GPU-accelerated Steady-state Computation of Large Probabilistic Boolean Networks

Andrzej Mizera[1], Jun Pang[1,2], and Qixia Yuan[1⋆]

[1] Faculty of Science, Technology and Communication, University of Luxembourg
[2] Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg
firstname.lastname@uni.lu

Abstract. Computation of steady-state probabilities is an important aspect of analysing biological systems modelled as probabilistic Boolean networks (PBNs). For small PBNs, efficient numerical methods can be successfully applied to perform the computation with the use of Markov chain state transition matrix underlying the studied networks. However, for large PBNs, numerical methods suffer from the state-space explosion problem since the state-space size is exponential in the number of nodes in a PBN. In fact, the use of statistical methods and Monte Carlo methods remain the only feasible approach to address the problem for large PBNs. Such methods usually rely on long simulations of a PBN. Since slow simulation can impede the analysis, the efficiency of the simulation procedure becomes critical. Intuitively, parallelising the simulation process can be an ideal way to accelerate the computation. Recent developments of general purpose graphics processing units (GPUs) provide possibilities to massively parallelise the simulation process. In this work, we propose a *trajectory-level parallelisation framework* to accelerate the computation of steady-state probabilities in large PBNs with the use of GPUs. To maximise the computation efficiency on a GPU, we develop a dynamical data arrangement mechanism for handling different size PBNs with a GPU, and a specific way of storing predictor functions of a PBN and the state of the PBN in the GPU memory. Experimental results show that our GPU-based parallelisation gains a speedup of approximately 400 times for a real-life PBN.

1 Introduction

Systems biology aims to model and analyse biological systems using mathematical and computational methods from a holistic perspective in order to provide a comprehensive, system-level understanding of cellular behaviour. Recent developments in systems biology have greatly promoted the discovery of unknown biological information, leading to the revealing of more and more large biological systems. This brings a significant challenge to computational modelling in terms of the state-space size of the system under study. Developed in 2002

⋆ Supported by the National Research Fund, Luxembourg (grant 7814267).

by Shmulevich et al. [1, 2], probabilistic Boolean networks (PBNs) is a well-suited framework for modelling large-size biological systems. Originally, PBNs is introduced as a probabilistic generalisation of the standard Boolean networks (BNs) to model gene regulatory networks (GRNs). The framework of PBNs not only takes the advantage of BNs to incorporate rule-based dependencies between genes and allow the systematic study of global network dynamics, but also is capable of dealing with uncertainty, which naturally occurs at different levels in the study of biological systems.

One of the key aspects of analysing biological systems, especially for those modelled as PBNs, is the comprehensive understanding of their long-run (steady-state) behaviour. This is vital in many contexts, e.g., attractors of a GRN were considered to characterise cellular phenotype [3]. There have been a lot of studies in analysing the steady-state behaviours of biological systems modelled as PBNs. As the dynamics of a PBN can be viewed as a discrete-time Markov chain (DTMC), it can be studied with the use of the rich theory of DTMCs. Relying on this, many numerical methods exist to compute steady-state probabilities for small-size PBNs [4, 5]. In the case of large-size PBNs, however, numerical methods face the state-space explosion problem. The use of statistical methods and Monte Carlo methods are then proposed to estimate the steady-state probabilities. These methods require simulating the PBN under study for a certain length and the simulation speed is an important factor in the performance of these approaches. For large PBNs and long trajectories, a slow simulation speed could render these methods infeasible as well. A natural way to address this problem is to parallelise the simulation process.

Recent improvements in the computing power and the general purpose graphics processing units (GPUs) enable the possibilities to massively parallelise this process. In this work, we propose a *trajectory-level parallelisation framework* to accelerate the computation of steady-state probabilities in large PBNs with the use of GPUs. The architecture of a GPU is very different from that of a central processing unit (CPU), and the performance of a GPU-based program is highly related to how the synchronisation between cores is processed and how memory accessing is managed. Our framework reduces the time-consuming synchronisation cost by allowing each core to simulate one trajectory. Regarding to the memory management, we contributes in two aspects. We first develop a dynamical data arrangement mechanism for handling different size PBNs with a GPU to maximise the computation efficiency on a GPU for relatively small-size PBNs. We then propose a specific way of storing predictor functions of a PBN and the state of the PBN in the GPU memory to reduce the memory consumption and to improve the accessing speed. We show with experiments that our GPU-based parallelisation gains a speedup of more than two orders of magnitudes.

Structure of the paper. We present preliminaries on PBNs and the architecture of GPUs in Section 2. The difficulties of parallelising the simulation of a PBN and how to overcome them are discussed in Section 3. We evaluate our GPU implementation in Section 4 and conclude our paper with some discussions in Section 5.

