Doctoral Dissertation

Parallel Processing Techniques for Parameter Estimation in Bayesian methods

Hiroki Nishimoto

March 17, 2023

Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Hiroki Nishimoto

Thesis Committee:

Professor Yasuhiko Nakashima	(Supervisor)
Professor Kazushi Ikeda	(Co-supervisor)
Associate Professor Renyuan Zhang	(Co-supervisor)
Assistant Professor Kan Yirong	(Co-supervisor)
Assistant Professor Pham Hoai Luan	(Co-supervisor)

Parallel Processing Techniques for Parameter Estimation in Bayesian methods*

Hiroki Nishimoto

Abstract

In recent years, the increase in computer data processing speeds and Internet communication speeds has made it possible to store a vast amount of data on a wide variety of events that occur in the world. This massive amount of data is called "big data," and the trend is to analyze it and use it for business and research. In addition to neural networks, which have been attracting attention recently, there is another method of data analysis that defines data as a mathematical model, sets the probability distribution that generates the data, and infers the parameters of the probability distribution from the data using an algorithm based on Bayesian inference. This method using Bayesian inference is more explanatory than the one using neural networks and is still used as the center of data analysis even now that neural networks are widely used.

The first part of this study consists of two parts. The first part is a study on speeding up a method called clustering, which is one of the typical big data analysis methods that classify data into groups by similarity and analyze their attributes from among various properties and characteristics that exist in a mixture. Clustering is a method that attracts attention as a means of analyzing user segments and recognizing brand positions, etc. There are various methods to realize clustering, but the method using a mixed Gaussian distribution that classifies data according to several Gaussian distributions is highly accurate, and furthermore, the method that uses the concept of Bayesian inference for the mixed Gaussian distribution has high accuracy. Clustering by Gaussian distribution obtained by parameter estimation using the variational inference method, which uses the concept of Bayesian inference and the parameters of the Gaussian distribution as random

*Doctoral Dissertation, Graduate School of Information Science,
Nara Institute of Science and Technology, March 17, 2023.

variables generated from the cointegration probability distribution, is characterized by its difficulty in overlearning and its ability to automatically determine the number of clusters. However, the variational inference method for mixed Gaussian distributions is known to take a long time to converge, and the computation time is enormous in proportion to the number of data. Therefore, as a first step in applying variational mixed Gaussian distributions to large-scale data, the variational inference method for Gaussian distributions in parallel using GPUs is implemented and evaluated, which is a parallel computing architecture. The proposed method was about 192 times faster when the number of data was about 1 million and about 107 times faster when the number of clusters was 256 while maintaining the same accuracy as the CPU implementation of the same algorithm. Compared to the method using the EM algorithm, it was also able to prevent excessive shrinkage in the number of clusters while maintaining the same speed and clustering score.

The second part is concerned with the design and evaluation of hardware to speed up parameter estimation in sequential Monte Carlo methods. The variational inference method described above is a powerful parameter estimation technique. However, it has a narrow range of applicability depending on the model and probability distribution used. There are only a few models, such as variational mixed Gaussian distributions, for which calculation algorithms have been established. Furthermore, complex models are difficult to implement. Instead, parameter estimation methods that use random number generators, such as Markov chain Monte Carlo methods, are used. The sequential Monte Carlo method generates many groups of probability distributions called particles. Then, random numbers are generated from each group, the likelihood of the data is calculated, and those with high likelihood are copied, and those with low likelihood are rejected, which is suitable for parallelization at the hardware level. This research is concerned with the hardware implementation of the process Resampling step, which is the bottleneck for speeding up parameter estimation using sequential Monte Carlo methods. In order to improve the efficiency of hardware dedicated to metropolis resampling, metropolis resampling is optimized for integer execution and evaluated the algorithm and hardware for 32, 16, and 8-bit data widths. The proposed method maintains the same resampling performance as the common 32-bit single-precision floating-point method in 8-bit execution. In the hardware evaluation, it reduces resource usage in any data width implementation. In the main modules such as the coefficient

generator and the execution of the Metropolis test, the LUT In the main modules, such as the coefficient generator and the Metropolis test, the system achieved a 31% reduction in 32-bit, 57% in 16-bit, and 64% in 8-bit, as well as up to 3.0x improvement in throughput and up to 75% reduction in memory usage at other bottlenecks.

Keywords:

Parallel Computing, Architecture, Approximate, Gaussian Mixture Models, Sequential Monte Carlo

Contents

List of Figures vii

List of Tables ix

I. GPGPU-oriented Optimization of Variational Gaussian Mixture Models 1

1. Introduction 2
- 1.1. Motivation . . . 2
- 1.2. Challenges and Contribution . . . 3
- 1.3. Composition of Part I . . . 3

2. Background Theory and Related Works 4
- 2.1. Structural Analysis of GPGPU . . . 4
- 2.2. Fundamentals of Gaussian Mixture Models . . . 5
- 2.3. Variational Inference for GMM . . . 7
- 2.4. Related Works of Gaussian Mixture Models on GPUs and FPGAs . . . 11

3. Proposed GPU Implementation of Variational Gaussian Mixture Models 12
- 3.1. Optimizing Memory Allocation . . . 13
- 3.2. Optimizing Number of Threads . . . 13
- 3.3. Changing Execution Order . . . 14
- 3.4. Details of each kernel . . . 14
- 3.5. Optimizing Data Transfer and CPU-GPGPU Co-operationscheme . . . 21

4. Evaluations and Results **23**
4.1. Evaluation with Artificial Data 24
4.1.1. Kernel processing time compared to CPU 24
4.1.2. Kernel processing time compared to OpenCL 24
4.1.3. Comparative experiments with varying parameters of the dataset 25
Varying number of data 25
Varying number of cluster 28
Varying number of dimensions 28
4.1.4. Comparison with FPGA implementation 29
4.2. Evaluation with Practical Data 32

5. Discussion **35**

II. Resampling High Efficient Hardware for Sequential Monte Carlo **38**

6. Introduction **39**
6.1. Motivation 39
6.2. Challenges and Contribution 40
6.3. Composition of Part II 41

7. Background Theory and Related Works **42**
7.1. Overview of Sequential Monte Carlo: SMC 42
7.2. Application of Sequential Monte Carlo 44
7.3. Problems of Sequential Monte Carlo 46
7.4. Related Works on Resamping and Its Speeding Up 46

8. Proposed Integer-Optimized Metropolis Resampling **50**

9. Evaluations and Results **53**
9.1. Evaluation Items 53
9.1.1. Root Mean Square Error : $RMSE$ 53
9.1.2. Effective Sample Size : ESS 54

9.2. Evaluation as the Resampling Algorithm 54
9.2.1. Evaluation Overview 55
9.2.2. Result 56
9.3. Evaluation of Resampling Quality by Randomness 56
9.3.1. Evalution Overview 57
9.3.2. Result 58
9.4. Evaluation as a part of Sequential Monte Carlo Sampler 62
9.4.1. Evaluation Overview 62
9.4.2. Result 63
9.5. Evaluation of Hardware Efficiency 71
9.5.1. Implementation Details of each IP 72
9.5.2. Result 76

10. Discussion **80**

III. Conclusion 83

References **87**

Publication List **99**

List of Figures

2.1. Architecture of GPUs. 5
2.2. Constitution of CUDA Kernel. 6
2.3. Example of a GMM. 7
2.4. Graphical models of Two types of GMMs. 8
2.5. (a) The clustering result of GMM using EM algorithm
(b) The clustering result of GMM using variational inference 11

3.1. Contiguous memory placement for data. 13
3.2. Overview of the GAUSS kernel 17
3.3. Overview of the WLP and LR kernels. 18
3.4. Overview of the SR kernel 20

4.1. Execution time and DB Score depend on the number of data. 27
4.2. Execution time and DB Score depending on the number of clusters. . 29
4.3. Execution time and DB Score depending on the number of dimensions. 30

7.1. The conceptual diagram of the SMC. 43
7.2. Breakdown of time in SMC in GPU implementation. 47
7.3. An example of the operation of the Simplified Random permutation Generator. 49

9.1. Distributions of each input weight. 55
9.2. Comparison result of RMSE
when varying the number of particles from 2^{10} to 2^{20}. 57
9.3. Comparison result of RMSE
when varying the number of Metropolis tests from 2^0 to 2^{10}. 58
9.4. Comparison result of RMSE when varying the number of particles
from 2^{10} to 2^{20} and using three types of RNGs 60

9.5. Comparison result of RMSE when varying the number of Metropolis tests from 2^0 to 2^{10} and using three types of RNGs 61
9.6. Comparison result of RMSE when varying the number of particles from 2^{10} to 2^{20} on SMC sampler 64
9.7. Comparison result of RMSE when varying the number of particles from 2^{10} to 2^{20} on SMC sampler (first ten trials) 65
9.8. Comparison result of ESS when varying the number of particles from 2^{10} to 2^{20} on SMC sampler (first ten trials) 66
9.9. Comparison result of RMSE when varying the number of Metropolis tests from 2^{10} to 2^{20} on SMC sampler 67
9.10. Comparison result of RMSE when varying the number of Metropolis tests from 2^{10} to 2^{20} on SMC sampler (first ten trials) 68
9.11. Comparison result of ESS when varying the number of Metropolis tests from 2^{10} to 2^{20} on SMC sampler (first ten trials) 69
9.12. Test design of integer-optimized Metropolis resampling architecture . 71
9.13. Detail of the Parallelized Metropolis Block: PMB. 74
9.14. Detail of the Random Permutation Generator: RPG. 76
9.15. Detail of the coefficients generator: U-Gen. 79

List of Tables

3.1. Correspondence between the implemented kernel and the equations in the variational-EM algorithm 15

4.1. Evaluation environment 23
4.2. Comparison of CUDA and CPU for the breakdown of execution time for each kernel in one iteration 25
4.3. Comparison of CUDA and OpenCL for the breakdown of execution time for each kernel in one iteration 26
4.4. Evaluation items and their summary 26
4.5. Result of evaluation with varying the number of data 27
4.6. Result of evaluation with varying the number of clusters 28
4.7. Result of evaluation with varying the number of dimensions 29
4.8. Comparison results with FPGA implementation of GMM by EM algorithm 31
4.9. Outline of Practical Data 32
4.10. Evaluation result with practical data sets 33

9.1. The evaluation parameters and its overview 56
9.2. Comparison result of the number of resampling run when varying the number of particles P from 2^{10} to 2^{20} 66
9.3. Comparison result of the number of resampling run when varying the number of Metropolis tests from 2^{0} to 2^{10} 70
9.4. Variables are used in Metropolis Resampling Circuit 72
9.5. Bus Overview of Metropolis Resampling Circuit 73
9.6. The input/output ports of the PMB 75
9.7. The input/output ports of the RPG 77
9.8. The input/output ports of the U-Gen 77

9.9. Comparison of resource usage for each data type and data width when parallelism M set to 16 and number of particles set to 2^{20} 78

Part I.

GPGPU-oriented Optimization of Variational Gaussian Mixture Models

1. Introduction

1.1. Motivation

Machine learning technologies have been developed along with the performance improvement of CPUs and general-purpose graphic processing units (GPGPUs). Various real-world applications, such as the analysis of big data, are well performed by using advanced machine learning technologies, where clustering is one of the significant tasks [1, 2].

Among all clustering algorithms, the Gaussian mixture model (GMM) is known as a representative methodology for the wide use of data mining and computer vision. By probability density modeling, GMMs perform near-nature applications well due to the soft clustering mechanism. Patel et al. [3] have presented a more sensitive clustering than K-means, also a clustering algorithm, in analyzing big data. For example, Reynold et al. [4] have constructed a speaker verification system using GMM to model the speaker's voice from speech data and have succeeded in the NIST speaker recognition evaluation. Also, Stauffer et al. [5] designed a system that uses probabilistic analyses to determine whether each pixel belongs to a background, instead of belonging and not belonging binary values in order to extract the background from a video stream using GMM.

However, the parameters of a GMM are always estimated from the datum by a combination of many methods. As typical fashions of parameter estimation, the expectation-maximization(EM) algorithm, Markov Chain Monte Carlo, and variational Bayesian method are applied.

Parameter estimation is one of the critical issues to clustering quality. The variational Bayesian Gaussian mixture model (VB-GMM), in which the parameters are estimated through variational Bayesian theory [6], is widely considered a high-performance candidate since it eliminates the over-fitting problem, and the appropriate number

of clusters can be determined in a single training run without cross-validation [5] Unfortunately, VB-GMM leads to the computation explosion. Much more iterations are necessary for convergence in contrast to most other parameter estimation fashions [7]. Thus, CPU-oriented implementations of VB-GMM are impractical in some application fields due to poor speed. For conventional GMM schemes such as those by expectation-maximization, the GPGPU implementations have been well investigated and proven helpful to speed up [8].

1.2. Challenges and Contribution

In this work, the GPGPU implementations of VB-GMM are developed by structure-oriented optimizations. Fitting the parallelism specification and memory structure of GPGPU, the CPU-GPGPU co-operation, execution re-order, and memory optimization propose the VB-GMM optimization strategy. An implementation flow with thirteen stages is presented in detail. Following this implementation flow, the VB-GMM, which is usually executed on the CPU, is migrated onto the GPGPU platform. Five types of real-world clustering tasks are verified by the proposed GPGPU implementation of VB-GMM, including MNIST, CIFAR10, PAMAP2, and Gas sensors for home activity monitoring Data Set [9–12], The experimental results show that the VB-GMM is successfully executed on GPGPU platforms and achieves 192 times speed-up compared to CPU. Moreover, It succeeded in finding the correct number of clusters, which is challenging to do with the EM algorithm.

1.3. Composition of Part I

The rest of this part is organized as follows. Section 2 provides an introduction to GMMs and the variational Bayesian method for GMM, an introduction to GPU architecture, and introduces previous work. The implementation of VB-GMM on GPGPU is explained in Section 3. Section 4 presents some evaluation of our implementation and the comparison of results with other implementations. The extended discussion is shown in Section 5. The conclusion is presented in Part III, along with those from Part II.

2. Background Theory and Related Works

2.1. Structural Analysis of GPGPU

Graphics processing units (GPUs) were initially developed as computing processors for generating and displaying 3D graphics. GPUs have massive homogeneous cores and achieve high speed by distributing the operations to those cores. Due to the high parallelism and processing capacity, the general-purpose utilization of GPUs leads the trend in many fields of high-performance computing [13, 14]. In general, GPUs perform well for executing single instruction multiple data (SIMD), in which a single instruction is applied to multiple data simultaneously, and all operations are processed in parallel, The key to speeding up processing lies in parallelizing the instructions.

The architecture of GPUs supporting NVIDIA's CUDA [15] is briefly shown in Fig. 2.1. This GPU consists of a streaming multiprocessor (SM) lined up on a block, a thread execution manager that controls the SM, and a video memory that stores the data. This video memory is called device memory, as opposed to host memory, which is managed by the CPU. The number of streaming multiprocessors (SMs) varies in the model. The GPU used in this study, the NVIDIA GTX RTX3090, has 82 SMs [16]. As mentioned earlier, each SM contains cores, shared memory, and registers, which are small in capacity but fast in access. Parallel operations are performed using these many cores and high-speed memory. The device memory is implemented with DRAM, which is slower in access speed than the shared memory but has a large capacity. In parallel operations using the GPU, data is moved among the host memory, device memory, and processing cores. After the calculation, the data is stored in the device memory; and then moved from the device memory to the main memory of the host device.

As shown in Fig. 2.2, the kernel of CUDA consists of Threads, which is the smallest unit of instructions; Thread Block, which summarizes Thread; and Grid, which summarizes Thread Blocks hierarchically. The user can arbitrarily change the

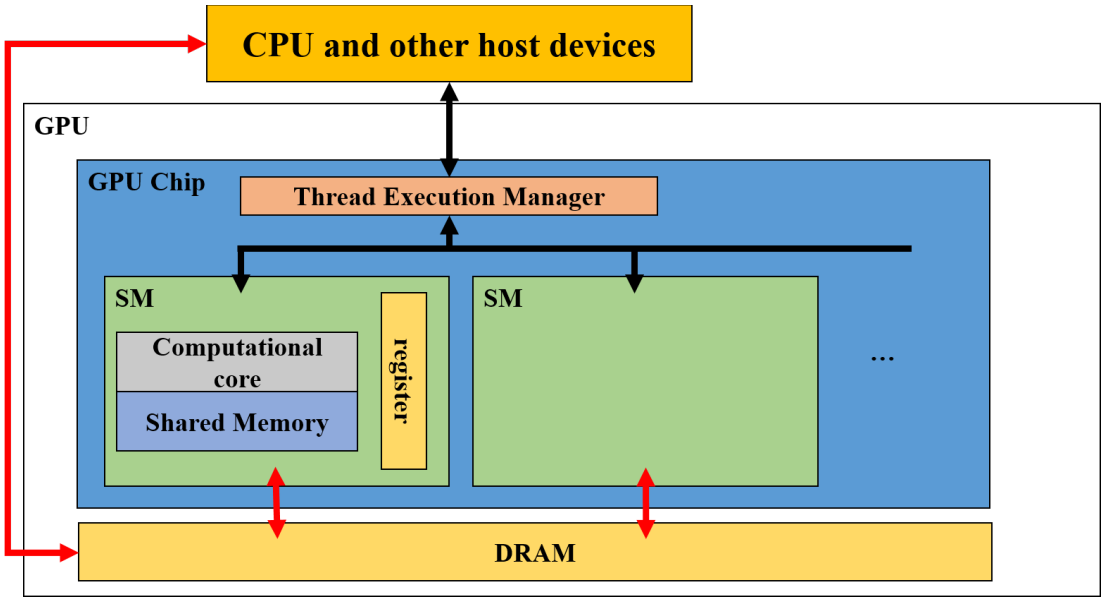


Figure 2.1.: Architecture of GPUs. The red arrows show the flow of data, and the black arrows show the flow of instructions.

number of dimensions of the Thread Block, as shown in BlockDim.x and y in the figure, as long as the limit is not exceeded. Within a thread block, 32 threads are divided into a group called a warp; and in the SM, memory readings and operations are performed in parallel in this warp. Since threads in the same warp are always executed simultaneously, if even one thread is delayed by a conditional branch etc., the efficiency of all threads in the warp is reduced. In addition, by accessing consecutive global memory addresses within the same warp, a high-speed memory access called coalesce access becomes feasible. Therefore, the efficiency of memory access and operations in each warp is important for GPU acceleration [17].

2.2. Fundamentals of Gaussian Mixture Models

Gaussian Mixture Models: GMM is a probabilistic model for clustering as represented by Eq. 2.1.

$$p(x|\pi,\mu,\Sigma) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k,\Sigma_k). \tag{2.1}$$

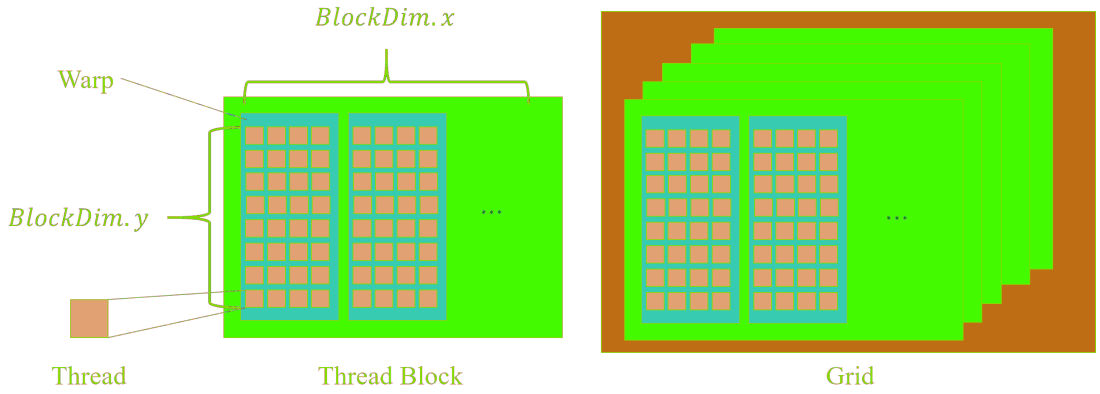


Figure 2.2.: Constitution of CUDA Kernel. The kernel of a GPU consists of threads, blocks, which are collections of threads, and grids, which are collections of blocks. The actual number of threads running on the SM is 32, which is called the warp.

It is expressed by the sum of the number of clusters K Gaussian distribution $\mathcal{N}(x|\mu_k, \Sigma_k)$ multiplied by each mixture weight π_k. Figure 2.3 shows an example of GMM, where the number of clusters and the dimension is 3 and 1, respectively.

In addition to clustering for data analysis, GMMs have been used in a variety of applications. Reynold et al. [4] used GMMs to model the speaker's voice from speech data to construct a speaker-matching system, which was successfully evaluated in the NIST speaker recognition evaluation. Toda et al. [18] have constructed a voice quality transformation framework using GMMs. Stauffer et al. [5] designed a probabilistic system for extracting background from video streams using GMMs, where each pixel is assigned to a background rather than a binary value of belonging or not belonging. Fujita et al. [19] used GMMs to model wireless location information for location estimation using wireless LANs, reducing the amount of data to about 5 percent of that of conventional methods and making the database lighter.

In recent years, more and more research has been combining GMM with Deep Neural Networks (DNN). Shahin et al. [20] combined GMM and DNN to design an emotion recognition model that is more robust to noise than existing methods. Koguchi et al. [21] overcame the problem of GMM-based speech synthesis, where GMM performs well in speech classification but poorly in speech synthesis, with a GMM-DNN composite model in which the synthesis part is replaced by a DNN, achieving better-synthesized

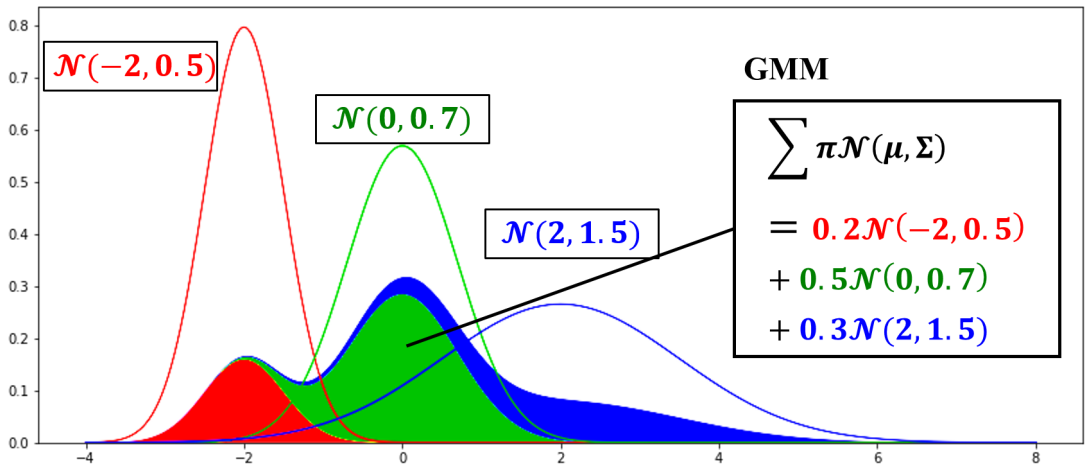


Figure 2.3.: Example of a GMM. This GMM consists of 3 weights(π) and 3 Gaussian distributions, and it is the sum of the 3 Gaussian distributions multiplied by π={0.2,0.5,0.3}.

speech than GMM alone and better initialization than DNN alone.

2.3. Variational Inference for GMM

For adapting the GMM to the dataset, three parameters π, μ, and Σ must be optimized for all K clusters.

However, the optimal GMM parameters are difficult to compute directly from the data. It is common to compute them using algorithms that maximize the likelihood by iterative computation or It is common to compute them using an iterative algorithm to maximize the likelihood of a sampling algorithm.

The variational Bayesian method is a general strategy for parameter estimation, which is widely applied to various problems of function optimization. In this work, this method is employed for the parameter estimation of GMM. Three distributions, Dirichlet, Gaussian, and Wishart are introduced as prior distributions for π, μ, and Λ, respectively as shown in Eq.s (2.2, 2.3 ,2.4).

$$\pi \sim Dir(\alpha). \tag{2.2}$$

$$\mu \sim \mathcal{N}(m, (\beta\Lambda)^{-1}). \tag{2.3}$$

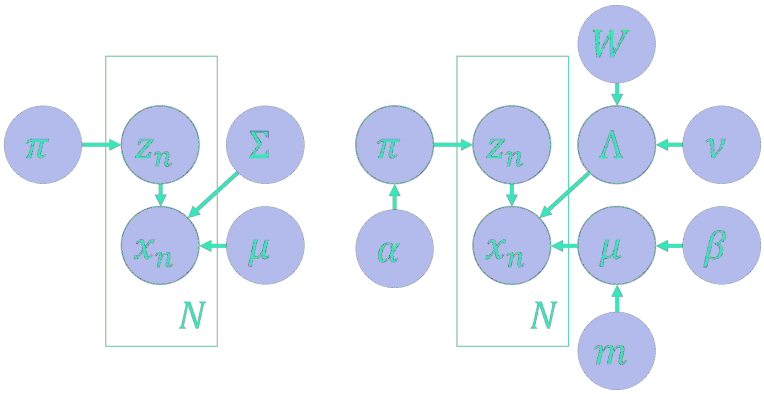


Figure 2.4.: Graphical models of Two types of GMMs. The left one shows GMM with the EM algorithm. The right one shows GMM with variational Inference.

$$\Lambda \sim \mathcal{W}(W, \nu). \tag{2.4}$$

where α is a parameter of Dirichlet distribution that is a prior distribution of π; m and β are parameters of a Gaussian distribution that is a prior distribution of μ; and W and ν are parameters of a Wishart distribution that is a prior distribution of an inverse matrix Λ of a covariance matrix Σ. Figure 2.4 shows graphical models of two types of GMMs with EM algorithms and variational inference, respectively. Thus, VB-GMM is seen as an extension of the parameters of EM-GMM as a probability distribution. Introducing these prior distributions makes it possible to obtain the parameters of GMM by an algorithm, which is called the variational-EM algorithm.

