Implementation of Sequential Importance Sampling in GPGPU

Keisuke Hayashi Masaya M. Saito Ryo Yoshida Tomoyuki Higuchi
The Institute of Statistical Mathematics
Midoricho 10-3, Tachikawa, Tokyo, 190-0014, JAPAN
{ghayashi, saitohm, yoshidar, higuchi}@ism.ac.jp

Abstract − *The estimation of many unknown parameters is carried out using a simplified Sequential Importance Sampling (SIS) algorithm which is implemented in a graphic processing unit (GPU). The aim of the present work is to show technical points to bring out the performance of GPU. Using the implemented code, two numerical experiments are demonstrated. In the first demonstration, it is shown that a parameter estimation involving* 10^9 *Monte Carlo samples is completed within eight hours. In the second demonstration, accuracy-guaranteed evaluation of the likelihood is carried out.*

Keywords: GPGPU programming, parameter estimation, Monte Carlo method

1 Introduction

Graphic processing units (GPU) have been acquired programmability for general scientific applications in recent years. When such a generality of the GPU is emphasised, it is called *general purpose* GPU (GPGPU). Dramatic improvements in computing speed have been achieved by appropriate use of GPUs in several fields. Data assimilation is a method combining models and observed data that was developed in atmospheric and oceanic simulation areas. This method is used to estimate appropriate initial values and parameters so that the simulation result well agrees with data.

Biological circuits in cells, such as transcription factor regulatory circuits and signal transduction pathways, can respond to extraneous stimuli by enhancing/repressing rates of gene decoding. Simulation models describing reaction kinetics of biochemical reactions realise experiments *in silico* to enhance our understanding of dynamic cellular activities.

Despite of a growing need for simulation-based approaches in life science, some fundamental issues limit drawing their potential in practical applications. One major problem arises from uncertainty in specifying model parameters, such as kinetic values and initial conditions of simulation, that are difficult to measure from *in vivo* experiments and theoretical analyses.

In the last several years, a variety of statistical technologies has been developed in bioinformatics and computational biology to explore kinetic model parameters using experimentally-observed profiles of biochemical variables, such as mRNA and proteins. In many cases [6, 7, 12, 13, 15], data are measured along time. Of numerous existing methods, Bayesian approach has an appealing general framework that enables us to reflect *a priori* knowledge on the parameter values. Since it is generally difficult to give a posterior distribution in an analytical form, the approximation of probability distributions by Monte Carlo ensembles is needed (we call their members *particles*). Particle Filter (PF) [4, 8] is a formulation in the context of time course analysis.

In our previous study [13] that applied Particle Filter (PF) to the search for 44 unknown parameters in a transcriptional circuit model, it turned out that a quite large number of particles (10^8) was needed to draw a good performance in approximating posterior distributions. The computation time was approximately eight days with a single core of CPU. Some methods which reduce the computation time to an acceptable level are necessary and we consider that GPGPU technique is a candidate of such methods. It should be noted that the scale of our problem is much larger than that of preceding reseraches (e.g. [5]) which used GPU to implement PF.

In this paper, we present two types of GPGPU-based acceleration techniques for sequential Monte Carlo computations. The first one is a simple shotgun search for kinetic parameters. Of interest is to draw a large number of particles effectively in order to explore a huge space of models parameters. The second type implementation focuses on making adaptively-controlled incremental process of particle size in accuracy-guaranteed Monte Carlo evaluation of posterior quantities, such as posterior means, likelihood values.

The present paper is organised as follows. In §2 and §3, we introduce the scheme of Bayesian parameter esti-

mation in dynamic system analyses. The heart of GPU programming is summarised in §4. We make demonstrations for the two types of GPGPU-based acceleration techniques in §5 and §6, respectively.

