



CPU- and GPU-Based Distributed Sampling in Dirichlet Process Mixtures for Large-Scale Analysis

Or Dinari 

Ben-Gurion University

Raz Zamir

Ben-Gurion University

John W. Fisher III 

Massachusetts Institute of Technology

Oren Freifeld 

Ben-Gurion University

Abstract

In the realm of unsupervised learning, Bayesian nonparametric mixture models, exemplified by the Dirichlet process mixture model (DPMM), provide a principled approach for adapting the complexity of the model to the data. Such models are particularly useful in clustering tasks where the number of clusters is unknown. Despite their potential and mathematical elegance, however, DPMMs have yet to become a mainstream tool widely adopted by practitioners. This is arguably due to a misconception that these models scale poorly as well as the lack of high-performance (and user-friendly) software tools that can handle large datasets efficiently. In this paper we bridge this practical gap by proposing a new, easy-to-use, statistical software package for scalable DPMM inference. More concretely, we provide efficient and easily-modifiable implementations for high-performance distributed sampling-based inference in DPMMs where the user is free to choose between either a multiple-machine, multiple-core, central-processing-unit (CPU) implementation (in Julia) and a multiple-stream graphics-processing-unit (GPU) implementation (in CUDA/C++). Both the CPU and GPU implementations come with a common (and optional) Python wrapper, providing the user with a single point of entry with the same interface. On the algorithmic side, our implementations leverage a leading DPMM sampler from [Chang and Fisher III \(2013\)](#). While Chang and Fisher III's implementation (in MATLAB/C++) used only CPU and was designed for a single multi-core machine, the packages we proposed here distribute the computations efficiently across either multiple multi-core machines or across multiple GPU streams. This leads to speedups, alleviates memory and storage limitations, and lets us fit DPMMs to significantly larger datasets and of higher dimensionality than was possible previously by either [Chang and Fisher III \(2013\)](#) or other DPMM methods.

Keywords: Dirichlet process mixtures, sampling, clustering, GPU, C++, CUDA, Julia, Python.

1. Introduction

In unsupervised learning, Bayesian nonparametric (BNP) mixture models, exemplified by the Dirichlet-process mixture model (DPMM), provide a principled approach for Bayesian modeling while adapting the model complexity to the data. This contrasts with finite mixture models whose complexity is determined manually or via model-selection methods. To fix ideas, an important DPMM example is the Dirichlet-process Gaussian mixture model (DPGMM), a Bayesian ∞ -dimensional extension of the classical Gaussian mixture model (GMM). Despite their potential, however, and although researchers have used them successfully in numerous applications during the last two decades, DPMMs still do not enjoy wide popularity among practitioners, largely due to computational bottlenecks that exist in current algorithms and/or implementations. In particular, one of the missing pieces is the availability of software tools that: 1) can efficiently handle DPMM inference in large datasets; 2) are user-friendly and can also be easily modified.

We argue that in order *for DPMMs to become a practical choice for large-scale data analysis, implementations of DPMM inference must leverage parallel- and distributed-computing resources* (in an analogy, consider how advances in graphics-processing-unit (GPU) computing and GPU software contributed to the success of deep learning). This is because of not only potential speedups but also memory and storage considerations. For example, this is especially true in distributed mobile robotic sensing applications where multiple autonomous agents working together have limited computational and communication resources. As another motivating example, consider unsupervised data-analysis tasks in large and high-dimensional computer-vision datasets.

In other words, while DPMMs are theoretically ideal for handling unlabeled datasets, current implementations of DPMM inference do not scale well with the size of the dataset and/or the dimensionality. This is partly since most existing implementations are serial and they do not harness the power of distributed computing. This does not mean that there do not exist distributed *algorithms* for DPMM inference. *There is, however, a large practical gap between designing such an algorithm and having it implemented efficiently in a way that fully utilizes the available computing resources.* Thus, the very few publicly-available implementations of such distributed algorithms are fairly limited in their capabilities (as well as their expressiveness; e.g., some support only isotropic Gaussians Wang and Lin 2017, etc.). Our work closes this gap by providing effective and scalable statistical software for typical large-scale inference.

Concretely, we propose two solutions from which the users can choose according to their needs, constraints, and available computing resources. The first proposed solution, a purely central-processing-unit (CPU) implementation, is based on distributing computations across multiple cores as well as multiple machines. The second proposed solution, based mostly on GPU, relies on distributing computations across multiple GPU streams in a single machine. See Table 1 for more details.

More generally (than the topic of DPMM inference), distributed implementations, at least in traditional programming languages used for such implementations, tend to be hard to debug, read, and modify. This clashes with the usual workflow of algorithm development. As a remedy, in our recent workshop paper (Dinari, Yu, Freifeld, and Fisher III 2019), which constitutes a preliminary and partial version of this paper, we proposed a Julia implementation for distributed sampling-based DPMM inference. Since the publication of (Dinari *et al.* 2019) we have improved the performance of our Julia implementation, have added its GPU

Package	Processor	Description
CUDA/C++	GPU	The fastest package for high N (number of data points) and d (data dimensions) on a single machine; supports multiple GPU streams.
Julia	CPU	Supports both multiple cores and multiple machines.
Python	Either CPU or GPU	Wrapper to the CUDA/C++ and Julia packages

Table 1: Overview of the proposed packages.

counterpart in CUDA/C++, and added an optional Python wrapper for both the CPU and GPU implementations.

To summarize, in this paper we explain how to use, via either Julia or CUDA/C++, an efficient distributed DPMM inference for large-scale analysis. Particularly, based on a leading parallel Markov chain Monte Carlo (MCMC) inference algorithm (Chang and Fisher III 2013) (to be discussed in Section 3) – originally implemented in C++ for a single multi-core CPU single machine in a highly-specialized fashion using a shared-memory model – we provide novel, more scalable, and easier-to-use-or-modify implementations that leverage either the latest Nvidia’s asynch memory allocation API for GPU or Julia’s capabilities for distributing CPU computations efficiently across multiple multi-core machines using a distributed memory model. This leads to speedups, alleviates memory and storage limitations, and lets us infer DPMMs from significantly larger and higher-dimensional datasets than was previously possible by either Chang and Fisher III (2013) or other DPMM inference methods. Our Julia and CUDA/C++ implementations are also accompanied by an optional Python wrapper which hides the Julia & CUDA/C++ code from the user and provides a single point of entry that lets the user employ, in the same settings and with the same code, either one of our CPU and GPU packages.

2. Demonstration of the model and software

Our Julia & CUDA/C++ packages provide a flexible and scalable implementation for DPMM inference. While full details about the model and software appear later in Section 3 and Section 4, respectively, in this section we provide a quick demonstration of how to use our code. Concretely, we show how our code is used to cluster the MNIST dataset (LeCun, Cortes, and Burges 1998). The latter is a well-known image dataset of handwritten digits. Moreover, and as mentioned below, we provide replication scripts that can be easily run in order to reproduce our results.