This algorithm consists of four steps, as shown below.

1. Initialization

   In this step, mean μ and precision Λ by are initialized by data X, and also initialize mixture weight π is initialized by the number of cluster K, which are as hyper-parameter Also, α, β, W, ν, m are initialized by each hyper-parameter.

2. Variational E Step

   In this step, π, Λ are estimated.

$$\ln \tilde{\pi} \equiv \mathbb{E}[\ln \pi_k] = \psi(\alpha_k) - \psi\left(\sum_{i=1}^{K}(\alpha_i)\right). \tag{2.5}$$

$$\begin{aligned}\ln \tilde{\Lambda}_k &\equiv \mathbb{E}[\ln |\Lambda_k|] \\ &= \sum_{d=1}^{D} \psi\left(\frac{\nu_k + 1 - d}{2}\right) + D \ln 2 + \ln |W_k|.\end{aligned} \tag{2.6}$$

Also, estimate correlation coefficient ρ by current parameters.

$$\begin{aligned}\rho_{nk} &= \pi_k |\Lambda_k|^{1/2} \\ &\quad \times \exp\left\{-\frac{D}{2\beta_k} - \frac{\nu_k}{2}(x_n - \mu_k)^T W_k (x_k - \mu_k)\right\}.\end{aligned} \tag{2.7}$$

At the end of the E step, normalize p to obtain the responsibility r by Eq. 2.8.

$$r_{nk} = \frac{\rho_n k}{\sum_{j=1}^{K} \rho_{nj}}. \tag{2.8}$$

3. Variational M Step

In this step, some parameters are computed by updated parameters in step 2.

$$N_k = \sum_{n=1}^{N} r_{nk}. \tag{2.9}$$

$$\bar{x}_k = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk} x_n. \tag{2.10}$$

$$S_k = \frac{1}{N_k} r_{nk} (x_n - \bar{x}_k)(x_n - \bar{x}_k)^T. \tag{2.11}$$

$$\alpha_k = \alpha_0 + N_k. \tag{2.12}$$

$$\beta_k = \beta_0 + N_k. \tag{2.13}$$

$$\nu_k = \nu_0 + N_k. \tag{2.14}$$

$$m_k = \frac{1}{\beta_k}(\beta_0 m_0 + N_k \bar{x}_k). \tag{2.15}$$

$$W_k^{-1} = W_0^{-1} + N_k S_k + \frac{\beta_0 N_k}{\beta_0 + N_k}(\bar{x}_k - m_0)(\bar{x}_k - m_0)^T. \tag{2.16}$$

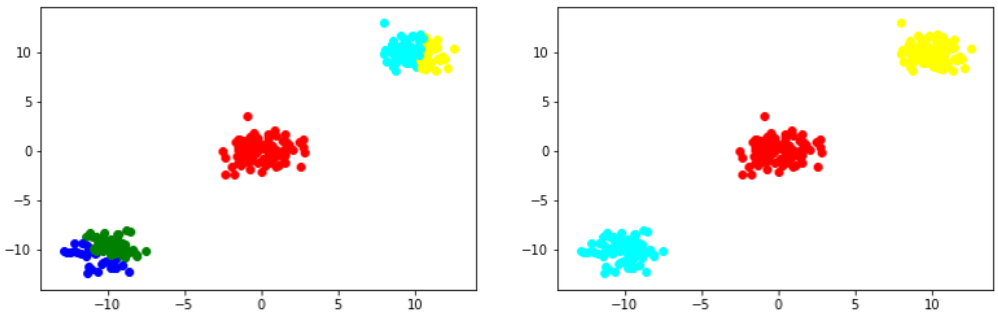
4. Convergence check

   Lower bound $\mathcal{L}_{new}$ is computed by Eq. 2.17. Comparing $\mathcal{L}_{new}$ with $\mathcal{L}_{old}$, if the difference between them falls below a pre-defined value, the process terminates. Otherwise, $\mathcal{L}_{new}$ is substituted into $\mathcal{L}_{old}$ and the process is returned to step 2.

$$\begin{aligned}\mathcal{L} = &- \sum_{n=1}^{N}\sum_{k=1}^{K}(e^{r_{nk}} \times r_{nk}) \\ &- \sum_{k=1}^{K}\left(\nu_k|W_k| + \frac{\nu_k D \ln 2}{2} - \sum_{k=1}^{K}\ln\Gamma(\nu_k)\right) \\ &- \left(\ln\Gamma\left(\sum_{k=1}^{K}\alpha_k\right) - \sum_{k=1}^{K}\ln\Gamma(\alpha_k)\right) \\ &- \frac{D\sum_{k=1}^{K}\ln\beta_k}{2}.\end{aligned} \tag{2.17}$$

The above process can obtain parameters that fit the data of interest. The GMM obtained by this algorithm optimizes the number of clusters K, which indicates how many Gaussian distributions the GMM is composed of. Unlike GMMs obtained by maximum likelihood estimation with the EM algorithm, they are fitted without excessive data decomposition.

Figure 2.5(a) shows data sampled from a GMM with K clusters and two dimensions. The hyperparameter, the number of clusters K, is given as 5. The clustering result is the GMM fitted by the maximum likelihood estimation using the EM algorithm. Figure 2.5(b) shows the result of clustering the same data with the same hyperparameter of five for the number of clusters K, fitted with the variational EM algorithm. The GMM clustering obtained by parameter estimation with the EM algorithm in Figure2.5(a) over-decomposes the data, whereas the clustering with the GMM fitted with the variational



(a) (b)

Figure 2.5.: (a) The clustering result of GMM using EM algorithm
(b) The clustering result of GMM using variational inference

EM algorithm in Figure2.5(a) over-decomposes the data. The clustering of GMMs obtained by parameter estimation with the variational EM algorithm in Figure2.5(b) shows that the data is divided into three clusters, which is the true number of clusters.

2.4. Related Works of Gaussian Mixture Models on GPUs and FPGAs

For related research on speeding up GMM, Guo et al. [22] made the EM algorithm for Gaussian mixture models suitable for pipeline processing in order to speed up execution on CPU and FPGA. In addition, He et al. [23] implemented FPGA processing by making the pipeline processing of [22] more efficient. Kumar et al. [8] proposed an EM algorithm model for GMM using a GPGPU. However, both are implementations of maximum likelihood estimation using the EM algorithm, and the speeding up of VB-GMM has not been realized.

3. Proposed GPU Implementation of Variational Gaussian Mixture Models

This implementation handles only diagonal components, that is, variances, instead of covariances, in order to reduce the amount of computation of matrix calculations in the same way as [23] and [8]. The main kernel implementation is described below. Let N, K, D be the number of data, the number of clusters, and the number of dimensions, respectively. In this section, the $O(x)$ means the order of computational complexity is x.

In VB-GMM, the variational-EM algorithm described in section is iterated, so once data is transferred to the GPU, there is no need to communicate with the host memory. Therefore, communication with the host is not the most worrisome aspect. This problem can be rephrased as "how to reduce the number of accesses using shared memory" and "how to achieve the coalesce-access for the minimum required access". The shared memory is a memory structure that threads can share in a block, and its access latency is smaller than that of global memory. Therefore, if a specific value is to be used repeatedly in a single kernel, it is faster to copy it from the global memory to the shared memory.

Also, in programming for CUDA-enabled GPU architectures, a critical performance consideration is coalescing accesses to global memory. The loading and storing of global memory by the warp threads are combined into as few transactions as possible by the device. Processing that can be SIMD parallelized according to the number of cores can ideally be parallelized to increase speed. However, processing that competes for access, such as computing the sum of arrays, i.e., Eq. (2.9) and (2.17) cannot be SIMD parallelized in VB-GMM.

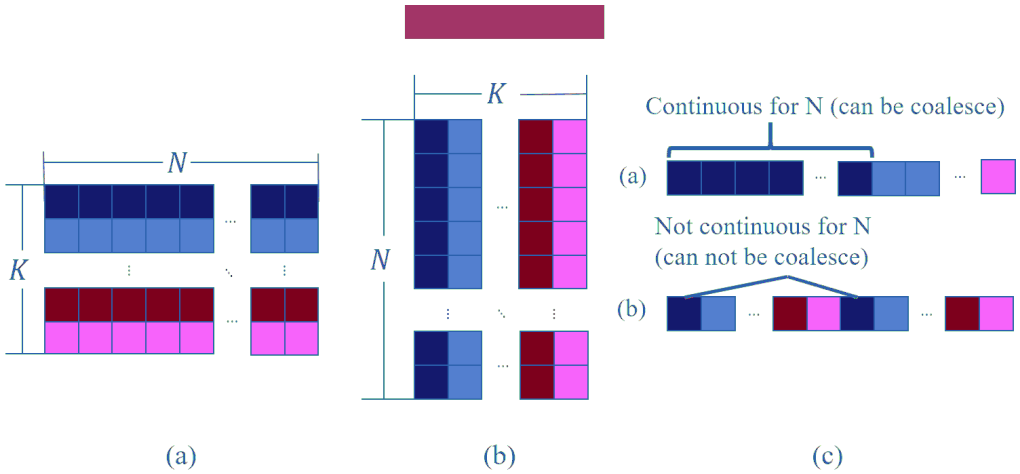


Figure 3.1.: (a) Contiguous with respect to the number of data N matrix. (b) Not contiguous with respect to the number of data N matrix. (c) The difference in physical memory layout between (a) and (b)

3.1. Optimizing Memory Allocation

In CUDA programming, paying attention to the memory layout and making it easy for coalescing access is essential. Variational inference in VB-GMM uses many arrays to perform calculations, but care must be taken with arrays related to the number of data: N.

In this case, care must be taken in calculating Eq.s (2.7, 2.9), etc. Particular attention should be paid to Eq.s (2.9) and (2.17) since they are processes of sums of elements with N of data. Therefore, in this implementation, the data is arranged so that the data size N is contiguous with the rows of the array. Figure 3.1 shows the difference in physical memory layout between contiguous arrays concerning the number of data N and those that are not, using an array size of $N \times K$.

3.2. Optimizing Number of Threads

In CUDA programming, the key to determining the number of threads per block is to maximize the coalescing to access the global memory and the memory allocation described above and select the number of threads that can effectively use the shared

memory in each block.

In this study, the optimal number of threads per block $(BlockDim_x, BlockDim_y)$ and blocks $(GridDim_x, GridDim_y)$ are set for each kernel. To enable coalescing, $BlockDim_x$ is set so that the data to be accessed is accessed at warp size in the continuous address direction, and $BlockDim_y$, $GridDim_x$, and $GridDim_y$ are set according to various parameters such as the number of data and the number of clusters. $BlockDim_y$ is set to a value close to even, which does not exceed the maximum number of threads inherent to each GPU and makes little difference in the processing time of each block, and $GridDim_x$ and $GridDim_y$ are set accordingly to the data to be calculated.

3.3. Changing Execution Order

In this implementation, E step shown in section 2 is calculated in logarithm to prevent overflow, etc. Therefore, when moving from the E step to the M step, processes such as summation are difficult in logarithms and must be exponentiated. When moving from the E step to the M step, all the values needed for the calculation are already in place. $\mathcal{L} = -\sum_{n=1}^{N}\sum_{k=1}^{K}(e^{r_{nk}} \times r_{nk})$ in Eq. (2.17) of Convergence Check shown in section 2. Therefore, the calculation of this value and the exponentiation process can be combined to improve efficiency.

3.4. Details of each kernel

The above decomposes and implements the parameter estimation of VB-GMM into 13 kernels. Tab. 3.1 shows the correspondence between the implemented kernel and the equations in the variational-EM algorithm.

1. Estimation of WEIGHT

   This kernel calculates the value of the array WEIGHT of size K according to Eq. (2.5). Since the computational cost of this kernel is not high, it is computed on the CPU.

2. Estimation of LAMBDA

Table 3.1.: Correspondence between the implemented kernel and the equations in the variational-EM algorithm

Kernel	Corresponding equation
WEIGHT	Eq. (2.5)
LAMBDA	Eq. (2.6)
GAUSS	$\frac{D}{2\beta_k} - \frac{\nu_k}{2}(x_n - \mu_k)^T W_k (x_k - \mu_k)$. in Eq. (2.7)
WLP	Eq. (2.7)
LR	Eq. (2.8)
SR	$\sum_{n=1}^{N} \sum_{k=1}^{K} (e^{r_{nk}} \times r_{nk})$. in Eq. (2.17)
NEC	Eq. (2.9)
MEC	Eq. (2.10)
CEC	Eq. (2.11)
PRI	Eq.s (2.12, 2.13, 2.14)
MEAN	Eq. (2.15)
PC	Eq. (2.16)
LB	Eq. (2.17)

Estimation of the array LAMBDA is shown in Eq. (2.6). This kernel's order of computation is also relatively small; it is straightforwardly computed in parallel since each element is independent.

3. Estimation of log Gaussian probability: GAUSS

   The purpose of the GAUSS kernel is to compute the $N \times K$ array GAUSS computed in $\frac{D}{2\beta_k} - \frac{\nu_k}{2}(x_n - \mu_k)^T W_k (x_k - \mu_k)$. This equation can be computationally transformed as shown in Eq. (3.1).

$$\begin{aligned} &\frac{D}{2\beta_k} - \frac{\nu_k}{2}(x_n - \mu_k)^T W_k (x_k - \mu_k) \\ &= -\frac{D}{2\beta_k} - \frac{\nu_k}{2}\left(\sum_{d=1}^{D} (\mu \circ \mu \circ W)_{(k,d)}\right. \\ &\qquad \left. -2x(\mu \circ W)^T + x \circ xW^T\right). \end{aligned} \tag{3.1}$$

In this kernel, $M_1 = \sum_{d=1}^{D} (\mu \circ \mu \circ W)_{(k,d)}$, $M_2 = x(\mu \circ W)^T$, and $M_3 = x \circ xW^T$ are calculated in parallel, and after each calculation, $-\frac{D}{2\beta_k} - \frac{\nu_k}{2} (M_1 - M_2 + M_3)$ is calculated. The $a \circ b$ is used in the computation of M1, M2, and M3 representing the Hadamard product, which is the computation of multiplying the elements of each matrix by each other and can be processed in SIMD. Therefore, the process of multiplying the elements is performed for each thread to speed up the process. For the matrix product, the cublasSgemm from cuBLAS, a numerical computing library of CUDA, is used. The $x \circ x$ used in the calculation of M3 is fixed when the data is given and is stored so that it does not need to be recomputed once. Finally, compute the Eq. (3.2) on the thread with row thread ID n and column thread ID k tuned for coalesce accessibility.

$$\mathrm{G}AUSS_{(n,k)} = M_{1(k)} - 2 \times M_{2(n,k)} + M_{3(n,k)}. \tag{3.2}$$

In this calculation, the computation shown in Eq. (3.2) is performed twice instead of once per block, considering the trade-off between the function call and memory access overhead. Figure 3.2 shows an overview of the calculation process for the block with the id of 1. Since $M1$ uses the same value for each block, it is stored in shared memory and then referenced.

4. Computation of weighted log probability: WLP

   Using the GAUSS, LAMBDA is calculated on the GPGPU, and the WEIGHT is calculated on the CPU and transferred to calculate the weighted log probability: WLP. This kernel is computed together with the next kernel, LR, to reduce overhead. The computation of array WLP is shown in Eq. (3.3).

$$\mathrm{W}LP_{(n,k)} = \mathrm{W}EIGHT_k + \frac{\mathrm{L}AMBDA_k}{2} + \mathrm{G}AUSS_{(n,k)}. \tag{3.3}$$

5. Estimation of log responsibility: LR

   Normalize the WLP to get array responsibility: LR in Eq. (3.4) to update parameters in the Variational M step.

$$\mathrm{L}R_k \quad = \quad \mathrm{W}LP_k - \sum_{i=1}^{K} (\mathrm{W}LP_i). \tag{3.4}$$

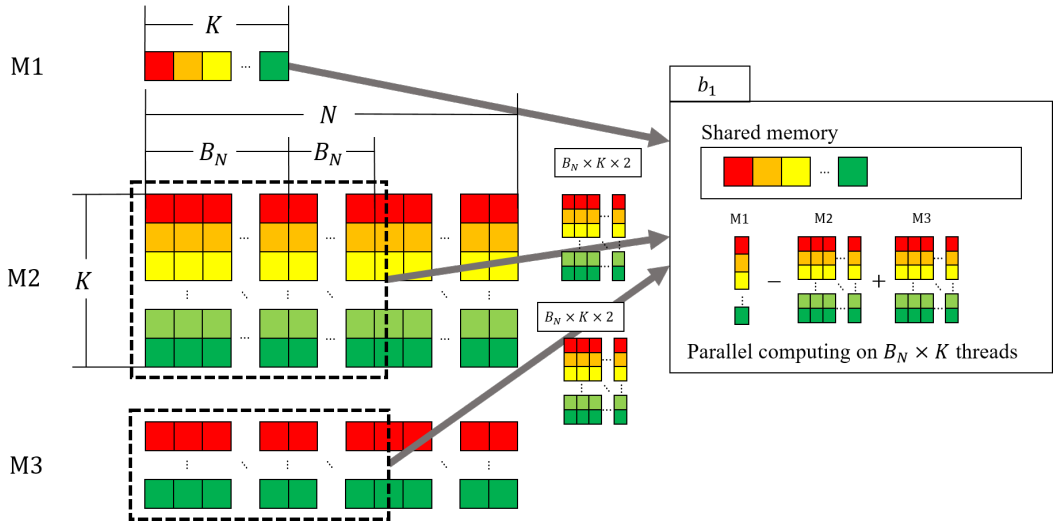


Figure 3.2.: Overview of the GAUSS kernel for the block with the id of 1. M1, M2, and M3 represent the data stored in the global memory. Block b_1 fetches the data and executes Eq. (3.2) with $B_n \times K$ threads according to the value B_n set based on the coalescing. The part indicated by "shared memory" indicates the data copied to the shared memory on the block.

Here, The method called $logsumexp$ is used, represented in Eq. (3.5) to prevent overflow and underflow.

$$\begin{aligned} & \log\left(\sum_{i=1}^{N}\exp(x_i)\right) \\ = \; & \log\left\{\exp(x_{max})\sum_{i=1}^{N}\exp(x_i - x_{max})\right\} \\ = \; & \log\left\{\sum_{i=1}^{N}\exp(x_i - x_{max}) + x_{max}\right\}. \end{aligned} \tag{3.5}$$

In this algorithm, each thread cannot complete the process independently because $\sum_{i=1}^{N}\exp(x_i - x_{max})$ is obtained after finding the maximum value in the column direction of WLP. Also, since this operation refers to the same value many times, it can be processed faster by making effective use of shared memory.

a) Compute WLP in each thread using Eq. (3.3), and store it in shared memory.

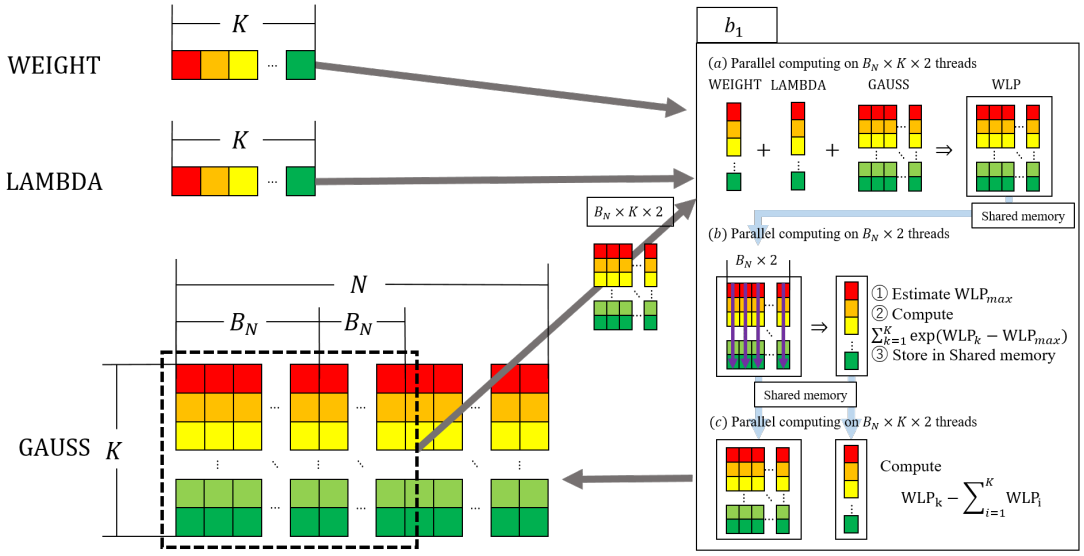


Figure 3.3.: Overview of the WLP and LR kernels for the block whose id is 1. GAUSS, WEIGHT, and LAMBDA represent the data stored in global memory. Block b_1 fetches the data according to the value B_n set based on the coalescing and executes Eq. (3.3,3.4) with $B_n \times K$ threads.

b) If the id of the threads in the column is 0, it will calculate $\sum_{k=1}^{K} \exp(WLP_k - WLP_{max})$ and store it in shared memory. Otherwise, it will wait for those threads to finish their calculations.

c) Compute LR in each thread using Eq. (3.4), and store it in global memory.

In this kernel, the computation shown above is performed twice instead of once per block, considering the trade-off between the function call and memory access overhead the same as GAUSS. Figure 3.3 shows an overview of the calculation process for the block with the id of 1.

6. Computation of Sum of Resp: SR

Calculate a value sum resp: SR, which is represented by Eq. (3.6), to calculate

the lower bound later.

$$SR = -\sum_{n=1}^{N}\sum_{k=1}^{K}(e^{LR_{nk}} \times LR_{nk}). \tag{3.6}$$

In the Variational M step, exponentiated values of LR are needed. But the calculation of lower bound needs, which is the next step of the Variational M step, needs the current LR value. Thus, SR is calculated before the Variational M step. As mentioned above, the process of finding the sum cannot be parallelized straightforwardly. This kernel was implemented as shown below, referring to Harris's method for speeding up the calculation of the sum of sequences [24].

a) Reads the value of $B_N \times 2$, the number of coalescing accessible threads, from global memory, calculates $e^{LR_{t_{id},nk}}$, and stores it in global memory for M step.

b) Calculate $e^{LR_{(t_{id},k)}} + LR_{(t_{id},k)}$ and $e^{LR_{(t_{id}+B_N,k)}} + LR_{(t_{id}+B_N,k)}$ and add them together and store it in the shared memory in each thread.

c) Set $SIZE = B_N$. A thread with id of $SIZE/2$ or less reads the value of its own t_{id} and the value of $t_{id} + (SIZE/2)$ from shared memory, adds them together and writes them to its own tid. Then, reduce $SIZE$ by half. This process is repeated until $SIZE$ becomes 32.

d) To make the addition even faster, the shfl_down_sync, a function that can refer to the same warp value without using shared memory, is used.

e) Once all the accumulated values in a block are obtained, they are stored in a temporary array for addition in global memory.

f) Further, the sum of the values in the temporary array for addition is calculated in the same way to obtain the total value.

g) This process is done for all the rows to get the exponentiated LR and SR.

Figure 3.4 shows an overview of the calculation process for the block with the id of 1.

7. Estimation of Num of Each Cluster: NEC

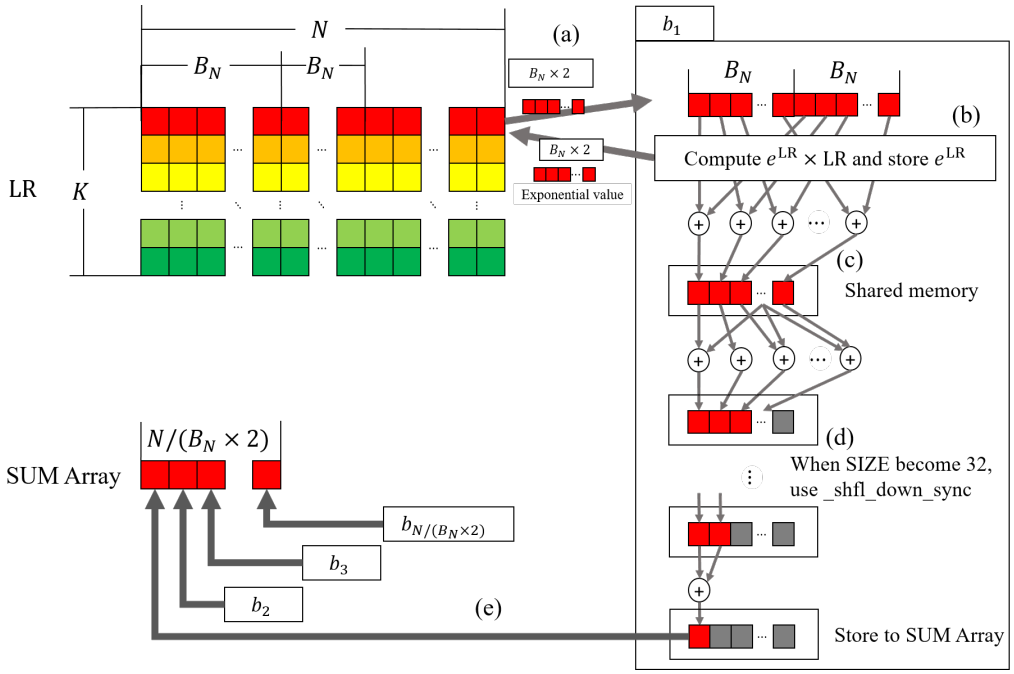


Figure 3.4.: Overview of the SR kernel for the block with the id of 1. LR and SUM Array represent the data stored in the global memory. Block b_1 fetches the data and executes Eq. (3.6) with $B_n \times K$ threads according to the value B_n set based on the coalescing. The part indicated by "shared memory" indicates the data copied to the shared memory on the block.