2 State space model

We introduce the *state space model* [9] to relate a simulation model to respective time course data. Let $\boldsymbol{x}_n = (x_{1,n}, \cdots, x_{1,p})$ be a vector of *simulation variables* at time n, $\boldsymbol{y}_n$ be its observation counterpart and $\boldsymbol{\theta}$ be parameters. The state space model then is given by

$$\begin{aligned} \boldsymbol{x}_n &= \boldsymbol{f}(\boldsymbol{x}_{n-1}, \boldsymbol{\theta}) + \boldsymbol{v}_n, & (1) \\ y_{i,n} &= x_{i,n} + w_{i,n}, \quad n \in \mathcal{N}_{\rm obs},\ i \in \mathcal{I}_{\rm obs}, & (2) \end{aligned}$$

with noise components $\boldsymbol{v}_n, w_{i,n}$ and indices $\mathcal{I}_{\rm obs}$ of components of $\boldsymbol{y}_n$ which are actually observed.

Estimation of the *augmented* parameters that aggregate $\boldsymbol{\theta}$ and the initial state variable $\boldsymbol{x}_0$ is of interest in the following arguments. Given a collection of observations, denoted by $\boldsymbol{Y} \equiv \{\boldsymbol{y}_n | n \in \mathcal{N}_{\rm obs}\}$, adaptation of the prior distributions $p(\boldsymbol{\theta})$ and $p(\boldsymbol{x}_0)$ to the data points is carried out in evaluating the posterior distribution having the form:

$$\begin{aligned} p(\boldsymbol{x_0}, \boldsymbol{\theta}|\boldsymbol{Y}) &\propto p(\boldsymbol{Y}, \boldsymbol{x}_0, \boldsymbol{\theta}) \qquad (3) \\ &= \int p(\boldsymbol{Y}, \boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N, \boldsymbol{\theta})\, d^N\boldsymbol{x} \\ &= p(\boldsymbol{\theta}) p(\boldsymbol{x_0}) \times \int \prod_{n \in \mathcal{N}_{\rm obs}} p(\boldsymbol{y}_n|\boldsymbol{x}_n) \prod_{n=1}^{N} p(\boldsymbol{x}_n|\boldsymbol{x}_{n-1}, \boldsymbol{\theta})\, d^N\boldsymbol{x}, \end{aligned}$$

where $d^N\boldsymbol{x} \equiv d\boldsymbol{x}_1 \cdots d\boldsymbol{x}_N$. Note that we use the Markov property [9], $p(\boldsymbol{y}_n|\boldsymbol{X}_n, \boldsymbol{Y}_{n-1}) \equiv p(\boldsymbol{y}_n|\boldsymbol{x}_n)$ and $p(\boldsymbol{x}_n|\boldsymbol{X}_{n-1}, \boldsymbol{Y}_{n-1}) \equiv p(\boldsymbol{x}_n|\boldsymbol{x}_{n-1})$, of Eqs. (1) and (2) to derive the last formula.

3 SIS algorithm

In this section, we introduce a variant of Particle Filter algorithms, (simplified) Sequential Importance Sampling (SIS) to calculate Eq. (3). The procedure consists of the following steps:

(i) Draw M particles of $(\boldsymbol{\theta}, \boldsymbol{x}_0)$ from the prior distributions, $\boldsymbol{\theta}^{(i)} \sim p(\boldsymbol{\theta})$ and $\boldsymbol{x}_0^{(i)} \sim p(\boldsymbol{x}_0)$ for $i = 1, \ldots, M$.

(ii) For $n \in \{1, \ldots, N\}$, iterate the following operation:

  (a) Mapping according to the stochastic variant of simulator, $\boldsymbol{x}_n^{(i)} = \boldsymbol{f}(\boldsymbol{x}_{n-1}^{(i)}, \boldsymbol{\theta}^{(i)}) + \boldsymbol{v}_n^{(i)}$ with $\boldsymbol{v}_n^{(i)} \sim q(\boldsymbol{v}_n)$, gives $\boldsymbol{x}_n^{(i)} \sim p(\boldsymbol{x}_n|\boldsymbol{x}_{n-1}, \boldsymbol{\theta})$.

  (b) If $n \in \mathcal{N}_{\rm obs}$, data are observed at nth time points, compute the incremental weight as $u_n^{(i)} = p(\boldsymbol{y}_n|\boldsymbol{x}_n^{(i)})$.