Our demonstration here includes a comparison between our two packages on one hand and two methods from `sklearn` (Pedregosa *et al.* 2011) on the other hand. As standard preprocessing, the dimensionality of the MNIST 28×28 grayscale images is first reduced to 32 using principal component analysis (PCA). Then, each of the four implementations is run on the 32-dimensional data. All this is done using the following Python code:

```
dim = 32
run_test(n_samples = 60000, d = dim, k = 10, numIter = 10, max_iter = 300,
        model = 'mnist', get_data = generate_mnist_data,
        prior = DPMPython.create_niw_prior(dim, 1.46, 456.8))
```

In the function call above, the `n_samples` argument is the number of points (i.e., MNIST images), `d` is the (reduced) dimension of the data, and the optional argument `k` is the expected number of clusters. Importantly, note that since the algorithm we use finds the number of clusters by itself, our packages do not, in fact, need that argument. However, **sklearn** does need it, so we include that argument in the interface only for the purpose of comparing our implementations to **sklearn**. We stress that our implementations make no use of this argument. The optional argument `get_data` is a helper to provide the data in the structure that the internal code expects. If the user already has prepared his or her data in the expected form then `get_data` is not required. As an example, above we used the `generate_mnist_data` method which returns the data and the ground-truth labels (used only for evaluation of the performance) as follows:

```
def generate_mnist_data(n_samples, d, k):
    data = np.load('mnist_images.npy')
    pca = PCA(n_components = d)
    data = pca.fit(data).transform(data)
    data = data - data.mean(axis = 0)
    data = data / data.std(axis = 0)
    data = np.swapaxes(data, 0, 1)
    gt = np.load('mnist_labels.npy')
    return data, gt
```

In our demonstration here we used the MNIST training set which contains 60,000 images. The number of features is $784 = 28 \times 28$ (pixels). We downloaded the mnist dataset by this code:

```
mnist = tf.keras.datasets.mnist
(train_images, train_labels), _ = mnist.load_data()
```

We reshaped the MNIST data to an array of size 60000×784 . We saved the images and the labels separately to two different files, `mnist_images.npy` and `mnist_labels.npy`, respectively. We performed a standard PCA on the data from the `mnist_images.npy` file. The number of principal components that we used is equal to the dimension provided by the user (e.g., in this example we used 32).

The `prior` object in the `run_test` method is defined based on the desired type of mixture components (e.g., Gaussian or Multinomial). In this example, the two values that we used for the prior's hyperparameters, 1.46 and 456.8, were found empirically using black-box optimization. That said, the user is of course free to configure the prior using different values. We will explain more about the (standard) priors and their hyperparameters in Section 3.

Next, we ran 4 different packages on the resulting dataset. Our Julia & CUDA/C++ packages and the two **sklearn** packages GaussianMixture and BayesianGaussianMixture. While GaussianMixture supports only inference for a mixture model with a finite number of Gaussians, BayesianGaussianMixture supports the broader case where the number of Gaussians is unknown.

We repeated these 4 tests 10 times according to the `numIter` parameter.

Internally, in each run we called the `fit` method of the relevant package which fits a model to the data. The key input arguments to our `fit` method appear in the function definition below:

```
def fit(data, alpha, prior = None,
        iterations = 100, verbose = False,
        burnout = 15, gt = None, gpu = True)
```

The `data` argument is a standard dataset. In this case it is the mnist dataset. The `alpha` argument is the DPMM concentration parameter. In this case we used the default value which is 1. The `gpu` argument defines whether to run the Julia or the CUDA/C++ package.

We collected all the results to a **pandas** DataFrame and at the end of all the iterations we saved it to csv files. The file `run_result_mnist_60000_32_10.csv` contains a detailed results per iteration. We did not have a way to get the time and normalized mutual information (NMI) from **sklearn** per iteration so all the rows reflect the final results. For our Julia & CUDA/C++ packages we provided detailed information as an output parameter. In the csv file each row represents a different iteration. Following is an illustration of one iteration (number 299) from the csv file.

#	NMI				Time			
	Cuda	Julia	BGM	GM	Cuda	Julia	BGM	GM
299	0.725	0.755	0.633	0.643	0.224	0.341	917.556	40.674

From these results we clearly see that our methods are faster than those of **sklearn** and achieve better performance in terms of NMI (a standard performance index for clustering).

For the final results we created 2 additional csv files, one for NMI and one for time, which summarize the results per model. For this demonstration, the NMI results are in `run_result_all_mnist_NMI_table.csv`:

Params	Cuda	Julia	Sklearn_GM	Sklearn_BGM	X	D	K
60000_32_10	0.712	0.722	0.632	0.622	60000	32	10

And the time results are in `run_result_all_mnist_time_table.csv`:

Params	Cuda	Julia	Sklearn_GM	Sklearn_BGM	X	D	K
60000_32_10	56.075	111.827	40.674	917.556	60000	32	10

Note it is unsurprising that `Sklearn_GM` was the fastest since it was given the true number of clusters and did not infer itself. Interestingly, however, our packages still beat it in terms of performance, despite the fact they were not given that value. This is largely because of the large moves (in terms of optimization) performed by the algorithm we implemented. The results of the comparison above will be discussed in more detail in Section 6.

Finally, in our Python GitHub repository, we also provided 3 replications scripts:

- `replication_short.py` for short run where only the MNIST dataset is used (takes about 20 minutes).
- `replication_full.py` for the entirety of the tests that we did on all of the datasets (this takes several days, partially since we performed many tests and comparisons, and partially since most of running time is due to the slowness of methods we compared to).
- `replication.py` that is being called by the two other scripts and is, in fact, the main replication script.

3. Models and the inference algorithm

3.1. Preliminaries: Finite mixture models and clustering

Let d and K be two positive integers and let \mathbf{x} be a generic point in \mathbb{R}^d . A K -component finite mixture model (FMM), also known as a parametric mixture model, is a probabilistic model whose associated d -dimensional probability density function (pdf) is

$$p(\mathbf{x}; \theta) = \sum_{k=1}^K \pi_k f(\mathbf{x}; \theta_k) \quad (1)$$

where $\theta = (\pi_k, \theta_k)_{k=1}^K$, $(\pi_k)_{k=1}^K$ form a convex combination (namely, $\sum_{k=1}^K \pi_k = 1$ and $\pi_k \geq 0 \forall k$), and $f(\mathbf{x}; \theta_k)$ is a d -dimensional pdf parameterized by θ_k . The $(f(\cdot; \theta_k))_{k=1}^K$ functions are called the mixture *components* while $(\pi_k)_{k=1}^K$ are called the mixture *proportions* (or *weights*).