This kernel computes values of array NEC according to Eq. (2.9). Since this kernel is a summation process, as in SR, and cannot be parallelized in a straightforward way, The method employed to speed up the Harris array summation calculation was also used in SR.

8. Estimation of Mean of each cluster: MEC

   The computation of matrix MEC is represented by Eq. (2.10). This calculation can be replaced by using data: X and LR and NEC in Eq. (3.7).

$$\mathrm{M}EC = \mathrm{L}R^T X/(\mathrm{N}EC). \tag{3.7}$$

   In this kernel, cublasSgemm is used the same as GAUSS.

9. Estimation of Covariance of Each Cluster: CEC

   The computation of matrix CEC is represented by Eq. (2.11), and this calculation can replace Eq. (3.8) if the GMM uses diagonal covariance.

$$S = \frac{rx \circ x}{N_k} - 2\frac{\bar{x} \circ rx}{N_k} + \bar{x} \circ \bar{x}. \tag{3.8}$$

   Therefore, $\mathrm{L}R^T(X \circ X)$ and $\mathrm{L}R^T X$ is able to be computed by using cublasSgemm and compute them as Eq. (3.8).

10. Estimation of α, β, ν: PRI

    The calculation cost of these values is shown in Eq.s (2.12, 2.13, 2.14) are not high. So, their calculations are conducted on the CPU.

11. Estimation of MEAN

    The Computation of prior of mean: MEAN is represented in Eq. (2.15). MEAN is also calculated on the CPU. Because the next calculation, which is for the precision matrix, needs x_k value.

12. Estimation of Precision Cholesky: PC

    The computation of prior precisions is represented by Eq. (2.16). After computing it, substitute its square root into the PC. It also computes the logarithmic determinant of the PC for the computation of the lower bound and the next variational E-step. This kernel's order of computation is also relatively small, and since each element is independent, it is straightforwardly computed in parallel.

13. Computation of lower bound: LB

    Lastly, the computation of lower bound functioned as the indicator computed by Eq. (2.17). Sum resp and log determinant precision Cholesky have already been calculated. The calculation cost of rest values is negligible. So, Their combinations are conducted on the CPU.

3.5. Optimizing Data Transfer and CPU-GPGPU Co-operationscheme

Frequent communication between host memory and device memory is a big loss when it comes to GPU acceleration. Therefore, in this implementation, communication

between host memory and device memory is basically performed only twice, before the start of the variational-EM algorithm and after convergence. Exceptionally, data necessary for WEIGHT, PRI, MEAN, and LB calculations are sent and received at each step of the variational-EM algorithm. These are because the computation and data transfer of the kernel is relatively very small when the amount of data is large, so it can overlap with the computationally large kernel running on the GPU. Therefore, this implementation cannot support the case where memory on the device cannot be allocated.

4. Evaluations and Results

To demonstrate the superiority of the GPGPU implementation of VB-GMM, we conducted an evaluation experiment comparing the CPU implementation of VB-GMM with the GPGPU implementation of the EM algorithm. The experiment is divided into two parts. One is an evaluation using artificial data, and the other is an experiment using actual data. Here, to avoid confusion, our implementation is called VB-GPU; CPU implementation of VB-GMM is VB-CPU; and GPU implementation of EM algorithm GMM is EM-GPU. The evaluation environment is shown in Tab. 4.1.

Table 4.1.: Evaluation environment

	VB-GPU	EM-GPU	VB-CPU
Algorithm	Variational inference	EM algorithm	Variational inference
OS	CentOS Linux release 7.9.2009 (Core)		
CPU	Intel(R) Core(TM) i9-10940X CPU @ 3.30GHz		
GPGPU	GeForce RTX3090		N/A
Memory	256GB		
Compiler	NVIDIA (R) Cuda compiler driver V11.2.142		gcc version 4.8.5

4.1. Evaluation with Artificial Data

4.1.1. Kernel processing time compared to CPU

For VB-GPU and VB-CPU, for each kernel shown in Section 3, the comparison of the processing time, one time of the variational-EM algorithm to see which kernel is faster and to confirm the validity of the experimental results, is conducted. In VB-GPU, NVIDIA Visual Profiler, Chrono library [25], and CUDA Runtime API are used to get percentages. In VB-CPU, C standard time function is used. For the kernels running in parallel on the GPU and CPU in Section 3.5, the two are described together, and the kernel running on the GPU is taken as its execution time.

In this evaluation, Artificially created data with 32, 16, and 1e+06 clusters, dimensions, and data, respectively, were used, and ran the variational-EM algorithm ten times, measured the execution time of each kernel, and divided by 10 to calculate the execution time per run for each kernel. Table 4.2 shows a comparison of the breakdown of the execution time of one variational-EM algorithm for the two implementations. From Tab. 4.2, kernels account for most of the execution time in the CPU implementation has been greatly accelerated, and speedup has been achieved.

4.1.2. Kernel processing time compared to OpenCL

CUDA is one of the most useful programming models for GPGPU. However, its portability is low. Therefore, in addition to comparing the kernels of each CPU, an additional OpenCL implementation of the kernel that had a significant parallelization is created for effect in the aforementioned experiment and compared to this implementation.

Data placement and kernel structure were basically the same as in the CUDA implementation. The data used for verification was the same as that used for comparing kernel time with the CPU. In the OpenCL implementation, the matrix product is the matrix product function of clBLAS [26], the arithmetic library for OpenCL, clblasSgemm. Table 4.3 shows a comparison of CUDA and OpenCL for the breakdown of execution time for each kernel in one iteration. From Tab. 4.3, the execution speed of CUDA is fast for many major kernels is indicated. Also, all major kernels are significantly faster than the CPU, even in implementations using OpenCL.

Table 4.2.: Comparison of CUDA and CPU for the breakdown of execution time for each kernel in one iteration

	CPU [ms] (t1)	GPU [ms] (t2)	Speed-Up t1/t2
WEIGHT	0.0059	2.1938	328.39
GAUSS	720.4222		
LAMBDA	0.0381	0.0278	1.37
WLP+LR	982.9521	1.3113	749.60
SR	773.6876	7.8857	98.11
NEC	58.7554	7.4401	7.90
MEC	108.8799	0.4742	229.59
PRI	0.0016		
CEC	128.9425	0.9323	138.31
MEAN	0.0055		
PC	0.0151	0.0213	0.71
LB	0.1013	0.0931	1.09

4.1.3. Comparative experiments with varying parameters of the dataset

These evaluations use data sampled by Gaussian mixture models whose parameters were set intentionally. This experiment consists of three experiments, each of which changes the number of data N, the number of clusters to sample K, and the number of dimensions D. The main evaluation items are execution time to evaluate execution speed and the Davis-Bouldin Score (DB Score) to evaluate the quality of clustering. All evaluation items and their contents are shown in Tab. 4.4.

Varying number of data

The number of dimensions and the number of clusters is fixed at 16 and 32, respectively. The number of data is varied from 1e+03 to 1e+07. The experimental results are shown in Tab. 4.5, and the extracted time and DB score are shown in Fig. 4.1.

Figure 4.1 indicates that the execution time of the CPU implementation increases as the number of data increases, while the execution speed of the two GPU implemen-

Table 4.3.: Comparison of CUDA and OpenCL for the breakdown of execution time for each kernel in one iteration

	OpenCL [ms] (t1)	CUDA [ms] (t2)	Speed-Up t1/t2
GAUSS	18.44	2.19	8.78
WLP+LR	0.54	1.31	0.41
SR	36.98	7.88	4.69
NEC	36.23	7.44	4.87
MEC	4.14	0.4742	8.73
CEC	5.79	0.9323	6.22

Table 4.4.: Evaluation items and their summary

Evaluation item	Summary
Time	The time it took for each implementation to converge. Unit: millisecond
DB score	Davis-Bouldin Score: DB score [27] signifies the average similarity between clusters, where the similarity is a measure that compares the distance between clusters with the size of the clusters themselves. Zero is the lowest possible score. Values closer to zero indicate a better partition. DB indicators for clustering results were calculated using scikit-learn's metrics library. [28].
Conv. Cluster.	The number of clusters at the time of convergence; one cluster was counted when the GMM weight exceeded the threshold.
Log Likelihood	Goodness of fit to the data. If it is excessively high, there is a high possibility of overtraining.
Conv. Iter.	The number of iterations each algorithm took to converge.

tations, VB-GPU and EM-GPU, remains flat until the number of data exceeds 1e+06. It can also be seen that VB-CPU and VB-GPU using variational inference have better DB scores than implementations using the EM algorithm. Also, Table 4.5 shows that compared to the implementation using EM-algorithm, both implementations using variational inference are closer to the true number of clusters of 32 at convergence. This means that the VB-GPU property of being able to find the number of clusters, introduced in Section 1.1, is reflected. When the number of data was 1e+07, VB-GPU successfully achieved 192 times speed-up compared to VB-CPU.

From Fig. 4.1 and Tab. 4.5, we can see that VB-GPU and VB-CPU use the same computation algorithm, and the results are basically the same, but when the data size

Table 4.5.: Result of evaluation with varying the number of data

Number of Data	Time[ms]			DB Score			Conv. Cluster.			Log Likelihood			Conv. Iter.		
	VG	VC	EG	VG	VC	EG	VG	VC	EG	VG	VC	EG	VG	VC	EG
1e+03	545	31	534	0.39	0.39	0.77	30	30	63	6.9	6.9	57.3	8	6	17
1e+04	554	263	596	0.26	0.26	0.98	31	31	59	21.5	23.9	47.2	8	7	28
1e+05	597	3189	553	0.25	0.25	0.53	31	31	62	36.3	39.0	53.7	8	7	6
1e+06	885	40264	1244	0.56	0.56	1.19	31	31	64	41.7	46.8	55.3	9	8	29
1e+07	5037	970074	3436	0.90	1.33	3.08	30	31	61	42.3	42.3	49.9	32	31	6

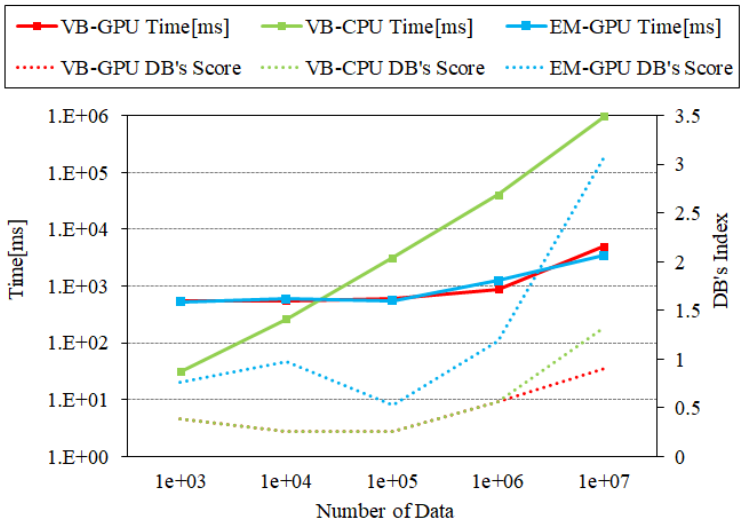


Figure 4.1.: Execution time and DB Score depend on the number of data.
(The number of clusters=32, The number of dimensions=16)

is 1e+07, there is a big difference between the two implementations. This is due to missing information when calculating the product of matrices in the GAUSS, MEC, and CEC kernels and when calculating the sum in the SR and NEC kernels. In particular, the SR and NEC kernels are prone to missing information because the GPU repeats the process of adding two values, which tends to cause differences in the absolute values of the values, while the CPU implements the process of adding the values in order from the front.

Table 4.6.: Result of evaluation with varying the number of clusters

Number of Cluster	Time[ms]			DB Score			Conv. Cluster.			Log Likelihood			Conv. Iter.		
	VG	VC	EG	VG	VC	EG	VG	VC	EG	VG	VC	EG	VG	VC	EG
4	699	2383	725	0.82	0.82	0.82	8	8	8	4.7	12.9	34.6	3	2	4
8	795	16105	757	0.41	0.41	0.43	7	7	16	3.5	33.8	44.4	17	18	6
16	815	23817	799	0.45	0.45	0.45	15	19	29	43.8	34.1	51.8	13	10	8
32	865	40312	1192	0.56	0.56	1.05	31	31	63	41.7	46.8	55.2	9	8	26
64	1099	85645	983	0.40	0.40	0.69	58	58	115	40.6	40.6	52.8	10	9	8
128	1333	143286	1264	0.80	0.80	0.73	107	107	219	37.6	37.6	50.7	8	7	9

Varying number of cluster

The number of dimensions and the number of data was fixed at 16 and 1e+06, respectively. The number of clusters varied from 4 to 128. The experimental results were shown in Tab. 4.6, and the extracted time and DB score are shown in Fig. 4.2.

Figure 4.2 indicates that the execution time of the CPU implementation increases as the number of clusters increases, while the execution speed of the two GPU implementations, VB-GPU and EM-GPU. It can also be seen that VB-CPU and VB-GPU using variational inference have better DB scores than implementations using the EM algorithm. Also, Table 4.6 shows that compared to the implementation using EM-algorithm, both implementations using variational inference are closer to the true number of clusters at convergence. This shows that the number of clusters, a property of variational inference, can be obtained in the same way as in Section 4.1.3. When the cluster number was 128, VB-GPU successfully executed and achieved 107 times speed-up compared to VB-CPU.

Varying number of dimensions

The number of clusters and the number of data fixed to 32 and 1e+06. The number of dimensions is varied from 4 to 128.

The experimental results are shown in Tab. 4.7, and the extracted time and DB score are shown in Fig. 4.3. Figure 4.3 and Tab. 4.7 show that VB-GPU has the same clustering capability as VB-CPU, regardless of the number of data, and can run at the same speed as EM-GPU. It can also be seen that the Gaussian distributions that make up the GMMs obtained by inference are close to the correct values when implemented using variational inference. When the number of dimensions was 128,

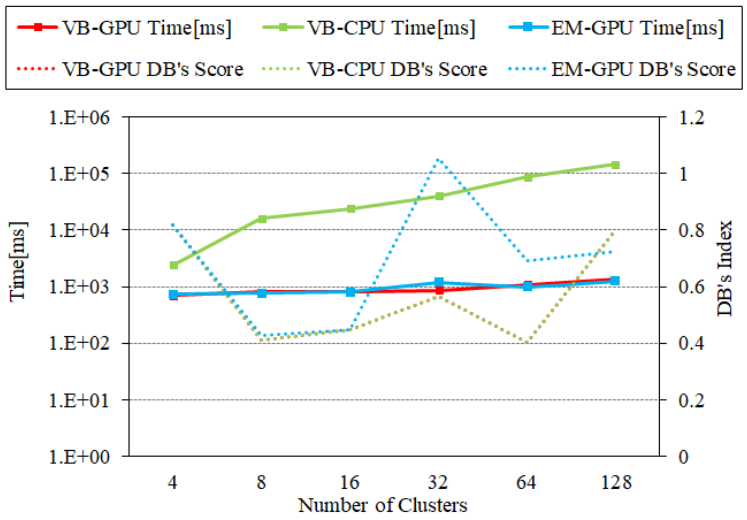


Figure 4.2.: Execution time and DB Score depending on the number of clusters. (The number of data=1e+06, The number of dimensions=32)

Table 4.7.: Result of evaluation with varying the number of dimensions

Number of dimensions	Time[ms]			DB Score			Conv. Cluster.			Log Likelihood			Conv. Iter.		
	VG	VC	EG	VG	VC	EG	VG	VC	EG	VG	VC	EG	VG	VC	EG
4	906	60639	823	0.36	0.36	0.25	29	29	45	8.1	8.1	10.2	19	19	15
8	860	44629	780	0.49	0.49	0.49	30	30	61	19.7	20.0	24.3	14	13	10
16	874	40190	1146	0.56	0.56	1.03	31	31	64	41.7	46.8	55.3	9	8	24
32	1070	57669	1062	0.46	0.46	0.77	29	29	56	83.9	87.2	105.0	7	6	8
64	1481	92933	1420	0.52	0.52	1.92	31	31	54	177.1	188.3	207.9	6	5	5
128	2333	163156	2822	0.95	0.95	3.18	23	23	56	260.4	260.3	355.1	5	4	31

VB-GPU successfully achieved 69 times speed-up compared to VB-CPU.

4.1.4. Comparison with FPGA implementation

This section shows the comparing evaluation with the FPGA model of EMGMM [23]. In this section, the FPGA implementation of expectation–maximization algorithm is referred to as EM-FPGA, and the GPU implementation of the proposed implementation of variational bayesian Gaussian mixture models is referred to as VB-GPU. Since the number of Data that can be processed Per Second: DPS is used as the evaluation value in the previous EM-FPGA study, the number of data that can be processed per second is

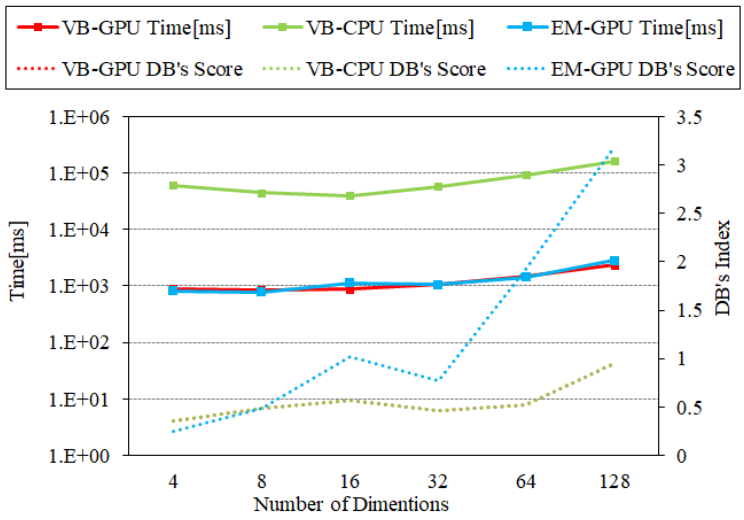


Figure 4.3.: Execution time and DB Score depending on the number of dimensions. (The number of data=1e+06, The number of clusters=32)

calculated and compared in this implementation as well. Equation 4.1 calculates DPS values of VB-GPU. In Equ. 4.1, N, L, T represent the number of data, the number of loops executed, and the execution time, respectively.

$$DPS = \frac{N \times L}{T} \tag{4.1}$$

DPS values for EM-FPGA were those described in previous research paper [23]. The evaluation data were sampled from GMMs following the number of dimensions D and the number of clusters K described in the paper [23], and the number of data N was evaluated at 10^7 in addition to 10^6 described in the previous research paper.

Table 4.8 shows Comparison results with FPGA implementation of GMM by EM algorithm. In Tab. 4.8, N, D, K represents the number of data, the dimensions of data, and the number of setting cluster size, respectively. DPS and SU represent the number of Data processed Per Second, Amount of performance improvement compared to FPGA implementation, respectively.

Table 4.8 shows that when the number of data is 10^6, the performance of the proposed method, VB-GPU, is lower than the EM-FPGA implementation in the previous study when the number of dimensions and clusters is small, but when the number of dimensions and clusters is large, the performance is more than four times higher. The

Table 4.8.: Comparison results with FPGA implementation of GMM by EM algorithm

		EM-FPGA [23]	VB-GPU			
N		10^6	10^6		10^7	
D	K	DPS	DPS	SU	DPS	SU
6	2	2.48E+08	1.11E+08	0.45x	4.71E+08	1.90x
6	4	2.48E+08	7.92E+07	0.32x	3.81E+08	1.54x
6	6	2.48E+08	6.45E+07	0.26x	3.10E+08	1.25x
16	8	9.15E+07	5.22E+07	0.57x	2.57E+08	2.81x
16	16	4.57E+07	3.27E+07	0.72x	1.68E+08	3.67x
16	32	2.28E+07	1.84E+07	0.81x	9.24E+07	4.05x
96	64	1.90E+06	8.76E+06	4.61x	3.99E+07	21.02x
96	128	9.50E+05	4.52E+06	4.76x	2.01E+07	21.16x
96	256	4.74E+05	2.33E+06	4.91x	1.02E+07	21.53x

number of data that can be processed per second is 4.91 times higher when the number of dimensions and the number of clusters are 96 and 256, respectively.

Also, in the case of the number of data is 10^6, the proposed method, VB-GPU, achieves better DPS than the previous study, EM-FPGA, for all the number of dimensions and clusters evaluated, and when the number of dimensions is 96 and clusters is 256, the number of data that can be processed per second is 21.53 times higher.

4.2. Evaluation with Practical Data

These evaluations use open data. Table 4.9 shows an outline of the data. Evaluation items took time to converge, and DB Score was the same as Section 4.1.3. For all datasets used in this evaluation, dimensionality reduction was performed by principal component analysis with whitening applied using scikit-learn as a preprocessing step, and dimensionality reduction was performed to the smallest number of dimensions with an explanatory variance ratio greater than 0.8. Alignment-accuracy was calculated as in the [29] experiment, using a majority vote of members to align each cluster to a single label and using that class.

Table 4.9.: Outline of Practical Data

N : Number of data, K : Number of classes, D : Number of dimensions

Name	N	K	D Original → After PCA	Overview
MNIST [9]	60000	10	64 → 39	28 x 28 Handwritten digits.
CIFAR10 [10]	60000	10	64 → 26	32 x 32 color images in 10 classes.
PAMAP2 [11]	376417	13	52 → 6	Physical Activity Monitoring dataset.
Gas sensors for home activity monitoring Data Set: GDS [12]	919438	3	11 → 2	Recordings of a gas sensor array composed of 8 MOX gas sensors and a temperature and humidity sensor.

1. MNIST

   The MNIST database of handwritten digits has a training set of 60,000 examples and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. Table 4.10 shows that the proposed method, VB-GPU, achieves better DB Scores than EM-GPU when the number of dimensions is 32 or more. Also, it can be confirmed that the number of clusters at convergence is reduced when the dimensionality is greater than the default number of clusters, which is 64 or more. In addition, VB-GPU achieved good Alignment-accuracies for any initial number of clusters.

Table 4.10.: Evaluation result with practical data sets

		Time[sec]		Time ratio	DB Score		Conv. Cluster		Log Likelihood		Alignment Accuracy	
Data Set	Init Cluster	VG	EG	VG/EG	VG	EG	VG	EG	VG	EG	VG	EG
MNIST	16	0.81	0.66	1.22	3.62	3.25	16	16	54.23	111.98	0.48	0.14
	32	1.25	0.83	1.50	3.39	6.75	32	32	56.66	121.00	0.63	0.22
	64	2.24	0.91	2.45	3.10	6.64	52	64	58.08	120.00	0.70	0.18
	128	2.88	1.71	1.68	2.94	6.35	63	128	59.07	127.26	0.72	0.20
CIFAR10	16	1.04	0.98	1.06	5.31	4.35	16	16	-36.07	-31.73	0.23	0.20
	32	1.27	1.12	1.14	4.81	4.06	28	32	-35.81	-30.95	0.26	0.24
	64	1.82	1.30	1.40	4.75	3.80	38	64	-35.60	-30.06	0.28	0.27
	128	3.19	1.66	1.93	4.15	3.25	41	128	-35.55	-29.27	0.28	0.30
PAMAP2	64	2.14	1.83	1.17	1.70	1.56	63	64	-5.06	63.58	0.69	0.67
	128	3.26	2.80	1.16	1.75	1.60	127	128	-4.22	-0.78	0.71	0.70
	256	6.78	4.17	1.63	1.65	1.56	253	256	-3.36	0.50	0.76	0.73
	512	10.09	8.11	1.25	1.57	1.56	466	512	-2.85	1.51	0.79	0.77
	1024	10.87	15.49	0.70	1.56	1.58	769	1024	-2.64	2.60	0.81	0.80
GDS	64	2.44	2.44	1.00	1.99	1.12	64	64	-1.30	0.42	0.50	0.47
	128	4.73	3.71	1.27	2.63	1.11	128	128	-1.18	0.52	0.52	0.48
	256	3.53	8.43	0.41	2.79	1.23	256	256	-1.27	0.97	0.51	0.48
	512	10.75	N/A	N/A	1.93	N/A	512	N/A	-1.22	N/A	0.49	N/A
	1024	22.33	31.45	0.71	2.30	1.13	1019	1024	-1.22	1.53	0.50	0.53

2. CIFAR10

   The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes. Value-based clustering of the pixels is applied to confirm if the ten classes could be clustered. As shown in Tab. 4.10, VB-GPU successfully reduced the number of clusters at convergence when the initial number of clusters was 32 or more in this experiment. However, in all cases, the DB Score was found to be below that of EM-GPU. Alignment-accuracies of VB-GPU and EM-GPU were comparable at any initial number of clusters.