(iii) Compute the cumulative weight for the ith sequence of simulations:

$$w^{(i)} = \prod_{n \in \mathcal{N}_{\rm obs}} p(\boldsymbol{y}_n|\boldsymbol{x}_n^{(i)}) = \prod_{n \in \mathcal{N}_{\rm obs}} u_n^{(i)}.$$

The final state of particles gives the Monte Carlo approximation of the posterior

$$p(\boldsymbol{x}_0, \theta|\boldsymbol{Y}) \propto \sum_{i=1}^{M} w^{(i)} \delta(\boldsymbol{\theta} - \boldsymbol{\theta}^{(i)}) \delta(\boldsymbol{x}_0 - \boldsymbol{x}_0^{(i)}). \qquad (4)$$

One obtains the original version of PF by adding the resampling operation [10] after step (b):

(c) If $n \in \mathcal{N}_{\rm obs}$, resample from the current particles $(\boldsymbol{x}_0^{(i)}, \boldsymbol{\theta}^{(i)})$, $i = 1, \ldots, M$, with the probability proportional to $u_n^{(i)}$, $i = 1, \ldots, M$.

It is proved that the both methods have the almost same efficiency as the posterior approximator. However, in a view to parallelism, the SIS obviously outperforms the PF, because each sequence of particles can be processed independently in the SIS, whereas the resample step (c) needs to distribute weights u_i of respective particles [2, 3].

4 GPGPU Programming

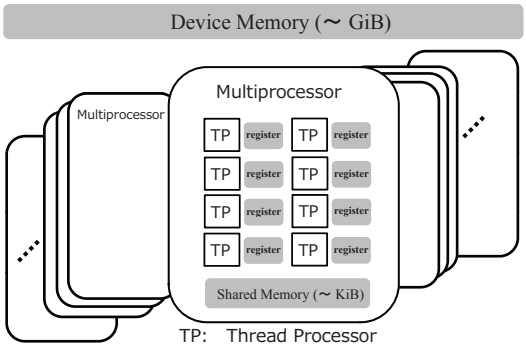


Figure 1: A schematic illustration of GPU.

In this section, we briefly describe the feature of GPU as well as key-points in GPGPU programming.

4.1 Architecture

A GPU is a board consisting of dozens of *multiprocessors* and one GB order *device memory*. The device memory is accessible from all of multiprocessors, but it is not cashed and hence inefficient. The schematic illustration of GPU is shown in Fig. 1. A multiprocessor consists of several thread processors, a *shared memory* and many registers. A total size of the shared memory and the registers is about 100 kB. A single control unit for computation flows is shared in one multiprocessor (such flows are called *threads* in CUDA). Hence, there is a set of 32 threads, called *wrap*, and each member can only execute the same operation.

```
// GPU code
float (*g)[N];
__global__ void foo() {
  // who am I?
  int  tid = blockDim.x * blockIdx.x + threadIdx.x;
  int  Nthreads = blockDim.x * gridDim.x;
  assert (Nthreads == N);
  ...
  for (i = 0; i < M; i++) { g[i][tid] = g[i][tid] + ... ;}  // do my own task.
  if  (tid == 0) { ... }  // do special task, if I am a special thread.
  ...
}

// CPU (host machine) code
float c[M][N];
int main(int argc, char** argv) {
  ...
  cudaMalloc(&g, sizeof(float)*N*M));
  cudaMemcpy(g, c, sizeof(float)*N*M, cudaMemcpyHostToDevice);
  nblocks = N*M/32;  nthreads = 32;
  foo<<<nblocks, nthreads>>>();
  cudaStreamSynchronize(0);
  cudaMemcpy(c, g, sizeof(float)*N*M, cudaMemcpyDeviceToHost);
}
```

Figure 2: Computation flow in CUDA

4.2 CUDA programming environment

High-level programming languages are prepared for parallel programming in GPU. In the case of NVIDIA's GPU products, C-like language called CUDA is prepared. CUDA employs a programming model called single program multiple data stream (SPMD). This is a direct reflection of the limitation such that a warp have to execute a single operation. In Fig. 2, we show a pseudo code. In this figure, we assume that the behaviour of threads in GPU are defined in the procedure `foo()`. First, data referred in GPU code is transfered by API call `cudaMemcpy()`. Then, copies of `foo()` are spawned by a special calling syntax `foo<<<...>>>()`, where the number of copies is specified in the bracket. While GPU works, CPU do some computation or simply go to wait by calling API `cudaStreamSynchronize()`. Finally, the host PC bring back the result of GPU's computation by calling again `cudaMemcpy()`.

