PAPER

GPGPU Implementation of Variational Bayesian Gaussian Mixture Models

Hiroki NISHIMOTO[†a)], ***Student Member***, **Renyuan ZHANG**[†], ***Member***, ***and*** **Yasuhiko NAKASHIMA**[†], ***Fellow***

SUMMARY The efficient implementation strategy for speeding up high-quality clustering algorithms is developed on the basis of general purpose graphic processing units (GPGPUs) in this work. Among various clustering algorithms, a sophisticated Gaussian mixture model (GMM) by estimating parameters through variational Bayesian (VB) mechanism is conducted due to its superior performances. Since the VB-GMM methodology is computation-hungry, the GPGPU is employed to carry out massive matrix-computations. To efficiently migrate the conventional CPU-oriented schemes of VB-GMM onto GPGPU platforms, an entire migration-flow with thirteen stages is presented in detail. The CPU-GPGPU co-operation scheme, execution re-order, and memory access optimization are proposed for optimizing the GPGPU utilization and maximizing the clustering speed. Five types of real-world applications along with relevant data-sets are introduced for the cross-validation. From the experimental results, the feasibility of implementing VB-GMM algorithm by GPGPU is verified with practical benefits. The proposed GPGPU migration achieves 192x speedup in maximum. Furthermore, it succeeded in identifying the proper number of clusters, which is hardly conducted by the EM-algotihm.

key words: *GPGPU, Gaussian mixture model, variational Bayes, machine learning, clustering*

1. introduction

The 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 the clustering is one of significant tasks. Among all of clustering algorithms, the Gaussian mixture model (GMM) is known as a representative methodology for a wide use of data mining and computer vision. By probability density modeling, GMMs well perform the near-nature applications due to the soft clustering mechanism. For example, Reynold et al. [1] have constructed a speaker verification system using GMM to model the speaker's voice from speech data, and have succeeded in NIST speaker recognition evaluation [1]. Also, Stauffer et al. [2] 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 [2]. However, the parameters of a GMM are always estimated from 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.

The parameter estimation is one of critical issues to clustering quality. The variational Bayesian Gaussian mixture model (VB-GMM), in which the parameters are estimated through variational Bayesian theory [3], is widely considered as 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 [3]. Unfortunately, VB-GMM leads to the computation explosion. Much more iterations are necessary for convergence in contrast to most of other parameter estimation fashions [18]. Thus, CPU-oriented implementations of VB-GMM is unpractical in some application fields due to the 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 [6].

In this work, the GPGPU implementations of VB-GMM is developed based on the structure-oriented optimizations. Fitting the parallelism specification and memory structure of GPGPU, the VB-GMM optimization strategy is proposed by using the CPU-GPGPU co-operation, execution re-order, and memory optimization. An implementation flow with thirteen stages is presented in detail. Following this implementation flow, the VB-GMM which is usually executed on CPU is migrated onto GPGPU platform. Five types of real-world clustering tasks are verified by the proposed GPGPU implementation of VB-GMM including MNIST, CIFAR10, CIFAR100, PAMAP2, and home-area activity monitoring. From the experimental results, the VB-GMM is successfully executed on GPGPU platforms and achieves 192x speed-up compared to that by CPU. Moreover, it succeeded in finding the true number of clusters, which is difficult to do with EM-algotihm.

The rest of this paper is organized as follows. Section 2 provides an introduction to GMMs and variational Bayesian method for GMM and introduces previous work. The implementation of VB-GMM on GPGPU is explained in Sect. 3. Section 4 presents some evaluation of our implementation and the comparison of results with other implementations. The extended discussion is shown in Sect. 5. The conclusion is made in Sect. 6.

Manuscript received May 28, 2021.
Manuscript revised October 18, 2021.
Manuscript publicized November 24, 2021.

[†]The authors are with Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan.
a) E-mail: nishimoto.hiroki.na7@is.naist.jp

DOI: 10.1587/transinf.2021EDP7121

2. Preliminary

2.1 Structural Analysis of GPGPU

Graphics processing units (GPUs) are originally developed as the computing processors for the generation and display of 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 [17]. 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 speed up processing lies on parallelizing the instructions.

The architecture of GPUs supporting NVIDIA's CUDA is briefly shown in Fig. 1. This GPU consists of 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 [10]. 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 device memory; and then moved from the device memory to the main memory of the host device.

As shown in Fig. 2, the kernel of CUDA consists of Thread, 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 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 these 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.

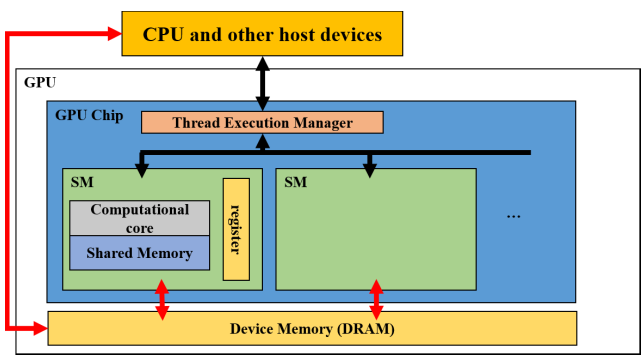


Fig. 1 Architecture of GPUs. The red arrows show the flow of data, and the black arrows show the flow of instructions.

2.2 Fundamentals of Gaussian Mixture Models

GMM is a probabilistic model for clustering as represented by Eq. (1).

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

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 3 shows an example of GMM, where the number of clusters and the dimension are 3 and 1, respectively.

