bandwidth because if the threads in a block are accessing consecutive global memory locations, then all the accesses are combined into a single request (or coalesced) by the hardware. Furthermore, a range of 128 to 256 threads is a recommended starting point for the number of threads in a block as the GPU architecture is optimised for calculating and swapping out threads waiting on memory reads.

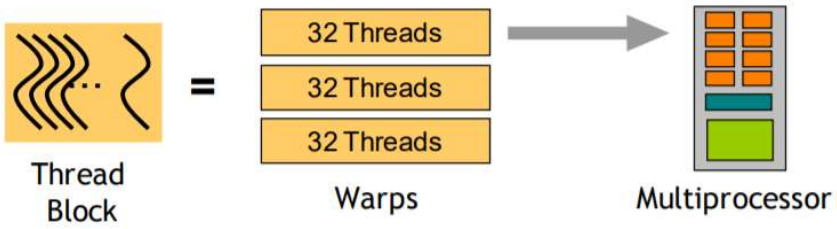

***Figure 7** – Mapping threads to warps (New York University Centre for Data Science, 2017).*

There are numerous factors involved in selecting block size, and some experimentation is often required. However, a few rules of thumb should be followed:

- Threads per block should be a multiple of warp size (which is 32 for most GPUs) to avoid wasting computation on warps that are not at full capacity and to facilitate memory coalescing.
- A minimum of 64 threads per block should be used, and only if there are multiple concurrent blocks per multiprocessor.
- Between 128 and 256 threads per block is a good initial range for experimentation with different block sizes.
- Use multiple thread blocks rather than one large thread block per multiprocessor if latency affects performance. This is particularly beneficial to kernels that frequently call *__syncthreads()* (which is a block level synchronization barrier) as thread blocks that aren't waiting for *__syncthreads()* can keep the hardware busy (NVIDIA, 2021).

From the programmer's perspective, an appropriate block dimension and grid dimension are crucial to achieving computational efficiency (time, resource utilisation, shared memory, coalescing) and they must be specified when launching the kernel. One of the most important CUDA design decision is to figure out how to split up tasks into threads. The reason why NVIDIA did not choose to make all the threads in one large block where every thread can communicate with each other, is because GPUs have different numbers of SMs but CUDA programs need to be able to run on any of them (consider a GPU with 1 SM). The only way to do this is to introduce the concept of a block so that GPUs with a smaller number of SMs can still operate serially, whereas a powerful GPU can run many blocks in parallel. Another important consideration from a hardware perspective, is that if the block is too large, then the circuitry overhead for the shared memory for the threads may become too large.

2.1.5 GPU Memory Hierarchy

Each thread block has its own shared memory (per-block shared memory) which is accessible only by threads within the block and is much faster than local or global memory.

However, it requires careful handling to ascertain optimal performance. Each thread also has its own private local memory which only exists for the lifetime of the thread and is generally handled by the compiler.

For the Turing GPU microarchitecture used in the NVIDIA GeForce RTX 2060, each block includes a new L0 instruction cache and a 64 KB register file. The four processing blocks share a combined 96 KB L1 data cache/shared memory which can be divided into 32 KB shared memory and 64 KB L1 cache, or alternatively, 64 KB shared memory and 32 KB L1 cache (NVIDIA, 2018).

2.1.6 Key Considerations and Limitations

When using CUDA, there are various key considerations such as possible memory bottlenecks and launch parameter configuration optimisation. Although shared memory and registers are high-speed memories with large bandwidth, they are available in limited amounts in a CUDA device. A programmer should be careful not to overuse these limited resources. The limited amount of these resources also caps the number of threads that can execute in parallel in a SM for a given application. The more resources a thread requires, the smaller the number of threads that can simultaneously reside in the SM due to a lack of resources.