3. PAMAP2

   The PAMAP2 (Physical Activity Monitoring) dataset contains data on 18 different physical activities (such as walking, cycling, playing soccer, etc.), performed by nine subjects wearing three inertial measurement units and a heart rate monitor. Table 4.10 shows that VB-GPU reduces the number of converging clusters when the number of set clusters is greater than 512, and the number of converging clusters is 769 when the initial number of clusters is 1024. When the initial number of clusters was 512 and 1024, the DB Score was the same as that of EM-GPU, despite the reduction in the number of clusters. Also, when the

initial number of clusters was 1024, the execution speed was 1.42 times faster. Alignment-accuracies of VB-GPU and EM-GPU were comparable at any initial number of clusters.

4. Gas sensor for home activity monitoring Data Set
   This dataset has recordings of a gas sensor array composed of eight MOX gas sensors and a temperature and humidity sensor. This sensor array was exposed to background home activity while subject to two different stimuli: wine and banana. The responses to banana and wine stimuli were recorded by placing the stimulus close to the sensors. We tried value-based clustering of the sensors to see if the three states banana, wine, and background could be clustered. Table 4.9 shows that in this experiment, VB-GPU did not achieve dimensionality reduction at any initial number of clusters, and DB Score was also high. Alignment-accuracies of VB-GPU and EM-GPU were comparable at any initial number of clusters.

5. Discussion

Firstly, we discuss the difference between VB-GPU and VB-CPU. Section 4.1.1 shows that using the GPU, we can see that the bottleneck in the CPU implementation has been greatly improved. These CPU bottlenecks include N in the computational order, indicating that GPUs are able to handle large amounts of data. This can also be able to confirm in evaluation with varying the number of data where we can see that the calculation time for VB-CPU increases as the amount of data increases, while for VB-GPU, it remains flat until the amount of data exceeds a certain level, and only when the amount of data exceeds 1e+07 does the calculation time increase. In addition, in evaluation with varying the number of clusters, we confirmed that VB-CPU increases the computation time as the number of clusters increases, but VB-GPU can suppress it. In the experiments with varying the number of dimensions, execution time did not correlate well with increasing dimensionality, and there was no significant difference between VB-GPU and VB-CPU. This may be due to the fact that some kernels other than GAUSS, which is the bottleneck, are not correlated with D.

Secondly, we will discuss the comparison with OpenCL. We will show that our CUDA implementation is faster than a similar implementation in OpenCL and more effective than using OpenCL in environments where CUDA is available. Also, all major kernels are significantly faster than the CPU, even in implementations using OpenCL. This shows that our implementation is effective even in environments where CUDA is not available.

Thirdly, the discussion of comparison with FPGA implementation is based on the result of Section 4.1.4. In this experiment, the proposed method is inferior to the FPGA implementation of the EM-algorithm, a previous study, in terms of the number of data processed per second, when the same number of data as in the previous study is used, when the number of dimensions and clusters is small. However, when the number of dimensions and clusters is large, the performance is about 4.7 times higher

than that of the previous study. In the runs with ten times the number of data used in the evaluation in the previous study, the proposed method was superior in all the validations measured, regardless of the number of dimensions and clusters, and achieved 21.5 times the performance when the number of dimensions and clusters were 96 and 256, respectively. This is thought to be because GPUs have high parallelism compared to FPGAs, and the larger the data to be processed at one time, the more advantageous they are. This suggests that the proposed method has an advantage over previous work using FPGAs in terms of processing large data.

Finally, we discuss the difference between VB-GPU and EM-GPU. We can see that in all experiments using artificial data in Section 4.1.3. VB-GPU achieves the same execution time as EM-GPU, and VB-GPU achieves a better DB Score than EM-GPU. Notably, in all experiments with arbitrary artificial data, the number of clusters at convergence is closer to the true number of clusters in the data than EM-GPU. This is a very important property in data analysis because it gives us an idea of what kind of clusters the data has. It also achieves a smaller Log likelihood than EM-GPU while achieving a good DB Score, indicating that it is able to perform good clustering while avoiding over-fitting the data.

Next, the discussion in the comparison of the proposed method, VB-GPU, and the previous study, EM-GPU, using practical data is presented. In experiments with the MNIST dataset of handwritten character images, the VB-GPU achieved better DB Scores than the EM-GPU at the number of initial clusters is larger than 16. It also confirmed that when the dimensionality is large, the number of clusters at convergence is successfully reduced. This indicates that the number of clusters in the VB-GMM has been optimized, as described in Section 2.3.

Despite the small number of clusters at convergence, the DB Score, which indicates the similarity between clusters, achieved good values, meaning that good clustering results were obtained. Alignment Accuracy is also higher than that of EM-GPU, indicating that the clustering is better than that of EM-GPU in the context of classifications that also follow the labels of the original data. In terms of execution speed, EM-GPU was faster in all cases.

In the clustering of CIFAR10, which is image data, similarly, when the initial number of clusters is large, VB-GPU shows that it is able to optimize the number of clusters. However, the DB Score was greater than that of EM-GPU. This indicates a high

degree of similarity between the classes at convergence, indicating that the clustering results overly grouped the classes together. Alignment Accuracy showed comparable performance in all cases. In terms of execution speed, EM-GPU was faster in all cases.

No significant difference in DB Score was found between the two implementations in the clustering of the dataset PAMAP2, which links the values of sensors attached to the body with Physical Activity. However, in this experiment, there were several cases where VB-GPU was able to optimize the number of converging clusters, converging to 466 clusters when the initial number of clusters was 512 and to 769 clusters when the initial number of clusters was 1024. The convergence to a smaller number of clusters while maintaining a similar DB Score indicates that better clustering is achieved compared to EM-GPU. In terms of execution time, it was faster than EM-GPU when the initial number of clusters was 1024.

In the GDS, a dataset that ties periodic sensor values to nearby objects, VB-GPU was unable to optimize the number of clusters at any initial number of clusters. The DB Score was also lower than that of the EM-GPU. This data set has 3 original classes and 2 dimensions after PCA, and the differences in information between classes are very small. It is thought that the optimization of the number of converging clusters did not occur as a result of attempting to separate the data by this small difference. In terms of execution time, the cluster was faster than EM-GPU when the initial number of clusters was 256 and 1024. The trend suggests that EM-GPU is also faster when the number of clusters is 512, which EM-GPU did not converge. As a summary of the comparison using practical data, VB-GPU has the advantage of optimizing the number of converging clusters when the number of initial clusters is large, and converges faster than EM-GPUs when the number of initial clusters is large in clustering with relatively large data such as PAMAP2 and GDS. In some cases, the clustering converged faster than EM-GPUs when the initial number of clusters was large. From this, VB-GPU is more effective than EM-GPU in clustering data where the actual number of clusters is unknown and the number of data is relatively large.

Part II.

Resampling High Efficient Hardware for Sequential Monte Carlo

6. Introduction

6.1. Motivation

The Bayesian method is one of the most popular methods for data analysis because it can express uncertainty in parameter estimates and analyze data with complex structures in a flexible model [30, 31]. Bayesian theory is expressed by Equ. 6.1.

$$p(\theta|y) = \frac{p(y|\theta)p(\theta)}{p(y)} \tag{6.1}$$

Where y represents the observed data, θ represents the unobserved parameters of the model, $p(\theta)$ is the prior probability of the parameter θ, and $p(y|\theta)$ is the likelihood, or likelihood, of y given θ, and the $p(y)$ is called as normalized constant or marginal likelihood. Unlike neural networks, the entire model can be explained by the data and the prior probabilities behind it, as shown in Equ. 6.1, so it is also attracting attention as a machine learning method with a high explanation.

However, the computation involved in general Bayesian tasks such as estimation, prediction, and model comparison using Bayesian methods is concentrated on integration to compute $p(y)$, and the dimensionality is too large for data analysis such as the huge amount of data that has become available due to recent improvements in data storage and communication technology. Therefore, most of the interest in Bayesian methods has focused on better approximate methods of inference in the form of Monte Carlo estimation and variational approximation.

Sampling methods using Markov Chain Monte Carlo (MCMC) are mainly used for sampling Bayesian inference problems because they can sample from the posterior distribution in Bayesian modeling, regardless of dimension or complexity. However, MCMC is too computationally intensive for practical use [32], and various improvements have been made for use on large data sets.

Sequential Monte Carlo (SMC) is an improved MCMC method characterized by the process of resampling, in which weights are calculated for each sampled value, and the next sampling is performed according to the weights [33].

SMC is suitable for parallelization due to its structure, and various parallelization implementations using multi-core CPUs, GPUs, FPGAs, etc., are being attempted to increase speed. Among them, resampling, which is a feature of SMC, is known to be a bottleneck because it is computationally expensive. Therefore, most SMC speed-up and efficiency improvement research focus on how to make resampling more efficient.

6.2. Challenges and Contribution

In this study, the Optimization of the dedicated hardware for the resampling step, which is one of the three steps in the sequential Monte Carlo method: sampling step, importance calculation step, and resampling step, and whose processing time increases in proportion to the number of particles. The specific optimization method is to optimize the metropolis resampling algorithm used in the resampling step, which is suitable for parallelization, from the currently used floating-point method to the integer-optimized method.

The proposed optimization of Metropolis Resampling to run on integers achieves the elimination of conversion and multiplication to the floating point in the random number generation part within Metropolis Resampling and also makes it hardware efficient by eliminating calculations in the floating-point in the Metropolis test. In the evaluation experiments, the proposed method was evaluated for 8-bit, 16-bit, and 32-bit integers, considering hardware efficiency. In the evaluation of several algorithms on CPUs, the proposed integer-optimized Metropolis resampling achieved resampling quality equivalent to that of single-precision floating-point methods for 8-bit integers, 16-bit integers, and 32-bit integers as resampling alone. In addition, the Sequential Monte Carlo Sampler, an algorithm using SMC, achieved the same performance as the floating-point method when the number of particles is large.

In the evaluation of the hardware implementation, the proposed integer-optimized Metropolis resampling reduced resource usage for all data width implementations compared to previous studies using the single-precision floating-point. It achieved up to 3.0 times LUT usage improvement in critical modules such as coefficient generation and

Metropolis test execution improvements in key modules such as coefficient generation and Metropolis test execution. LUT utilization reductions of 31% at 32 bits, 57% at 16 bits, and 64% at 8 bits were achieved in key modules such as coefficient generation and Metropolis test execution, while other bottlenecks achieved up to 3.0 times throughput improvement and up to 75% memory utilization reduction.

6.3. Composition of Part II

The rest of this paper is organized as follows. Section 7 provides an introduction to SMC and introduces previous work. Section 8 presents the proposed integer-optimized Metropolis Resampling algorithm and its application to hardware. Section 9 presents some evaluation of the proposed algorithm for conforming validation of this algorithm. The extended discussion is shown in Section 10. The conclusion is presented in Part III, along with those from Part I.

7. Background Theory and Related Works

7.1. Overview of Sequential Monte Carlo: SMC

Sequential Monte Carlo: SMC [33–35], is a class of algorithms for estimating the posterior distribution of a state in a dynamic Bayesian model. This algorithm is a type of approximate Bayesian method whose basic concept was presented in 1996 as a Monte Carlo filter [34] or bootstrap filter [33], an iterative method that approximates the posterior distribution using a set of weighted samples called particles.

The algorithm uses these particles to estimate posterior distribution in the procedure described below.

1. Sampling:
   The algorithm generates a set of particles by sampling from the posterior distribution at the previous time step. The particles represent different possible states of the system at the current time step and are used to approximate the posterior distribution.

2. Importance Computation:
   The algorithm assigns a weight to each particle based on its likelihood, given the current observations. The weight reflects the relative importance of the particle in approximating the posterior distribution.

3. Resampling:
   The algorithm selects a new set of particles from the current set of particles based on their weights. The resampling step is used to ensure that the new set of particles more accurately represents the posterior distribution. It also helps to

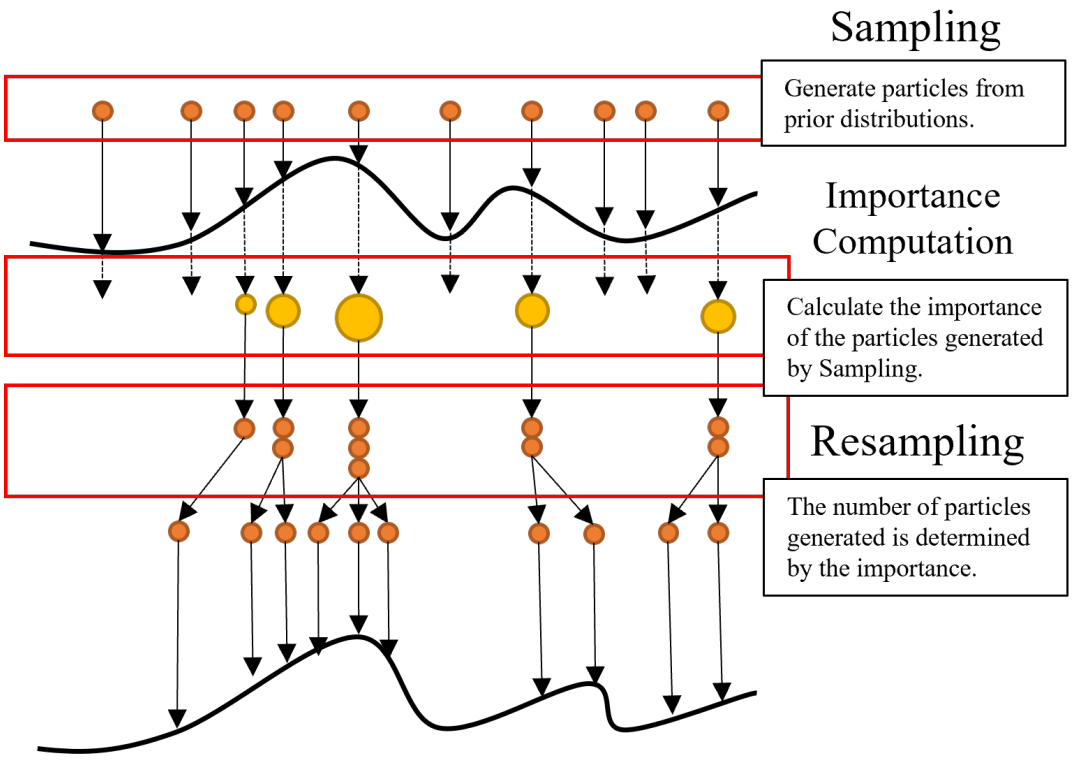


Figure 7.1.: The conceptual diagram of the SMC.
SMC converges to distribution in line with the data by repeating the three steps of Sampling, Importance Computation, and Resampling.

prevent the loss of diversity in the particle set, which can occur if a small number of particles dominate the distribution.

After the resampling step, the algorithm returns to the sampling step and begins the next iteration. The process is repeated until the desired level of accuracy is achieved or the predetermined number of iterations have been completed. The conceptual diagram of the SMC is shown in Fig. 7.1.

One of the main advantages of SMC is that it can handle high-dimensional state spaces and complex non-linear models [33, 34]. Therefore, as the next section will show, SMC is used in a wide range of fields, including many-object tracking, physics, financial economics, and statistics, where nonlinearities and non-Gaussianities need to be modeled.

7.2. Application of Sequential Monte Carlo

SMC is a beneficial technique in signal processing due to its robustness since there is much noise in the input. Early studies have shown that Djuric et al. [36] took advantage of SMC's sequential importance-oriented nature in signal processing problems, which are generally highly uncertain, and applied it to parameter estimation. For example, there are many applications for time-period analysis as a fundamental task in signal processing. [37–39]. In the equally essential task of radar analysis, Boers et al. [40] showed that using SMC for the likelihood ratio, a signal detection method, is effective for radar detection of stealth and dim objects. Decoding sparse signals is another essential task. By using SMC-based models in the sparse signal decoding task, Yoo et al. [41] achieved a lower error rate than existing methods. Take the audio signal, for example; SMC has been used in many studies of speaker tracking [42–44] using speech signals, which is one of the most typical signal-processing tasks. In a recent study, Liu et al. [45] designed a two-layer particle filter to overcome the drawback of classical particle filter-based methods: poor performance when measurements are disturbed by noise. Also, Wang et al. [46] proposed a distributed review speaker estimation method using an unscented particle filter [47], a type of Particle Filter, and data correlation in noisy and reverberant environments, which has been difficult in the past, and showed that multiple speakers could be distributed and tracked even in reverberant and noisy environments.

In addition to basic signal processing, applied areas include motion tracking and simultaneous localization and mapping (SLAM).

In motion-tracking, While MCMC was initially the mainstay of motion-tracking applications, Blake et al. [48] introduced SMC, and various studies have since been conducted on the use of SMC for motion tracking [49–53]. Recently, Zhang et al. [54] combined existing convolution neural network: CNN-based models with SMC to make them more robust than models using only CNNs [55].

SLAM is a technique used by robots and autonomous vehicles to build a map of an unknown environment while simultaneously determining their own location within that environment [56]. It involves the integration of sensory data, such as that obtained from lasers, cameras, or inertial measurement units, with algorithms that estimate the robot's position and orientation relative to the environment. Many algorithms of mobile robot SLAM (Simultaneous Localization and Mapping) have been researched at present,

however, the SLAM algorithm of mobile robots based on probability is often used in the unknown environment [57].

The application of SMC to SLAM has been studied for a long time, and Fox et al. [58] published their adaptation in 2001, surpassing the state-of-the-art method at the time [59]. Subsequently, it was extended to the Rao-Blackwellised Particle Filter [60,61], and it is still a typical application of SMC today. In SLAM, as in motion tracking, methods combining deep neural networks have been attracting attention in recent years. Karkus et al. [62] have achieved higher accuracy than existing methods with a model combining SMC and DNN.

Many studies have been done as an alternative to Markov chain Monte Carlo, Chopin [63] and Moral et al. [64] sampled the posterior distribution using SMC and showed that it could be applied to a wide range of models. Moral [65] followed suit, presenting a simple unifying framework that allows for extending both the SMC methodology and its applicability to a wider range of models. The combined variational inference method of Naesseth [66] et al. and Sequential Monte Carlo was used in the state-space model. They demonstrated its usefulness in stochastic variation models of fusion data and deep Markov models of cranial neural circuits. Hadian et al. [67] proposed a hybrid model that combines a multi-objective particle swarm optimization algorithm and sequential Monte Carlo simulations to find the optimal placement of electric vehicle charging stations. Chen et al. [68] improved the particle filter algorithm and developed a new particle swarm optimization particle filter algorithm with a mutation operator that avoids local solutions. By applying this algorithm to noise reduction, they achieved lower error rates and faster execution than previous studies.

Researchers have also been conducted in recent years to try to overcome the weaknesses of SMC. SMC had a problem with high-dimensional reasoning. However, Naesseth et al. [69]used an improved version of Sequential Monte Carlo to achieve better results than state-of-the-art methods for three tasks: Gaussian Model, Soil Carbon Cycles, and Mixture Model, even for high-dimensional problems, which were originally considered weak. Dai et al. [70] discussed sequential Monte Carlo samplers and their possible implementations, noting that despite their potential advantages, such as the ability to perform sequential inference and take advantage of parallel processing resources, their software is not as extensive as MCMC and is still underutilized in the statistics field.

7.3. Problems of Sequential Monte Carlo

There are several potential limitations and challenges associated with using sequential Monte Carlo (SMC) algorithms:

- Computational complexity:
  SMC algorithms can be computationally intensive, particularly for high-dimensional state spaces or complex models. The number of particles required to achieve a desired level of accuracy increases with the dimensionality of the state space [71], which can lead to longer run times and greater computational resources.
- Weight degeneracy:
  The particles' weight can degenerate, meaning that a small number of particles dominate the distribution, while the rest have very low weights [72, 73]. This can lead to a loss of diversity in the particle set and hinder the algorithm's performance.
- Model misspecification:
  If the model used in the SMC algorithm is misspecified or lacks sufficient flexibility, the algorithm may not be able to accurately estimate the posterior distribution [74].

Overall, while SMC algorithms can be effective in many situations, it is important to carefully consider their limitations and potential challenges when applying them to real-world problems. These limitations frequently occur when the number of particles is small, but increasing the number of particles increases the computational load, so parallelization and other methods have been used to speed up the process.

Gustaf et al. [75] evaluated each step in the GPU implementation of SMC and found that the resampling step accounts for most of it, as shown in Fig. 7.2. This means that the bottleneck of SMC in parallel architectures is Resampling, and in order to make SMC even faster, Resampling needs to be accelerated.

7.4. Related Works on Resamping and Its Speeding Up

As indicated in Section 7.4, the resampling step is a bottleneck in SMC, and many studies have been on speeding up this step. Resampling is the process of generating

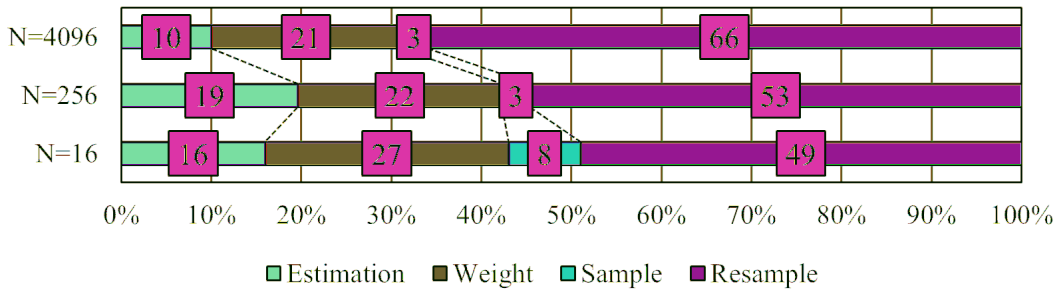


Figure 7.2.: Breakdown of time in SMC in GPU implementation [75].
The larger the number of particles, the greater the share of resampling in the total processing time.

the next particle according to the importance of the particle, as shown in the bottom of Fig. 7.1.

Various resampling algorithms have been developed, such as multinomial, residual, stratified and systematic [76–80]. These resampling algorithms involve the normalization of the weights, and the normalization requires the calculation of the cumulative sum of the quantum weights.

For speeding up these algorithms, performance was improved by parallelizing the cumulative sum of resampling and particle selection [81–83]. However, when particle weights are expressed in the single-precision floating point, calculating cumulative sums becomes difficult as the number of particles increases due to problems such as overflow. Although double-precision floating-point can be used to handle an increase in the number of particles, modern hardware is more specialized for single-precision floating-point than double-precision floating-point, and single-precision is preferable for production-scale algorithms.

Murray et al. [84] propose a GPU implementation of two new resampling algorithms (Metropolis resampling and Rejection resampling) that can be easily parallelized in hardware. The authors show that these alternative approaches are significantly faster than the commonly used systematic resampling algorithms on GPUs. However, when implemented on GPUs, these algorithms are prone to the warp divergence problem.

The algorithm of Metropolis resampling is shown in Alg. 1. As shown in Alg. 1, The algorithm is based on the Metropolis method, which compares a randomly generated

Algorithm 1 Metropolis Resampling

```
for pi = 1 to P do
    k = pi
    for bi = 1 to B do
        u ~ U[0, 1]
        j ~ U[1, ..., N]
        if u < w_j / w_k then
            k = j
        end if
    end for
    a_pi = k
end for
```

value to a value calculated from the weights of the particle under verification and a randomly selected particle to decide whether to adopt it or not. Hence, the weight calculations are independent of each other, indicating that parallelization is ready.

Liu et al. implemented a highly efficient addressing model on FPGA [85] using a Simplified Random Permutation Generator(SRPG) for Metropolis and Rejection resampling and achieved higher execution speed than Murray et al. [84]. The SRPG, had been developed by Liu et al., is a system in which indices from 0 to M-1 are stored consecutively in $log_2 M$-bit, and by shifting the number of values given by the random number generator, M independent random indices can be obtained in one random number generation. An example of the operation of the Simplified Random permutation Generator when the number of parallel particles is M, and the number of shifts obtained is two is shown in Fig. 7.3. However, this implementation requires the use of off-chip memory when the number of particles grows to about 1 million, a problem that severely limits FPGA performance.