Threads are hierarchically grouped: Threads are grouped into a *block* and blocks are further grouped into a *grid*. A rough image of task distribution is the following: A blocks is assigned to a multiprocessor, it is divided into warps, and then a warp is processed by thread processors. Here, one have to note that threads in a block are guaranteed to be executed synchronously (namely two warps can be synchronised), whereas blocks are not. If one have to synchronise among block after some task, they should prepare a GPU code that ends at the task and a CPU code that waits all tasks of all blocks, by calling `cudaThreadsSynchronize()`.

4.3 Remarks on efficiency improvement

Memory usage First the device memory access should be avoided as much as possible. Its latency is several $\times 100$ clocks, whereas the latency of the shared memory access and basic arithmetic operations except division are 4 clocks.

Nonetheless, some device memory accesses are indispensable. In those cases, one should design data structures such that the access to them is *coalesced*, that is the i-th member thread of a warp should access i-th element of an array. Otherwise, memory access of respective threads are serialised.

If many blocks are provided to GPU, the latency may be hidden. Each multiprocessor has a queue of blocks, and if some block goes to access the device memory, another block becomes active with this memory access continued by DMA.

Avoidance of divergent branches CUDA fully support if-then-else structure and threads can do their own task by referring a unique thread ID. However, if different operations are tried to be executed by threads in a wrap, operations in if-clause and those in else-clause are sequentially and both the results are alternately accepted and ignored by respective threads.

5 Parameter estimation

In this section, we demonstrate the parameter estimation of a transcription network using SIS algorithm implemented in CUDA.

5.1 Code design

SIS part In SIS algorithm defined in §3, respective particles are independently computed from start to end. We then straightforwardly assign one particle to one thread. Moreover, due to the absence of resampling, all particles need not be integrated in parallel. This allows us to reduce the required size of memory and the simulation can be done using only shared memory and registers, if the dimension of $\boldsymbol{x}_n$ is not so large.

Model part We use a transcription network model of circadian clock [11], defined by the following differential equations of variables $x_1, \cdots, x_{12}$:

$$
\begin{aligned}
\dot{x}_1 &= -d_1 x_1 + C_1 I(x_5 < s_1^5) I(x_{12} > s_1^{12}) + c_1, \\
\dot{x}_2 &= -d_2 x_2 + L_2 x_1 - b_2^4 x_2 x_4, \\
\dot{x}_3 &= -d_3 x_3 + C_3 I(x_5 < s_3^5) I(x_{12} > s_3^{12}) + c_3, \\
\dot{x}_4 &= -d_4 x_4 + L_4 x_3 - b_2^4 x_2 x_4, \\
\dot{x}_5 &= -d_5 x_5 + b_2^4 x_2 x_4, \\
\dot{x}_6 &= -d_6 x_6 + C_6 I(x_5 < s_6^5) I(x_{12} > s_6^{12}) + c_6, \\
\dot{x}_7 &= -d_7 x_7 + L_7 x_6, \\
\dot{x}_8 &= -d_8 x_8 + c_8, \\
\dot{x}_9 &= -d_9 x_9 + L_9 x_8 - b_9^{11} x_9 x_{11},
\end{aligned}
$$