Example 1 In the (finite) Gaussian mixture model (GMM), $\theta_k = (\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ and $f(\mathbf{x}; \theta_k) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ where the latter is a multivariate Gaussian pdf evaluated at \mathbf{x} and parameterized by a mean vector, $\boldsymbol{\mu}_k \in \mathbb{R}^d$, and a symmetric positive-definite (SPD) $d \times d$ covariance matrix, $\boldsymbol{\Sigma}_k$. In other words, in this example

$$f(\mathbf{x}; \theta_k) = \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \triangleq (2\pi)^{-d/2} (\det \boldsymbol{\Sigma}_k)^{-1/2} \exp(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1} (\mathbf{x} - \boldsymbol{\mu}_k)). \quad (2)$$

The case where \mathbf{x} takes values in a discrete set is similar, except that each component is a probability mass function (pmf), not a pdf. A popular example is a mixture of categorical distributions or, more generally, multinomial distributions.

Let $\mathbf{X} = (\mathbf{x}_i)_{i=1}^N$ stand for N data points and again let $K > 0$ be an integer. *Clustering* is the task of partitioning \mathbf{X} into K parts, called *clusters* and denoted by $\mathbf{C} = (C_k)_{k=1}^K$. Let z_i denote the latent point-to-cluster assignment of \mathbf{x}_i ; i.e., cluster k is defined as $C_k = (\mathbf{x}_i)_{i:z_i=k}$. In the standard formulation of FMM-based clustering, the data points are assumed to be independent and identically-distributed (i.i.d.) draws from an FMM and one typically tries to maximize the corresponding likelihood function, $p(\mathbf{X}|\theta)$, over θ . The most popular approach for (locally-) maximizing that likelihood is via an expectation-maximization (EM) algorithm (Dempster, Laird, and Rubin 1977). As the time-honored Bayesian formulation helps avoiding problems such as over-fitting and enables encoding prior knowledge, the FMM also has a Bayesian (but still finite) variant, where θ is assumed to be random and drawn from a suitable prior (Gelman, Stern, Carlin, Dunson, Vehtari, and Rubin 2013) (specifically, the prior over $(\pi_k)_{k=1}^K$ is usually a Dirichlet *distribution*, not to be confused with a Dirichlet *process*). In which case, one targets the *posterior* distribution, $p(\theta|\mathbf{X})$, rather than the likelihood; e.g., one can try to maximize $p(\theta|\mathbf{X})$ over θ , sample θ from it, compute posterior expectations, etc.

3.2. The Dirichlet process mixture model (DPMM)

Below we provide a brief and *informal* introduction to the DPMM, focusing only on the essentials required for understanding this manuscript and make it self-contained as possible. For a comprehensive and formal mathematical treatment, see Ghosal and Van der Vaart (2017). For an applied data-analysis perspective, see Müller, Quintana, Jara, and Hanson (2015). For a gentle introduction with machine-learning and/or computer-vision readers in mind, see the theses by Sudderth (2006) or Chang (2014).

The DPMM is a BNP extension of the FMM (Antoniak 1974). Loosely speaking, a DPMM entertains the notion of a mixture of infinitely-many *components*. The weights, $\boldsymbol{\pi} = (\pi_k)_{k=1}^\infty$, are drawn from a Griffiths-Engen-McCloskey (GEM) stick-breaking process with a concentration parameter $\alpha > 0$ (Pitman 2002), while the components, $(\theta_k)_{k=1}^\infty$, are drawn from a prior as in the Bayesian FMM case.

Example 2 *The Dirichlet process GMM (DPGMM) is a GMM with infinitely-many Gaussians.*

Like the FMM, the DPMM is often used for clustering. With a DPMM, however, the number of *clusters*, K , is not assumed to be known; rather, it is viewed as a latent random variable whose value is inferred with the rest of the model.

While in an FMM the number of clusters and the number of components are typically equal, and are thus both denoted by a single symbol, K , with a DPMM the situation is different: although there are infinitely-many components, the (latent and random) number of clusters, K , is finite (particularly, $K \leq N$). The inferred value of K depends on α (the higher α is, the more clusters are expected), on the complexity of the data, and the (hyper-parameters of the) prior over the components (that said, when N is large, the last two factors are usually more influential than α).

Example 3 *Recall that the standard prior for Gaussian components is a normal inverse-Wishart (NIW) distribution (Gelman et al. 2013). If the NIW's hyper-parameters are set to strongly favor small covariances (hence small clusters), then this will implicitly favor a large K . Likewise, if the NIW prior strongly favors larger covariances (hence large clusters), then K will tend to be small. In the lack of prior knowledge, however, the NIW prior can be set to be very weak (i.e., high uncertainty), letting the data speak for itself.*

For simplicity, our text below implicitly assumes that all the random vectors involved have either a pdf or a pmf. One known mathematical construction of the DPMM uses the following distributions:

$$\boldsymbol{\pi} | \alpha \sim \text{GEM}(\boldsymbol{\pi}; \alpha), \quad (3)$$

$$\theta_k | \lambda \stackrel{i.i.d.}{\sim} f_\theta(\theta_k; \lambda), \quad \forall k \in \{1, 2, \dots\}, \quad (4)$$

$$z_i | \boldsymbol{\pi} \stackrel{i.i.d.}{\sim} \text{Cat}(z_i; \boldsymbol{\pi}) \quad \forall i \in \{1, 2, \dots, N\}, \quad (5)$$

$$\mathbf{x}_i | z_i, \theta_{z_i} \sim f_{\mathbf{x}}(\mathbf{x}_i; \theta_{z_i}), \quad \forall i \in \{1, 2, \dots, N\}. \quad (6)$$

Here, $f_\theta(\cdot; \lambda)$ is the pdf or pmf (associated with some base measure) parameterized by λ , the infinite-length vector $\boldsymbol{\pi} = (\pi_k)_{k=1}^\infty$ is drawn from a GEM stick-breaking process with a concentration parameter $\alpha > 0$ (particularly, $\pi_k > 0$ for every k and $\sum_{k=1}^\infty \pi_k = 1$) while θ_k is drawn from f_θ . Each of the N i.i.d. observations $(\mathbf{x}_i)_{i=1}^N$ is generated by first drawing a label, $z_i \in \mathbb{Z}^+$, from $\boldsymbol{\pi}$ (i.e., Cat is the categorical distribution), and then \mathbf{x}_i is drawn from (a pdf or a pmf) $f_{\mathbf{x}}$ parameterized by θ_{z_i} . Informally,

$$\mathbf{x}_i \stackrel{i.i.d.}{\sim} \sum_{k=1}^\infty \pi_k f_{\mathbf{x}}(\mathbf{x}_i; \theta_k). \quad (7)$$

Here too, each $f_{\mathbf{x}}(\cdot; \theta_k)$ is called a *component* and we make no distinction between a component, $f_{\mathbf{x}}(\cdot; \theta_k)$, and its parameter, θ_k . The so-called labels $(z_i)_{i=1}^N$ encode the observation-to-component assignments. A cluster is a collection of points sharing a label; i.e., \mathbf{x}_i is in cluster

k , denoted by C_k , if and only if $z_i = k$. Let (the random variable) K be the number of unique labels: $K = |\{k : z_i = k \text{ for some } i \in \{1, \dots, N\}\}|$; i.e., K is also the number of clusters and is bounded above by N . Typically, and as assumed in this manuscript, f_θ is chosen to be a conjugate prior (Gelman *et al.* 2013) to f_x . The latent variables here are K , $(\theta_k)_{k=1}^\infty$, $\boldsymbol{\pi}$, and $(z_i)_{i=1}^N$. For more details (and other constructions), see Sudderth (2006). and Gelman *et al.* (2013).