To launch a CUDA kernel, the block dimension and grid dimension need to be specified by the host (CPU) code. If only a single value is supplied for each parameter, then the *y* and *z* dimensions are assumed to be 1. For example, to launch a kernel named "square" with 1 block of 64 threads: *square<<<1,64>>>* or equivalently, *square<<<dim3(1,1,1), dim3(64,1,1)>>>*. More generally, this can be expressed as:

KERNEL<<<DIM OF GRID OF BLOCKS, DIM OF BLOCK OF THREADS>>>(...)
KERNEL<<<dim3(bx,by,bz), dim3(tx, ty, tz)>>>(...)

The configuration parameters are important to ensure that the entire GPU is kept busy. The number of threads per block should be a multiple of 32 threads so that all warps within that block use all cores on that multiprocessor. The number of blocks in a grid should be larger than the number of multiprocessors so that all multiprocessors have at least one block to execute (e.g., for the GeForce RTX 2060, use at least 30 blocks as there are 30 SMs).

It is important that the block and grid variables do not exceed the boundary values for the GPU device architecture, otherwise the kernel will not be launched. For the NVIDIA GeForce RTX 2060, the max dimension size of a thread block (x, y, z) is (1024, 1024, 64) and the grid size (x, y, z) is (2147483647, 65535, 65535). The maximum number of threads per block is 1024 (with older GPUs tending to only support 512). For example, a possible thread block configuration may be 1024×1×1 (a one-dimensional thread block), or 16 × 16 × 2 (a three-dimensional thread block equivalent). In all cases, the CUDA model restricts the number of threads per block. These dimension boundaries can be ascertained by executing the sample code, '*deviceQueryDrv*', that comes with the CUDA installation.

2.2 Bayesian Networks

2.2.1 Key Concepts

G (DAG). A Bayesian network consists of a set of nodes (vertices) and a set of directed edges (Figure 8). The nodes represent the variables, and the edges represent conditional dependencies (Ben-Gal, 2008).

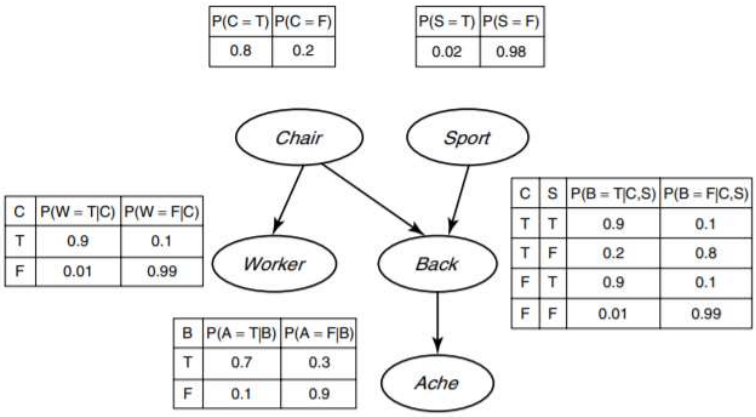

Figure 8 *– Example of a simple Bayesian Network (Ben-Gal, 2008).*

Bayesian networks can visualise probabilistic models, quantify interrelationships and make inferences to support real-world decisions in a variety of scientific fields. They are a clean and clear language to handle uncertainty with many practical applications including inference (P(this problem | these symptoms)), anomaly detection (event rareness) and active data collection (how to choose the next diagnostic test given some observations).

The parameters follow the Markovian property, where the conditional probability distribution (CPD) at each node depends only on its parents. For discrete random variables, the conditional probabilities are represented in a table which denotes each combination of values of its parents. The joint distribution can therefore be calculated by these local conditional probability tables (CPTs). However, if there is a joint distribution to be calculated for every single variable (dimension), the computational complexity is $O(2^n)$ so the time required to solve the problem using any currently known algorithm increases rapidly as the size of the problem grows. The complexity class of general querying of Bayes nets is NP-complete (which means that it is possible to verify if a solution is correct in polynomial time).