2 Preliminaries

2.1 Probabilistic Boolean networks (PBNs)

A PBN $G(X, F)$ consists of a set of binary-valued nodes (also known as genes) $X = \{x_1, x_2, \ldots, x_n\}$ and a list of sets $\mathcal{F} = (F_1, F_2, ..., F_n)$. For each $i \in \{1, 2, ..., n\}$, the set $F_i = \{f_1^{(i)}, f_2^{(i)}, \ldots, f_{\ell(i)}^{(i)}\}$ is a collection of Boolean functions, known as *predictor functions*, for node x_i, where $\ell(i)$ is the number of predictor functions for node x_i. Each $f_j^{(i)}$ is a Boolean function defined using a subset of the nodes, referred to as *parent nodes* of x_i. At each time point t, the value of each node x_i is updated with one of its predictor functions. The predictor function is selected in accordance with a probability distribution $C_i = (c_1^{(i)}, c_2^{(i)}, \ldots, c_{\ell(i)}^{(i)})$, where the individual probabilities are the *selection probabilities* for the respective elements of F_i and they sum to 1. Several variants of PBNs exist due to the different way of selecting predictor functions and the synchronisation of nodes update. In this paper, we focus on the *independent synchronous* PBNs, i.e., the choice of predictor functions for each node is made independently and the values of all the nodes are updated synchronously. We use $x_i(t)$ to denote the value of node x_i at time point t, and $s(t) = (x_1(t), x_2(t), \ldots, x_n(t))$ to denote the state of the PBN at time point t. The state space of the PBN is $S = \{0, 1\}^n$ and it is of size 2^n. The transition from state $s(t)$ to state $s(t+1)$ is performed by randomly selecting a predictor function for each node x_i from F_i and by applying those selected predictor functions to update the values of all the nodes synchronously. Let $f(t)$ be the combination of all the selected predictor functions at time point t. The transition of state $s(t)$ to $s(t+1)$ can then be denoted as

$$s(t+1) = f(t)(s(t)). \tag{1}$$

A PBN can therefore be viewed as a discrete-time Markov chain (DTMC) with state space $S = \{0, 1\}^n$ and transition relation defined by Equation 1.

In a PBN with *perturbations*, a perturbation rate $p \in (0, 1)$ is introduced and the dynamics of a PBN is guided with both perturbations and predictor functions: at each time point t, the value of each node x_i is flipped with probability p; and if no flip happens, the value of each node x_i is updated with selected predictor functions synchronously. Let $\boldsymbol{\gamma}(t) = (\gamma_1(t), \gamma_2(t), \ldots, \gamma_n(t))$ be a perturbation vector, where each element is a Bernoulli distributed random variable with parameter p, i.e., $\gamma_i(t) \in \{0, 1\}$ and $\mathbb{P}(\gamma_i(t) = 1) = p$ for all t and $i \in \{1, 2, \ldots, n\}$. Extending Equation 1, the transition from $s(t)$ to $s(t+1)$ in PBNs with perturbations is given as

$$s(t+1) = \begin{cases} s(t) \oplus \gamma(t) & \text{if } \gamma(t) \neq 0 \\ f(t)(s(t)) & \text{otherwise,} \end{cases} \tag{2}$$

where $\oplus$ is the element-wise exclusive or operator for vectors. According to Equation (2), from any state, the system can move to any other state with one transition due to perturbations. Therefore, the underlying Markov chain is in

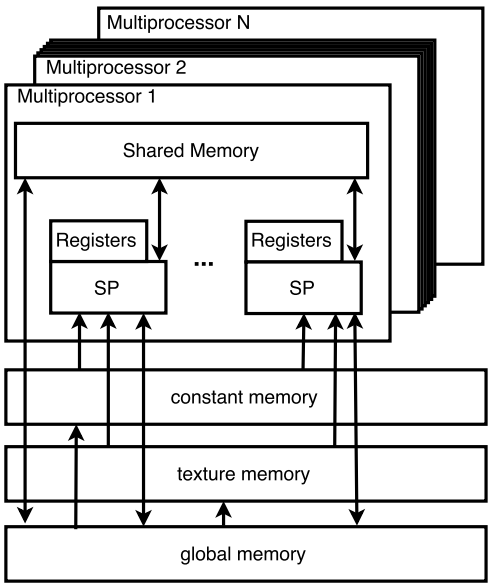

Fig. 1: Architecture of a GPU.

fact irreducible and aperiodic. Thus, the dynamics of a PBN with perturbations can be viewed as an ergodic DTMC [1]. Based on the ergodic theory, the long-run dynamics of a PBN with perturbations is governed by a unique limiting distribution, convergence to which is independent of the choice of the initial state.

The density of a PBN is measured with the number of predictor functions and the number of parent nodes for each predictor function. For a PBN G, its density is defined as $\mathcal{D}(G) = \frac{1}{n}\sum_{i=1}^{M}\phi(i)$, where n is the number of nodes in G, M is the total number of predictor functions in G, and $\phi(i)$ is the number of parent nodes for the ith predictor function.

2.2 GPU architecture

We review the basics of GPU architecture and its programming approach, i.e., common unified device architecture (CUDA) released by NVIDIA.

At the physical hardware level, an NVIDIA GPU usually contains tens of streaming multiprocessors (SMs, also abbreviated as MPs), each containing a fixed number of streaming processors (SPs), fixed size of *registers*, fast *shared memory* (as shown in Figure 1, with N being the number of MPs).

Accessing registers and shared memory is fast, but the size of these two types of memory is very limited. In addition, a large size *global memory*, a small size *texture memory* and *constant memory* are available outside the MPs. Global memory has a high bandwidth (128 bytes in our GPU), but also a high latency.

Accessing global memory is usually orders of magnitude slower than accessing registers or shared memory. Constant memory and texture memory are memories of special type which can only store read-only data. Accessing constant memory is most efficient if all threads are accessing exactly the same data; and texture memory is better for dealing with random access. We refer to registers and shared memory as *fast memory*; global memory as *slow memory*; and constant memory and texture memory as *special memory*.