Example 4 In the case of Gaussian components, where $f_\theta(\cdot; \lambda)$ is an NIW pdf, we have

$$f_\theta(\overbrace{\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k}^{\theta_k}; \overbrace{\kappa, \mathbf{m}, \nu, \boldsymbol{\Psi}}^{\lambda}) = \text{NIW}(\boldsymbol{\mu}, \boldsymbol{\Sigma}; \kappa, \mathbf{m}, \nu, \boldsymbol{\Psi}) \triangleq \mathcal{N}(\boldsymbol{\mu}; \mathbf{m}, \frac{1}{\kappa} \boldsymbol{\Sigma}) \mathcal{W}^{-1}(\boldsymbol{\Sigma}; \nu, \boldsymbol{\Psi}) \quad (8)$$

where $\mathcal{W}^{-1}(\boldsymbol{\Sigma}; \nu, \boldsymbol{\Psi})$ is the Inverse-Wishart distribution (over $d \times d$ SPD matrices), the hyperparameters are

$$\lambda = (\mathbf{m}, \boldsymbol{\Psi}, \kappa, \nu) \quad (9)$$

where $\mathbf{m} \in \mathbb{R}^d$, the $d \times d$ matrix $\boldsymbol{\Psi}$ is SPD, and the two real numbers κ and ν satisfy $\kappa > 0$ and $\nu > d - 1$ (do not confuse κ (“kappa”) with k , the index of the component).

3.3. Inference via Chang and Fisher III’s DPMM sampler

We now briefly review a DPMM sampler proposed by Chang and Fisher III (2013). That sampler consists of a restricted Gibbs sampler (Robert and Casella 2013) and a split/merge framework (Jain and Neal 2004) which together form an ergodic Markov chain. The operations in each step of that sampler are highly parallelizable. Importantly, the splits and merges let the sampler make *large moves* along the (posterior) probability surface as in such operations multiple labels change their label *simultaneously* to the same different label. This is unlike what happens, e.g., in methods that must change each label separately from the others. We now describe the essential details.

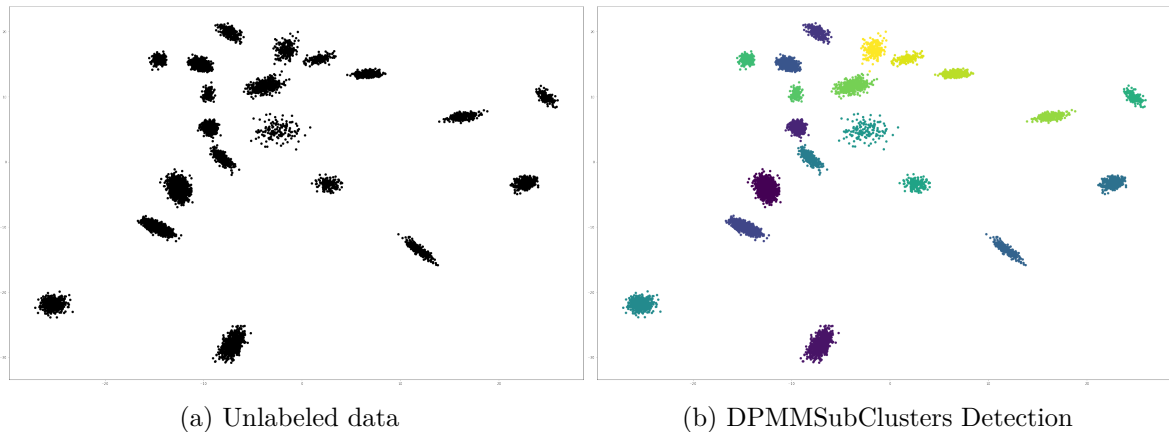
The augmented space

The latent variables, $(\theta_k)_{k=1}^\infty$, $\boldsymbol{\pi}$, and $(z_i)_{i=1}^N$, are augmented with auxiliary variables. For each component θ_k two subcomponents (conceptually thought of as l –“left” and r –“right”) are added, $\bar{\theta}_{k,l}, \bar{\theta}_{k,r}$, with subcomponent weights $\bar{\boldsymbol{\pi}}_k = (\bar{\pi}_{k,l}, \bar{\pi}_{k,r})$. Implicitly, this means that every cluster C_k is augmented with two subclusters, $\bar{C}_{k,l}$ and $\bar{C}_{k,r}$. For each cluster label z_i , an additional *subcluster label*, $\bar{z}_i \in \{l, r\}$, is added; i.e., subcluster $\bar{C}_{k,l} \subset C_k$ consists of all the points in C_k whose subcluster label is l (the other subcluster, $\bar{C}_{k,r}$, is defined similarly).

The restricted Gibbs sampler

This restricted sampler is not allowed to change (the current estimate of) K ; rather, it can change only the parameters of the existing clusters and subclusters, and when sampling the labels, it can assign an observation only to an existing cluster. For each instantiated component k , changing

$$\theta_k, \bar{\theta}_{k,l}, \text{ and } \bar{\theta}_{k,r} \quad (10)$$

Figure 1: 20 clusters detected by **DPMMSubClusters**.

is done using

$$p(\theta_k | C_k; \lambda), p(\bar{\theta}_{k,l} | \bar{C}_{k,l}; \lambda), \text{ and } p(\bar{\theta}_{k,r} | \bar{C}_{k,r}; \lambda), \quad (11)$$

respectively, where the latter three are the conditional distributions of the cluster or subcluster parameters given the cluster or subclusters (and the prior hyperparameters, λ). In Section 5, we will dive deeper into the details of the restricted Gibbs sampler.