2.3 Variational Inference for GMM

For adapting GMM to a data set, we have to optimize three parameters of π, μ, and Σ for all K clusters. The variational

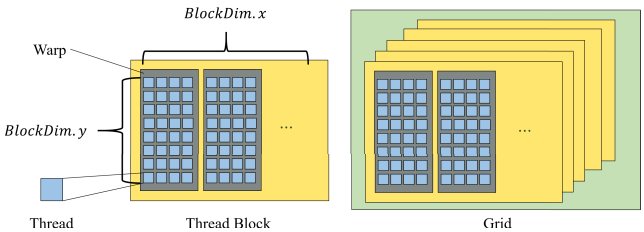


Fig. 2 Constitution of CUDA kernel. The kernel of a GPU consists of threads, blocks, which are collections of threads, and grids, which are collections of blocks. The actual number of threads running on the SM is 32, which is called the warp.

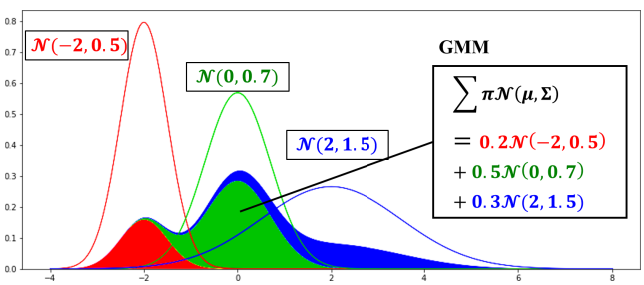


Fig. 3 Example of a GMM. This GMM consists of 3 weights (π) and 3 Gaussian distributions, and it is the sum of the 3 Gaussian distributions multiplied by π={0.2,0.5,0.3}.

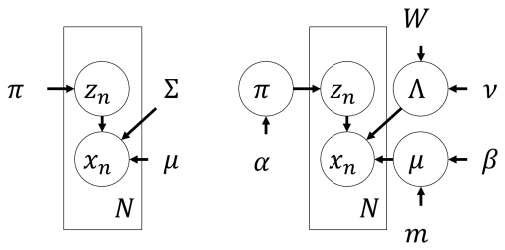


Fig. 4 Graphical models of two types of GMMs. Left one shows GMM with EM algorithm. Right one shows GMM with variational inference.

Bayesian method is a general strategy for parameter estimation, which is widely applied in 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 Eqs. (2), (3), and (4).

$$\pi \sim Dir(\alpha). \tag{2}$$

$$\mu \sim \mathcal{N}(m, (\beta\Lambda)^{-1}). \tag{3}$$

$$\Lambda \sim \mathcal{W}(W, \nu). \tag{4}$$

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 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 parameters of GMM by an algorithm, which is called variational-EM algorithm. This algorithm consists of four steps as shown below.

1. Initialization
   In this step, we initialize mean μ and precision Λ by data X, and also initialize mixture weight π by the number of clusters K, which is a hyper-parameter. Also we initialize α, β, W, ν, m by each hyper-parameter.
2. Variational E Step
   In this step, we estimate π, Λ. In addition, ψ represents the digamma function.

$$\ln \tilde{\pi} \equiv \mathbb{E}[\ln \pi_k] = \psi(\alpha_k) - \psi\left(\sum_{i=1}^{K}(\alpha_i)\right). \tag{5}$$

$$\begin{aligned}\ln \tilde{\Lambda}_k &\equiv \mathbb{E}[\ln |\Lambda_k|] \\ &= \sum_{d=1}^{D}\psi\left(\frac{\nu_k + 1 - d}{2}\right) + D\ln 2 + \ln |W_k|.\end{aligned} \tag{6}$$

   Also estimate correlation coefficient ρ by current parameters.

$$\rho_{nk} = \pi_k |\Lambda_k|^{1/2} \times \exp\left\{-\frac{D}{2\beta_k} - \frac{\nu_k}{2}(x_n - \mu_k)^T W_k (x_k - \mu_k)\right\}. \tag{7}$$

   At the end of the E step, normalize p to obtain the responsibility r by Eq. (8).

$$r_{nk} = \frac{\rho_n k}{\sum_{j=1}^{K}\rho_{nj}}. \tag{8}$$

3. Variational M Step
   In this step, we compute some parameters by updated parameters in step 2.

$$N_k = \sum_{n=1}^{N} r_{nk}. \tag{9}$$

$$\bar{x}_k = \frac{1}{N_k}\sum_{n=1}^{N} r_{nk} x_n. \tag{10}$$

$$S_k = \frac{1}{N_k} r_{nk}(x_n - \bar{x}_k)(x_n - \bar{x}_k)^T. \tag{11}$$

$$\alpha_k = \alpha_0 + N_k. \tag{12}$$

$$\beta_k = \beta_0 + N_k. \tag{13}$$

$$\nu_k = \nu_0 + N_k. \tag{14}$$

$$m_k = \frac{1}{\beta_k}(\beta_0 m_0 + N_k \bar{x}_k). \tag{15}$$

$$W_k^{-1} = W_0^{-1} + N_k S_k + \frac{\beta_0 N_k}{\beta_0 + N_k}(\bar{x}_k - m_0)(\bar{x}_k - m_0)^T. \tag{16}$$

4. Convergence check
   Lower bound $\mathcal{L}_{new}$ is computed by Eq. (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.