2.2.2 The Hidden Markov model (HMM) special case

Hidden Markov models are a special case of BNs where the modelled system is assumed to be a Markov process with an unobserved state sequence. In Markov models, the

state is directly visible to the observer, and the state transition probabilities are the only parameters to be considered. To compare, in a hidden Markov model, only the series of events (symbols), which depend upon the internal factors (hidden states) are observable (Amayri et al., 2017). An HMM is specified by the following components:

$Q = q_1 q_2 \dots q_N$	A set of N **states**
$A = a_{11} \dots a_{ij} \dots a_{NN}$	**Transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. the sum of the a_{ij} for each state i is 1.
$O = o_1 o_2 \dots o_T$	A sequence of T **observations**, each one drawn from a vocabulary $V = v_1, v_2, \dots, v_V$.
$B = b_i(o_T)$	A sequence of **observation likelihoods** (**emission probabilities**) each expressing the probability of observation o_T being generated from a state i.
$\pi = \pi_1, \pi_2, \dots, \pi_N$	An **initial probability distribution** over states, where π_i is the probability that the Markov chain will start in state i. (Note that some states may have $\pi_j = 0$ meaning that they cannot be initial states).

Some of the common algorithms and the problems HMMs are applied to include:

- *Forward-backward procedure* – given a model and sequence of observations, how can we compute the probability that the observed sequence was produced by the model?
- *Viterbi algorithm* – how do we uncover the hidden part of the model (the state sequence)?
- *Baulm-Welch algorithm* - How do we optimise the model parameters to best describe how the observed sequence comes about?

2.3 Bayesian Inference

2.3.1 Key Concepts

The dominant computational task in Bayesian inference is numerical integration. The goal of Bayesian inference is to update the probability for a hypothesis as more evidence or information becomes available. Inferences can be drawn based on the posterior distribution using Bayes' theorem given the observed data (D).

$$p(x|D) = \frac{p(x,D)}{p(D)} = \frac{p(x,D)}{\int_x p(x,D)dx} \tag{1.1}$$

In realistic scenarios, there will be numerous unknowns (y, z) which must be marginalised out of the joint distribution which involves further integration (Minka, 2001).

$$p(x|D) = \frac{\int_{y,z} p(x,y,z,D)}{\int_{x,y,z} p(x,y,z,D)dx} \tag{1.2}$$

Therefore, numerical integration is imperative to practical Bayesian inference. Numerical integration algorithms can be divided up into two classes which are either deterministic (exact) or non-deterministic (approximate). The deterministic method seeks to

approximate the integrand with another integral that is known exactly by using the integrand's various properties such as its maxima and curvature. To compare, non-deterministic methods sample from the integrand to get a stochastic estimate. The latter approach is more general and works for almost any integrand, and exact inference methods have complexity that is exponential in the network's treewidth which can make them computationally intractable (Minka, 2001). Table 1 provides a brief overview of some common algorithms for Bayesian inference.

Name of Method	Type	Description
Variable Elimination	Exact	Eliminates (by integration or summation) the non-observed non-query variables one by one by distributing the sum over the product.
Clique Tree Propagation	Exact	Caches the computation so that many variables can be queried at one time and new evidence can be propagated quickly.
Importance Sampling	Approximate	Sample from a distribution and reweigh the samples in a principled way, so that their sum approximates the desired integral (has large, possibly infinite variance so it is an unreliable estimator).
Markov chain Monte Carlo Sampling Methods (Metropolis-Hastings, Gibbs, Reversible-jump)	Approximate	Constructs a Markov chain that has the desired distribution as its equilibrium distribution and samples of the desired distribution can be taken by recording the chain's states. The more steps that are included, the more closely the distribution of the sample matches the desired distribution.
Variational Bayesian Methods (Expectation Propagation, Mean Field)	Approximate	Iterative approach that leverages the factorization structure of the target distribution by minimizing the KL (Kullback-Leibler) divergence.

Table 1 *– An overview of common Bayesian inference methods (Wainwright & Jordan, 2007).*

2.3.2 Markov Chain Monte Carlo (MCMC)

Recently, statistical literature on parallel and distributed inference has begun to emerge, with the current focus on "embarrassingly parallel" approaches. One class of algorithms that is well-suited to this approach is Markov chain Monte Carlo (MCMC). In a distributed approach, samples are obtained independently in parallel on local machines, which are then communicated to a central node, and combined to form an approximation to the posterior distribution.