The split/merge framework

Splits and merges allow the sampler to change K using the Metropolis-Hastings framework (Hastings 1970). Particularly, the auxiliary variables are used to propose splitting an existing cluster or merging two existing ones. When a split is accepted, each of the newly-born clusters is augmented with two new subclusters. The Hastings ratio of a split is (Chang and Fisher III 2013):

$$H_{\text{split}} = \frac{\alpha \Gamma(N_{k,l}) f_{\mathbf{x}}(\bar{C}_{k,l}; \lambda) \Gamma(N_{k,r}) f_{\mathbf{x}}(\bar{C}_{k,r}; \lambda)}{\Gamma(N_k) f_{\mathbf{x}}(C_k; \lambda)} \quad (12)$$

where Γ is the Gamma function, N_k , $N_{k,l}$ and $N_{k,r}$ are the numbers of points in C_k , $\bar{C}_{k,l}$ and $\bar{C}_{k,r}$, respectively, while

$$f_{\mathbf{x}}(C_k; \lambda), f_{\mathbf{x}}(\bar{C}_{k,l}; \lambda), \text{ and } f_{\mathbf{x}}(\bar{C}_{k,r}; \lambda) \quad (13)$$

represent the *marginal* likelihood of C_k , $\bar{C}_{k,l}$ and $\bar{C}_{k,r}$ respectively. Concrete expressions for the marginal likelihood, in the case of Gaussian or Multinomial components (the component types considered in our experiments) appear in Chang (2014).

Finally, a *merge* proposal is based on taking two existing clusters and proposing merging them into one. The corresponding Hastings ratio is $H_{\text{merge}} = 1/H_{\text{split}}$ where $\bar{C}_{k,l}$ and $\bar{C}_{k,r}$ are replaced with the two clusters, and C_k is replaced with the result of the merge. For the derivation behind H_{split} and H_{merge} , see Chang and Fisher III (2013).

Importantly, and as any successful DPMM inference method, Chang and Fisher III's sampler can detect different numbers of clusters according to the complexity of the dataset. For

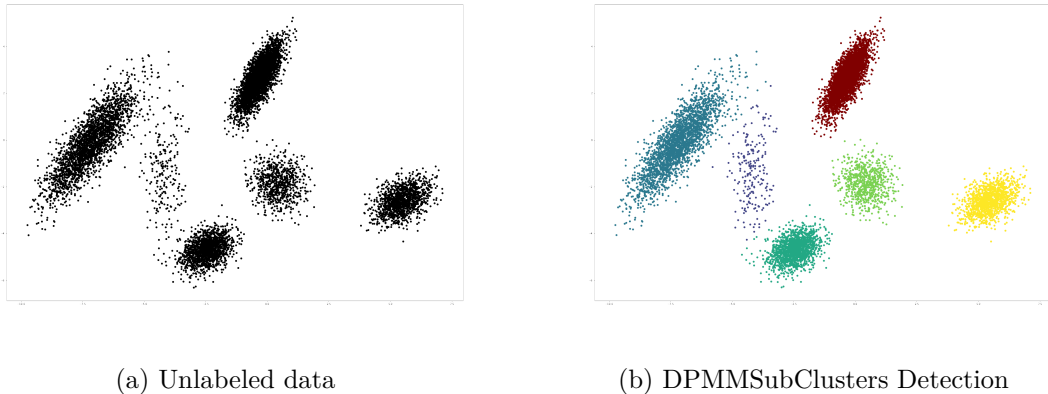


Figure 2: 6 clusters detected by **DPMMSubClusters**.

example, in Figure 1 we demonstrate unlabeled data with 20 clusters while in Figure 2 we show data consisting of 6 clusters. Using our implementation (the topic of the next section), the sampler correctly detected the different numbers of clusters in each dataset, while using the same code and the same hyperparameters.

4. Software

The proposed software supports three different software languages (Julia, CUDA/C++, Python), two operating systems (Windows & Linux) as well as multiple interface options. In terms of performance, the proposed software is faster than other publicly-available packages that exist today and that we were able to test. Particularly, as long as the product of N (number of data points) times d (the dimension of the data) is not too low, then our GPU version is consistently at least a few times faster than any other implementation we are aware of.

4.1. Installation

Our software (licensed under GPLv2) with installation instructions is available on GitHub. The URLs for our packages appear in Table 2. In order to compile the CUDA/C++ package the user will need C++14 (or higher) and CUDA 11.2 (or higher). For visualization purposes (i.e., plotting points in 2D), the CUDA/C++ windows version also requires **OpenCV**. However, this visualization is used only for debugging purposes so the installation of **OpenCV** is not mandatory for using our implementation. For the CPU package the user will need Julia 1.5 (or higher). For Python we used 3.8. The installation steps below were tested successfully on Windows 10 (Visual Studio 2019), Ubuntu 18.04, and Ubuntu 21.04.

Package	GitHub
CUDA/C++	https://github.com/BGU-CS-VIL/DPMMSubClusters_GPU
Julia	https://github.com/BGU-CS-VIL/DPMMSubClusters.jl
Python	https://github.com/BGU-CS-VIL/dpmpython

Table 2: GitHub URL for each package.

4.2. Executing

Below are listed several options to run DPMM inference using our software. Usually the best performance will be achieved by running the CUDA/C++ version which uses the GPU (when available).

From Julia code – CPU

The following sample code generates a synthetic GMM dataset with $N = 10^5$ points, dimension $d = 2$ and $K = 10$ clusters, and then fits a DPMM to the data (without knowing K or the other parameters of the GMM) using the sampler.

```
using DPMMSubClusters

x, labels, clusters = DPMMSubClusters.generate_gaussian_data(10^5, 2, 10, 100.0)
hyper_params = DPMMSubClusters.niw_hyperparams(Float32(1.0),
  zeros(Float32, 2), Float32(5), Matrix{Float32}(I, 2, 2)*1)
DPMMSubClusters.dp_parallel(x, hyper_params, Float32(100000.0), 100, 1,
  nothing, true, false, 15, labels)
```

From a C++ code – GPU

`dp_parallel()` in ‘`dp_parallel_sampling_class`’ is the function that should be called in order to run the program. The sample code below will first generate a synthetic random dataset and will then run the sampler to fit a DPMM to it.

```
srand(12345);
data_generators data_generators;
MatrixXd x;
std::shared_ptr<LabelsType> labels = std::make_shared<LabelsType>();
double** tmean;
double** tcov;
int N = (int)pow(10, 5);
int D = 2;
int numClusters = 2;
int numIters = 100;

data_generators.generate_gaussian_data(N, D, numClusters, 100.0, x,
  labels, tmean, tcov);
std::shared_ptr<hyperparams> hyper_params =
  std::make_shared<niw_hyperparams>(1.0, VectorXd::Zero(D), 5,
  MatrixXd::Identity(D, D));
dp_parallel_sampling_class dps(N, x, 0, prior_type::Gaussian);
ModelInfo dp = dps.dp_parallel(hyper_params, N, numIters, 1, true,
  false, false, 15, labels);
```

From the command line – GPU

Running the CUDA/C++ program can be done from the command line (in both Linux and

Windows). There are a few parameters that can be used to run the program. In order to set the hyperparams the `params_path` parameter can be used. The value of the parameter should be a path for a JSON file which includes the hyperparams (i.e., `alpha` or `hyper_params` for the prior) . In order to use this parameter follow this syntax:

```
--params_path=<PATH_TO_JSON_FILE_WITH_MODEL_PARAMS>
```

There are few more parameters like `model_path` for the path to a npy file which include the model and `result_path` for the path of the output results.