At the programming level, the programming interface CUDA is in fact an extension of C/C++. A segment of code to be run in a GPU is put into a function called a *kernel*. The kernels are then executed as a grid of blocks of threads. A thread is the finest granularity in a GPU and each thread can be viewed as a copy of the kernel. A block is a group of threads executed together in a batch. Each thread is executed in an SP and threads in a block can only be executed in one MP. One MP, however, can launch several blocks in parallel. Communications between threads in the same block are possible via shared memory. NVIDIA GPUs use a processor architecture called single instruction multiple thread (SIMT), i.e., a single instruction stream is executed via a group of 32 threads, called a *warp*. Threads within a warp are bounded together, i.e., they always execute the same instruction. Therefore, branch divergence can occur within a warp: if one thread within a warp moves to the 'if' branch of an 'if-then-else' sentence and the others choose the 'else' branch, then actually all the 32 threads will "execute" both branches, i.e., the thread moving to the 'if' branch will wait for other threads when they execute the 'else' branch and vice versa. If both branches are long, then the performance penalty is huge. Therefore, branches should be avoided as much as possible in terms of performance. Moreover, the data accessing pattern of the threads in a warp should be taken care of as well. We consider the access pattern of shared memory and global memory in this work. Accessing shared memory is most efficient if all threads in a warp are fetching data in the same position or each thread is fetching data in a different position. Otherwise, the speed of accessing shared memory is reduced by the so-called *bank conflict*. Accessing global memory is most efficient if all threads in a warp are fetching data in a *coalesced* pattern, i.e., all threads in a warp are reading data in adjacent locations in global memory. In principle, the number of threads in a block should always be an integral multiple of the warp size due to the SIMT architecture; and the number of blocks should be an integral multiple of the number of MPs since each block can only be executed in one MP.

An important task for GPU programmer is to hide latency. This can be done via the following four ways:

1. increase the number of active warps;
2. reduce the access to global memory by caching the frequently accessed data in fast memory, or in constant memory or texture memory, if the access pattern is suitable;
3. reduce bank conflict of shared memory access;
4. coalesce accesses to the global memory to use the bandwidth more efficiently.

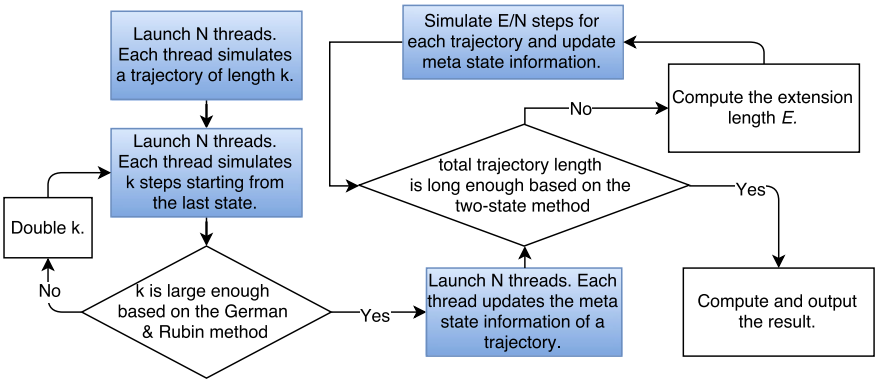

Fig. 2: Workflow of steady-state analysis using trajectory-level parallelisation.

However, the above four methods often compete with one another due to the restrictions of the hardware resources. For example, using more shared memory would restrict the number of active blocks and hence the number of active warps is limited. Therefore, a trade-off between the use of fast memory and the number of threads has to be considered carefully. We discuss this problem and provide our solution to it in Section 3.2.

3 PBN Simulation in GPU

In this section, we present how simulation of a PBN is performed in a GPU, while addressing the problems identified at the end of Section 2.

3.1 Trajectory-level parallelisation

In general, there are two ways for parallelising the PBN simulation process. One way is to update all nodes synchronously, i.e., each GPU thread only updates one node of a PBN; the other way is to simulate multiple trajectories simultaneously. The first way requires synchronisation among the threads, which is time-consuming in the current GPU architecture. Therefore, in our implementation, we take the second way to simulate multiple trajectories concurrently. Samples from multiple trajectories can be merged together to compute steady-state probabilities of a PBN using a combination of the two-state Markov chain approach [6] and the Gelman & Rubin method [7]. A detailed description for this combination can be found in [8]. Note that merging is performed in a CPU and no synchronization is required. We show in Figure 2 the workflow for computing steady-state probabilities based on trajectory-level parallelisation.