$$\begin{aligned}\mathcal{L} = &-\sum_{n=1}^{N}\sum_{k=1}^{K}(e^{r_{nk}} \times r_{nk}) \\ &- \sum_{k=1}^{K}\left(\nu_k |W_k| + \frac{\nu_k D \ln 2}{2} - \sum_{k=1}^{K}\ln\Gamma(\nu_k)\right) \\ &- \left(\ln\Gamma\left(\sum_{k=1}^{K}\alpha_k\right) - \sum_{k=1}^{K}\ln\Gamma(\alpha_k)\right) \\ &- \frac{D\sum_{k=1}^{K}\ln\beta_k}{2}.\end{aligned} \tag{17}$$

2.4 Previous Works

For related research on speeding up GMM, Guo et al. [4] 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. [5] implemented FPGA processing by making the pipeline processing of [4]

more efficient. Kumar et al. [6] 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. Implementation

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 [5] and [6]. The main kernel implementation is described below. Let N, K, D be number of data, number of clusters, number of dimension, respectively. In this section, the $O(x)$ means the order of computational complexity is x. In VB-GMM, the variational-EM algorithm described in Sect. 2 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 coalesce access for the minimum required accesses".

The shared memory is a memory structure that can be shared by threads 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 shared memory. Also, in programming for CUDA-enabled GPU architectures, a very important performance consideration is the coalescing of accesses to global memory. The loading and storing of global memory by the threads of the warp 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 compete for access, such as computing the sum of arrays, i.e., Eqs. (9) and (17) cannot be SIMD parallelized in VB-GMM.

3.1 Optimizing Memory Allocation

In CUDA programming, it is important to pay attention to the memory layout and make it easy for coalescing access. VB-GMM's variational inference uses many arrays to perform calculations, but we need to be careful about the arrays related to the number of data: N. In this case, we need to pay attention to the calculations in Eqs. (7), (9) etc. Particular attention should be paid to Eqs. (9) and (17), since they are the process of the summation of the elements of the number of data N. Therefore, in this implementation, we decided to place the data in such a way that the data size N is contiguous to the rows of the array. Figure 5 shows the difference in physical memory layout between arrays that are contiguous with respect to the number of data N and those that are not, using an array size of $N \times K$.

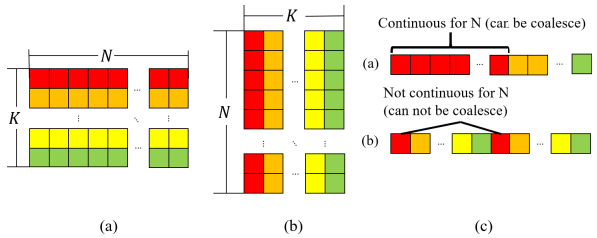


Fig. 5 (a) Contiguous with respect to the number of data N matrix. (b) Not contiguous with respect to the number of data N matrix. (c) Difference in physical memory layout between (a) and (b).

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 as well as the memory allocation described above, and to select the number of threads that can effectively use the shared memory in each block. In this study, we set the optimal number of threads per block ($BlockDim_x$, $BlockDim_y$) and blocks ($GridDim_x$, $GridDim_y$) for each kernel. Basically, 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 variable parameters such as number of data and 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 Sect. 2.3 is calculated in logarithm to prevent overflow, etc. Therefore, when moving from E step to M step, processes such as summation are difficult in logarithm, and must be exponentiated. When moving from E step to M step, we already have all the necessary values to calculate $\mathcal{L} = -\sum_{n=1}^{N}\sum_{k=1}^{K}(e^{r_{nk}} \times r_{nk})$ in Eq. (17) of Convergence Check shown in Sect. 2.3. Therefore, the calculation of this value and the exponentiation process can be combined to improve efficiency.

3.4 Details of Each Kernel

Based on the above, we have decomposed the parameter estimation of VB-GMM into 13 kernels and implemented them. Table 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. (5). Since the computational cost of this kernel is not high, it is computed on the

Table 1 Correspondence between the implemented kernel and the equations in the variational-EM algorithm.

Kernel	Corresponding equation
WEIGHT	Eq. (5)
LAMBDA	Eq. (6)
GAUSS	$\frac{D}{2\beta_k} - \frac{\nu_k}{2}(x_n - \mu_k)^T W_k(x_k - \mu_k)$ in Eq. (7)
WLP	Eq. (7)
LR	Eq. (8)
SR	$\sum_{n=1}^{N}\sum_{k=1}^{K}(e^{r_{nk}} \times r_{nk})$ in Eq. (17)
NEC	Eq. (9)
MEC	Eq. (10)
CEC	Eq. (11)
PRI	Eqs. (12), (13), and (14)
MEAN	Eq. (15)
PC	Eq. (16)
LB	Eq. (17)

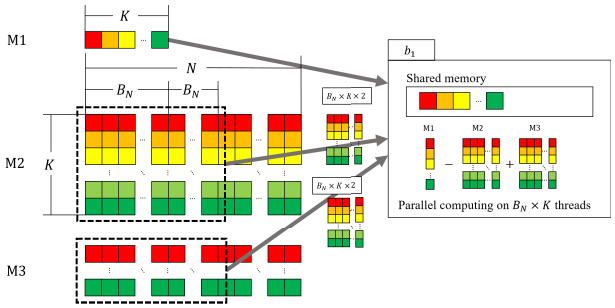


Fig. 6 Overview of the GAUSS kernel for the block with id of 1. M1, M2, and M3 represent the data stored in the global memory. Block b_1 fetches the data and executes Eq. (19) with $B_n \times K$ threads according to the value B_n set based on the coalescing. The part indicated by "shared memory" indicates the data copied to the shared memory on the block.

CPU.