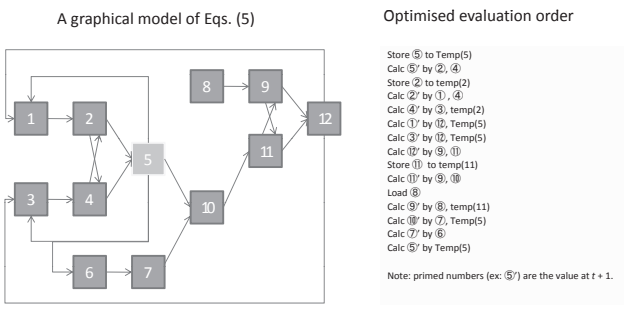


Figure 3: A graphical model of differential equations Eqs. (5). The box 5 corresponding to x_5 has the most connections. If updating from box 5 to develop the time, the number of required temporaries are minimised.

$$\begin{aligned}
\dot{x}_{10} &= -d_{10}x_{10} + C_{10}I(x_5 > s_{10}^5)I(x_7 < s_{10}^7) + c_{10},\\
\dot{x}_{11} &= -d_{11}x_{11} + L_{11}x_{10} - b_9^{11}x_9x_{11},\\
\dot{x}_{12} &= -d_{12}x_{12} + b_9^{11}x_9x_{11},
\end{aligned} \tag{5}$$

where $\dot{x}_i \equiv dx_i/dt$, $I(\text{true}) = 1$ and $I(\text{false}) = 0$. The parameters $\boldsymbol{\theta}$ consists of $(d_i, c_i, C_i, L_i, s_i^j, b_i^j)$. If we symbolically write these equations as $\dot{x} = g(x|\boldsymbol{\theta})$ and their solution as $x = \phi(t)$, then system model Eq. (1) is given by

$$\boldsymbol{x}_n = \boldsymbol{x}_{n-1} + \int_0^{\Delta t} g(\boldsymbol{\phi}(t)|\theta)dt \quad \text{with} \quad \boldsymbol{\phi}(0) = \boldsymbol{x}_{n-1} + \boldsymbol{v}_n. \tag{6}$$

We approximated this by Euler method with $\Delta t = 0.01$. The data for components $\mathcal{I}_{\text{obs}} = \{1, 3, 6, 10\}$ of $\boldsymbol{y}_{i,n}$ are available at 12 time points $n \in \{n_1, \cdots, n_{12}\}$ such that $n_{k+1} - n_k \fallingdotseq 64$ [14]. We assume the likelihood at time n as follows:

$$u_n = p(\boldsymbol{y}_n|\boldsymbol{x}_n) \propto \exp[-\sum_{i\in\mathcal{I}_{\text{obs}}}(y_{i,n} - x_{i,n})^2/2\sigma_i^2] \tag{7}$$

with $\sigma_1 = 0.64$ and $\sigma_i = 0.25$ $(i \neq 1)$.

Other implementation note If one integrates Eqs. (5) using Euler method, the order of evaluations of $x_{i,n}$ $(i = 1, \cdots, 12)$ is arbitrary. By evaluating variable with the most connections, one can reduce the number of temporary variables used to hold previous state $x_{i,n-1}$. The schematics illustration is shown in Fig. (3).

Overall structure The structure of our SIS code is summarised in Fig. 4.

5.2 Result

The result of the estimation involving 10^9 particles in GPU is shown as follows. We here use Tesla C 870 GPU. The whole computation time is within 8 hours, where it would take several days in a single core CPU. Table 1 compares computation times in the GPU, a PC cluster that consists of 96 CPUs, and a single CPU (interpolation from the PC cluster).

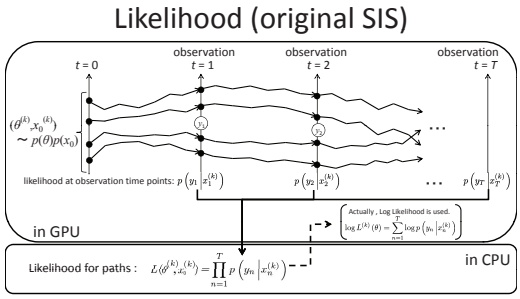


Figure 4: The structure of SIS code for GPU.