MCMC methods use an adaptive proposal $Q(x'||x)$ to sample from the true distribution $P(x)$ using a Markov Chain. These methods can be used to draw samples from high dimensional distributions without knowing much about the distribution. In MCMC, sample z_{i+1} is drawn from a transition probability $T(z_{i+1}|z_i)$ where z_i is the previous sample. Therefore, the samples $(z_1, z_2, ...)$ form a Markov chain. The transition probability depends on an adaptive proposal density $q(z_{i+1}|z_i)$ and an acceptance rule.

The major benefit of using MCMC is that it guarantees asymptotically exact recovery of the posterior distribution as the number of posterior samples grows. However, MCMC methods may require a prohibitively long time, as most methods must perform $O(N)$ operations to draw a sample. Furthermore, MCMC methods may also require a significant number of "burn-in" steps before beginning to generate representative samples.

Metropolis-Hastings Algorithm

The Metropolis-Hastings algorithm designs a Markov process by constructing transition probabilities from the proposal density $q(x'|x)$ which can be any fixed density from which samples can be drawn from (Wainwright & Jordan, 2007). The steps of the algorithm are:

1. Pick an initial state x at random.
2. Sample from the proposal distribution as $x \sim q(x'|x)$.
3. Accept with probability $\min\left(1, \frac{p(x')q(x|x')}{p(x)p(x'|x)}\right)$.
4. If rejected, the next state is a repeat of the current state.
5. Iterate Step 2 to 4.

In Metropolis-Hastings, the form of the proposal distribution evolves based on the present state. A common choice of family for the proposal distribution is the isotropic Gaussian. The choice of the variance for this distribution is critical for the performance of the sampling. A smaller variance will increase the number of accepted samples but will reduce the speed of search in of the space. Thus, choosing a proposal distribution forces us to perform a trade-off between speed and acceptance rate, and requires care.

One disadvantage of the Metropolis algorithm is that it is sensitive to step size. Small step size may cause a random walk whereas a large step size may cause a high rejection rate. Slice sampling is a suggested method to overcome this by automatically adjusting step size to fit the distribution (Wainwright & Jordan, 2007).

Gibbs Sampling – A special case of Metropolis-Hastings

Gibbs sampling is a method for sampling from distributions over at least two dimensions. It can be viewed as a Metropolis-Hastings method in which a sequence of proposal distributions $q(x|x')$ are defined in terms of the conditional distributions of the joint distribution $p(x)$. It is assumed that, whilst $p(x)$ is too complex to draw samples from directly, its conditional distributions $p(x_i|x_{j\neq i})$ are tractable to work with. The algorithm is as follows:

1. Initialize $x_1, x_2, \ldots, x_K$;
2. Sample conditional distribution: $x_1^{(t+1)} \sim p(x_1|x_2^{(t)}, x_3^{(t)}, \ldots, x_K^{(t)})$;
3. Sample conditional distribution: $x_2^{(t+1)} \sim p(x_2|x_2^{(t)}, x_3^{(t)}, \ldots, x_K^{(t)})$;
4. Sample conditional distribution: $x_3^{(t+1)} \sim p(x_3|x_2^{(t)}, x_3^{(t)}, \ldots, x_K^{(t)})$;
5. Iterates Step 2 to 4 for all variables.

There are numerous advantages of Gibbs sampling including: it is easy to evaluate the conditional distributions, conditionals may be conjugate and can then be sampled from exactly, conditionals will be lower dimensional and rejection sampling or importance sampling can be applied. However, the major drawback is that when variables have strong dependencies it is difficult to move around. We can introduce auxiliary variables to help move around when such high dimensional variables are correlated (Wainwright & Jordan, 2007).

3.0 Research Progress

3.1 Methodology

Whilst there are numerous parallel implementations of Bayesian Inference available including the Sequential MC, Consensus Monte Carlo, Latent Dirichlet allocation (LDA), the embarrassingly parallel MCMC algorithm presented by Neiswanger et al. (2014) was implemented as it had minimal communication overheads and was simplest of the methods explored to understand.