Dülger et al. [86] developed Metropolis-C1 and Metropolis-C2, which are modified versions of the Metropolis Resampling algorithm for GPUs, and proposed an algorithm to prevent warp divergence in GPU implementations of Murray et al.'s algorithm, and confirmed faster execution than the original Metropolis Resampling in GPU implementations. Chesser et al. [87] developed Megopolis resampling, a resampling algorithm

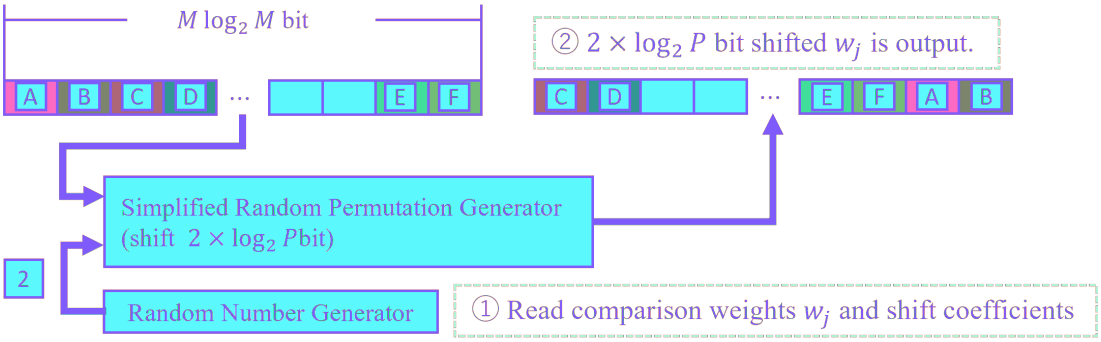


Figure 7.3.: An example of the operation of the Simplified Random permutation Generator. It receives M data, shifts it by the value obtained from the random number generator multiplied by $Mlog_2P$, and outputs it.

based on Metropolis resampling that prevents warp divergence and achieves faster execution by using warp-forcing access. However, it is generally known that execution on a GPU is less power efficient than execution on an FPGA, and many recent applications [88–91] have seen FPGAs achieve several times the power efficiency of GPUs. In recent years, the increase in power consumption by computers and servers has become an issue [92], and the ability to accurately execute large amounts of data on FPGAs is a major advantage in light of the attention paid to high computational efficiency per unit of power.

This research proposes solutions to the problems of low throughput and small memory faced by FPGAs in executing power-efficient metropolis resampling algorithms on FPGAs: increasing throughput by using integers, increasing parallelism by reducing resources, and reducing memory usage by reducing data width.

8. Proposed Integer-Optimized Metropolis Resampling

In this section, the proposed method, optimization of metropolis resampling by integerization, is presented.

Hardware efficiency by quantization to integers is very effective for hardware and power efficiency, and quantization to 8 bits is also used in the Tensor Processing Units [93], a dedicated deep neural network inference chip used in Google's data centers. Hashemi et al. [94] evaluated various deep neural network inputs and network parameters with varying bit precision of data width. The results showed that in most cases, using low-precision data offers significant advantages in terms of power, energy consumption, and design area at a slight reduction in network accuracy. Also, although this is a deep neural network, Jacob et al. [95] trained a network using a quantization scheme to integer data types and presented results that outperformed the floating-point type in hardware efficiency and latency while maintaining accuracy comparable to existing floating-point types.

It is common practice to improve power and area efficiency and latency by quantizing floating-point data and reducing it to integer execution. Power, area efficiency, and latency improvements can also be expected when applied to SMC.

Metropolis resampling is as shown in Alg.1. The integer optimization is performed on the random number generation part in line four and the comparison part in line six. First, about the processing in line six, division generally requires a large clock in CPUs and is a cause of latency reduction even if the circuitry is used. Therefore, if conversion to multiplication is possible, it is better to perform the conversion. In this case, the processing in line six can be converted to the expression 8.1 by transition because w_k and w_j are greater than or equal to 0 due to integer conversion.

Algorithm 2 The Generation Process of 0 to 1 floationg-point random value

1: $u' \sim U[0, ..., 2^N - 1]$
2: $u_f = u'$ // Conversion from integer to floating point representation
3: $u = \frac{u_f}{2^N - 1}$

$$u < \frac{w_j}{w_k}$$
$$u \times w_k < w_j \tag{8.1}$$

Next, the fourth line of processing is generated by the Alg.2 procedure when $u \sim U[0, 1]$ in the case of Nbit floating point.

Integers use the exponential part mantissa part, and unlike floating point, the value can be doubled by shifting one bit to the left. In metropolis resampling with integerization, the metropolis test for metropolis resampling can be transformed as Equ. 8.2.

$$\begin{aligned}
\frac{u'}{2^N - 1} \times w_k &< w_j \\
u' \times w_k &< w_j \times (2^N - 1) \\
u' \times w_k &< w_j \times 2^N - w_j \\
u' \times w_k &< (w_j << N) - w_j \\
u' \times w_k + w_j &< (w_j << N)
\end{aligned} \tag{8.2}$$

Here, $2N$ bits value $(w_j << N)$, which is the value of N bits integer w_j shifted left N times, is equivalent to the concatenated value of N bits w_j and Nbit 0-filled value That is, the 0 to $N - 1$ bits of $(w_j << N)$ are zero, and given that Let $[n : m]$ represent the value cut out from the m-th bit to the N-th bit of the value, then $(w_k \times u' + w_j)[2N - 1 : N] = (wj << N)[2N - 1 : N]$ and $(w_k \times u' + w_j)[N - 1 : 0] > 0$, unless. The upper N bits values of $2N$ bits, $u' \times w_k + w_j$, and the Nbit w_j values only need to compare as Equ. 8.3.

$$(u' \times w_k + w_j)[2N - 1 : N] < w_j \tag{8.3}$$

Algorithm 3 Proposed integer-optimized metropolis resampling

for $pi = 1\ to\ P$ **do**
  $k = pi$
  for $bi = 1\ to\ B$ **do**
    $u' \sim U[0, ..., N-1]$
    $j \sim U[1, ..., P]$
    if $(w_k \times u' + w_j)[2N-1 : N] < w_j$ **then**
      $k = j$
    end if
  end for
  $a_i = k$
end for

The integer-optimized metropolis resampling described above is shown in Alg.3. This optimization achieves the following two things.

- Elimination of input integer to floating point conversion costs
- Elimination of floating point operations to convert 0 to 1 in the random number generator

9. Evaluations and Results

In this section, evaluations of the proposed method, which is the optimization of the Metropolis resampling to integer, and their results and a brief explanation of items to evaluate the performance of SMC are shown. Three evaluations were conducted: as a resampling algorithm, a comparison of resampling quality using arbitrarily generated weights with previous work is shown in Section 9.2; as an evaluation for use in applications, a comparison of resampling performance in applications using the SMC sampler with previous work is shown in Section 9.4; and as an evaluation of hardware compatibility, a comparison of resource utilization when implemented in hardware with previous work is shown in Section 9.5.

9.1. Evaluation Items

In this section, Two brief explanations of items to evaluate the performance of SMC are shown.

9.1.1. Root Mean Square Error : $RMSE$

Root Mean Square Error: $RMSE$, shown in Equ 9.1, was used as the resampling quality. The $RMSE$ is calculated by how many particles: p_i with weight: w_i are employed: o_i, and its value takes a good value if more of the larger weights are adopted and fewer of the smaller weights are adopted. The range of values is greater than or equal to 0, with 0 being the best value.

$$RMSE = \sqrt{\frac{1}{P}\sum_{i=1}^{P}\left(\frac{o_i}{P} - \frac{w_i}{sum(w)}\right)^2}$$

w_i : The $i-$th weight

o_i : The number of adopted w_i

P : The number of particles (9.1)

9.1.2. Effective Sample Size : ESS

The effective sample size: ESS is a measure of the number of effective samples in a particle set used in the SMC method. It is a measure of the diversity of the particles and indicates how well the particle set represents the distribution. The ESS is calculated as follows:

1. Calculate the weights of the particles. The weights of the particles reflect the importance of each particle in representing the distribution.
2. Normalize the weights so that they sum to 1. This step is necessary to ensure that the weights can be interpreted as probabilities.
3. Calculate the ESS. ESS is calculated by Equ. 9.2

$$ESS = \frac{1}{\sum_{i=1}^{P} w_i^2} \tag{9.2}$$

The ESS will be a value between 0 and P, where P is the number of particles. A high ESS indicates that the particle set is diverse and represents the distribution well, while a low ESS indicates that the particle set is not diverse and may not be representative of the distribution.

9.2. Evaluation as the Resampling Algorithm

An evaluation of the resampling quality of the proposed method as a resampling algorithm and its results are presented in this section.

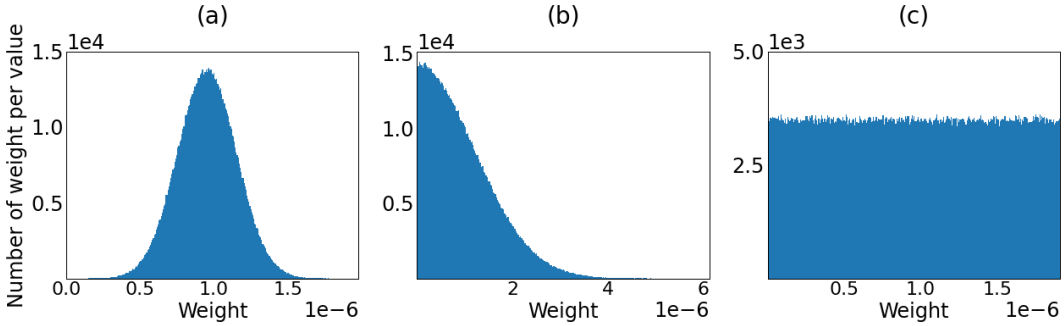


Figure 9.1.: Distributions of each input weight.
(a) Input weights that follow a normal distribution
(b) Input weights that are biased toward lower weights
(c) Input weights that follow a uniform distribution

9.2.1. Evaluation Overview

Experiments were conducted by comparing the resampling quality of the proposed and existing methods using three patterns of input weights: input weights that follow a normal distribution, input weights that are biased toward lower weights, and input weights that follow a uniform distribution.

The evaluation parameters were: the data type and data width were conventionally used as single-precision floating-point; the data width of the proposed method was 32-bit, 16-bit, and 8-bit nonnegative integers; the number of particles : P was 2^{10} to 2^{20} and the number of Metropolis tests within the Metropolis resampling algorithm : B was 2^0 to 2^{10}. Table 9.1 summarizes the evaluation parameters.

The evaluation process is described below.

1. Generation of weights
   Generate P weights according to each distribution.

2. Resampling
   Perform resampling on each implementation and verify how many p_i have been adopted o_i.

3. Calculate $RMSE$
   Calculate RMSE from o_i obtained by resampling and w_i based on equ. 9.1.

Table 9.1.: The evaluation parameters and its overview

Parameter	Summary	Range
Data Type	Type and width of data used to evaluate the algorithm.	Floating point : 32bit Unsigned integer : {32, 16, 8} bit
P	Total number of particles to be resampled	$P = 2^{p_P}, p_P = \{10, ..., 20\}$
B	The number of Metropolis tests within the Metropolis resampling algorithm:	$P = 2^{p_B}, p_B = \{0, ..., 10\}$

9.2.2. Result

Figure 9.2 shows results of varying the number of particles. The proposed integer-optimized Metropolis resampling algorithm was archived to obtain equivalent $RMSE$ to the single-precision floating-point on all setting numbers of particle size. This result indicates that integer-optimized metropolis resampling can achieve the same quality as single-precision floating-point runs, regardless of the quantum number and quantum weight bias, for the 32-, 16-, and 8-bit runs that were verified.

Figure 9.3 shows results of varying the number of Metropolis tests within the Metropolis resampling algorithm. The proposed integer-optimized Metropolis resampling algorithm was archived to obtain equivalent ESS to a single-precision floating-point on all setting numbers of Metropolis test within the Metropolis resampling algorithm. This result indicates that integer-optimized metropolis resampling can achieve the same quality as single-precision floating-point runs, regardless of the bias in the number and quantum weights of the metropolis test for the 32-, 16-, and 8-bit runs that were verified.

9.3. Evaluation of Resampling Quality by Randomness

This section describes the evaluation and results of resampling quality by randomness.

In general, obtaining true random numbers in numerical calculations is complex, and pseudo-random numbers generated from a random number generator: RNG, are often used. The difference between pseudo-random and true random numbers is that

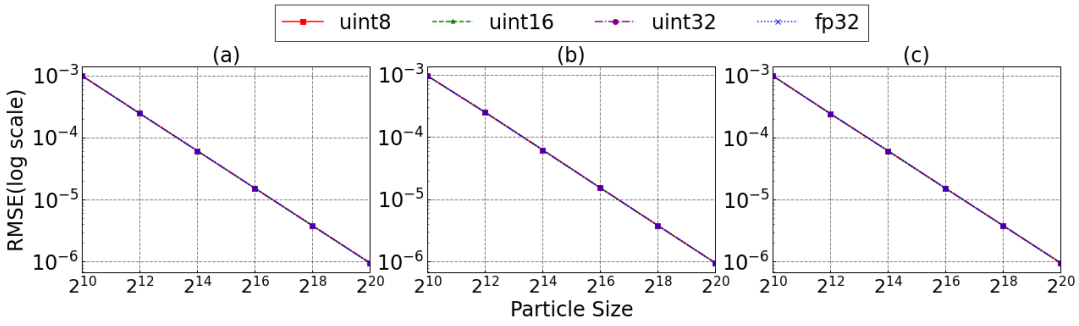


Figure 9.2.: Comparison of RMSE, a measure of resampling quality for each data width and implementation, in experiments with three types of input data shown in Fig. 9.1, varying particle count from 2^{10} to 2^{20}.

true random numbers do not have a cycle, whereas pseudo-random numbers generate identical permutations with a constant cycle. The larger the period, the more random the random number, but the larger the period, the more complex the calculations to generate the random number and the greater the hardware implementation cost. This study aimed to improve the efficiency of the Metropolis resampling hardware and reduce the cost required for RNGs. Therefore, comparing the resampling results of each RNG is conducted and looking for the RNG with a small implementation cost while maintaining the resampling quality.

9.3.1. Evalution Overview

There are several methods for generating pseudo-random numbers, such as methods based on linear congruence and M-sequences. However, the M-sequence with Linear Feedback Shift Register: LFSR is the most common and efficient hardware implementation. Therefore, evaluation is conducted using three random number generators, Galois LFSR: GLFSR as pure LFSR, XorShift: XoS and Mersenne Twister: MT as LFSR-based random number generation methods.

The linear feedback shift register: LFSR is a type of shift register that has feedback connections that allow it to generate a sequence of pseudo-random bits. The key characteristic of an LFSR is that its output is a linear function of its previous state. An LFSR's sequence of bits generated can be used as a pseudorandom number generator,

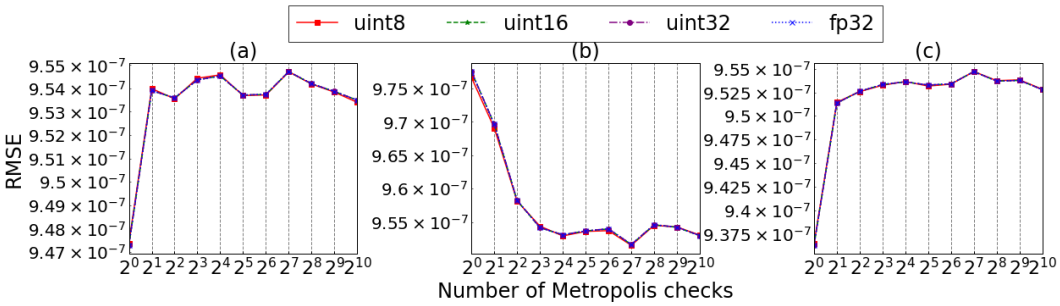


Figure 9.3.: Comparison of RMSE, a measure of resampling quality for each data width and implementation, for experiments varying the number of Metropolis tests from 2^0 to 2^{10} using three types of input data in Fig. 9.1.

stream cipher, or spreading code. LFSRs are often used in digital communications and digital signal processing because they are relatively simple to implement and have good statistical properties.

Galois FLSR has the same output as the Fibonacci LFSR to which the term LFSR usually refers, but with an optimized number of clocks for random number generation. It has a period corresponding to the number of bits, and when the number of bits is B, the period is 2^{B-1}. XorShift [96,97] is a class of pseudorandom number generators that use a linear feedback shift register to generate a sequence of bits, which are then converted to random numbers. The name "XorShift" comes from the generator using bitwise exclusive or and bit shifts to generate new random numbers. Mersenne Twister [98]: MT is an algorithm that yields good quality random numbers with a considerable period of $2^{19937} - 1$. MT has a considerable period and provides better random numbers than the GLFSR and XoS of good quality, but the hardware efficiency is low [99].

The experiment was conducted using the same evaluation axes as shown in Table 9.1 in Section 9.2 using the three types of random numbers used in Section 9.2.

9.3.2. Result

Figure 9.4 shows the comparison of RMSE quality results for each implementation when varying the number of particles from 2^{10} to 2^{20} using three different RPGs and three different input data. It indicates that the value of RMSE, the resampling quality,

is the same for any random number generator. The comparison of the RMSE quality results for each implementation when varying the number of Metropolis tests from 2^0 to 2^{10} using three different RPGs and three different input data types are shown in Fig. 9.4. In the experiment, the results were almost the same for all random number generators and for the experiments with different numbers of particles.

From this, the XorShift is the most hardware-efficient random number generator among those shown in [99], which is optimal for integer-optimized metropolis resampling.

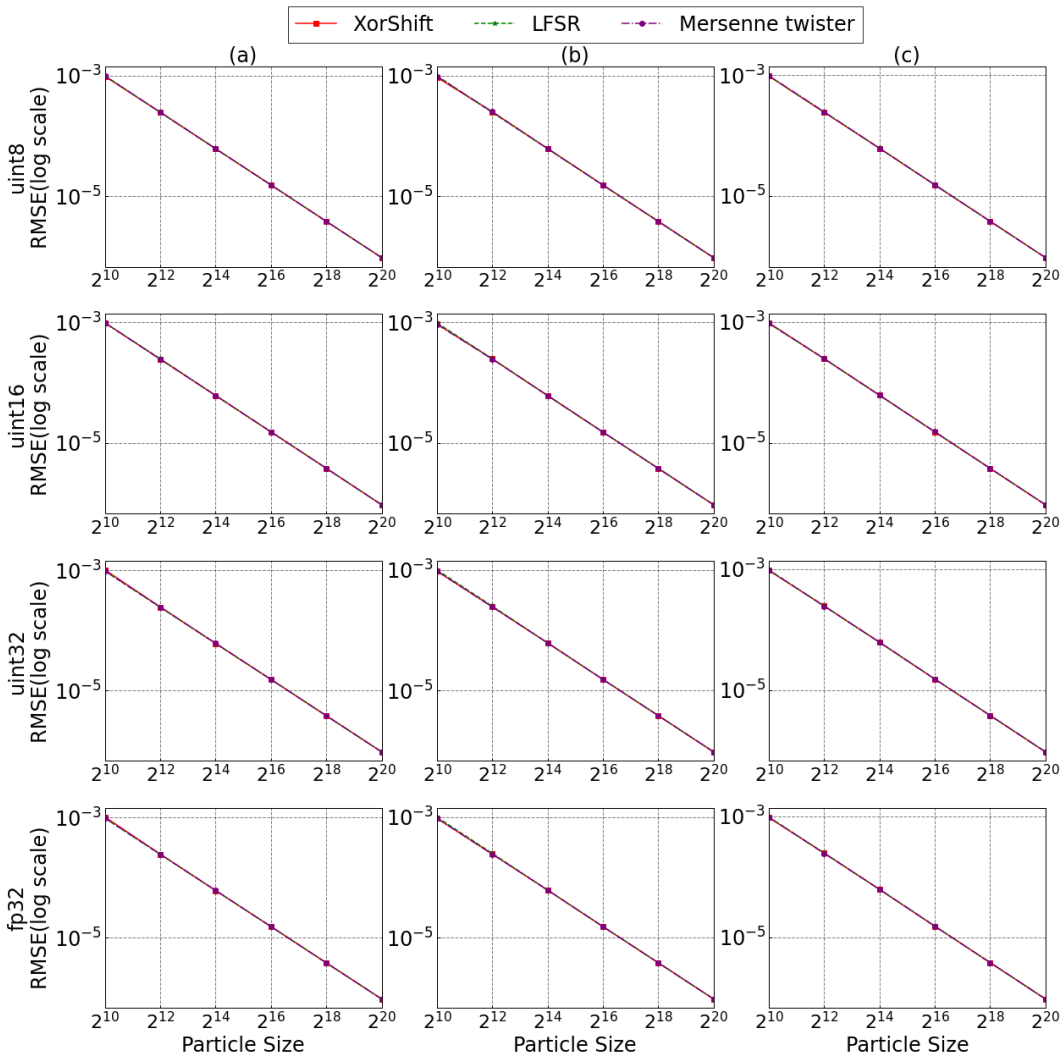


Figure 9.4.: Comparison of RMSE quality results for each implementation when varying the number of particles from 2^{10} to 2^{20} using three different RPGs and three different input data shown in Fig 9.1.
This graph indicates that the value of RMSE, the resampling quality, is the same for any random number generator.

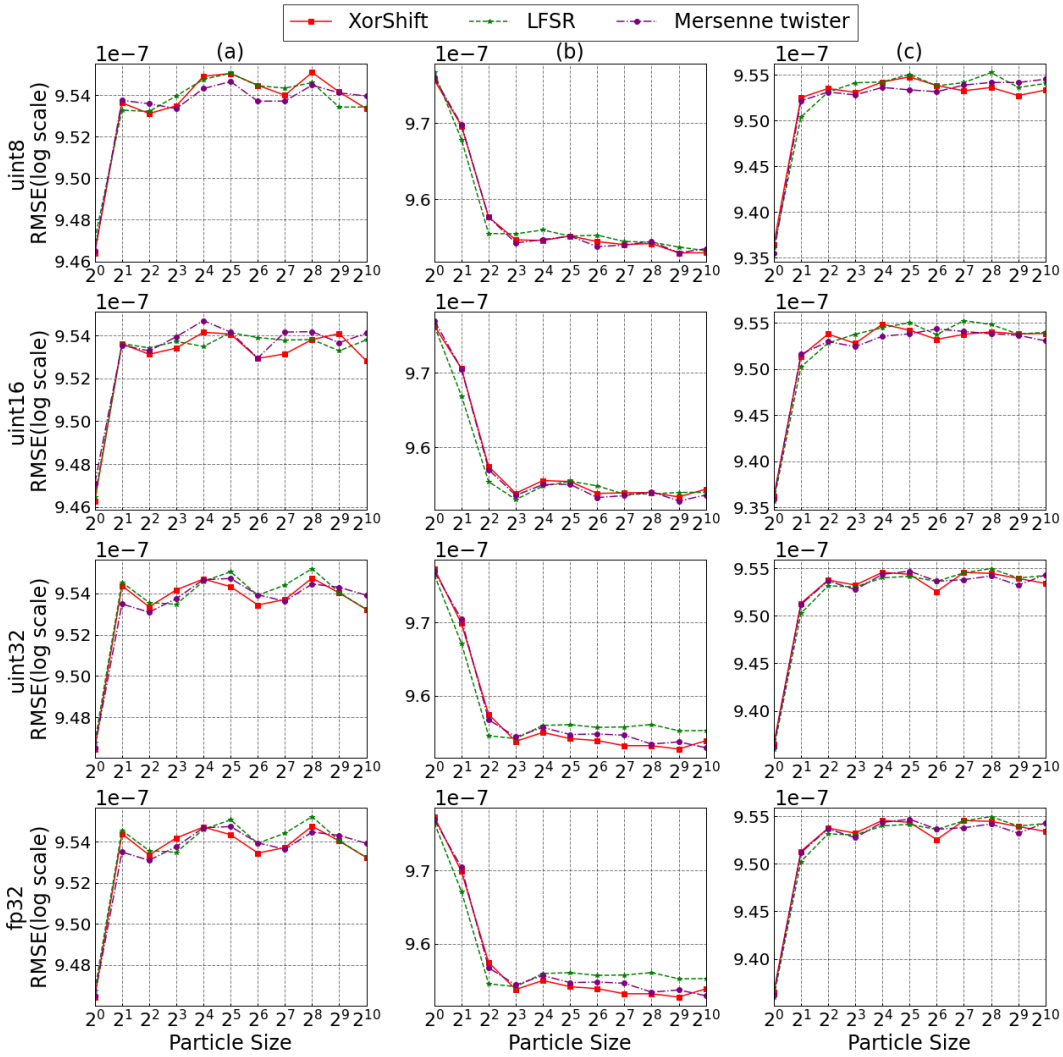


Figure 9.5.: Comparison of RMSE quality results for each implementation when varying the number of Metropolis tests from 2^0 to 2^{10} using three different RPGs and three different input data shown in Fig 9.1.
This graph indicates that the value of RMSE, the resampling quality, is the same for any random number generator.

9.4. Evaluation as a part of Sequential Monte Carlo Sampler

This section shows evaluations of the resampling quality of the proposed integer-optimized Metropolis resampling as a part of a sequential Monte Carlo sampler(SMC sampler) [65] and its results.

9.4.1. Evaluation Overview

The SMC sampler is an application of the sequential Monte Carlo method, an algorithm that estimates the parameters of a probability distribution by sampling.

This evaluation was conducted with data sampled from a normal distribution with mean set to 100 and variance set to 1: $\mathcal{N}(100, 1)$ are estimated. Particles have a Gaussian distribution as a prior for the mean and a gamma distribution as a prior for the variance. Particles are initialized $\mathcal{N}(0, 10)$ and $\mathcal{G}(1, 1)$.