2. Estimation of LAMBDA
   Estimation of the array LAMBDA is shown in Eq. (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. (18).

$$\begin{aligned}&\frac{D}{2\beta_k} - \frac{\nu_k}{2}(x_n - \mu_k)^T W_k(x_k - \mu_k)\\ &= -\frac{D}{2\beta_k} - \frac{\nu_k}{2}\left(\sum_{d=1}^{D}(\mu \circ \mu \circ W)_{(k,d)}\right.\\ &\quad \left. - 2x(\mu \circ W)^T + x \circ xW^T\right).\end{aligned} \tag{18}$$

   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, we calculate $-\frac{D}{2\beta_k} - \frac{\nu_k}{2}(M_1 - M_2 + M_3)$. The $a \circ b$ is used in the computation of M1, M2, and M3 represents 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, we use cublasSgemm from cuBLAS, a numerical computing library of CUDA. The $x \circ x$ use 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, we compute Eq. (19) in a thread with row thread id n and column thread id k that has been tuned for coalesce accessibility.

$$\mathrm{GAUSS}_{(n,k)} = M_{1(k)} - 2 \times M_{2(n,k)} + M_{3(n,k)}. \tag{19}$$

   In this calculation, the computation shown in Eq. (19) is performed twice instead of once per block, considering the trade-off between function call and memory access overhead. Figure 6 shows an overview of the calculation process for the block with 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 are calculated on the GPGPU, and the WEIGHT calculated on 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. (20).

$$\mathrm{WLP}_{(n,k)} = \mathrm{WEIGHT}_k + \frac{\mathrm{LAMBDA}_k}{2} + \mathrm{GAUSS}_{(n,k)}. \tag{20}$$

5. Estimation of log responsibility: LR
   Normalize the WLP to get array responsibility: LR in Eq. (21) to update parameters in Variational M step.

$$\mathrm{LR}_k = \mathrm{WLP}_k - \sum_{i=1}^{K}(\mathrm{WLP}_i). \tag{21}$$

   Here, we use a method that is called *logsumexp*, represent in Eq. (22) to prevent overflow and underflow.

$$\begin{aligned}&\log\left(\sum_{i=1}^{N}\exp(x_i)\right)\\ &= \log\left\{\exp(x_{max})\sum_{i=1}^{N}\exp(x_i - x_{max})\right\}\\ &= \log\left\{\sum_{i=1}^{N}\exp(x_i - x_{max}) + x_{max}\right\}.\end{aligned} \tag{22}$$

   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

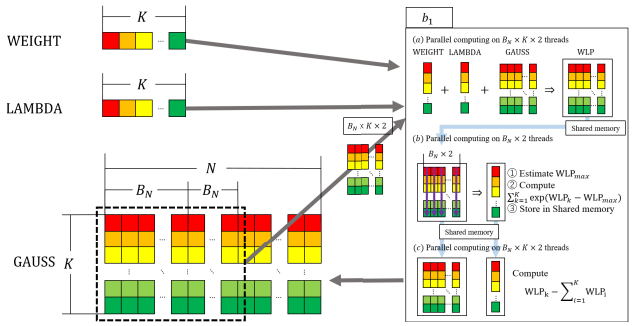


Fig. 7 Overview of the WLP and LR kernels for the block whose id is 1. GAUSS, WEIGHT, and LAMBDA represent the data stored in global memory. Block b_1 fetches the data according to the value B_n set based on the coalescing and executes Eqs. (20) and (21) with $B_n \times K$ threads.

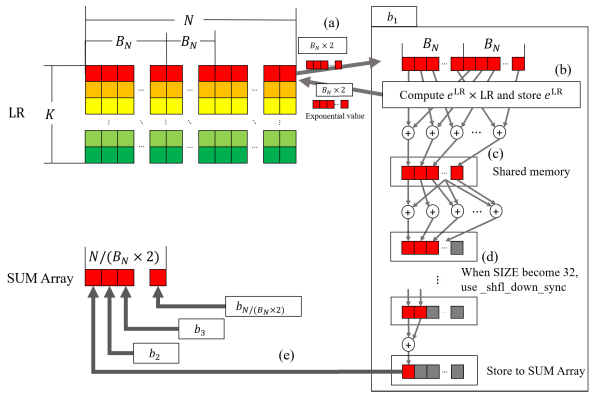


Fig. 8 Overview of the SR kernel for the block with id of 1. LR and SUM array represent the data stored in the global memory. Block b_1 fetches the data and executes Eq. (23) with $B_n \times K$ threads according to the value B_n set based on the coalescing. The part indicated by "shared memory" indicates the data copied to the shared memory on the block.

by making effective use of shared memory.

a. Compute WLP in each thread using Eq. (20), and store it in shared memory.
b. If the id of the threads in the column is 0, it will calculate $\sum_{k=1}^{K} \exp(\mathrm{WLP}_k - \mathrm{WLP}_{max})$ 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. (21), 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 function call and memory access overhead same as GAUSS. Figure 7 shows an overview of the calculation process for the block with id of 1.

6. Computation of Sum of Resp: SR
Calculate a value sum resp: SR, which is represented by Eq. (23), to calculate lower bound later.

$$\mathrm{SR} = -\sum_{n=1}^{N}\sum_{k=1}^{K}(e^{LR_{nk}} \times LR_{nk}). \tag{23}$$

In the Variational M step, we need exponentiated values of LR. But the calculation of lower bound needs, which is the next step of Variational M Step needs current LR value. Thus, we calculate SR before Variational M Step. As mentioned above, the process of finding the sum cannot be parallelized straight forward. This kernel was implemented as shown below, referring to Harris's method for speeding up the calculation of the sum of sequences [19].