To allow for quicker burn-in and sampling in settings where data are partitioned among machines, Neiswanger et al. (2014) developed the following approach: on each machine, run MCMC on only a subset of the data (independently, without communication between machines), and then combine the samples from each machine to construct samples from the full-data posterior distribution. Unlike previous strategies, this strategy does not involve multiple, duplicate chains (as each chain uses a different portion of the data and samples from a different posterior distribution), nor does it involve parallelizing a single chain (as there are multiple chains operating independently).

Neiswanger et al. (2014) partitions N independent and identically distributed data points $x^N = \{x_1, \dots, x_N\}$ into M subsets and taking a power $\frac{1}{M}$ of the prior in each case. Samples from the *subposterior* (the posterior given a data subset with an underweighted prior) in parallel, and then combines the samples to form samples from the full-data posterior. More formally, the procedure is:

1. For $m = 1, \dots, M$ (in parallel):
   Sample from the subposterior p_m where:

$$p_m(\theta) \propto p(\theta)^{\frac{1}{M}} p(x^{n_m}|\theta) \quad [1]$$

2. Combine the subposterior samples to produce samples from an estimate of the subposterior density product $p_1, \dots, p_M$ which is proportional to the full-data posterior.

$$p_1 \dots p_m(\theta) \propto p(\theta|x^N) \quad [2]$$

Neiswanger et al. (2014) suggests to implicitly sample from the product of nonparametric density estimates which allows asymptotically exact samples to be produced from the full posterior as a consistent density product estimator is used. Given T samples $\{\theta_{t_m}^m\}_{t_m=1}^T$ from a subposterior p_m, the kernel density estimator $\hat{p}_m(\theta)$ is:

$$\hat{p}_m(\theta) = \frac{1}{T} \sum_{t_m=1}^{T} \frac{1}{h^d} K\left(\frac{\left\|\theta - \theta_{t_m}^m\right\|}{h}\right)$$

$$\hat{p}_m(\theta) = \frac{1}{T} \sum_{t_m=1}^{T} N_d\left(\theta|\theta_{t_m}^m, h^2 I_d\right)$$

Where a Gaussian kernel with parameter h has been used and the nonparametric density product estimator for the full posterior is the product for the M subposteriors. As each (subsample) non-parametric estimate involves T terms, the resulting sample from the true posterior distribution contains T^M terms and it is suggested to use an independent Metropolis-

within-Gibbs (IMG) sampler to handle this complexity (Neiswanger et al., 2014). A single dimension of the current state is sampled at each step in the Markov chain independently of its current value (with the other dimensions fixed) and then based on its mixture weight it is accepted or rejected (see Algorithm 1 lines 3-12). The input of the algorithm is the subposterior samples and the output is the posterior samples.

Input: Subposterior samples: $\{\theta_{t_1}^1\}_{t_1=1}^T \sim p_1(\theta), \ldots,$
$\{\theta_{t_M}^M\}_{t_M=1}^T \sim p_M(\theta)$
Output: Posterior samples (asymptotically, as
$T \to \infty$): $\{\theta_i\}_{i=1}^T \sim p_1 \cdots p_M(\theta) \propto p(\theta|x^N)$
1: Draw $t_\cdot = \{t_1, \ldots, t_M\} \overset{\text{iid}}{\sim} \text{Unif}(\{1, \ldots, T\})$
2: **for** $i = 1$ **to** T **do**
3: Set $h \leftarrow i^{-1/(4+d)}$
4: **for** $m = 1$ **to** M **do**
5: Set $c_\cdot \leftarrow t_\cdot$
6: Draw $c_m \sim \text{Unif}(\{1, \ldots, T\})$
7: Draw $u \sim \text{Unif}([0, 1])$
8: **if** $u < w_{c_\cdot}/w_{t_\cdot}$ **then**
9: Set $t_\cdot \leftarrow c_\cdot$
10: **end if**
11: **end for**
12: Draw $\theta_i \sim \mathcal{N}_d(\bar{\theta}_{t_\cdot}, \frac{h^2}{M} I_d)$
13: **end for**