Each blue box represents a kernel to be parallelised in a GPU. The first and second blue boxes perform the same task except that trajectories in the first blue box are abandoned while those in the second blue box are stored in global memory. This is due to the requirement of the Gelman & Rubin method [7] that

Algorithm 1 Simulate one step of a PBN in a GPU

```
1: procedure SimulateOneStep(n, F, extraF, p, S)
2:     perturbed := false;
3:     for (i := 0; i < n; i++) do
4:         if rand() < p then perturbed := true; S[i/32] := S[i/32] ⊕ (1 ≪ (i%32));
5:         end if                                   //the result of i/32 is ⟨⟨
6:     end for
7:     if perturbed then return S;
8:     else
9:         set array nextS to 0;
10:        for (i := 0; i < n; i++) do
11:            index := nextIndex(i); //sample the Boolean function index for node i
12:            compute the entry of the Boolean function based on index and S;
13:            v := F[index];
14:            if entry > 31 then                    //entry starts with 0
15:                get index of the Boolean function in extraF;      //see Section 3.3
16:                v := extraF[index]; entry := entry%32;
17:            end if
18:            v := v ≫ entry; nextS[i/32] := nextS[i/32] | ((v&1) ≪ (i%32));
19:        end for
20:    end if
21:    S := nextS; return S.
22: end procedure
```

only the second half samples are used for computing steady-state probabilities. Based on the last k samples simulated in the second blue box, the third blue box computes the *meta state* information required by the two-state Markov chain approach [6]. The two-state Markov chain approach determines whether the samples are large enough based on the meta state information. If not enough, the last (forth) kernel is called again to simulate more samples; otherwise, the steady-state probability is computed.

The key part of the four kernels is the simulation process. We describe in Algorithm 1 the process for simulating one step of a PBN in a GPU. The four inputs of this algorithm are respectively the number of nodes n, the Boolean functions F, the extra Boolean functions $extraF$ and the current state S. The extra Boolean functions are generated due to that we optimise the storage of Boolean functions and split them into two parts in order to save memory (see Section 3.3 for details). Due to this optimisation, an 'if' sentence (lines 14 to 17) has to be added. This 'if' sentence fetches the Boolean function stored in the second part ($extraF$). The probability that this sentence is executed is very small due to the way we split the Boolean functions and the time cost of executing this sentence is also very small. Therefore, by paying a small penalty in terms of computational time, we are able to store Boolean functions in fast memory and gain much more speedups with the use of fast memory.

data	data type	stored in
random number generator	CUDA built in	registers
node number	integer	constant memory
perturbation rate	float	constant memory
cumulative number of functions	short array	constant memory
selection probabilities of functions	float array	constant memory
indices of positive nodes	integer array	constant memory
indices of negative nodes	integer array	constant memory
cumulative number of parent nodes	short array	shared memory
Boolean functions	integer array	shared memory
indices for extra Boolean functions	short array	shared memory
parent nodes indices for each function	short array	shared/texture memory
current state	integer array	registers/global memory
next state	integer array	registers/global memory

Table 1: Frequently accessed data arrangement.

3.2 Data arrangement

As mentioned in Section 2.2, suitable strategy for hiding latency should be carefully considered for a GPU program. Since the simulation process requires accessing the PBN information (in a random way) in each simulation step and the latency cost for frequently accessing data in slow memory is really huge, caching those information in fast and special memory results in a more efficient computation comparing to allowing more active warps. Therefore, we first try to arrange all frequently accessed data in fast and special memory as much as possible; then based on the remaining resources we calculate the optimal number of threads and blocks to launch. Since the size of fast memory is limited and the memory required to store a PBN varies from PBN to PBN, a suitable data arrangement policy is necessary. In this section, we discuss how we dynamically arrange the data in different GPU memories for different PBNs.

In principle, frequently accessed data should be put in fast memory as much as possible. We list all the frequently used data and how we arrange them in GPU memories in Table 1. As the size of the fast memory is limited and has different advantages for different data accessing modes, we save different data in different memories. Namely, those read-only data that are always or most likely to be accessed simultaneously by all threads in a warp, are put in constant memory; other read-only data are put in shared memory if possible; and the rest of the data are put in registers if possible. Since the memory required to store the frequently used data varies a lot from PBN to PBN, we propose to use a dynamic decision process to determine how to arrange some of the frequently accessed data, i.e., the data shown in the last four rows of Table 1. The dynamic process calculates the memory required to store all the data for a given PBN, and determines where to put them based on their memory size. If the shared memory and registers are large enough, all the data will be stored in these two fast memories. Otherwise, they will be placed in the global memory. For the

data stored in the global memory, we use two ways to speed up their access. One way is to use texture memory to speed up the access for read-only data, e.g., the parent node indices for each function. The other way is to optimise the data structure to allow a coalesced accessing pattern, e.g., the current state. We explain this in details in Section 3.3. This dynamical arrangement of data allows our program to explore the computation ability of a GPU as much as possible, leading to faster speedups for relatively small sparse networks.

3.3 Data optimisation

As mentioned in Section 2.2, a GPU usually has a very limited size of fast memory and the latency can vary a lot when accessing the same memory in a different way, e.g., accessing shared memory with or without bank conflict. Therefore, we optimise the data structure of two important data, i.e., the Boolean functions (stored as truth tables) and the states of a PBN, to save space and to maximise the access speed.