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 t_{id}. Then, reduce $SIZE$ by half. This process is repeated until $SIZE$ becomes 32.
d. To make the addition even faster, we use shfl_down_sync, a function that can refer to the same warp value without using shared memory.
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 8 shows an overview of the calculation process for the block with id of 1.

7. Estimation of Num of Each Cluster: NEC
This kernel compute values of array NEC according to Eq. (9). Since this kernel is a summation process, as in SR, and cannot be parallelized in a straightforward way, we adopted a method to speed up the calculation of the sum of Harris sequences, which was also use in SR.
8. Estimation of Mean of each cluster: MEC
Computation of matrix MEC is represented by Eq. (10). This calculation can be replaced by using data: X and LR and NEC in Eq. (24).

$$\mathrm{MEC} = \mathrm{LR}^T X/(\mathrm{NEC}). \tag{24}$$

In this kernel, we use cublasSgemm same as GAUSS.

9. Estimation of Covariance of Each Cluster: CEC
Computation of matrix CEC is represented by Eq. (11) and this calculation can replace Eq. (25) if the GMM using diagonal covariance.

$$S = \frac{rx \circ x}{N_k} - 2\frac{\bar{x} \circ rx}{N_k} + \bar{x} \circ \bar{x}. \tag{25}$$

Therefore, we can compute $\mathrm{LR}^T(X \circ X)$ and $\mathrm{LR}^T X$ by

Table 2 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 @ 3.30 GHz		
GPGPU	GeForce RTX3090		N/A
Memory	256 GB		
Compiler	NVIDIA (R) Cuda compiler driver V11.2.142		gcc version 4.8.5

using cublasSgemm and compute them as Eq. (25).

10. Estimation of α, β, ν: PRI
    The calculation cost of these values shown in Eqs. (12), (13), and (14) are not high. So we did the calculations on the CPU.
11. Estimation of MEAN
    Computation of prior of mean: MEAN is represented in Eq. (15). We also calculate MEAN on the CPU. Because, next calculation, which is for the precision matrix, needs x_k value.
12. Estimation of Precision Cholesky: PC
    Computation of prior of precisions is represented by Eq. (16). After computing it, substitute its square root into PC. Also, we calculate log determinant values of PC for computation of lower bound and 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, computation of lower bound which is functioned as the indicator computed by Eq. (17). Sum resp and log determinant precision cholesky have already been calculated. The calculation cost of rest values negligible. So, we compute them on CPU.

3.5 Optimizing Data Transfer and CPU-GPGPU Co-Operation Scheme

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 conduct experiments to evaluate the comparison between the CPU implementation of VB-GMM and the GPGPU implementation of the EM algorithm.

The CPU implementation of VB-GMM is a sequential execution type implementation in C language, using OpenBLAS for matrix product. The GPU implementation of the EM algorithm was reproduced with a parallel execution model implemented in CUDA based on the kernel presented by Kumar et al. [6]. Although the NEC summation process in Sect. 3 is not presented in Kumar et al.'s implementation, we diverted the NEC implementation because it is a common process between the EM algorithm and VB-GMM and may be a bottleneck.

The experiment is divided into two parts. One is 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 Table 2.

4.1 Kernel Processing Time Compared to CPU

We compare the processing time for one iteration of variational-EM algorithms in the VB-GPU and VB-CPU for each of the kernels shown in Sect. 3 to determine which kernel is faster and to confirm the validity of our experimental results. In VB-GPU, we use NVIDIA Visual Profiler, Chrono library, CUDA Runtime API to get percentages. In VB-CPU, we use time functions. For the kernels running in parallel on the GPU and CPU in Sect. 3.5, the two are described together, and the kernel running on the GPU is taken as its execution time.

In this evaluation, we use artificially created data with 32, 16, and 1e+06 clusters, dimensions, and data, respectively, ran the variational-EM algorithm 10 times, measured the execution time of each kernel, and divided it by 10 to calculate the execution time per run for each kernel. Table 3 shows a comparison of the breakdown of the execution time of one variational-EM algorithm for the two implementations. From Table 3, we can see that the kernel, which accounts for most of the execution time in the CPU implementation, has been greatly accelerated, and speedup has been achieved.

4.2 Kernel Processing Time Compared to OpenCL

CUDA is one of most useful programming model for GPGPU. But its portability is low. Therefore, in addition to comparing the kernels of each CPU, we also created an additional OpenCL implementation of the kernel that had a sig-

Table 5 Evaluation items and their summary.

Evaluation item	Summary
Time	The time it took for each implementation to converge. Unit: millisecond
DB's score	Davis-Bouldin Score: DB's score [9] 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. We computed the DB's index of result of clustering, using the metrics library of scikit-learn [7].
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.

Table 3 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 GAUSS	0.0059 720.4222	2.1938	328.39
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 PRI	108.8799 0.0016	0.4742	229.59
CEC MEAN	128.9425 0.0055	0.9323	138.31
PC	0.0151	0.0213	0.71
LB	0.1013	0.0931	1.09

Table 4 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

nificant parallelization effect in the aforementioned experiment and compared it 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 [8], the arithmetic library for OpenCL, clblasSgemm. Table 4 shows a comparison of CUDA and OpenCL for the breakdown of execution time for each kernel in one iteration. From Table 4, we can see that the execution speed of CUDA is fast for many major kernels. Also, all major kernels are significantly faster than the CPU, even in implementations using OpenCL.

4.3 Evaluation with Artificial Data

This 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 Davis-Bouldin Score (DB's Score) to evaluate the quality of clustering. All evaluation items and their contents are shown in Table 5.