Algorithm 1 *– Asymptotically Exact Sampling via Non-parametric Density Product Estimation (Neiswanger et al., 2014).*

3.2 Experimental Setup

To set up the simulation to measure speed-up, a Python script was used to create a dataset (by generating random samples from a multivariate normal distribution) and save it to a text file which could then be parsed by the C++ program. To make use of this script, the user needs to use a command prompt terminal window and select the number of dimensions and samples to be used by calling: "python n_dimensions n_samples". This simplified the input to the program (instead of needing to convert discrete variables from a specific Bayesian Network) and allowed various combinations of dimensions and samples to be trialled. This approach focused on achieving speedup of the parallel MCMC algorithm and the communication between R is left for future work. It was also attempted to use shared and global memory, however, for some datasets, the shared memory on the NVIDIA GeForce RTX 2060 was insufficient to accommodate the problem's size so global memory had to be utilised when the problem grew too large. To record the time taken to run the mcmc program, the cudaEventRecord() inbuilt function was used to record the start and stop time (Figure 9).

```
int grid_dim = n_blocks; // Default to 1D structures
int block_dim = n_threads / n_blocks;
printf("Grid dim: %d, Block dim: %d\n", grid_dim, block_dim);
cudaEventRecord(start);
// Usage: mykernel<<<grid_dim, block_dim, shared_mem>>>(args);
shared_mcmc<<<grid_dim, block_dim, n_threads * n_data_per_thread * sizeof(float)>>>
cudaEventRecord(stop);
gpuErrChk(cudaDeviceSynchronize()); // Error check for shared memory sync
```

Figure 9 *– Code snippet of the event timing using cudaEventRecord().*

3.3 Discussion of Results

An NVIDIA GeForce RTX 2060 card was the hardware used to run the experiment and 40,000 samples with 10 dimensions were selected as this was determined to be a sufficiently large problem size but did not require copious amounts of time to run to completion (Figure 10). The CPU sequential time was 25.6s and a time of 0.814s was achieved with 30 blocks and 128 threads/block (31.5x speedup). These results were ascertained by taking an average of 10 runs of the program for each input combination. The blue line represents the set-up using 128 threads/block and the red line represents when 256 threads/block were used. For the 256 threads/block set up, the speedup at 10 blocks was slightly greater than the 128 threads/block set up. However, the speedup was observed to drop significantly when the number of blocks was increased to 30 for the 256 threads/block set up (potentially due to load imbalances). Overall, up to 31.5 orders of magnitude in speedup for the parallelized MCMC algorithm was achieved with the experiment, however, this varied depending upon sample size, the number of dimensions and the GPU hardware used.

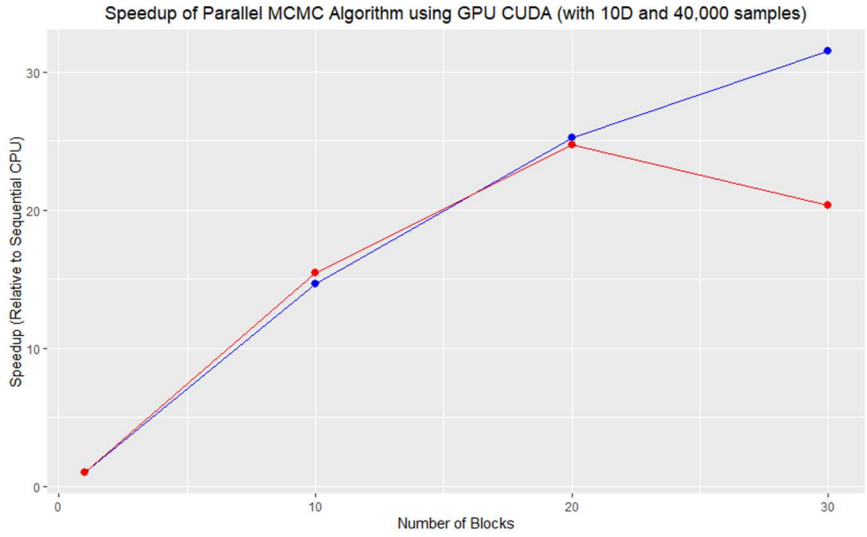

Figure 10 *– Speedup of Parallel MCMC Algorithm using 10 dimensions and 40,000 samples.*