Figure 5 shows the observed time course versus the simulation time course of the maximum a posteriori (MAP) solution among 10^9 particles. We notice that the simulation time course well reproduces the periodicity inherent in the observation time course.

environment	GPU	1 CPU core	96 CPU cores
comp. time	7h45m	(8 days)	2h

Table 1: Computation times of GPU and CPU(s). The time in a single core is scaled from the time measured using 96 cores of a PC cluster.

6 Accurate evaluation of posterior

6.1 Code modification

The Monte Carlo approximation of the likelihood is given by

$$p(\boldsymbol{Y}|\boldsymbol{\theta}, \boldsymbol{x}_0) = \lim_{M\to\infty} \frac{1}{M}\sum_{i=1}^{M}\prod_{n\in\mathcal{N}_{\text{obs}}} p(y_n|x_n^{(i)}), \tag{8}$$

where $(x_n^{(i)})_{i=1}^M$ are M stochastic variants of the solution of Eq. (2), for a given configuration of $(\boldsymbol{\theta}, \boldsymbol{x}_0)$. In actual computations, this is approximated by a certain finite value of M. In this section, we study how the accuracy of the likelihood depends on the number of particles M.

The procedure to calculate Eq. (8) is constructed as a modified version of procedure to calculate the posterior for a certain set $\{(\boldsymbol{\theta}, \boldsymbol{x}_0)\}$, shown in Fig. 4. All needed modifications of the procedure are to assign a common configuration of $(\boldsymbol{\theta}, \boldsymbol{x}_0)$ to all threads (particles) and to add a step which averages likelihoods of respective variants. The schematic illustration of this modification is shown in Fig. 6. However, since all threads take a common value of $(\boldsymbol{\theta}, \boldsymbol{x}_0)$, it is enough to prepare a single area containing this value in the shared memory, instead of preparing areas for each threads. Therefore, the consumption of the shared memory per thread becomes smaller than that of the original SIS. This reduction of the memory consumption increases the number of active threads in a multiprocessor.