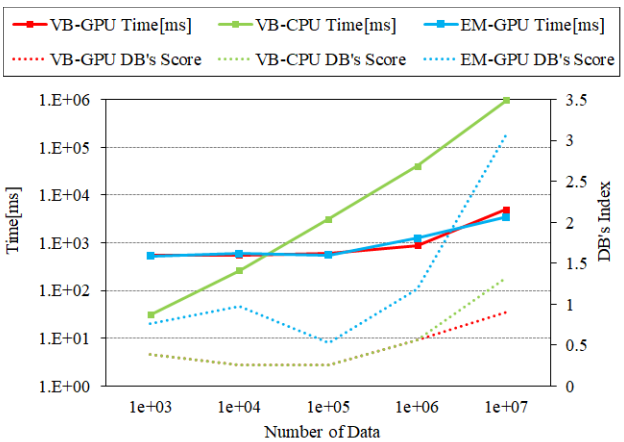


Fig. 9 Execution time and DB's index depend on the number of data. (The number of clusters = 32, The number of dimensions = 16)

4.3.1 Varying Number of Data

The number of dimensions and the number of clusters are fixed to 16 and 32, respectively. The number of data is varied from 1e+03 to 1e+07. The experimental results are shown in Table 6, and the extracted time and DB's score are shown in Fig. 9.

From Fig. 9, we can see that the execution time of the CPU implementation increases as the number of data increases, while the execution speed of the two GPU implementations, VB-GPU and EM-GPU, remains flat until the number of data exceeds 1e+06. We can also see that VB-CPU and VB-GPU using variational inference have better DB's Score than the implementation using EM algorithm. Also, Table 6 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 Sect. 1, is reflected. When the number of data was 1e+07, VB-GPU is successfully executed achieves 192x speed-up compared to VB-CPU.

From Fig. 9 and Table 6, 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 is 1e+07,

Table 6 Result of evaluation with varying the number of data.

Number of Data	Time[ms]			DB's 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

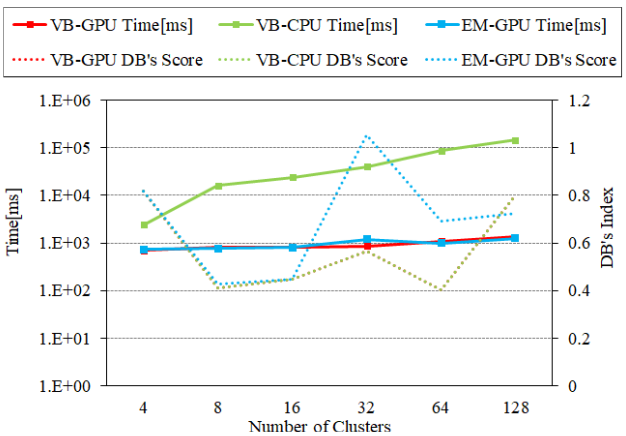


Fig. 10 Execution time and DB's index depending on the number of clusters. (The number of data = 1e+06, The number of dimensions = 32)

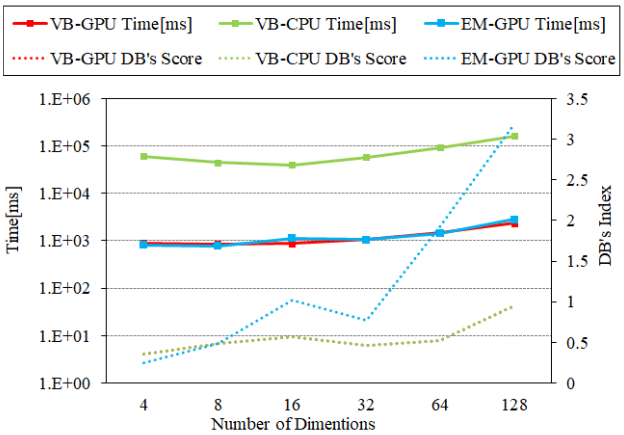


Fig. 11 Execution time and DB's index depending on the number of clusters. (The number of data = 1e+06, The number of clusters = 32)

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.

4.3.2 Varying Number of Cluster

The number of dimensions and the number of data fixed to 16 and 1e+06. The number of clusters is varied from 4 to 128. The experimental results are shown in Table 7, and the extracted time and DB's score are shown in Fig. 10.

From Fig. 10, we can see that the execution time of the CPU implementation increases as the number of cluster increases, while the execution speed of the two GPU implementations, VB-GPU and EM-GPU. We can also see that VB-CPU and VB-GPU using variational inference have better DB's Score than the implementation using EM algorithm. Also, Table 7 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 Sect. 4.3.1. When the number of cluster was 128, VB-GPU is successfully executed achieves 107x speed-up compared to VB-CPU.

4.4 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 Table 8, and the extracted time and DB's score are shown in Fig. 11. Figure 11 and Table 8 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 dimension was 128, VB-GPU is successfully executed achieves 69x speed-up compared to VB-CPU.

4.5 Evaluation with Practical Data

This evaluations use open data. Table 9 shows an outline of the data. Evaluation items were time to converge and DB's index same as Sect. 4.3. In addition, we also evaluated Ajusted Random Index (ARI) [15]. ARI is a function that measures the similarity of the two assignments, ignoring permutations and with chance normalization. If the labels are randomly assigned, this index will be close to 0.0, and if the labels match exactly, this index will be 1.0. We computed the ARI of the ground truth and clustering results of the dataset, using metrics library of scikit-learn [7]. We also aligne each cluster to one label by majority vote of the members and used that class to calculate the alignenment

Table 7 Result of evaluation with varying the number of cluster.