The evaluation items consisted of $RMSE$, which is used in sec 9.2 and ESS, the Number of times resampling was performed. In this experiment, the SMC sampler performs resampling when the ESS is calculated with the current particle parameters and the t-th target data is half the number of particles. Hence, the number of times resampling was performed indicates how many times the ESS was below the reference value, and if this value is large, it indicates that the particles obtained by resampling are often not in line with the data. The evaluation parameters were the same as sec 9.2 and summarized in tab. 9.1.

The flow of the experiment is shown below.

1. Sample data from a normal distribution with mean set to 100 and variance set to 1: $\mathcal{N}(100, 1)$ as evaluation input data.
2. Initialize particle are initialized $\mathcal{N}(0, 10)$ and $\mathcal{G}(1, 1)$.
3. Calculate the ESS for the t-th data of the current particle.
4. If the ESS value is more than half of the number of particles, return to 3; if less than half, proceed to 5.
5. Perform resampling and update particle weights.

This processing flow is performed using 100 pieces of input data.

9.4.2. Result

Figure 9.6 is a graph of the RMSE in the experiment in which the number of Metropolis tests was fixed at 32, and the number of particles was changed from 10^{10} to 10^{20}. In this figure, the position at which resampling occurs is indicated.

Figure 9.6 shows that the change in RMSE is close to that of single-precision floating-point, except for 32-bit integers when the particle size is 10^{10}.

Figures 9.7 and 9.8 show the changes in RESE and ESS for the first ten trials of the same experiment, respectively. Figures 9.7 and 9.8 indicate that the RMSE/ESS transition for the integer-optimized Metropolis test approaches that of a single-precision floating-point run as the number of particles increases and is comparable to that of a single-precision floating-point run for any data width as the number of particles increases above 10^{18}. This may be because the larger the number of particles, the smaller the weight per particle, and the less significant the difference in weights between particles. Therefore, the larger the particles, the more influential the optimization to reduce the data width.

Table 9.4.2 shows the number of resampling runs of the SMC Sampler when the number of Metropolis tests is fixed at 32 and the number of particles is changed. Table 9.4.2 indicates that when the number of data is 1024, the SMC Sampler using the proposed integer-optimized metropolis resampler performs on average 2.7 times more resamplings than the SMC Sampler using single-precision floating-point sampling. However, when the number of particles is set to 4096 or more, the number of resamplings is almost the same.

Figure 9.9 shows the RMSE graph of an experiment in which the number of particles was fixed at 10^{20}, and the number of Metropolis test runs was changed from 1 to 1024. In this figure, the position at which resampling occurs is indicated.

The changes in RESE and ESS for the first thirty trials of the same experiment are shown in Figures 9.7 and 9.8, respectively.

From Figures 9.10 and 9.11, it can be seen that for all implementations, resampling occurs when the number of Metropolis test runs is small, that is, when the ESS is below the standard value many times, and resampling occurs every time when the number of Metropolis test runs is one. Also, up to 2^2 Metropolis test runs, the RMSE is large for

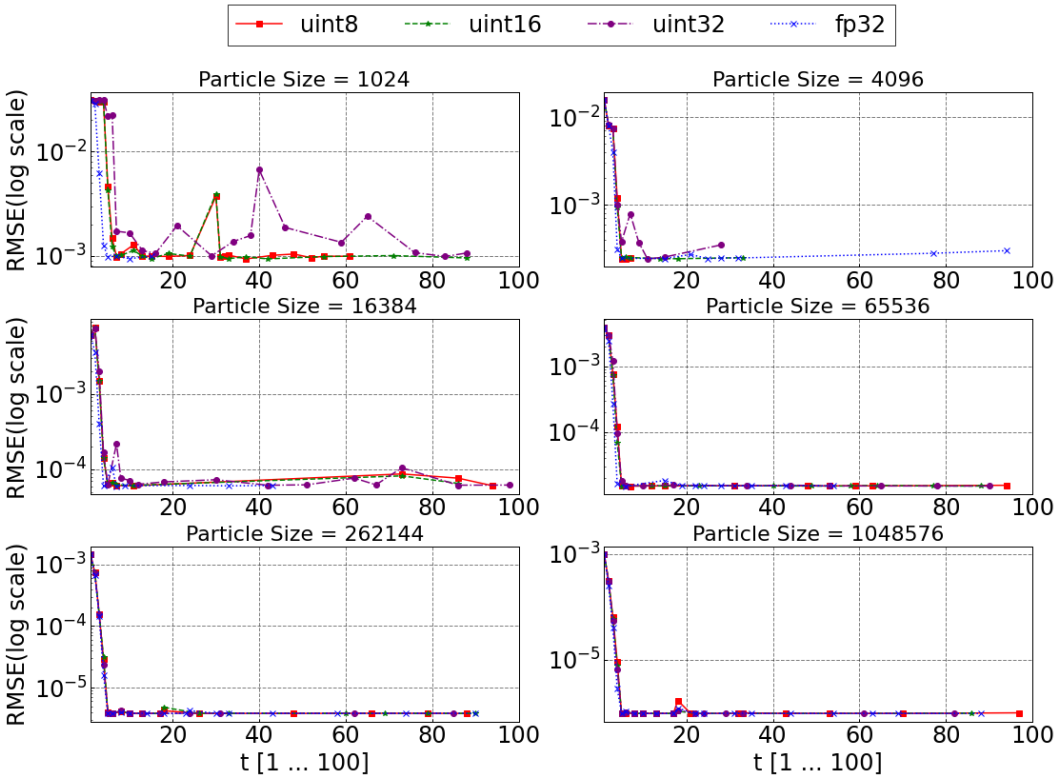


Figure 9.6.: Comparison of RMSE, a measure of resampling quality for each data width and implementation, for experiments varying the number of particles from 2^{10} to 2^{20} in parameter inference with the SMC sampler for the evaluated data.

integers 8, 16, 32, and single-precision floating-point, in that order, indicating that the quality of resampling is relatively poor. However, this property also does not change when the number of Metropolis test runs is 2^4 times. In other words, the single-precision floating-point case, more than 16 times This property is confirmed when the number of Metropolis test runs reaches 2^4, or more than 16 times, as in the single-precision floating-point case. This indicates that with integer-optimized Metropolis resampling, the resampling quality is equivalent to the single-precision floating-point when the Metropolis test is performed 16 times or more.

Table 9.4.2 shows the number of resamplings performed by the SMC sampler for each metropolis run and each data width when the number of data is fixed at 10^{20},

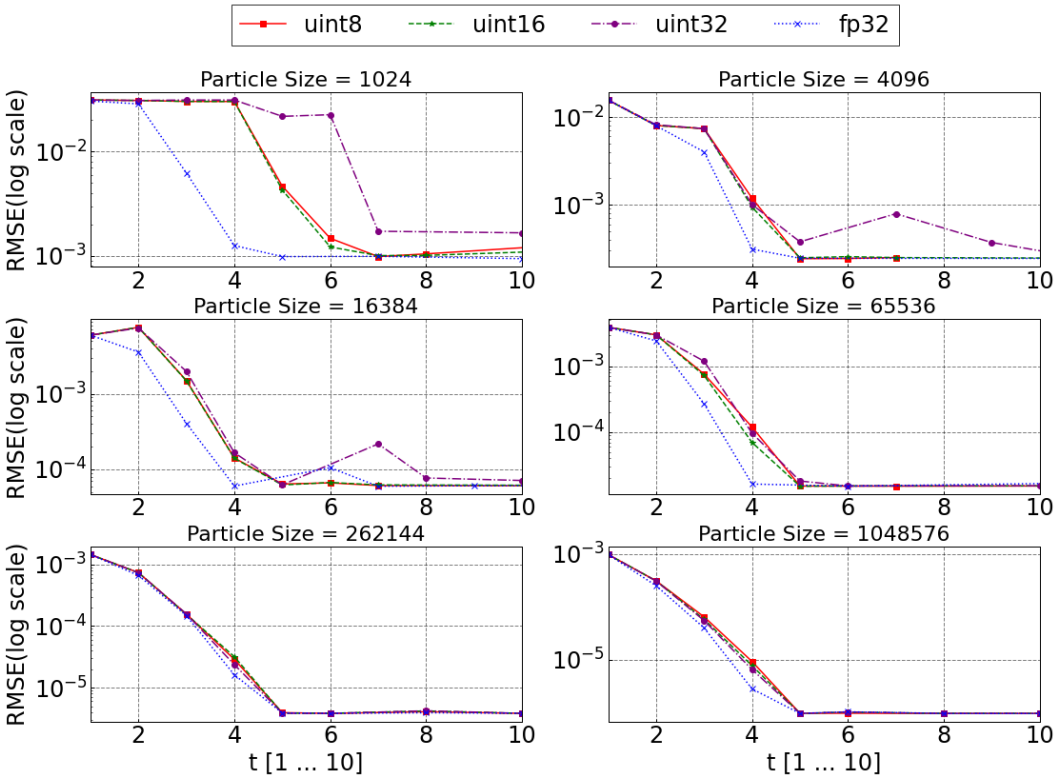


Figure 9.7.: Comparison of RMSE of first ten times, a measure of resampling quality for each data width and implementation, for experiments varying the number of particles from 2^{10} to 2^{20} in parameter inference with the SMC sampler for the evaluated data.

and the number of metropolis test runs is varied. It can be seen that the differences in the number of resampling times when the number of Metropolis tests is varied are not significantly different from those of single-precision floating-point, unlike when the number of particles is varied, as shown in Tab. 9.4.2.

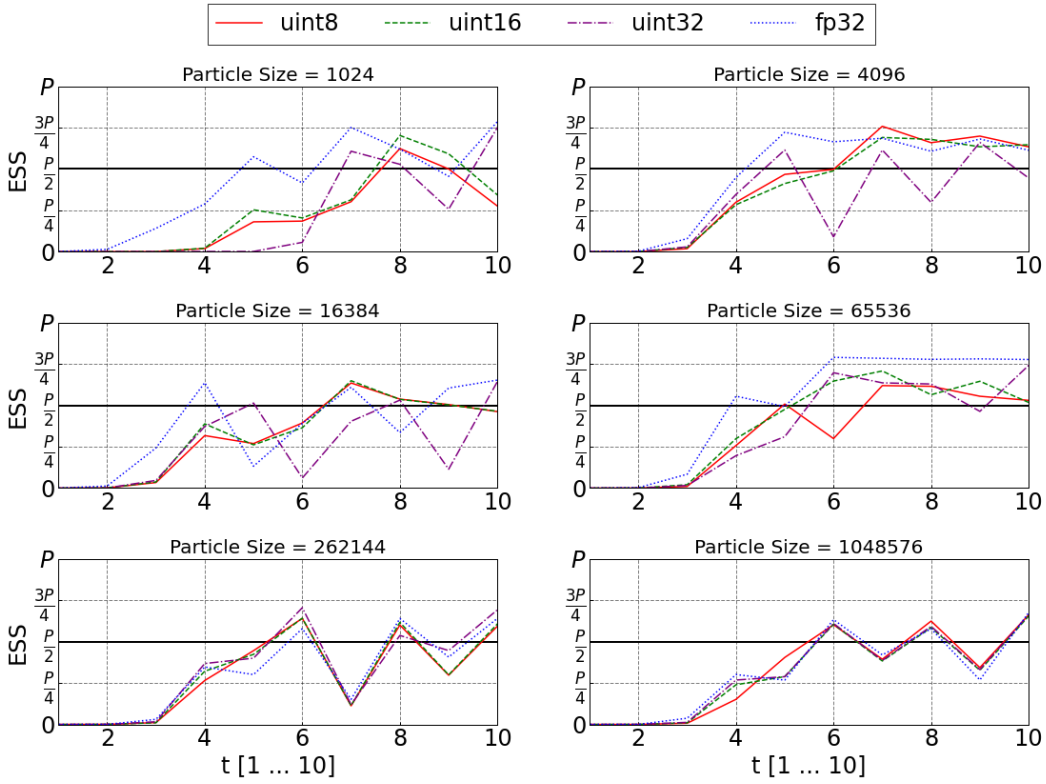


Figure 9.8.: Comparison of ESS of first ten times, a measure of performance of resampling in SMC sampler for each data width and implementation, for experiments varying the number of particles from 2^0 to 2^{10} in parameter inference with the SMC sampler for the evaluated data.

Table 9.2.: Comparison result of the number of resampling run when varying the number of particles P from 2^{10} to 2^{20}

P	uint8	uint16	uint32	fp32
2^{10}	23 (+15)	21 (+13)	21 (+13)	8
2^{12}	7 (−05)	10 (−02)	10 (−02)	12
2^{14}	11 (+01)	10 (+00)	18 (+08)	10
2^{16}	14 (+00)	15 (+01)	14 (+00)	14
2^{18}	16 (−02)	17 (−01)	14 (−04)	18
2^{20}	18 (−02)	17 (−03)	17 (−03)	20

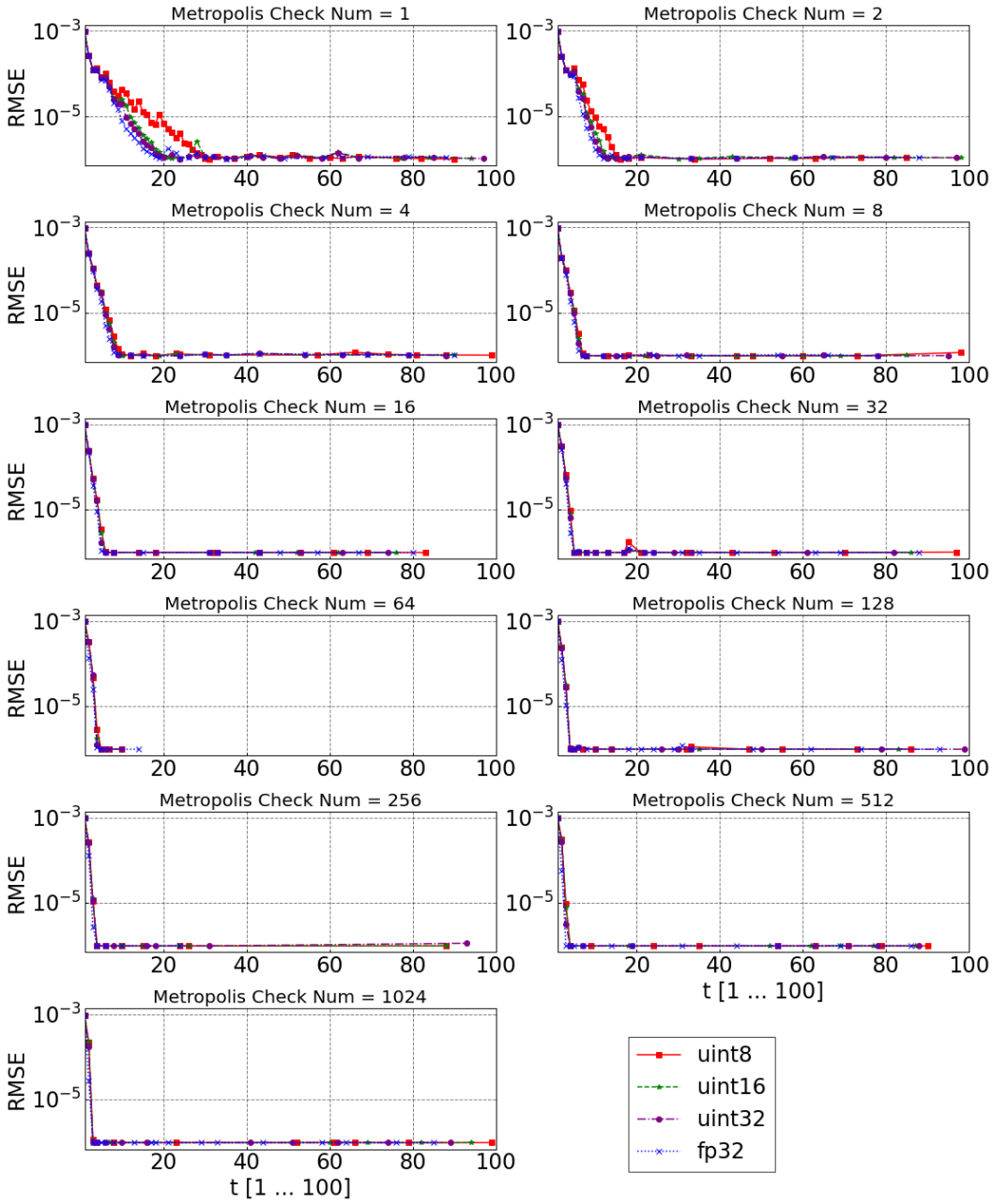


Figure 9.9.: Comparison of RMSE, a measure of resampling quality for each data width and implementation, for experiments varying the number of metropolis tests from 2^{10} to 2^{20} in parameter inference with the SMC sampler for the evaluated data.

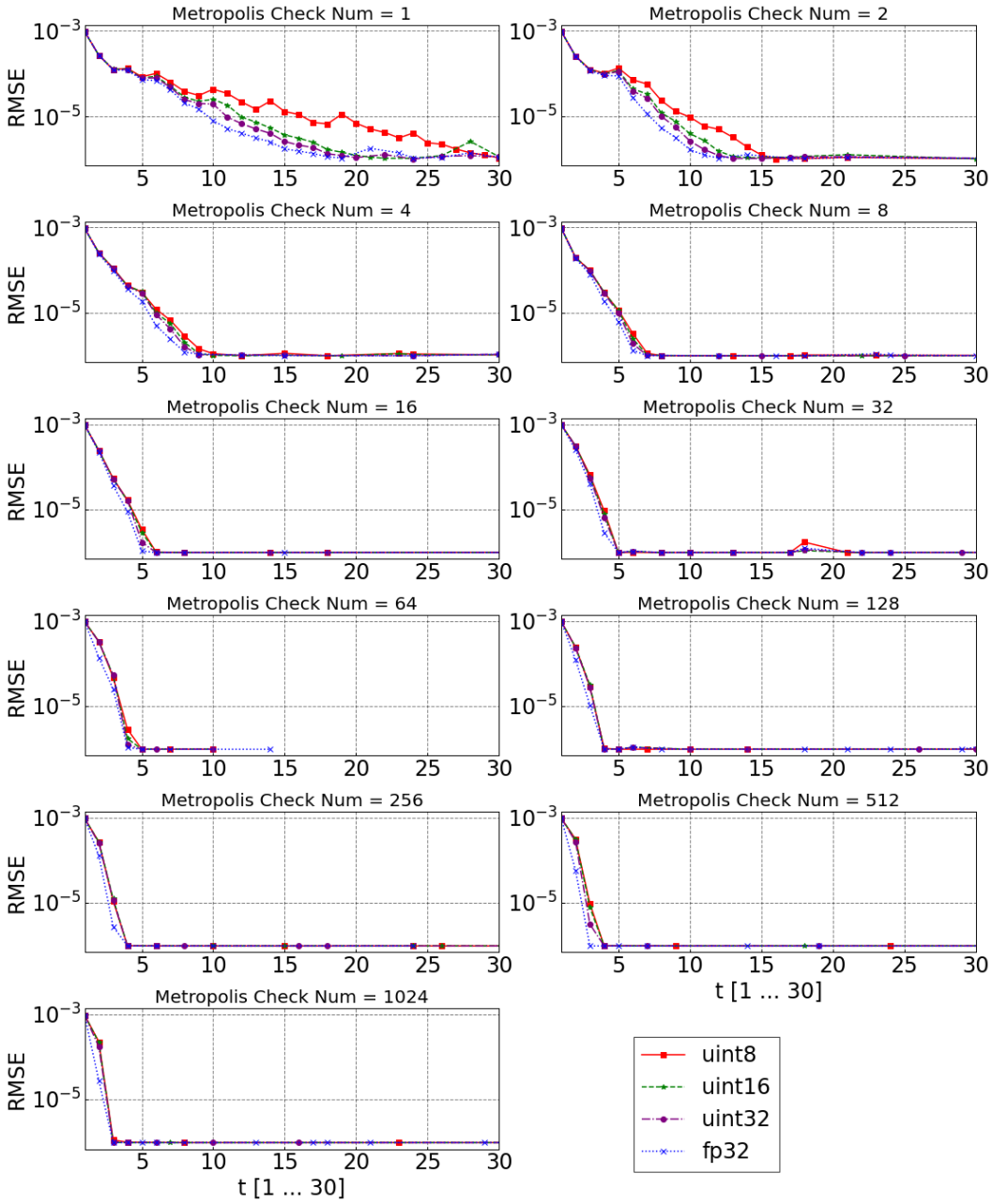


Figure 9.10.: Comparison of RMSE of first ten times, a measure of resampling quality for each data width and implementation, for experiments varying the number of metropolis tests from 2^{10} to 2^{20} in parameter inference with the SMC sampler for the evaluated data.

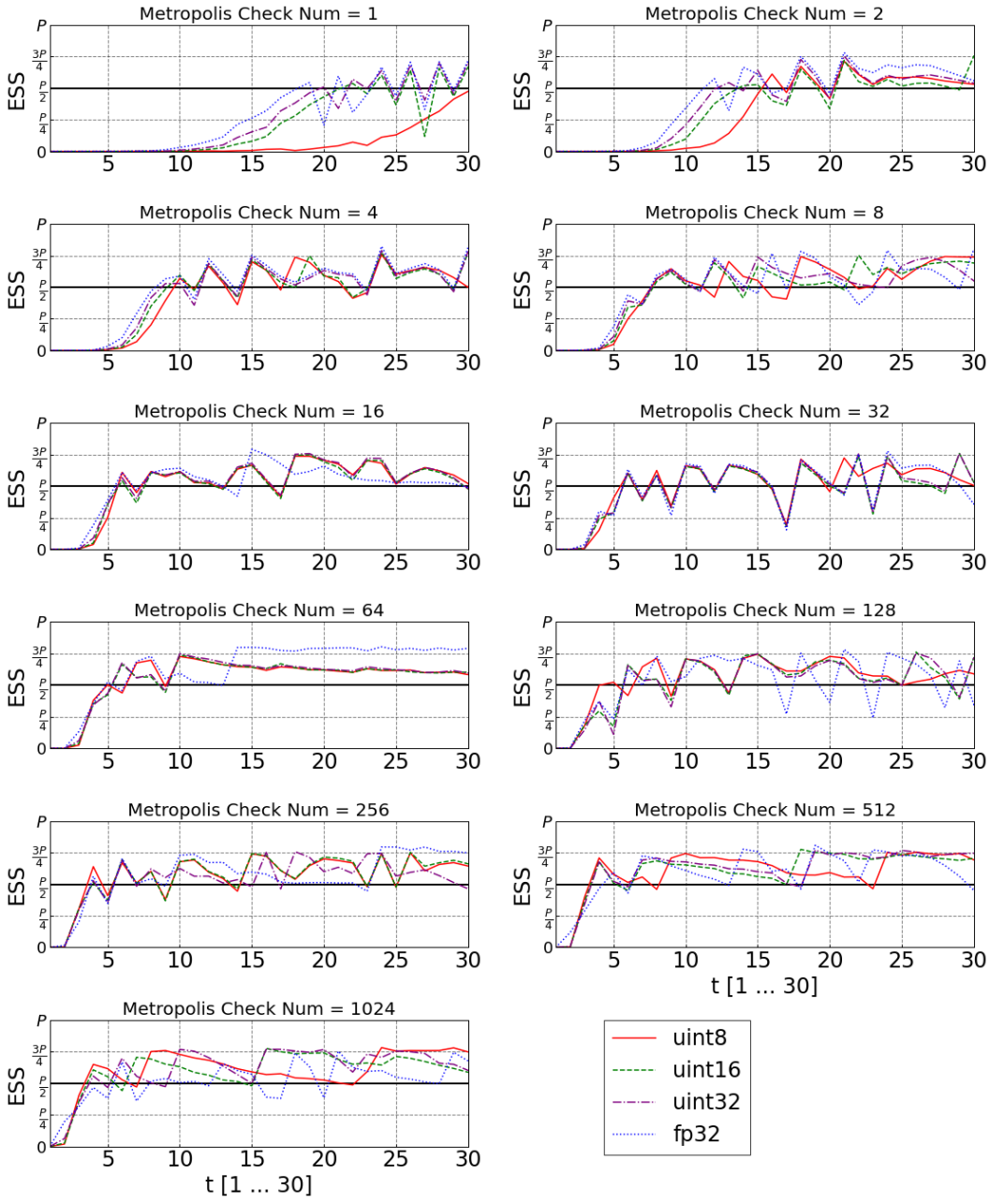


Figure 9.11.: Comparison of ESS of first ten times, a measure of performance of resampling in SMC sampler for each data width and implementation, for experiments varying the number of metropolis tests 2^0 to 2^{10} in parameter inference with the SMC sampler for the evaluated data.