Optimisation on Boolean functions. A direct way to store a truth table is to use a Boolean array, which consumes one byte to store each element. Accessing an element of the truth table can be directly made by providing the index of the Boolean array. Instead, we propose to use a primitive 32-bit integer (4 bytes) type to store the truth table. Each bit of an integer stores one entry of the truth table and hence the memory usage can be reduced by 8 in maximum: 4 bytes compared to 32 bytes of a Boolean array. A 32-bit integer can store a truth table of at most 32 elements, corresponding to a Boolean function with max. 5 parent nodes. Since for real biological systems the number of parent nodes is usually small [9], in most cases one integer is enough for storing the truth table of one Boolean function. In the case of a truth table with more than 32 elements, additional integers are needed. In order to save memory and quickly locate a specific truth table, we save the additional integers in a separate array. More precisely, we use a 32-bit integer array F of length M to store the truth tables for all the M Boolean functions and the ith ($i \in [0, M-1]$) element of F stores only the first 32 elements of the ith truth table. If the ith truth table contains more than 32 elements, the additional integers are stored in an extra integer array *extraF*. In addition, two index arrays *extraFIndex* and *cumExtraFIndex* are needed to store the index of the ith truth table in *extraF*. Each element of *extraFIndex* stores one index value of the truth table which requires additional integers. The length of *extraFIndex* is at most M. Each element of *cumExtraFIndex* stores the cumulative number of additional required integers for all the truth tables whose indices are stored in *extraFIndex*.

As an example, we show how to store a truth table with 128 elements in Figure 3. We assume that this 128-element truth table is the ith one among all M truth tables and that it is the jth one among those m truth tables that require additional integers to store. Therefore, its first 32 (0-31th) elements are stored in the ith element of F and its index i is stored in the jth element of *extraFIndex*, denoted as e_j. The jth element of *cumExtraFIndex*, denoted as

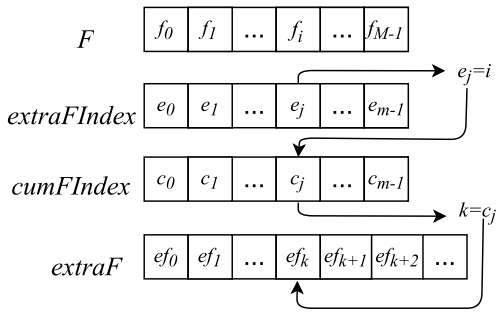

Fig. 3: Demonstration of storing Boolean functions in integer arrays.

c_j, stores the total number of additional integers required to store the $j-1$ truth tables whose indices are stored in the first $j-1$ elements of *extraFIndex*. Let $cumExtraFIndex[j] = k$. The kth, $(k+1)$th, and $(k+2)$th elements of *extraF* store the 32-127th elements of the ith truth table. After storing the truth tables in this way, accessing the tth element of the ith truth table can be performed in the following way. When $t \in [0, 31]$, $F[i]$ directly provides the information, and when $t \in [32, 127]$, three steps are required: 1) search the array *extraFIndex* to find the index j such that $extraFIndex[j]$ equals to i, 2) fetch the jth value of array *cumFIndex* and let $k = cumFIndex[j]$, 3) the integer $extraF[k + (t-32)\%32]$ contains the required information. Since in most cases the number of parent nodes is very limited, the array *extraFIndex* is very small. Hence, the search of the index j in the first step can be finished very quickly. In the rare case where the *extraFIndex* array would be large, e.g., M is large and the length of *extraFIndex* would be close to M, it is preferable to store *extraFindex* as an array of length M and let $extraFindex[i]$ store the entry in $cumFIndex$ for the ith truth table so that the search phase of the first step is eliminated. The required memory for storing this truth table is reduced from 128 bytes (stored as Boolean arrays) to 20 bytes (6 integers to store the truth table and 2 shorts to store the index). In addition to saving memory, the above optimisation can also reduce the chances of bank conflict in shared memory due to the fact that accessing any entry of a truth table is performed by fetching only one integer in array F in most cases. Accessing the elements in *extraFIndex* requires additional memory fetching; however, as mentioned before, the chance for such cases to happen is very small in a real-life PBN and the gained memory space and improved data fetching pattern can compensate for this penalty.

Optimisation on PBN states. The optimisation of the data structure for states is similar to that for Boolean functions, i.e., states are stored as integers and each bit of an integer represent the value of a node. Therefore, a PBN with n nodes requires $\lceil n/32 \rceil$ integers $(4 * \lceil n/32 \rceil$ bytes) to be stored, compared to n bytes when stored as a Boolean array. During the simulation process, the current state and the next state of a PBN have to be stored. As shown in Table 1, the

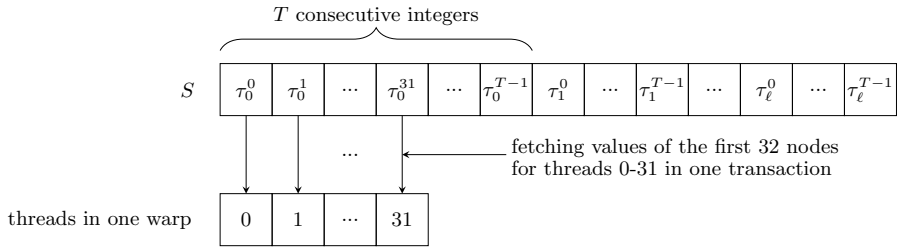

Fig. 4: Storing states in one array and coalesced fetching for threads in one warp.