4.0 Future Work and Conclusion

This report provides insight into the progress made on the 'Development of a GPU-Accelerated Computational Platform for Bayesian Networks project. Significant work has been conducted to achieve a 31.5x speedup in the parallel MCMC algorithm. To continue this progress, further work should be done to allow the program to accept a wider range of input data (i.e., handling discrete states from existing Bayesian Networks and communication with R). Future work is required to integrate this GPU-based solution into R for it to be used with existing Bayesian Networks in a more accessible way (such as through the bnlearn library). For example, discrete values and CPTs for simple cases could be parsed and the current input could be able to handle samples in the DAG space constrained by a given Bayesian Network structure. Another limitation is that the algorithm selected holds only for posterior distributions over finite-dimensional graphical models with unconstrained variables. Finally, better speedup could be achieved with alternative hardware/distributed systems and the accessibility of powerful, CUDA-abled GPUs should be further considered.

5.0 References

Amayri, M., Ngo, Q.-D., Safadi, E. A. E., & Ploix, S. (2017). Bayesian network and Hidden Markov Model for estimating occupancy from measurements and knowledge. *2017 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. https://doi.org/10.1109/idaacs.2017.8095179

Ben-Gal, I. (2008). Bayesian Networks. *Encyclopedia of Statistics in Quality and Reliability*, 1. https://doi.org/10.1002/9780470061572.eqr089.

Fang, J., Varbanescu, A. L., & Sips, H. (2011). A Comprehensive Performance Comparison of CUDA and OpenCL. *International Conference on Parallel Processing*, 216–225. https://doi.org/10.1109/ICPP.2011.45.

Korb, K. B., & Nicholson, A. E. (2010). *Bayesian Artifiicial Intelligence (Chapman & Hall/CRC Computer Science & Data Analysis Book 2)* (2nd ed.). CRC Press.

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., & Dubey, P. (2010). Debunking the 100X GPU vs. CPU myth. ACM SIGARCH Computer Architecture News, 38(3), 451–460. https://doi.org/10.1145/1816038.1816021

Lounis, M., Bounceur, A., Laga, A., & Pottier, B. (2015, June). GPU-based parallel computing
of energy consumption in wireless sensor networks. 2015 European Conference on Networks and Communications (EuCNC). https://doi.org/10.1109/eucnc.2015.7194086

Minka, T. P. (2001). A family of algorithms for approximate Bayesian inference. MIT. https://escholarship.org/content/qt53n4f34m/qt53n4f34m.pdf

Neiswanger, W., Wang, C., & Xing, E. P. (2014, March). *Asymptotically Exact, Embarrassingly Parallel MCMC*. https://arxiv.org/abs/1311.4780

New York University Centre for Data Science. (2017). Introduction to GPUs: CUDA. Nyu-Cds.Github.Io. https://nyu-cds.github.io/python-gpu/02-cuda/

NVIDIA. (2009). NVIDIA's Next Generation CUDA Compute Architecture: Fermi (WP_V1.1). https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

NVIDIA. (2018). NVIDIA Turing GPU Architecture (WP-09183-001_v01). https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

NVIDIA. (2021, February 9). CUDA C++ Best Practices Guide. *CUDA Toolkit Documentation* V11.2.1. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference (Morgan Kaufmann Series in Representation and Reasoning)* (1st ed.). Morgan Kaufmann.

Silva, A. F. (2010). cudaBayesreg: Bayesian Computation in CUDA. *The R Journal*, *2*(2), 48. https://doi.org/10.32614/rj-2010-015

Wainwright, M. J., & Jordan, M. I. (2007). Graphical Models, Exponential Families, and Variational Inference. *Foundations and Trends® in Machine Learning*, *1*(1–2), 1–305. https://doi.