Table 9.3.: Comparison result of the number of resampling run when varying the number of Metropolis tests from 2^0 to 2^{10}

B	uint8	uint16	uint32	fp32
2^0	44 (+09)	39 (+04)	38 (+03)	35
2^1	23 (+02)	24 (+03)	23 (+02)	21
2^2	22 (+02)	23 (+03)	20 (+00)	20
2^3	18 (+00)	17 (−01)	16 (−02)	18
2^4	16 (+01)	15 (+00)	15 (+00)	15
2^5	18 (−02)	17 (−03)	17 (−03)	20
2^6	7 (+00)	7 (+00)	7 (+00)	7
2^7	14 (−03)	12 (−05)	14 (−03)	17
2^8	10 (+03)	10 (+03)	10 (+03)	7
2^9	12 (−01)	11 (−02)	11 (−02)	13
2^{10}	11 (+07)	12 (−06)	12 (−06)	18

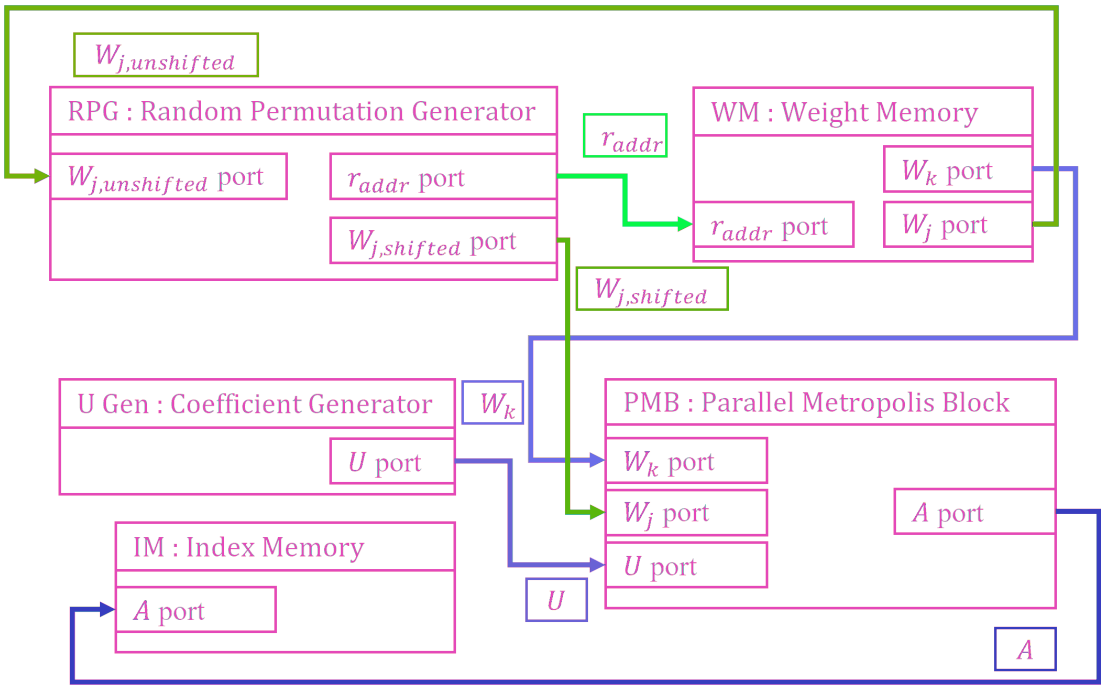


Figure 9.12.: Test design of integer-optimized Metropolis resampling architecture
The squares indicate IP cores with their respective functions, and the arrows indicate busses.

9.5. Evaluation of Hardware Efficiency

For evaluating the performance of the proposed integer-optimized Metropolis resampling in hardware, a hardware implementation was performed, and the circuit area was assessed.

The metropolis resampling circuit designed in this study is shown in Figure9.12. This design is based on the implementation of Liu et al. [85] with each part corresponding to metropolis resampling in integers. The squares indicate IP cores with their respective functions, and the arrows indicate buses. The contents of the variables in the figure are shown in Tab. 9.5, and the summary of the buses indicated by each arrow is shown in Tab. 9.5. The implementation for hardware evaluation was done using AMD's Vivado [100] and Vitis HLS [101]

Table 9.4.: Variables are used in Metropolis Resampling Circuit

Variable	Content	Value
M	Parallelism of the circuit.	An arbitrarily determined value of 2 squared.
L_D	Length of data. As an example, 32 for single-precision floating-point data and 16 for 16-bit integers.	An arbitrarily determined value of 2 squared.
L_I	Length of the index.	$log_2 P$

9.5.1. Implementation Details of each IP

- Parallelized Metropolis Block: PMB

  In Parallelism Metropolis Block(PMB), the Metropolis test is performed in M parallel. This block has three input ports, the weights $W_k = \{w_{k_1}, ..., w_{k_M}\}$ of the particles to be verified, the weights $W_{j,shifted} = \{w_{j_{1,shifted}}, ..., w_{j_{M,shifted}}\}$ of the particles to be compared, which is shifted by RPG, These inputs are distributed to M metropolis blocks, each of which receives B metropolis test runs.

  The port of input W_k is connected to the weight memory: WM, and the weight of particles is read from WM in order according to the counter that PMB has inside, and sent to each Metropolis block.

  Input W_j port is connected to Random Permutation Generator: RPG. The W_j port receives the shifted weights and the indexes of those weights sent by RPG.

  Input U port is connected to Coefficient Generator: U-Gen. M coefficients generated by U-Gen are inputted.

  The Metropolis test is performed with Equ. 9.3 when the data type is a float and with Equ. 8.3 when the data type is an integer.

$$u \times w_k < w_j \tag{9.3}$$

  The particles that result from the Metropolis test are weighted together and sent to the memory, where the results are stored.

  This block is implemented using RTL for random number generation, and high-level synthesis for shifting and supporting AXI4-Stream, and parallelized to M

Table 9.5.: Bus Overview of Metropolis Resampling Circuit

Bus Name	Content	Width
r_{addr}	Random address generated from random numbers to obtain W_j. It is a multiple of $L_D/8 \times M$.	32
W_{k_t}	The group of w_k, the comparative importance of the t-th input in the M-parallel metropolis resampling circuit. $W_{k_t} = \{w_{k_{t\times M}}, w_{k_{t\times M+1}}, ..., w_{k_{t\times M+M-1}}\}$	$L_D \times M$
$W_{j,unshifted}$	Unshifted weights read from the weight memory. when $i = r_{addr}/((L_D/8) \times M)$, $W_{j_i} = \{w_{j_{i\times M}}, w_{j_{i\times M+1}}, ..., w_{j_{i\times M+M-1}}\}$	$L_D \times M$
$W_{j,shifted}$	Shifted weights read from the weight memory.	$L_D \times M$
U	Random numbers for Metropolis resampling. $U = \{u_1, u_2, ..., u_{M-1}\}$	$L_D \times M$
A_t	Result of t-th run.	$L_I \times M$

by #pragma HLS unroll.

The input/output ports of the PMB and their roles are shown in Tab. 9.5.1.

The processing of sending and receiving data in the AXI4-Stream protocol and the computation of the Metropolis test in this block is implemented with high-level synthesis using Vitis HLS [101], and M parallelism is achieved by #pragma HLS unroll.

The hardware resources evaluated in this IP are Look Up Table (LUT) and Digital Signal Processing (DSP). IPs other than this IP are capable of outputting data every clock cycle and supplying data every clock cycle. This means that the throughput of this block is directly related to performance. Therefore, Latecy, the number of clocks until this block receives input and outputs, is added as a performance evaluation item.

- Random Permutation Generator: RPG
  This block is used to shift a random number of randomly read weights and send them to the PMB. The process flow is shown below.

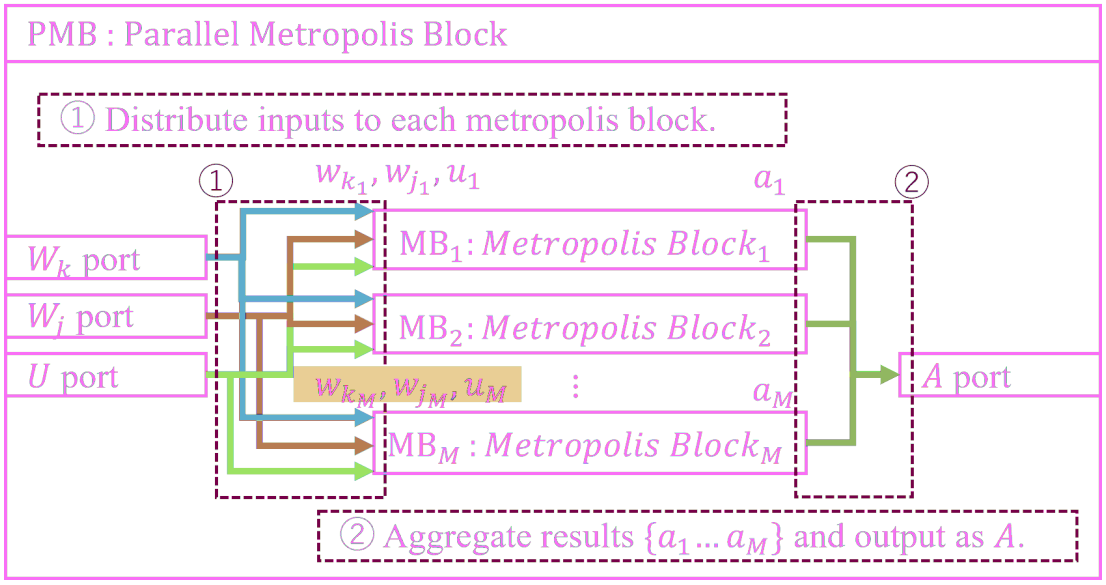


Figure 9.13.: Detail of the Parallelized Metropolis Block: PMB.
This block receives inputs from three ports, W_k, W_j, and U, distributes the inputs to M metropolis blocks, and aggregates the results and outputs them as A.

1. Generate random numbers for read addresses r_{addr} adn random number for shift value r_{shift}.
2. Read r_{addr} value $W_{j,unshifted}$ from WM.
3. Shift the read data $W_{j,unshifted}$ according to r_{shift}.
4. Transmit shifted weight $W_{j,shifted}$ to PMB.

This block is implemented using RTL for random number generation, and high-level synthesis for shifting and supporting AXI4-Stream using Vitis HLS [101], and is pipelined internally by Vitis HLS optimization directive #pragma HLS pipeline, and parallelized to M by #pragma HLS unroll. Thus, pipelining and parallelization allow this block to read out the weights of M comparison particles every clock.

XorShift [96] is used as a random number generator which is hardware-efficient. Since the only computation performed in this IP is the shift operation, no DSP is used. Therefore, only the LUT was used as the evaluation item.

Table 9.6.: The input/output ports of the PMB

Port Name	In/Out	Width(bit)	Overview
W_k port	In	$L_D \times M$	Input port of particles to be verified
W_j port	In	$L_D \times M$ $+L_I \times M$	Input port of particles to be compared
U port	In	$L_D \times M$	Input port of coefficients
A port	Out	$L_I \times M$	Output port of result indexes

- Coefficients Generator: U-Gen
  This block generates M coefficients U for the Metropolis test. The random number generator employs Xorshift as in RPG. For integer implementations, random numbers are output as they are. In the case of a floating-point implementation, the random number generated by Xorshift is converted to floating-point format and multiplied by the inverse of the maximum value that can be obtained in the bit width.

  This block is implemented using RTL for random number generation, and high-level synthesis for multiplication and supporting AXI4-Stream using Vitis HLS [101], and is pipelined internally by Vitis HLS optimization directive #pragma HLS pipeline, and parallelized to M by #pragma HLS unroll. Thus, pipelining and parallelization allow this block to read out the weights of M comparison particles every clock.

  As with the PMB, the evaluation items for this IP were the number of LUTs and DSPs used.

- Weight Memory: WM
  This block is the memory that stores weights. It consists of Block RAM and has ports to read W_k and W_j, respectively. This IP evaluates BRAM usage to assess memory resource usage.

- Index Memory: IM
  This block is the memory that stores results. It consists of Block RAM and has ports to read W_k and W_j, respectively. Since this IP has the same implementation in integer as in the previous studies and the hardware used is the same for all data

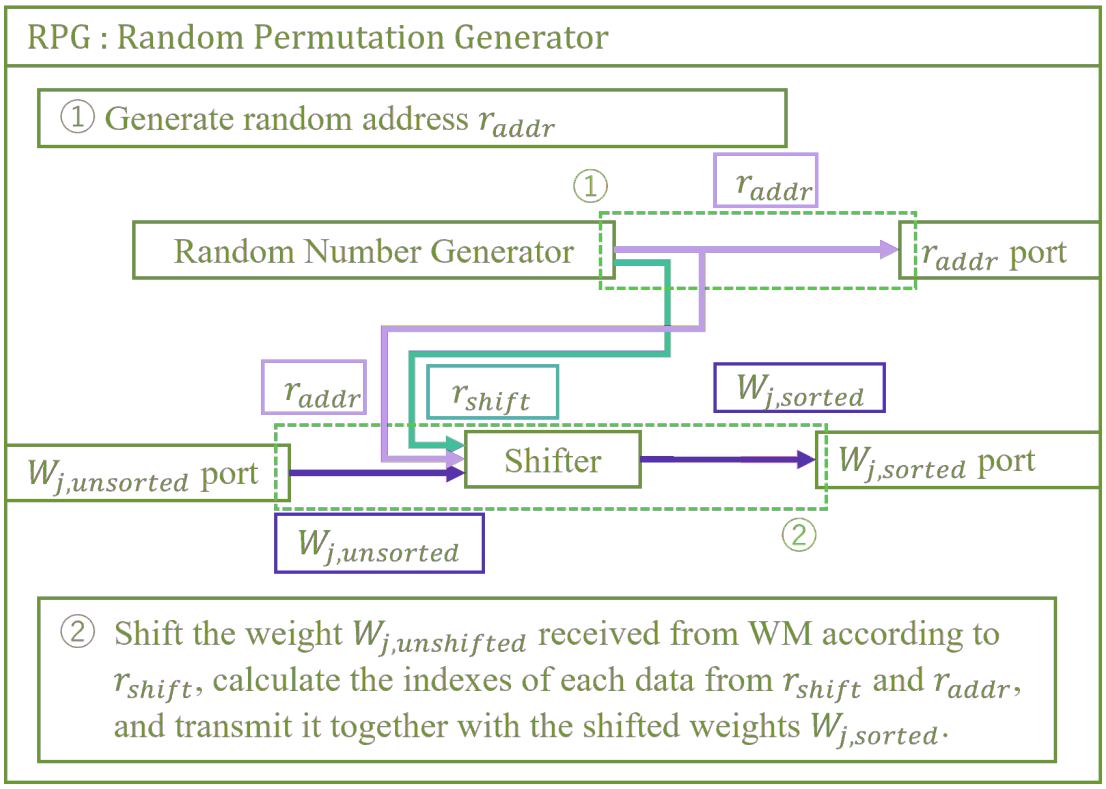


Figure 9.14.: Detail of the Random Permutation Generator: RPG.
This block reads out M randomly weights of particles with random addresses and shifts by randomly generated values and sent to the PMB along with their indices.

types and widths to be evaluated, the evaluated values are omitted.

9.5.2. Result

Table9.9 shows a table comparing the per-resource utilization of IP when the degree of parallelism M is set to 16. From Tab. 9.9, it can be seen that the resampling module based on the proposed integer-optimized Metropolis resampling algorithm saves a significant amount of resources, except for IM, the memory that stores the resampling results.

Adopting the proposed integer optimization achieved to reduce LUT usage for integer implementations of all data widths compared to single precision floating point

Table 9.7.: The input/output ports of the RPG

Port Name	In/Out	Width(bit)	Overview
$W_{j,unshifted}$ port	In	$D_W \times M$	Input port of particles to be compared from WM
$W_{j,shifted}$ port	Out	$D_W \times M + P\log_2(P) \times M$	Output port of particles are sorted in this block
r_{addr} port	Out	A_W	Output port of address for reading random particles

Table 9.8.: The input/output ports of the U-Gen

Port Name	In/Out	Width(bit)	Overview
U port	Out	$D_W \times M$	Output port of coefficients for Metropolis test.

implementations, reducing it to 25% for 32-bit integers, 60% for 16-bit integers, and 69% for 8-bit integers on PMB. DSP usages were also improved, reduced to 25% for 16-bit integer and 8-bit integer implementations. Latency per Metropolis test runs as 3, 4, and 2 for 32-, 16-, and 8-bit integers, respectively, compared to 6 for single-precision floating-point.

Since the module that reads the random weights for the Metropolis test: RPG, does not use floating-point operations, there is no difference in resources used between single-precision floating-point and 32-bit integers, which are also 32 bits. Implementing 16-bit and 8-bit integers achieved resource savings of 40% and 51%, respectively.

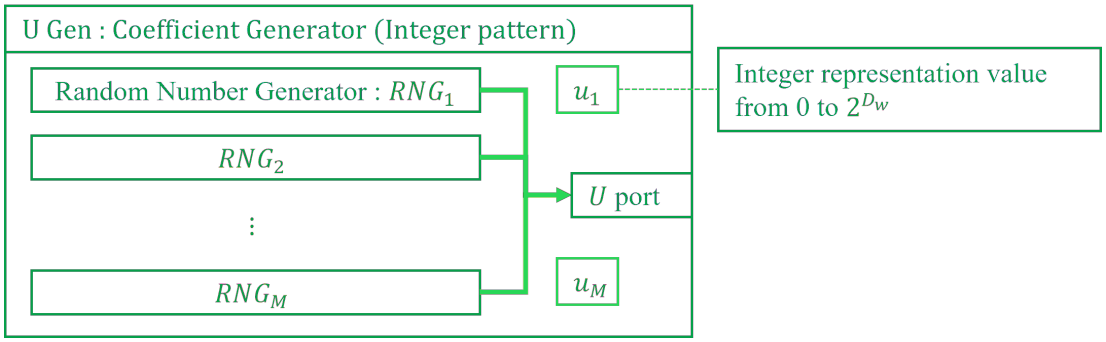
The circuit that generates the random numbers that serve as the coefficients for the Metropolis test has achieved significant resource savings, reducing the LUT by an average of 27%. In addition, the number of DSPs used uniquely by data type in this IP was 0 for all data width implementations with integers, while 48 were used for

Table 9.9.: Comparison of resource usage for each data type and data width when parallelism M set to 16 and number of particles set to 2^{20}

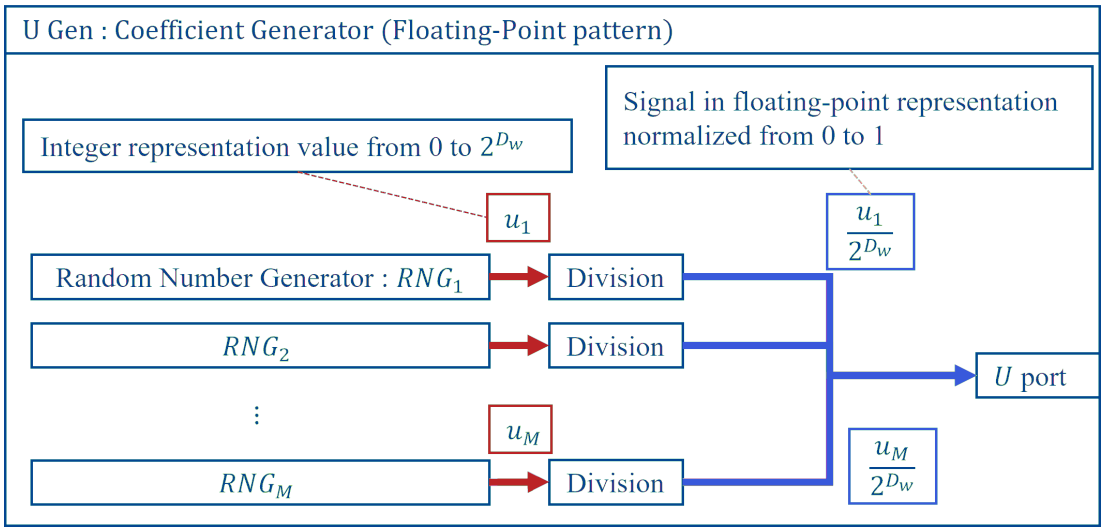
	Data Type	Floating-Point	Integer		
	Width (Bits)	32	32	16	8
IP	Item				
PMB	LUT	6415	4832 (75%)	2561 (40%)	2051 (32%)
	DSP	64	64	16	16
	Latency	6	3	4	2
RPG	LUT	6128	6128 (100%)	3707 (60%)	3057 (50%)
U-Gen	LUT	5561	1542 (28%)	1686 (30%)	1430 (17%)
	DSP	48	0	0	0
WM	BRAMB36 Tiles	512	512	256	128

single-precision floating-point.

In WM, its resource usage was reduced according to the data width. The 32-bit integer is the same as the single-precision floating-point one, but the implementation with 16-bit integers reduces the BRAM usage, which is resource usage, to 50% and 25% for 8-bit integers.



(a) Implementation of U-Gen by integer



(b) Implementation of U-Gen by floating-point

Figure 9.15.: Detail of the coefficients generator: U-Gen.
This block generates the attendants for the Metropolis test and generates random numbers for each clock data width, and the implementation is different for integer and floating-point.

10. Discussion

Firstly, based on the results of Section 9.2, the Discussion of the integer-optimized resampling performance of the proposed method, the Metropolis resampling algorithm, is shown. While Section 9.2 was performed on three patterns of datasets: input data with weights following a normal distribution, input data with a majority of small weights, and data with weights following a uniform distribution, the proposed metropolis resampling algorithm, optimized to run on integers, achieved For all data widths, the resampling quality was equivalent to that of single-precision floating-point, regardless of the number of particles and the number of Metropolis tests. This shows that although quantization generally causes data bias and loss of difference after one resampling, the proposed method of resampling with integer quantization can guarantee resampling quality without problems in the three cases tested in this study.

Secondly, the discussion of the results from the experiments with random number quality is conducted in Section 9.3. In this experiment, the same experiments as in Section 9.2 were conducted using three random number generators. The results show that, as in Experiment Section 9.2, the proposed method, integer-optimized metropolis resampling, achieved RMSEs comparable to single-precision floating-point results on all data sets. This indicates that it is better to use XorShift-based hardware, which has the best hardware efficiency among those verified when implementing hardware since it can achieve the same resampling quality as floating-point ones regardless of the random number quality.

Thirdly, the discussion of the proposed method in the SMC sampler is shown based on the experiments and results presented in Section 9.4. In this experiment, unlike the two experiments described, there was a large difference between floating point and integer runs. Both single-precision floating-point and integer implementations showed a tendency to lower the RMSE over the number of trials and stabilize at some lower level. When the number of particles was 2^{10}, more resampling was performed in the

integer run than in the single-precision floating-point result. This is likely due to the fact that when the data is small, the per-particle weights tend to be larger, and finer differences become more important, but these differences are lost due to quantization. As the number of data increases to 2^{16}, the difference in weights for each particle data becomes smaller, even for single-precision floating point, indicating that the trends in RMSE and ESS are almost the same as for integer data. In particular, for particle counts of 2^{18} and 2^{20}, the proposed method shows almost the same trend as the single-precision floating-point method for both ESS and RMSE at all data widths. This is a very convenient result for the reduction of data width, which is being verified as a countermeasure to the increased hardware usage due to the larger number of particles.

The comparison by changing the number of Metropolis test runs also shows that the number of trials required for the RMSE to reach some stable value for all implementations differs depending on the number of Metropolis test runs. The overall trend is that the greater the number of Metropolis tests, the fewer the number of trials required before stabilization. This may be due to the nature of metropolis resampling: the fewer the number of trials, the lower the probability that particle replacement will occur, resulting in poor-quality resampling. Among the implementations, we found that running the Metropolis test with 8-bit integers required more trials to stabilize than the other implementations when the number of Metropolis test runs was small. This phenomenon is thought to be caused by a combination of the following two factors. (i) the number of quantization bits makes it relatively easy for weights with small differences to be quantized to the same weight, and (ii) the small number of trials reduces the probability of comparison with particles with large differences. This is thought to be the reason for the low RMSE in integer 8-bit execution, as even comparisons that would be exchanged in the original floating-point execution remain low-weighted particles because quantization eliminates the difference, and the comparison does not occur. This problem was also observed not to occur when the number of Metropolis tests is four or more, and the RMSE transition of four or more times is equivalent to that of the single-precision floating-point for all data widths.

Thus, the proposed metropolis resampling with quantization to integer can perform as well as the single-precision floating-point one when the number of particles is larger than 2^{16}, and the number of metropolis tests is more significant than 4.

Finally, the discussion about hardware efficiency from the results in Section 9.5 is

given.

The parallel execution IP (PMB) of the Metropolis test confirmed that the integer implementation could reduce the LUT by 25%, even when implemented at 32 bits, the same width as the single-precision floating-point implementation, and that the 16-bit and 8-bit implementations can significantly reduce usage by more than 60%. Latency was also confirmed to be improved in all implementations. This is thought to be because the number of clocks and amount of resources required for single-precision floating-point operations are more significant than those for integer operations. In the Metropolis resampling circuit, the Metropolis test performed in this PMB is the bottleneck, and processing the Metropolis test for P particles requires the number of clocks obtained by multiplying the latency of this circuit by the number of Metropolis tests and the number of particles and dividing by the parallelism degree M. The proposed method's ability to reduce latency by integerization is a major advantage.

For RPG, the IP that reads the comparison weights for the Metropolis test, the 16- and 8-bit implementations with reduced data widths showed better hardware efficiency than the single-precision floating-point ones. 32-bit implementations are the same as the single-precision floating-point ones because the only processing performed within this circuit is shifting, which is not dependent on the data type. The reason is that the processing performed in this circuit is shift-only and does not depend on the data type.