Our code has support for both Gaussian and Multinomial distributions. It can be easily adapted to other component distributions, e.g., Poisson, as long as they belong to an exponential family. The default distribution is Gaussian. To specify a distribution other than a Gaussian, use the `prior_type` parameter. For example:

```
--prior_type="Multinomial"
```

The JSON file containing the model parameters can contain many parameters that can be controlled by the user. A few examples are: `alpha`, `prior`, number of iterations, `burn_out` and `kernel`. The full list of parameters can be seen in the function `init()` in `'global_params'`. The result file is a JSON file which by default contains the predicted labels, the weights, the normalized mutual information (NMI) score and the running time per iteration. A few other parameters can be added to the result file. Samples for these additional parameters are commented out in the `main.cpp` file.

From Python code – CPU and GPU

The `dpmmwrapper.py` file contains an example for how to run either the GPU or CPU packages from Python (look for the code in the function `main`). In the following example we are generating a GMM synthetic dataset for 10^5 point, 2 dimensions and 10 clusters. We are running it on the GPU.

```
from julia.api import Julia
jl = Julia(compiled_modules=False)
from dpmmpython.dpmmwrapper import DPMMPython
from dpmmpython.priors import niw

data, gt = DPMMPython.generate_gaussian_data(sample_count=10000,
      dim=2, k=10, var=100.0)
prior = niw(kappa = 1, mu = np.zeros(2), nu = 3, psi = np.eye(2))
labels, clusters, results = DPMMPython.fit(data = data,
      alpha = 100, prior = prior, verbose = True,
      gt = gt, gpu = True)
```

Here, the variable `results` depends on the backend: for the CUDA/C++ backend it will be null, while for the Julia backbone `results` will contain other information (see the documentation of our Julia `fit` function). The `gt` variable stands for the ground-truth labels.

From a **Jupyter** notebook – CPU and GPU

The notebook `dpmmwrapper.ipynb` can be executed to test different packages including our GPU and CPU packages on multiple datasets.

4.3. Test

Our CUDA/C++ package was built with the test-driven development (TDD) methodology. The Windows version comes with 53 unit tests which cover more than 60% of the code. Usually it takes less than 3 minutes to run all tests. Those tests are for both Gaussian and Multinomial distributions, and include the different CUDA kernels (for matrix multiplication) mentioned in Section 5 below. We used the GoogleTest framework to write those tests and they were validated with a Visual Studio 2019 test engine.

5. The proposed implementation

5.1. Design and implementation

In our implementation(s) after we initialize all the required objects, we start running the iterations of the sampler. For each iteration in `group_step()` we execute the algorithm which is described below in detail. For concreteness, below we will refer to the CUDA/C++ code during the algorithm’s description and not to the Julia code. However, the structure of the code in both packages is similar enough to be followed as long as the reader is familiar with any of those languages.

We use the same notation as in [Chang and Fisher III \(2013\)](#) where N is the number of data points and K is the (current estimate of the) number of clusters.

- For each iteration of the restricted Gibbs sampling:

- (a) Sample cluster weights $\pi_1, \pi_2, \dots, \pi_K$:

$$(\pi_1, \dots, \pi_K, \tilde{\pi}_{K+1}) \sim \text{Dir}(N_1, \dots, N_K, \alpha). \quad (14)$$

In our code we built a vector `points_count` that holds the number of points for each cluster and apply a Dirichlet-distribution sampling at once for all clusters:

```
dirichlet_distribution<std::mt19937> d(points_count);
std::vector<double> dirichlet = d(*globalParams->gen);
```

- (b) Sample sub-cluster weights from a 2D Dirichlet distribution:

$$(\bar{\pi}_{kl}, \bar{\pi}_{kr}) \sim \text{Dir}(N_{kl} + \alpha/2, N_{kr} + \alpha/2), \quad \forall k \in \{1, \dots, K\}. \quad (15)$$

In order to propose meaningful splits that are likely to be accepted, the algorithm uses auxiliary variables such that each cluster consists of 2 sub-clusters (conceptually thought of as “left” and “right”). $\bar{\pi}_k = \{\bar{\pi}_{kl}, \bar{\pi}_{kr}\}$ denote the weights of the sub-clusters of cluster k . The code is implemented in `sample_cluster_params()` in ‘`shared_actions`’.

(c) Sample cluster parameters:

$$\theta_k \propto f_{\mathbf{x}}(\mathbf{x}_{\mathcal{I}_k}; \theta_k) f_{\theta}(\theta_k; \lambda), \quad \forall k \in \{1, \dots, K\}. \quad (16)$$

Here, $f_{\mathbf{x}}(X; \theta_k)$ is the likelihood of a set of data points under the parameter θ_k , $f_{\theta}(\theta; \lambda)$ is the likelihood of the parameter θ under the prior $f_{\theta}(\cdot; \lambda)$, \propto denotes sampling proportional to the right-hand side of the equation, and $\mathcal{I}_k = \{i : z_i = k\}$ is the set of indices of points labeled as belonging to cluster k . Each distribution (e.g., a Gaussian or a Multinomial) has its own `sample_distribution()` function which calculates the cluster’s parameters. The prior classes for the Gaussian and Multinomial distributions are ‘`niw`’ and the ‘`multinomial_prior`’ respectively which inherit from the ‘`prior`’ class. The likelihood is calculated in each class within the function `log_marginal_likelihood()`. In order to extend and support to more distributions new classes which inherit from ‘`prior`’ may be added.

(d) Sample sub-cluster parameters:

$$\bar{\theta}_{kh} \propto f_{\mathbf{x}}(\mathbf{x}_{\mathcal{I}_{kh}}; \bar{\theta}_{kh}) f_{\theta}(\bar{\theta}_{kh}; \lambda), \quad \forall k \in \{1, \dots, K\}, \quad h \in \{l, r\}. \quad (17)$$

$\bar{\theta}_k = \{\bar{\theta}_{kl}, \bar{\theta}_{kr}\}$ denotes the parameters of the sub-clusters of cluster k . In the code we used the same functions that we used for the calculation of the cluster’s parameters. We maintain the following objects in memory: a `std::vector` of `local_cluster` type where each item in that vector holds the cluster and sub-cluster parameters (e.g., $\theta_k, \mu_k, \Sigma_k, \theta_{kl}, \theta_{kr}, \bar{\pi}_{kl}$ and $\bar{\pi}_{kr}$ for the k^{th} Gaussian), the point counts (i.e., N_k, N_{kl}, N_{kr}) and the sufficient statistics as well as the cluster weights (i.e., $\pi_1, \pi_2, \dots, \pi_k$).