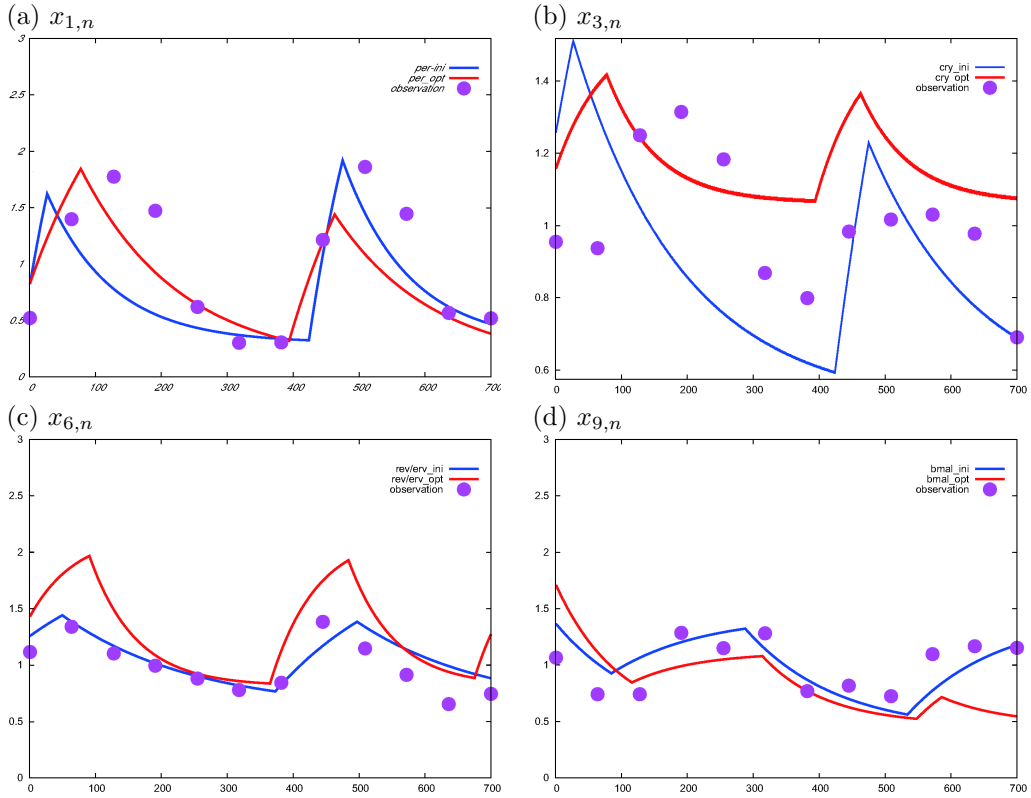


Figure 5: Comparison of the observation time course and a simulation one. This simulation time course is the MAP solution of 10^9 particles.

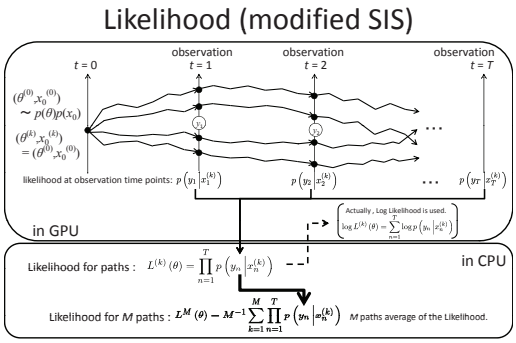


Figure 6: Modified of SIS code to improve the accuracy of likelihoods.

6.2 Numerical Experiment

Let us observe how the precision of the posterior is improved as the number M of particles increases. For simplicity, we study the posterior as a function of single parameter C_3, for fixed values of the other parameters.

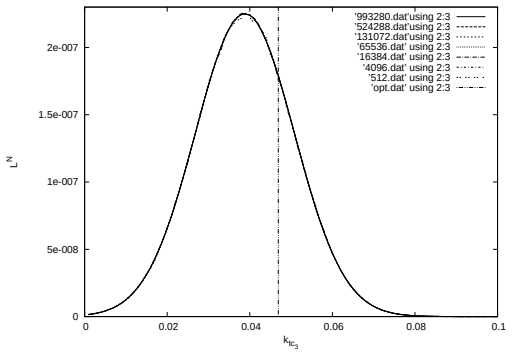
Distributions of the posterior, as a function of C_3, is shown in Fig. 7 for different M from 512 to 993,280. The effect of increased number M is displayed in the vicinity of the peaks (Fig, 7). Intuitively, to capture a smoothness of the function, an order of thousand particles seems to be needed. In order to select some appropriate number M from this result, one have to decide the required precision in $\boldsymbol{\theta}$. The value at $C_3 = 39\times10^{-3}$ is less than or nearly equal to in $M \leq 131,072$ the value at $C_3 = 38 \times 10^{-3}$, whereas the order is reversed in $M = 524,288$ and 993,280. Hence, 5×10^5 particles are necessary if C_3 should be determined in a precision of two significant figures. The number of particles can be reduced to less than 10^4, if $C_3 = 39\times10^{-3}$ is acceptable as a value with the error of 10^{-3}.

In the vicinity of the peak, which has a actual influence in the selection of the MAP solution, the difference of the value of the posterior is $\sim 3\%$. Hence, the Monte Carlo error in the evaluated posterior is considered to be not serious in the parameter selection.

7 Summary

We have demonstrated the two types of the GPGPU-implementation of SIS, and applied to the search for kinetics parameters in a biological circuit model. In the first demonstration, we show that the computation time is reduced from 8 days to 8 hours. It is important that a straightforward approach becomes applicable under the high performance of GPU. In the second demonstration, we examine how the Monte Carlo error in the value of the posterior calculated by SIS depends on the number of particles dedicated to a given configuration of parameters. The result shows that the Monte Carlo error is not serious to the parameter selection.