In U-Gen, the circuit that creates the coefficients for the Metropolis test, the integer implementation produced higher hardware efficiency than the single-precision floating-point one. This can be attributed first of all to the fact that the integer implementation omits the process of conversion and multiplication of random numbers to floating point, as also mentioned in Section 8.

It can be seen that WM, the memory that stores the weights, reduces its resource usage according to the data width. In the case of the integer 8-bit implementation, it is possible to store four times the number of particles with the same resource usage compared to the implementation in the previous study by Liu et al. [85]

With this, the three goals described in Section 7.4 and the proposed integerization, increased throughput, increased parallelism due to resource reduction, and reduced memory usage due to reduced data width was achieved.

Part III.

Conclusion

This study shows the optimization of parallel implementations of parameter estimation tasks in two types of Bayesian inference.

The first part is GPU acceleration of parameter estimation by the variational inference method of GMM used for clustering. The Gaussian mixture model based on variational Bayesian estimation is implemented on GPU for high-speed clustering applications. Employing the proposed strategies, including CPU-GPU co-optimization, execution reorder, and memory management, the VB-GMM is efficiently conducted by GPU with high parallelism. Various data sets for real-world clustering applications are introduced for validations. The experimental results show that convergence is generally faster than the same algorithm implemented on the CPU, and comparable convergence scores are achieved. As a typical example, the proposed VB-GMM on GPU is 192 times faster than the CPU when the number of data is 1E+07 and 107 times faster when the number of clusters is 128. Compared with the state-of-art GPU implementations conducted by the EM algorithm, the proposed VB-GMM on GPU can suppress degeneracy even on data sets where the EM algorithm would have degenerated; fair performances over speed, clustering scores, and clustering distributions are achieved. Experiments on practical data showed that VB-GMM on GPUs is more effective than the GPU implementation of the EM algorithm for clustering data with a relatively large number of data, where the actual number of clusters is unknown.

In the second part, the Optimization of the dedicated hardware for the resampling step, which is one of the three steps in the sequential Monte Carlo method: sampling step, importance calculation step, and resampling step, and whose processing time increases in proportion to the number of particles. The specific optimization method is to optimize the metropolis resampling algorithm used in the resampling step, which is suitable for parallelization, from the currently used floating-point method to the integer-optimized method.

The proposed optimization of Metropolis Resampling to run on integers achieves the elimination of conversion and multiplication to the floating point in the random number generation part within Metropolis Resampling. Also, it makes it hardware efficient by eliminating calculations in the floating-point in the Metropolis test.

In the evaluation experiments, the proposed method was evaluated for 8-bit, 16-bit, and 32-bit integers, considering hardware efficiency. In evaluating several algorithms on CPUs, the proposed integer-optimized Metropolis resampling achieved resampling

quality equivalent to that of single-precision floating-point methods for 8-bit integers, 16-bit integers, and 32-bit integers as resampling alone. In addition, the Sequential Monte Carlo Sampler, an SMC application, achieved the same performance as the floating-point method when the number of particles was large.

In the evaluation of the hardware implementation, the proposed integer-optimized Metropolis resampling reduced resource usage for all data width implementations compared to previous studies using the single-precision floating-point. It achieved up to 3.0 times LUT usage improvement in critical modules such as coefficient generation and Metropolis test execution improvements in key modules such as coefficient generation and Metropolis test execution. LUT utilization reductions of 31% at 32 bits, 57% at 16 bits, and 64% at 8 bits were achieved in key modules such as coefficient generation and Metropolis test execution, while other bottlenecks achieved up to 3.0 times throughput improvement and up to 75% memory utilization reduction.

Acknowledgements

I would like to start by thanking my advisor, Professor Yasuhiko Nakashima, for their guidance, support, and encouragement throughout the entire process of researching and writing this dissertation. Their expertise in the field and willingness to help at all times were invaluable to me.

I would also like to express my gratitude to the members of my dissertation committee, Professor Kazushi Ikeda, Associate Professor Renyuan Zhang, Assistant professor Kan Yirong, Assistant professor Pham Hoai Luan, , for their constructive feedback and valuable insights. I appreciate the time and effort they took to review my work and provide me with the guidance I needed to improve it.

I would like to thank all lab members. I gave much advice from them.

I would also like to thank Mr. Nishida and Mr. Kaji, supervisors of my internship company. They were given the opportunity to learn the general hardware implementation flow.

I would like to thank my wife and friends for their unwavering support and encouragement. Their belief in me helped me to persevere through the challenges of this process.

I would also like to extend my thanks to the Nara Institute of Science and Technology for providing me with the resources and opportunities to pursue this degree. I am grateful for the support of the faculty and staff who have helped me along the way.

References

[1] Jiawei Han, Jian Pei, and Hanghang Tong. *Data mining: concepts and techniques*. Morgan kaufmann, 2022.

[2] Krzysztof J Cios, Witold Pedrycz, and Roman W Swiniarski. *Data mining methods for knowledge discovery*, volume 458. Springer Science & Business Media, 2012.

[3] Eva Patel and Dharmender Singh Kushwaha. Clustering cloud workloads: K-means vs gaussian mixture model. *Procedia Computer Science*, 171:158–167, 2020.

[4] Douglas A Reynolds, Thomas F Quatieri, and Robert B Dunn. Speaker verification using adapted gaussian mixture models. *Digital signal processing*, 10(1-3):19–41, 2000.

[5] Chris Stauffer and W Eric L Grimson. Adaptive background mixture models for real-time tracking. In *Proceedings. 1999 IEEE computer society conference on computer vision and pattern recognition (Cat. No PR00149)*, volume 2, pages 246–252. IEEE, 1999.

[6] Adrian Corduneanu and Christopher M Bishop. Variational bayesian model selection for mixture distributions. In *Artificial intelligence and Statistics*, volume 2001, pages 27–34. Morgan Kaufmann Waltham, MA, 2001.

[7] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[8] NSL Phani Kumar, Sanjiv Satoor, and Ian Buck. Fast parallel expectation maximization for gaussian mixture models on gpus using cuda. In *2009 11th IEEE*

International Conference on High Performance Computing and Communications, pages 103–109. IEEE, 2009.

[9] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[10] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[11] Attila Reiss and Didier Stricker. Introducing a new benchmarked dataset for activity monitoring. In *2012 16th international symposium on wearable computers*, pages 108–109. IEEE, 2012.

[12] Ramon Huerta, Thiago Mosqueiro, Jordi Fonollosa, Nikolai F Rulkov, and Irene Rodriguez-Lujan. Online decorrelation of humidity and temperature in chemical sensors for continuous monitoring. *Chemometrics and Intelligent Laboratory Systems*, 157:169–176, 2016.

[13] John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, 30(2):56–69, 2010.

[14] Tosiyuki Nakaegawa. High-performance computing in meteorology under a context of an era of graphical processing units. *Computers*, 11(7):114, 2022.

[15] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit, 2020. Accessed: 2023-01-31.

[16] NVIDIA. Nvidia ampere ga102 gpu architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf. Accessed: 2023-01-31.

[17] Paulius Micikevicius. Gpu performance analysis and optimization. 2012.

[18] Tomoki Toda, Alan W Black, and Keiichi Tokuda. Voice conversion based on maximum-likelihood estimation of spectral parameter trajectory. *IEEE Transactions on Audio, Speech, and Language Processing*, 15(8):2222–2235, 2007.

[19] 藤田迪, 梶克彦, 河口信夫, et al. Gaussian mixture model を用いた無線 lan 位置推定手法. **情報処理学会論文誌**, 52(3):1069–1081, 2011.

[20] Ismail Shahin, Ali Bou Nassif, and Shibani Hamsa. Emotion recognition using hybrid gaussian mixture model and deep neural network. *IEEE access*, 7:26777–26787, 2019.

[21] Junya Koguchi, Shinnosuke Takamichi, Masanori Morise, Hiroshi Saruwatari, and Shigeki Sagayama. Dnn-based full-band speech synthesis using gmm approximation of spectral envelope. *IEICE TRANSACTIONS on Information and Systems*, 103(12):2673–2681, 2020.

[22] Ce Guo, Haohuan Fu, and Wayne Luk. A fully-pipelined expectation-maximization engine for gaussian mixture models. In *2012 International Conference on Field-Programmable Technology*, pages 182–189. IEEE, 2012.

[23] Conghui He, Haohuan Fu, Ce Guo, Wayne Luk, and Guangwen Yang. A fully-pipelined hardware design for gaussian mixture models. *IEEE Transactions on Computers*, 66(11):1837–1850, 2017.

[24] M. Harris. Optimizing parallel reduction in cuda. https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf. Accessed: 2023-01-31.

[25] Alessandro Tasora, Radu Serban, Hammad Mazhar, Arman Pazouki, Daniel Melanz, Jonathan Fleischmann, Michael Taylor, Hiroyuki Sugiyama, and Dan Negrut. Chrono: An open source multi-physics dynamics engine. In *High Performance Computing in Science and Engineering: Second International Conference, HPCSE 2015, Soláň, Czech Republic, May 25-28, 2015, Revised Selected Papers 2*, pages 19–49. Springer, 2016.

[26] clblas. https://github.com/clMathLibraries/clBLAS. Accessed: 2023-01-31.

[27] David L Davies and Donald W Bouldin. A cluster separation measure. *IEEE transactions on pattern analysis and machine intelligence*, (2):224–227, 1979.

[28] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron

Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[29] Michael C Hughes and Erik Sudderth. Memoized online variational inference for dirichlet process mixture models. *Advances in neural information processing systems*, 26, 2013.

[30] Elaine Angelino, Matthew James Johnson, Ryan P Adams, et al. Patterns of scalable bayesian inference. *Foundations and Trends® in Machine Learning*, 9(2-3):119–247, 2016.

[31] Elaine Angelino, Eddie Kohler, Amos Waterland, Margo Seltzer, and Ryan P Adams. Accelerating mcmc via parallel predictive prefetching. *arXiv preprint arXiv:1403.7265*, 2014.

[32] Iain Murray. *Advances in Markov chain Monte Carlo methods*. University of London, University College London (United Kingdom), 2007.

[33] Arnaud Doucet, Nando de Freitas, and Neil Gordon. An introduction to sequential monte carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer, 2001.

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

[35] Jun S Liu and Rong Chen. Sequential monte carlo methods for dynamic systems. *Journal of the American statistical association*, 93(443):1032–1044, 1998.

[36] Petar M Djuric. Sequential estimation of random parameters under model uncertainty. In *2000 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No. 00CH37100)*, volume 1, pages 297–300. IEEE, 2000.

[37] Corentin Dubois, Manuel Davy, and Jérôme Idier. Tracking of time-frequency components using particle filtering. In *Proceedings.(ICASSP'05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, volume 4, pages iv–9. IEEE, 2005.

[38] Nattapol Aunsri and Kosin Chamnongthai. Stochastic description and evaluation of ocean acoustics time-series for frequency and dispersion estimation using particle filtering approach. *Applied Acoustics*, 178:108010, 2021.

[39] Yi Zhang, Yong Lv, and Mao Ge. Time–frequency analysis via complementary ensemble adaptive local iterative filtering and enhanced maximum correlation kurtosis deconvolution for wind turbine fault diagnosis. *Energy Reports*, 7:2418–2435, 2021.

[40] Yvo Boers and Pranab K Mandal. Optimal particle-filter-based detector. *IEEE signal processing letters*, 26(3):435–439, 2019.

[41] Jin Hyeok Yoo, Sun Hong Lim, Byonghyo Shim, and Jun Won Choi. Estimation of dynamically varying support of sparse signals via sequential monte-carlo method. *IEEE Transactions on Signal Processing*, 68:4135–4147, 2020.

[42] Yunqiang Chen and Yong Rui. Real-time speaker tracking using particle filter sensor fusion. *Proceedings of the IEEE*, 92(3):485–494, 2004.

[43] Darren B Ward, Eric A Lehmann, and Robert C Williamson. Particle filtering algorithms for tracking an acoustic source in a reverberant environment. *IEEE Transactions on speech and audio processing*, 11(6):826–836, 2003.

[44] Kai Nickel, Tobias Gehrig, Rainer Stiefelhagen, and John McDonough. A joint particle filter for audio-visual speaker tracking. In *Proceedings of the 7th international conference on multimodal interfaces*, pages 61–68, 2005.

[45] Hong Liu, Yidi Li, and Bing Yang. 3d audio-visual speaker tracking with a two-layer particle filter. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 1955–1959. IEEE, 2019.

[46] Rong Wang, Zhe Chen, and Fuliang Yin. Distributed multiple speaker tracking based on unscented particle filter and data association in microphone array networks. *Circuits, Systems, and Signal Processing*, pages 1–23, 2022.

[47] Rudolph Van Der Merwe, Arnaud Doucet, Nando De Freitas, and Eric Wan. The unscented particle filter. *Advances in neural information processing systems*, 13, 2000.

[48] Andrew Blake, B Bascle, M Isard, and J MacCormick. Statistical models of visual shape and motion. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 356(1740):1283–1302, 1998.

[49] M Sanjeev Arulampalam, Simon Maskell, Neil Gordon, and Tim Clapp. A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. *IEEE Transactions on signal processing*, 50(2):174–188, 2002.

[50] Juan José Pantrigo, Angel Sánchez, Kostas Gianikellis, and Antonio S Montemayor. Combining particle filter and population-based metaheuristics for visual articulated motion tracking. *ELCVIA Electronic Letters on Computer Vision and Image Analysis*, 5(3):68–83, 2005.

[51] Jayant R Mahajan, Neetu Agarwal, and Chandansingh Rawat. Motion object tracking for thermal imaging using particle filter. In *Applied Computer Vision and Image Processing: Proceedings of ICCET 2020, Volume 1*, pages 161–168. Springer, 2020.

[52] Asfak Ali, Avra Ghosh, and Sheli Sinha Chaudhuri. Determination of optimum dynamic threshold for visual object tracker. In *2021 International Conference on Automation, Control and Mechatronics for Industry 4.0 (ACMI)*, pages 1–5. IEEE, 2021.

[53] Pei-Hsuan Chiu, Po-Hsuan Tseng, and Kai-Ten Feng. Interactive mobile augmented reality system for image and hand motion tracking. *IEEE Transactions on Vehicular Technology*, 67(10):9995–10009, 2018.

[54] Tianzhu Zhang, Changsheng Xu, and Ming-Hsuan Yang. Multi-task correlation particle filter for robust object tracking. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4335–4343, 2017.

[55] Chao Ma, Jia-Bin Huang, Xiaokang Yang, and Ming-Hsuan Yang. Hierarchical convolutional features for visual tracking. In *Proceedings of the IEEE international conference on computer vision*, pages 3074–3082, 2015.

[56] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. *IEEE robotics & automation magazine*, 13(2):99–110, 2006.

[57] Feng Zhang, Siqi Li, Shuai Yuan, Enze Sun, and Languang Zhao. Algorithms analysis of mobile robot slam based on kalman and particle filter. In *2017 9th International Conference on Modelling, Identification and Control (ICMIC)*, pages 1050–1055. IEEE, 2017.

[58] Dieter Fox, Sebastian Thrun, Wolfram Burgard, and Frank Dellaert. Particle filters for mobile robot localization. In *Sequential Monte Carlo methods in practice*, pages 401–428. Springer, 2001.

[59] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. Markov localization for mobile robots in dynamic environments. *Journal of artificial intelligence research*, 11:391–427, 1999.

[60] Robert Sim, Pantelis Elinas, Matt Griffin, James J Little, et al. Vision-based slam using the rao-blackwellised particle filter. In *IJCAI Workshop on Reasoning with Uncertainty in Robotics*, volume 14, pages 9–16, 2005.

[61] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling. In *Proceedings of the 2005 IEEE international conference on robotics and automation*, pages 2432–2437. IEEE, 2005.

[62] Peter Karkus, Shaojun Cai, and David Hsu. Differentiable slam-net: Learning particle slam for visual navigation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2815–2825, 2021.

[63] Nicolas Chopin. A sequential particle filter method for static models. *Biometrika*, 89(3):539–552, 2002.

[64] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential monte carlo for bayesian computation. *Bayesian statistics*, 8(1):34, 2007.

[65] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential monte carlo samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(3):411–436, 2006.

[66] Christian Naesseth, Scott Linderman, Rajesh Ranganath, and David Blei. Variational sequential monte carlo. In *International conference on artificial intelligence and statistics*, pages 968–977. PMLR, 2018.

[67] Emad Hadian, Hamidreza Akbari, Mehdi Farzinfar, and Seyedamin Saeed. Optimal allocation of electric vehicle charging stations with adopted smart charging/discharging schedule. *IEEE Access*, 8:196908–196919, 2020.

[68] Hanxin Chen, Dong Liang Fan, Lu Fang, Wenjian Huang, Jinmin Huang, Chenghao Cao, Liu Yang, Yibin He, and Li Zeng. Particle swarm optimization algorithm with mutation operator for particle filter noise reduction in mechanical fault diagnosis. *International journal of pattern recognition and artificial intelligence*, 34(10):2058012, 2020.

[69] Christian A Naesseth, Fredrik Lindsten, and Thomas B Schön. High-dimensional filtering using nested sequential monte carlo. *IEEE Transactions on Signal Processing*, 67(16):4177–4188, 2019.

[70] Chenguang Dai, Jeremy Heng, Pierre E Jacob, and Nick Whiteley. An invitation to sequential monte carlo samplers. *Journal of the American Statistical Association*, 117(539):1587–1600, 2022.

[71] Neil J Gordon, David J Salmond, and Adrian FM Smith. Novel approach to nonlinear/non-gaussian bayesian state estimation. In *IEE proceedings F (radar and signal processing)*, volume 140, pages 107–113. IET, 1993.

[72] 上野玄太. 粒子フィルタとデータ同化. **統計数理**, 67(2):241–253, 2019.

[73] Fred Daum and Jim Huang. Particle degeneracy: root cause and solution. In *Signal Processing, Sensor Fusion, and Target Recognition XX*, volume 8050, pages 367–377. SPIE, 2011.

[74] Liang Meng, Mark A Kramer, and Uri T Eden. A sequential monte carlo approach to estimate biophysical neural models from spikes. *Journal of neural engineering*, 8(6):065006, 2011.

[75] Gustaf Hendeby, Rickard Karlsson, and Fredrik Gustafsson. Particle filtering: the need for speed. *EURASIP Journal on Advances in Signal processing*, 2010:1–9, 2010.

[76] Miodrag Bolic, Petar M Djuric, and Sangjin Hong. Resampling algorithms and architectures for distributed particle filters. *IEEE Transactions on Signal Processing*, 53(7):2442–2450, 2005.

[77] Randal Douc and Olivier Cappé. Comparison of resampling schemes for particle filtering. In *Ispa 2005. proceedings of the 4th international symposium on image and signal processing and analysis, 2005.*, pages 64–69. IEEE, 2005.

[78] Jeroen D Hol, Thomas B Schon, and Fredrik Gustafsson. On resampling algorithms for particle filters. In *2006 IEEE nonlinear statistical signal processing workshop*, pages 79–82. IEEE, 2006.

[79] Tiancheng Li, Miodrag Bolic, and Petar M Djuric. Resampling methods for particle filtering: classification, implementation, and strategies. *IEEE Signal processing magazine*, 32(3):70–86, 2015.

[80] BG Sileshi, Carles Ferrer, and Joan Oliver. Particle filters and resampling techniques: Importance in computational complexity analysis. In *2013 Conference on Design and Architectures for Signal and Image Processing*, pages 319–325. IEEE, 2013.

[81] Peng Gong, Yuksel Ozan Basciftci, and Fusun Ozguner. A parallel resampling algorithm for particle filtering on shared-memory architectures. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1477–1483. IEEE, 2012.

[82] Mark Harris et al. Optimizing parallel reduction in cuda. *Nvidia developer technology*, 2(4):70, 2007.

[83] Gustaf Hendeby, Jeroen D Hol, Rickard Karlsson, and Fredrik Gustafsson. A graphics processing unit implementation of the particle filter. In *2007 15th European Signal Processing Conference*, pages 1639–1643. IEEE, 2007.

[84] Lawrence M Murray, Anthony Lee, and Pierre E Jacob. Parallel resampling in the particle filter. *Journal of Computational and Graphical Statistics*, 25(3):789–805, 2016.

[85] Shuanglong Liu, Grigorios Mingas, and Christos-Savvas Bouganis. Parallel resampling for particle filters on fpgas. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 191–198. IEEE, 2014.

[86] Özcan Dülger, Halit Oğuztüzün, and Mübeccel Demirekler. Memory coalescing implementation of metropolis resampling on graphics processing unit. *Journal of Signal Processing Systems*, 90(3):433–447, 2018.

[87] Joshua A Chesser, Hoa Van Nguyen, and Damith C Ranasinghe. The megopolis resampler: Memory coalesced resampling on gpus. *Digital Signal Processing*, 120:103261, 2022.

[88] Tan Nguyen, Colin MacLean, Marco Siracusa, Douglas Doerfler, Nicholas J Wright, and Samuel Williams. Fpga-based hpc accelerators: An evaluation on performance and energy efficiency. *Concurrency and Computation: Practice and Experience*, 34(20):e6570, 2022.

[89] Murad Qasaimeh, Joseph Zambreno, Phillip H Jones, Kristof Denolf, Jack Lo, and Kees Vissers. Analyzing the energy-efficiency of vision kernels on embedded cpu, gpu and fpga platforms. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 336–336. IEEE Computer Society, 2019.

[90] Mark Klaisoongnoen, Nick Brown, and Oliver Brown. Fast and energy-efficient derivatives risk analysis: Streaming option greeks on xilinx and intel fpgas. *arXiv preprint arXiv:2212.13977*, 2022.

[91] Hao Sun, Qi Deng, Xinzhe Liu, Yuhao Shu, and Yajun Ha. An energy-efficient stream-based fpga implementation of feature extraction algorithm for lidar point clouds with effective local-search. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2022.

[92] Japan Science Center for Low Carbon Society Strategy and Technology Agency. Impact of progress of information society on energy consumption (vol. 4): Feasibility study of technologies for decreasing energy consumption of data centers. 4, 2022.

[93] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[94] Soheil Hashemi, Nicholas Anthony, Hokchhay Tann, R Iris Bahar, and Sherief Reda. Understanding the impact of precision quantization on the accuracy and energy of neural networks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1474–1479. IEEE, 2017.

[95] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

[96] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8:1–6, 2003.

[97] Richard Brent et al. Note on marsaglia's xorshift random number generators. American Statistical Association, 2004.

[98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[99] Mohammed Bakiri, Jean-François Couchot, and Christophe Guyeux. Fpga implementation of $\mathbb{F}_2$ -linear pseudorandom number generators based on zynq mpsoc: a chaotic iterations post processing case study. *arXiv preprint arXiv:1611.08410*, 2016.

[100] Xilinx. Vivado design suite user guide: Design flows overview (ug892). https://docs.xilinx.com/r/en-US/ug892-vivado-design-flows-overview/RTL-Design, 2022. Accessed: 2023-01-31.

[101] Xilinx. Introduction to fpga design with vivado high-level synthesis (ug998). https://docs.xilinx.com/v/u/en-US/ug998-vivado-intro-fpga-design-hls, 2019. Accessed: 2023-01-31.

Publication List

Peer Review Journal Paper

1. Hiroki NISHIMOTO, Renyuan ZHANG, Yasuhiko NAKASHIMA, GPGPU Implementation of Variational Bayesian Gaussian Mixture Models, IEICE Transactions on Information and Systems, 2022, Volume E105.D, Issue 3, Pages 611-622, Released on J-STAGE, Mar. (2022)

Peer Review Conference Paper

1. H. Nishimoto, T. Nakada and Y. Nakashima, "GPGPU Implementation of Variational Bayesian Gaussian Mixture Models," 2019 Seventh International Symposium on Computing and Networking (CANDAR'19), Nov. (2019)

2. H. Nishimoto, R. Zhang and Y. Nakashima, "Application and Evaluation of Quantization for Narrow Bit-width Resampling of Sequential Monte Carlo," 2022 IEEE 35th International System-on-Chip Conference (SOCC'22), Sep. (2022)

3. Honda, Taku, Hiroki Nishimoto, and Yasuhiko Nakashima. "Speeding Up VBGMM By Using Logsumexp With the Approximate Exp-function." 2020 Eighth International Symposium on Computing and Networking Workshops (CANDARW). IEEE, 2020.

Misc

1. 西本宏樹, 中田尚, 中島康彦. "変分混合ガウスモデルアクセラレータ設計のための変分推論アルゴリズムの解析." 研究報告システムと LSI の設計技術 (SLDM) 2018.29 (2018): 1-6.

2. 西本宏樹, 中田尚, 中島康彦. "GPGPU を用いた変分混合ガウスモデルのパラメータ推定高速化." 研究報告システム・アーキテクチャ (ARC) 2019.1 (2019): 1-5.

3. 上垣柊季, 芦原佑樹, 阪口喜晃, 西本宏樹. "ロケット GNSS-TEC による電離圏電子密度構造観測", 第 4 回観測ロケットシンポジウム, Apr. (2022)

4. 西本宏樹, 木村睦. "AI による画像認識のしくみ / AIS の講演の紹介" IDW'20 チュートリアル Dec. (2020)