states are put in registers whenever possible, i.e., when the number of nodes is smaller than 129. In the case of a PBN with nodes number equal to or larger than 129, the global memory has to be used due to the limited register size (shared memory are used to store other data and would not be large enough to store states in this case). To reduce the frequency of accessing global memory, one register (32 bits) is used to cache the integer that stores the values of 32 nodes. Updating of the 32 node values is performed via the register and stored in the global memory with a single access only once after all the 32 node values are updated in the register. Moreover, states for all the threads are stored in one large integer array S in the global memory and we arrange the content of this array to allow for a coalesced accessing pattern. More specifically, starting from the 0th integer, every consecutive T integers store the values of 32 nodes in the T threads (assuming there are T threads in total). Figure 4 shows how to store states of a PBN with n nodes for all the T threads in an integer array S and how the 32 threads in the first warp fetch the first integer in a coalesced pattern. We denote τ_i^j as the ith integer to store values of 32 nodes for thread j and let $\ell = \lceil n/32 \rceil$. For threads in one warp, accessing the values of the same node can be performed via fetching the adjacent integers in the array S. This results in a coalesced accessing pattern of the global memory. Hence, all the 32 threads in one warp can fetch the data in a single data transaction.

4 Evaluation

We evaluate our GPU-based parallelisation framework for computing steady-state probabilities of PBNs in both randomly generated networks and a real-life biological network. All the experiments are performed on a high performance computing (HPC) machines, each of which contains a CPU of Intel Xeon E5-2680 v3 @ 2.5 GHz and an NVIDIA Tesla K80 Graphic Card with 2496 cores @824MHz. The program is written in a combination of both Java and C, and the initial and maximum Java virtual machine heap size is set to 4GB and 11.82GB, respectively. The C language is used to program operations on GPUs due to the fact that no suitable Java library is currently provided for operations on NVIDIA GPUs. When launching the GPU kernels, the kernel configurations (the number of threads and blocks) are dynamically determined as mentioned in Section 3.2.

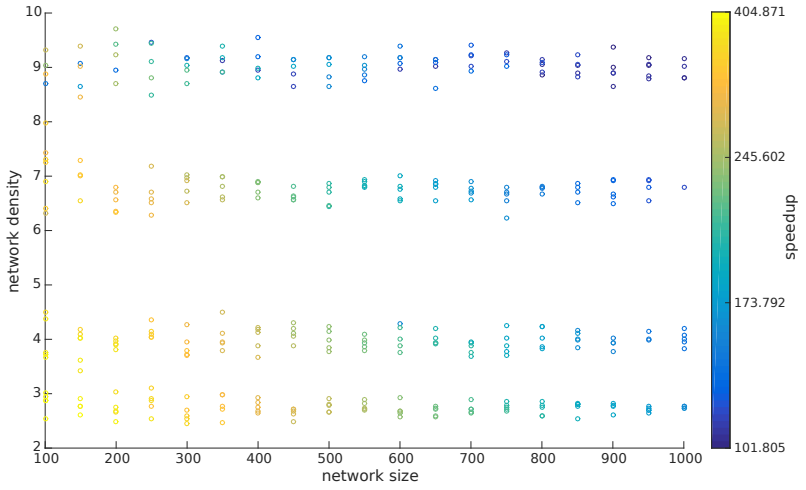

Fig. 5: Speedups of GPU-accelerated steady-state computation.

4.1 Randomly generated networks

The evaluation on randomly generated networks is performed on 380 PBNs, which are generated using the tool ASSA-PBN [10]. The nodes number of these networks ranges in the set $\{100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750, 800, 850, 900, 950, 1000\}$. For each of the 380 networks, we compute one steady-state probability using both the sequential two-state Markov chain approach and our GPU-accelerated parallelisation framework. We set the three precision requirements of the two-state Markov chain approach, i.e., the confidence level s, the precision r, and the steady-state convergence parameter ϵ to 0.95, 5×10^{-5}, and 10^{-10} respectively. The computation time limit is set to 10 hours. In the end, we obtain 366 pairs of valid results. The 14 invalid pairs are due to that the sequential version two-state Markov chain approach is timed out (the parallel version is not). Among the 366 results, 355 (96.99%) are comparable, i.e., the differences of computed probabilities satisfy the specified precision requirement. This result meets our confidence level requirement.

We compute the speedups of the GPU-accelerated parallelisation framework with respect to the sequential two-state Markov chain approach for those 366 valid results with the formula $\mathit{speedup} = \frac{s_{pa}/t_{pa}}{s_{se}/t_{se}}$, where s_{pa} and t_{pa} are respectively the sample size and time cost of the parallelisation framework, and s_{se} and t_{se} are respectively the sample size and time cost of the sequential approach. The speedups are ploted in Figure 5. As can be seen from this figure, we obtain speedups approximately between 102 and 405. There are some small gaps with respect to the densities of those networks, e.g., no networks with density between 5 and 6. Those gaps are due to the way how those networks are randomly gener-

node #	density	probability		sample size (million)		time (s)		speedup
		seq.	par.	seq.	par.	seq.	par.	
100	2.53	0.24409	0.24401	350	367	2637.06	6.84	405
100	9.32	0.36221	0.36217	426	427	3429.06	13.04	264
400	2.75	0.12003	0.12002	316	318	7615.72	26.77	286
400	8.98	0.04657	0.04660	135	137	3908.25	20.79	190
700	2.64	0.05800	0.05794	259	261	8567.52	39.27	220
700	9.41	0.10632	0.10634	438	441	16541.79	121.60	137
1000	2.73	0.14675	0.14673	838	839	30626.44	184.44	166
1000	8.81	0.00298	0.00293	20	21	792.86	8.10	103