(e) Sample cluster assignments for each point:

$$z_i \propto \sum_{k=1}^K \pi_k f_{\mathbf{x}}(\mathbf{x}_i; \theta_k) \mathbb{1}(z_i = k), \quad \forall i \in \{1, \dots, N\}. \quad (18)$$

To sample from a probability distribution efficiently we implemented a GPU kernel in

`sample_by_probability()` based on a C algorithm (Smith 2002). To summarize all k of a point we wrote the `dcolwise_dot_all_kernel()` kernel. These kernels are working in parallel on all components. Each component is working in a different GPU stream while in each stream the points are aggregated to blocks that are running in parallel. We set a number of 512 threads per block. This parameter can be fine-tuned to optimize the performance based on the GPU hardware being used.

(f) Sample sub-cluster assignments for each point:

$$\bar{z}_i \propto \sum_{h \in \{l, r\}} \pi_{z_i h} f_{\mathbf{x}}(\mathbf{x}_i; \bar{\theta}_{z_i h}) \mathbb{1}(\bar{z}_i = h), \quad \forall i \in \{1, \dots, N\}. \quad (19)$$

$\bar{z}_i \in \{l, r\}$ variables are (conceptually thought of as “left” and “right”) indicate which of the sub-cluster the i^{th} point is assigned to. We have a thin version of the cluster and sub-cluster parameters `thin_cluster_params`. For a single machine this structure is unneeded since the data already exists in ‘`local_cluster`’. However, we maintain it not only for consistency with the Julia package but also because this option may be valuable in the future to scale our CUDA/C++ implementation to multiple machines.

For efficiency we are not copying the data; rather, we use `std::shared_ptr` between the structures. We have also chunks of the data per GPU in the device memory as part of the ‘`gpuCapability`’ structure. The structure contains the following 3 properties: List of points \mathbf{x}_i (`d_points`), chunks of the cluster assignments z_i (`d_labels`) and sub-cluster assignments \bar{z}_i (`d_sub_labels`).

```
struct gpuCapability
{
    int* d_labels;
    int* d_sub_labels;
    double* d_points;
};

class cudaKernel
{
protected:
    std::map<int, gpuCapability> gpuCapabilities;
};
```

- Propose and accept splits:
By sampling the sub-clusters, one is able to propose and accept meaningful splits that divide a cluster into its 2 sub-clusters.

- Propose to split cluster k into its 2 sub-clusters for all $k \in \{1, 2, \dots, K\}$.
- Calculate the Hastings ratio H and accept the split with probability $\min(1, H)$:

$$H_{\text{split}} = \frac{\alpha \Gamma(N_{kl}) f_{\mathbf{x}}(\mathbf{x}_{\mathcal{I}_{kl}}; \lambda) \cdot \Gamma(N_{kr}) f_{\mathbf{x}}(\mathbf{x}_{\mathcal{I}_{kr}}; \lambda)}{\Gamma(N_k) f_{\mathbf{x}}(\mathbf{x}_{\mathcal{I}_k}; \lambda)}. \quad (20)$$

The proposal is done in the function `should_split_local()` and the split itself is done in the kernel `split_cluster_local_worker()`.

- Propose and accept merges:
Merges are proposed by merging 2 sub-clusters into one with each of the original clusters becoming a sub-cluster of the new merged cluster.

- Propose to merge clusters k_1, k_2 for all pairs $k_1, k_2 \in \{1, 2, \dots, K\}$.
- Calculate the Hastings ratio H_{merge} ,

$$H_{\text{merge}} = \frac{\Gamma(N_{k_1} + N_{k_2}) p(\mathbf{x}|\hat{z})}{\alpha \Gamma(N_{k_1}) \Gamma(N_{k_2}) p(\mathbf{x}|z)} \times \frac{\Gamma(\alpha)}{\Gamma(\alpha + N_{k_1} + N_{k_2})} \times \frac{\Gamma(\frac{\alpha}{2} + N_{k_1}) \Gamma(\frac{\alpha}{2} + N_{k_2})}{\Gamma(\frac{\alpha}{2}) \Gamma(\frac{\alpha}{2})}, \quad (21)$$

and accept the merge with probability $\min(1, H_{\text{merge}})$. The proposal is done in function `should_merge()` and the merge itself is done in the kernel `merge_clusters_worker()`.

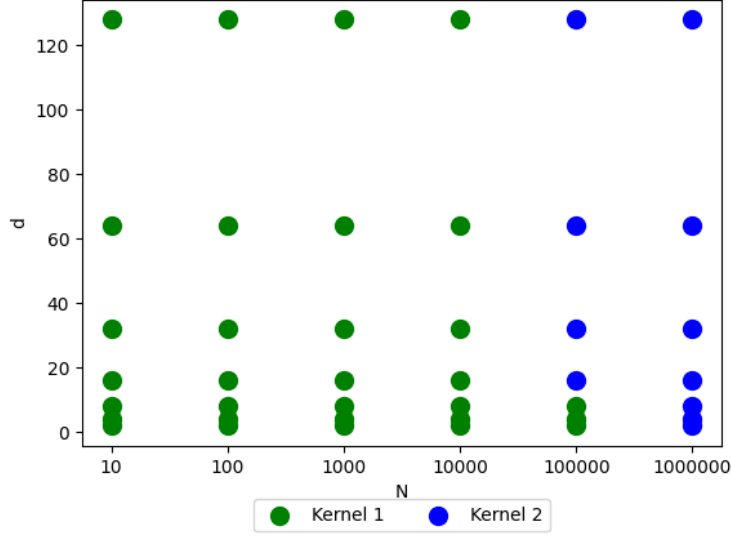


Figure 3: Two CUDA kernels running on different sizes of labels and dimensions. The faster kernel was selected for each pair (d and N).

5.2. Optimizing processing

Two kernels for optimization

In order to optimize the performance of the matrix multiplication which is required in our algorithm we used two different CUDA kernels and decide which one to use based on the size of the $d \times N$ matrix (where d is the dimension of the data and N is the number of points). We added a built-in capability to automatically select, at run-time, the best kernel automatically based on d , N and the GPU capabilities. For more optimization in case that the best kernel is known we provide an option to set the kernel which can save some time at the beginning of the run (especially with a high N and a high d). We measured the optimal kernel on an NVidia Quadro RTX 4000 card. On matrices whose size was below 640,000 size ($d \times N$), Kernel #1 is optimized and above this number Kernel #2 is optimized. Kernel #1 was written in a native way for CUDA to multiply two matrices. Kernel #2 was written with cublas API which is more optimal for big matrices. The results (using N between 10 to 10^6 and d between 2 to 128) are displayed in Figure 3.