Number of Cluster	Time[ms]			DB's 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

Table 8 Result of evaluation with varying the number of dimensions.

Number of dimensions	Time[ms]			DB's 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

Table 9 Outline of practical data.

Name	Number of data	Number of classes	Number of dimensions	Overview
MNIST [11]	60000	10	64	28 x 28 Handwritten digits.
CIFAR10 [12]	60000	10	64	32 x 32 color images in 10 classes.
PAMAP2 [14]	376417	13	52	Physical Activity Monitoring dataset.
Gas sensors for home activity monitoring Data Set: GDS [13]	919438	3	11	Recordings of a gas sensor array composed of 8 MOX gas sensors,and a temperature and humidity sensor.

Table 10 Evaluation result with practical data sets.

Data Set	Init	Time[ms]		DB's Score		ARI		Alignment Accuracy		Conv. Iter.		Conv. Cluster		Log Likelihood	
		VG	EG	VG	EG	VG	EG	VG	EG	VG	EG	VG	EG	VG	EG
MNIST	16	807	660	3.62	3.25	0.138	0.003	0.48	0.14	62	43	16	16	54.2	112.0
	32	1246	833	3.39	6.75	0.168	0.034	0.63	0.22	95	58	32	29	56.7	121.0
	64	2237	914	3.10	6.64	0.134	0.031	0.70	0.18	118	45	52	25	58.1	120.0
	128	2878	1713	2.94	6.35	0.120	0.049	0.72	0.20	83	77	63	71	59.1	127.3
CIFAR10	16	758	727	24.46	N/A	0.015	0.000	0.15	0.10	50	59	5	12	140.1	284.2
	32	844	917	39.94	0.62	0.028	0.000	0.16	0.10	46	98	5	18	142.3	304.1
	64	836	970	22.22	0.54	0.005	0.000	0.13	0.10	23	66	3	16	133.9	291.0
	128	964	1216	39.73	2.82	0.026	0.000	0.16	0.10	19	61	3	23	143.4	296.1
PAMAP2	16	917	812	3.90	3.24	0.294	0.377	0.71	0.68	37	23	16	16	47.6	44.6
	32	1594	1324	3.24	3.43	0.226	0.275	0.73	0.73	79	76	32	32	53.9	53.0
	64	1581	1835	3.02	3.39	0.126	0.192	0.77	0.78	41	81	64	64	61.1	63.6
	128	2941	2423	3.09	3.12	0.091	0.126	0.86	0.85	52	66	126	127	67.5	71.0
GDS	16	748	796	3.23	2.39	0.057	0.064	0.57	0.58	27	27	16	16	-4.8	-0.9
	32	1271	1129	3.05	2.26	0.045	0.045	0.63	0.62	57	47	32	32	-2.9	1.9
	64	1606	1278	2.61	2.13	0.026	0.040	0.67	0.66	43	35	64	64	-0.9	4.6
	128	2532	2700	2.22	1.76	0.020	0.025	0.76	0.71	41	61	126	125	1.7	8.1

accuracy as in the experiment in [16].

4.5.1 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. We preprocess as in [16]. Each image down to number of dimensions = 64 via PCA and normalization each value 0 to 1 using preprocessing library of scikit-learn.

As shown in Table 9, we can see that regardless of the initial number of clusters, VB-GPU has higher ARI and alignment accuracy.

4.5.2 CIFAR10

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes. We attempt value-based clustering of the pixels to see if the 10 classes could be clustered. We preprocess each image down to number of dimensions = 64 via PCA and normalize each pixel value 0 to 1 using preprocessing library of scikit-learn.

From the results in Table 9, we can see that VB-GPU has higher ARI and Alignment Accuracy than EM-GPU, similar to the experiment at MNIST. However, both ARIs

are low, indicating that the accuracy as clustering is not good.

4.5.3 PAMAP2

The PAMAP2 (Physical Activity Monitoring) dataset contains data of 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. As preprocessing, we removed record of time, and standardized using preprocessing library of scikit-learn.

From Table 9, we can see that the values of most of the results are close for both implementations, and the ARI and Alignment Accuracy are high, indicating that both implementations are able to cluster well.

4.5.4 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. As preprocessing, we removed record of time, and standardized using preprocessing library of scikit-learn. Table 9 shows the two implementations did not make much difference.

5. Discussion

Firstly, we discuss the difference between VB-GPU and VB-CPU. Section 4.1 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 Sect. 4.3.1, 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 Sect. 4.3.2, we confirmed that VB-CPU increases the computation time as the number of clusters increases, but VB-GPU can suppress it. In the experiments in Sect. 4.4, 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.

Finally, we discuss the difference between VB-GPU and EM-GPU. We can see that in all experiments using artificial data in Sects. 4.1, 4.2, and 4.3, VB-GPU achieves the same execution time as EM-GPU and VB-GPU achieves a better DB's 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's Score, indicating that it is able to perform good clustering while avoiding over-fitting the data.

In the experiments using practical data, ARI and alignment accuracy of VB-GPU are higher and Log likelihood of VB-GPU is lower than EM-GPU for all initial cluster values in the experiments with MNIST. This is due to the feature of Bayesian inference that suppresses degeneracy by introducing a prior distribution. In the experiment with CIFAR10, this phenomenon can be clearly confirmed because the ARI of EM-GPU becomes zero due to degeneracy, while the ARI of VB-GPU does not become zero. Experiments with PAMAP2 and Gas sensors for home activity monitoring DataSet showed comparable results for both, and did not confirm the superiority of VB-GPU. In the experiment using the practical data, we were not able to confirm the phenomenon that the implementation obtains a number close to the true number of clusters using variational inference at convergence, which was confirmed in the experiment using the artificial data. This is thought to be due to differences in data complexity.