Table 2: Speedups of GPU-accelerated steady-state computation of 8 randomly generated networks. seq. is short for the sequential two-state Markov chain approach; while par. is short for the GPU-accelerated parallel approach.

ated, i.e., one cannot force the ASSA-PBN tool to generate a PBN with a fixed density, but can only provide the following information to affect the density: the number of nodes, the maximum (minimum) number of functions for each node, and the maximum (minimum) number of parent nodes for each function. However, even with the gaps, the tendency of the changes of speedups with respect to densities can be well observed. In fact, this observation is similar to that of the network size. With the network size decreasing and the density decreasing, our GPU-accelerated parallelisation framework gains higher speedups. This is due to our dynamic way of arranging data for different size PBNs: data for relatively small[3] and sparse networks can be arranged in the fastest memory.

To demonstrate the computation details, we select 8 pairs among the 366 results and show in Table 2 the computed probabilities, the sample size (in millions) and the time cost (in seconds) for computing the steady-state probabilities using both the sequential two-state Markov chain approach and the GPU-accelerated parallelisation framework. The two approaches generated comparable results using similar length of samples while our GPU-accelerated parallelisation framework shows speedups of more than two orders of magnitude. All detailed results for the 380 networks can be found at `http://satoss.uni.lu/software/ASSA-PBN/benchmark/`.

4.2 An apoptosis network

We have analysed a PBN model of an apoptosis network using the sequential two-state Markov chain approach in [6]. The apoptosis network was originally published in [**?**] as a BN model and cast into the PBN framework in [5]. The PBN model (as shown in Figure 6) contains 91 nodes and 107 Boolean functions. The selection probabilities of the Boolean functions were fitted to experimental data

[3] In fact all the networks used in this subsection should be called large-size PBNs since the network with the smallest size has already contained $2^{100} \approx 10^{30}$ states.

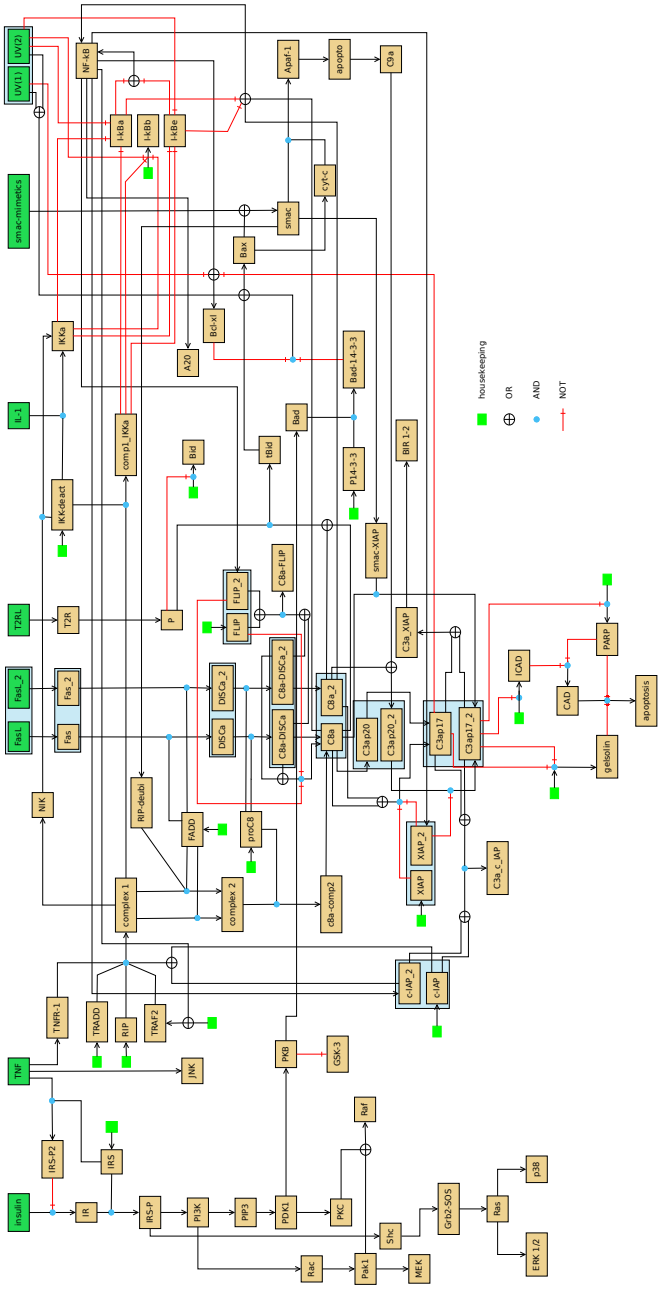

Fig. 6: The wiring of the probabilistic Boolean model of apoptosis in [5].

in [5]. We took the 20 best fitted parameter sets and performed the influence analyses for them. Although we managed to finish this analysis in an affordable

steady-state R C F	probability		sample size (million)		time (s)		speedup
	seq.	par.	seq.	par.	seq.	par.	
0 1 1	0.003236	0.003237	589.05	590.77	3866.04	9.28	417.81
1 1 1	0.990053	0.990046	1809.27	1811.71	11476.00	28.08	409.20
1 0 1	0.005592	0.005590	1015.95	1021.07	6662.26	15.89	421.47
1 1 0	0.001082	0.001080	197.80	200.12	1281.45	3.27	396.60
* 1 1	0.993289	0.993288	1222.83	1241.06	7967.42	19.30	418.99
* 1 0	0.001082	0.001087	197.29	206.37	1096.90	3.36	341.62
* 0 1	0.005614	0.005624	1021.87	1039.35	6725.25	16.17	422.98

Table 3: Speedups of GPU-accelerated steady-state computation of a real-life apoptosis network.

amount of time due to an efficient implementation of a sequential PBN simulator, the analysis was still very expensive in terms of computation time since the required trajectories were very long and we needed to compute steady-state probabilities for a number of different states.