(a) whole distribution



(b) magnification around the peak

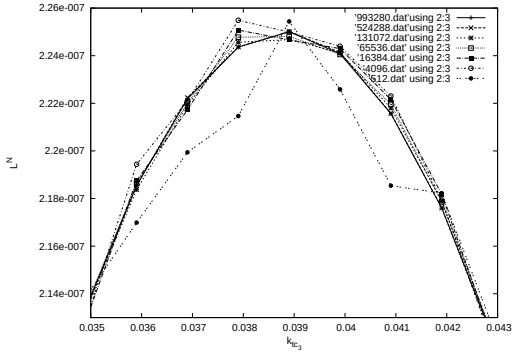


Figure 7: The posterior of a single parameter C_3 for different number of particles.

However, the possibility that the improvement of the posterior yield the improvement of its derivative and gradient methods becomes applicable is remained as a problem that should studied.

References

[1] Nvidia cuda compute unified device architecture programming guide version 2.2.1. . [http : //www.nvidia.co.jp/object/cuda_home.html]

[2] M. Bolic, P.M. Djuric, and S. Hong. Resampling algorithms for particle filters: a computational complexity perspective. *Journal on Applied Signal Processing*, 55:2267–2277, 2004.

[3] M. Bolic, P.M. Djuric, and S. Hong. Resampling algorithms and architectures for distributed particle filters. *IEEE Transactions on Signal Processing*, 5:2442–2450, 2005.

[4] N. J. Gordon, D.J. Salmond, and A.F.M. Smith. Novel approach to nonlinear/non-gassian bayesian state estimation. *IEE Proceedings-F*, 140:107–113, 1993.

[5] N. Ikoma, R. Yamaguchi, H. Kawano, and H. Maeda. Tracking of multiple moving objects in dynamic image of omni-directional camera using phd filter. *Journal of Advanced Computational Intelligence and Intelligent Informatics*, 12:16–25, 2008.

[6] S. Kikuchi, D. Tominaga, M. Arita, K. Takahashi, and M. Tomita. Dynamic modeling of genetic networks using genetic algorithm and s-system. *Bioinformatics*, 19:643–650, 2003.

[7] S. Kimura, K. Ide, A. Kashihara, M. Kano, M. Hatakeyama, R. Masui, N. Nakagawa, S. Yokoyama, S. Kuramitsu, and A. Konagaya. Inference of s-system models of genetic networks using a cooperative coevolutionary algorithm. *Bioinformatics*, 21:1154–1163, 2004.

[8] G. Kitagawa. Monte carlo filter and smoother for non-gaussian nonlinear state space models. *Journal of computational and graphical statistics*, 5:1–25, 1996.

[9] G. Kitagawa and W. Gersch. *Smoothness priors analysis of time series*. Springer-Verlag, New York, 1996.

[10] J. S. Liu. *Monte Carlo strategies in scientific computing*. Springer-Verlag, 2001.

[11] H. Matsuno, S. T. Inouye, Y. Okitsu, Y. Fujii, and S. Miyano. A new regulatory interactions suggested by simulations for circadian genetic control mechanism in mammals. *Journal of Bioinformatics and Computational Biology*, 4:139–153, 2006.

[12] M. Nagasaki, R. Yamaguchi, R. Yoshida, S. Imoto, A. Doi, Y. Tamada, H. Matsuno, S. Miyano, and T. Higuchi. Genomic data assimilation for estimating hybrid functional petri net from time-course gene expression data. *Genome Informatics*, 17:46–61, 2006.

[13] K. Nakamura, R. Yoshida, M. Nagasaki, S. Miyano, and T. Higuchi. Parameter estimation of *in silico* biological pathways with particle filtering towards peta-scale computing. *Pacific Symposium on Biocomputing*, pages 227–238, 2009.

[14] H. R. Ueda, W. Chen, A. Adachi, H. Wakamatsu, S. Hayashi, T. Takasugi, M. Nagano, K. Nakahama, Y. Suzuki, S. Sugano, M. Iino, Y. Shigeyoshi, and S. Hashimoto. A transcription factor response element for gene expression during circadian night. *Nature*, 418:534–539, 2002.

[15] R. Yoshida, M. Nagasaki, R. Yamaguchi, R. Imoto, S. Miyano, and T. Higuchi. Bayesian learning of biological pathways on genomic data assimilation. *Bioinformatics*, 24:2592–2601, 2008.