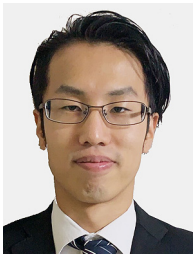
6. Conclusion

The Gaussion mixture model based on variational Bayesian estimation is implemented on GPGPU for the high speed clustering applications. Employing the proposed strategies including CPU-GPU co-optimization, execution re-order and memory management, the VB-GMM is efficiently conducted by GPGPU with high parallelism. Various data-sets for real-world clustering applications are introduced for validations.

From the experimental results, the convergence is generally faster than the same algorithm implemented on CPU; and the comparable convergence scores are achieved. As a typical example, the proposed VB-GMM on GPGPU is 192x faster than the CPU when the number of data is 1E+07, and 107x faster when the number of clusters is 128. Compared with the state-of-art GPGPU implementations conducted by the EM algorithm, the proposed VB-GMM on GPGPU is able to suppress degeneracy even on data sets where the EM algorithm would have degenerated; fair performances over speed, clustering scores, clustering distributions are achieved.

References

[1] D.A. Reynolds, T.F. Quatieri, and R.B. Dunn, "Speaker verification using adapted Gaussian mixture models," Digital Signal Process, vol.10, no.1-3, pp.19–41, 2000.
[2] C. Stauffer and W.E.L. Grimson, "Adaptive background mixture models for real-time tracking," IEEE Computer Society Conference on Computer Vision and Pattern Recognition, vol.2, pp.246–252, 1999.
[3] A. Corduneanu and C.M. Bishop, "Variational Bayesian model selection for mixture distribution," 2001 Artificial Intelligence and Statistics, pp.27–34, 2001.
[4] C. Guo, H. Fu, and W. Luk, "A fully-pipelined expectation-maximization engine for Gaussian mixture models," 2012 International Conference on Field-Programmable Technology, pp.182–189, 2012.
[5] C. He, H. Fu, C. Guo, W. Luk, and G. Yang, "A fully-pipelined hardware design for Gaussian mixture models," IEEE Trans. Comput., vol.66, no.11, pp.1837–1850, 2017.
[6] N.S.L.P. Kumar, S. Satoor, and I. Buck, "Fast parallel expectation maximization for Gaussian mixture models on GPUs Using CUDA," 2009 11th IEEE International Conference on High Performance Computing and Communications, Seoul, pp.103–109, 2009.
[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay, "Scikit-learn: Machine learning in Python," Journal of Machine Learning Research, vol.12, no.85, pp.2825–2830, 2011.
[8] clBLAS, [Online]. Available: https://github.com/clMathLibraries/clBLAS
[9] D.L. Davies and D.W. Bouldin, "A cluster separation measure," IEEE Trans. Pattern Anal. Mach. Intell., vol.PAMI-1, no.2, pp.224–227, 1979.
[10] NVIDIA, "NVIDIA AMPERE GA102 GPU ARCHITECTURE," [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf
[11] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.
[12] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.
[13] R. Huerta, T. Mosqueiro, J. Fonollosa, N. Rulkov, and I. Rodriguez-Lujan, "Online decorrelation of humidity and temperature in chemical sensors for continuous monitoring," Chemometrics and Intelligent Laboratory Systems, vol.157, pp.169–176, 2016.
[14] A. Reiss and D. Stricker, "Introducing a new benchmarked dataset for activity monitoring," The 16th IEEE International Symposium on Wearable Computers (ISWC), pp.108–109, 2012.
[15] L. Hubert and P. Arabie, "Comparing partitions," Journal of Classification, vol.2, pp.193–218, 1985.
[16] M.C. Hughes and E.B. Sudderth, "Memoized online variational inference for Dirichlet process mixture models," Advances in Neural Information Processing Systems, 2013.
[17] J. Nickolls and W.J. Dally, "The GPU computing era," IEEE Micro, vol.30, no.2, pp.56–69, 2010.
[18] C.M. Bishop, Pattern Recognition and Machine Learning, pp.474–486, Springer, New York, 2006.
[19] M. Harris, "Optimizing parallel reduction in cuda," [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf



Hiroki Nishimoto received his M.E. from Nara Institute of Science and Technology in 2020. He has been an doctor student in Nara Institute of Science and Technology now. His research interests include computer architecture. He is a student member of IEICE.

Renyuan Zhang received M.E. and Ph.D. degrees from Waseda University and the University of Tokyo in 2010 and 2013, respectively. He was an assistant professor with Japan Advanced Institute of Science and Technology from 2013 to 2017. He has been an assistant professor and associate professor with Nara Institute of Science and Technology since 2017 and 2021, respectively. His research interests include analog-digital-mixed circuits and approximate computing. He is a member of IEEE and IEICE.

Yasuhiko Nakashima received B.E., M.E., and Ph.D. degrees in Computer Engineering from Kyoto University in 1986, 1988 and 1998, respectively. He was a computer architect in the Computer and System Architecture Department, FUJITSU Limited from 1988 to 1999. From 1999 to 2005, he was an associate professor in the Graduate School of Economics, Kyoto University. Since 2006, he has been a professor in the Graduate School of Information Science, Nara Institute of Science and Technology. His research interests include computer architecture, emulation, circuit design, and accelerators. He is a fellow of IEICE, a senior member of IPSJ, a member of IEEE CS and ACM.