In this work, we re-perform part of the influence analyses from [6] using our GPU-accelerated parallel two-state Markov chain approach. In the influence analysis, we consider the PBN with the best fitted values and we aim to compute the *long-term influences* on complex2 from each of its parent nodes: RIP-deubi, complex1, and FADD, in accordance with the definition in [11]. In order to compute this long-term influence, seven different steady-state probabilities are required. We show in the first column of Table 3 the values of the seven steady-states. The three numbers or "*" with two numbers respectively represent the steady-state values of the three genes RIP-deubi, complex1, and FADD: 0 represents active; 1 represents inactive; and "*" represents irrelevant. We compute those seven different steady-state probabilities using both the sequential two-state Markov chain approach and the GPU-accelerated parallelisation framework. We show in Table 3 the computed steady-state probabilities, the sample size (in millions), the time cost (in seconds), and the speedups we obtain for this computation. The confidence level s, precision r, and the steady-state convergence parameter ϵ of this computation are set to 0.95, 5×10^{-6}, and 10^{-10} respectively. The density of the network is approximately 1.78. The two approaches compute comparable steady-state probabilities with similar trajectory lengths; while our GPU-accelerated parallelisation framework reduces the time cost by approximately 400 times. The total time cost for computing the seven probabilities is reduced from about 11 hours to approximately 1.5 minutes.

5 Conclusion and Future Works

In this paper, we have proposed a trajectory-level parallelisation framework to accelerate the computation of steady-state probabilities for large PBNs with the use of GPUs. Our work contributes in three aspects in maximising the perfor-

mance of a GPU when computing the steady-state probabilities. First, we reduce the time consuming synchronisation cost between GPU cores by allowing each core to simulate all nodes of one trajectory. Secondly, we propose a dynamical data arrangement mechanism for handling different size PBNs with a GPU. This leads to large speedups for handling relatively small-size PBNs. Lastly, we develop a specific way of storing predictor functions of a PBN and the state of the PBN in the GPU memory to save space and to accelerate the memory accessing. We show with experiments that our GPU-based parallelisation gains a speedup of more than two orders of magnitudes. Evaluation on a real-life apoptosis network shows that our GPU-based parallelisation obtains a speedup of approximately 400 times.

There are two directions for our future works. One is to apply our work to analyse large realistic biological models. The other one is to optimise the current structure to better handle very dense and huge networks.

References

1. Shmulevich, I., Dougherty, E.R.: Probabilistic Boolean Networks: The Modeling and Control of Gene Regulatory Networks. SIAM Press (2010)
2. Trairatphisan, P., Mizera, A., Pang, J., Tantar, A.A., Schneider, J., Sauter, T.: Recent development and biomedical applications of probabilistic Boolean networks. Cell Communication and Signaling **11** (2013) 46
3. Kauffman, S.A.: Homeostasis and differentiation in random genetic control networks. Nature **224** (1969) 177–178
4. Shmulevich, I., Gluhovsky, I., Hashimoto, R., Dougherty, E., Zhang, W.: Steady-state analysis of genetic regulatory networks modelled by probabilistic Boolean networks. Comparative and Functional Genomics **4**(6) (2003) 601–608
5. Trairatphisan, P., Mizera, A., Pang, J., Tantar, A.A., Sauter, T.: optPBN: An optimisation toolbox for probabilistic boolean networks. PLOS ONE **9**(7) (2014)
6. Mizera, A., Pang, J., Yuan, Q.: Reviving the two-state markov chain approach (technical report). Available online at `http://arxiv.org/abs/1501.01779` (2015)
7. Gelman, A., Rubin, D.: Inference from iterative simulation using multiple sequences. Statistical Science **7**(4) (1992) 457–472
8. Mizera, A., Pang, J., Yuan, Q.: Parallel approximate steady-state analysis of large probabilistic Boolean networks. In: Proc. 31st ACM Symposium on Applied Computing. (2016) 1–8
9. Harri, L., Sampsa, H., Ilya, S., Olli, Y.H.: Relationships between probabilistic Boolean networks and dynamic Bayesian networks as models of gene regulatory networks. Signal Processing **86**(4) (2006) 814–834
10. Mizera, A., Pang, J., Yuan, Q.: ASSA-PBN: An approximate steady-state analyser of probabilistic Boolean networks. In: Proc. 13th International Symposium on Automated Technology for Verification and Analysis. Volume 9364 of LNCS., Springer (2015) 214–220
11. Shmulevich, I., Dougherty, E.R., Kim, S., Zhang, W.: Probabilistic Boolean networks: a rule-based uncertainty model for gene regulatory networks. Bioinformatics **18**(2) (2002) 261–274