Data structure

To optimize the operations that needed to be done on the matrices and in the device, all the data in the GPU and in Eigen's structure are stored in a column-major order. In general, in places which we needed to perform operations on all data points, we iterate on the dimensions and run (in parallel) each point calculation on a different GPU core.

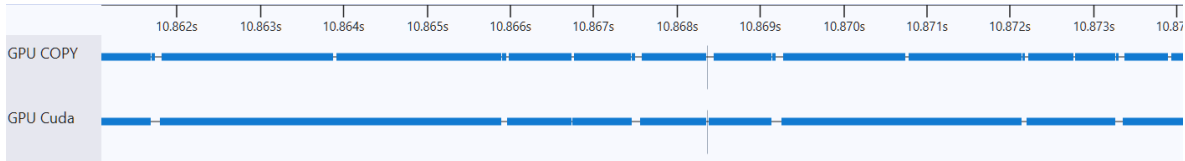


Figure 4: CUDA running in multi-streams. Copy and kernels are working in parallel. Blue color indicates an active kernel. The horizontal axis stands for time (in [s]).

5.3. Parallelism

The sampler from [Chang and Fisher III \(2013\)](#) was designed with massive parallelization in mind. Thus, most of its operations are parallelizable. Particularly, sampling cluster parameters is parallelizable over the clusters, sampling assignments can be computed independently for each point, and cluster splits can be proposed in parallel. Thus, a multi-core implementation is quite straightforward, with the caveat that one needs to be careful with merges; i.e., to prevent more than 2 clusters merge into the same single cluster; e.g., if clusters 1 and 2 are merged at the same time when clusters 2 and 3 are merged, this would imply the three clusters (1,2, and 3) would be merged into a single one. However, this would be inconsistent with the underlying model. Recall that [Chang and Fisher III \(2013\)](#) released a multi-core single-machine C++ implementation (for CPU). The nature of the algorithm, however, let us build an implementation that goes beyond their original implementation, by allowing parallelization across multiple machines, not just multiple CPU cores. In turn, this enables us to not only leverage more computing power and gain speedups but also provide practical benefits in terms of memory and storage. For example, our Julia implementation can be used within a distributed network of weak agents (e.g., small robots collecting data). It also never transfers data (which is expensive and slow); rather, we transfer only sufficient statistics and parameters. Thus, the proposed implementation is also well suited for a network of low-bandwidth communication. Likewise, on a single machine, we are able to use the GPU's cores powers and Nvidia streams and asynchronous calls.

Multiple GPU streams

Most of the code was written as separated individual streams. Each stream contains sequences of operations that execute in issue-order on the GPU. CUDA operations in different streams may run concurrently. In many places we tied the stream to a specific cluster. Thus, we could run close to $O(1)$ instead of $O(K)$. We took advantage of CUDA's asynchronous memory allocation API which was exposed lately in version 11. We were able to allocate and deallocate the parameters as part of the sequence of operations and in this way to improve the concurrency of other kernels that were running on other streams. In Figure 4 we show an example of memory allocation operations that were performed on the GPU in parallel and at the same time that the kernel operation was running.

Multiple GPU cards

We tested the performance of our implementation by parallelizing the processing across multiple GPUs. The logic that we adopt was to parallelize the parts that we were running with multi-streams on multi-GPUs. We implemented a container with all available GPUs: `std::map<int, gpuCapability> gpuCapabilities`. In the places that we sample the cluster

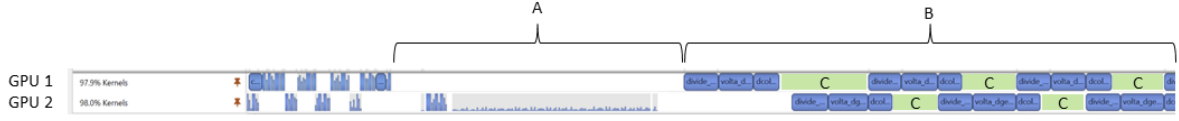


Figure 5: CUDA running in multi GPU cards. Kernels from both GPU cards are working in parallel. The horizontal axis stands for time (in [s]).

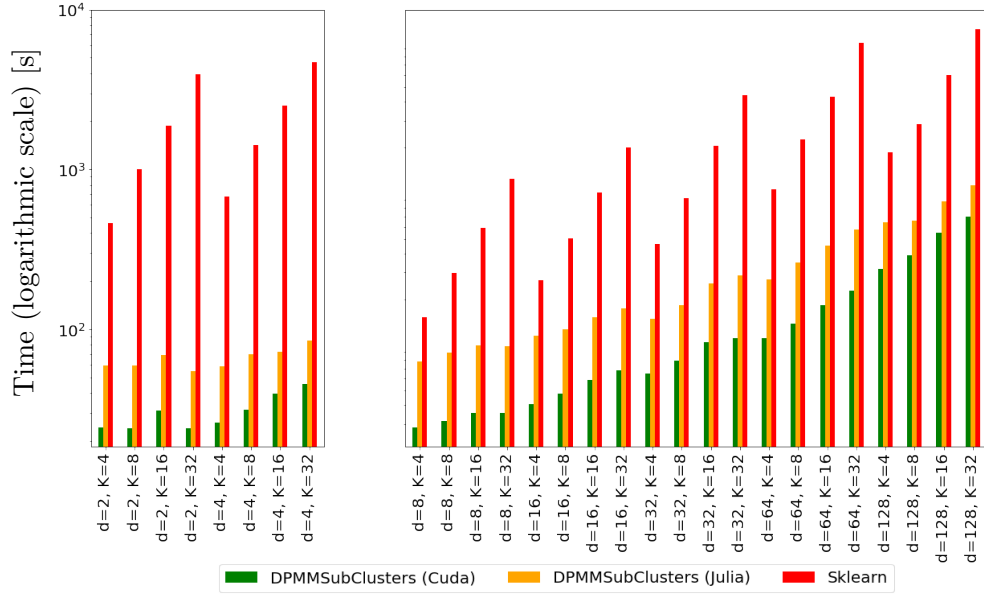


Figure 6: DPGMM synthetic data, time, $N = 10^6$. In the right panel, due to sklearn's slowness, we had to give it the unfair advantage of using the true K as the upper bound of the number of clusters.

and sub-cluster parameters and where the sufficient statistics are been calculated, instead of creating a stream on the same GPU we took the next available GPU by calling to the function `pick_any_device` and created a stream on the current selected device. These areas are marked as B in Figure 5. Our hypothesis was that in places which K is large there should be more impact. Our observation when we tested 2 GPUs (of type Quadro RTX 4000) was that the performance is not better. In Figure 5 we can see that during the timeline that marked as B the two GPUs are working in parallel on different kernels. The areas that are marked in C are the places which there were some work that was done on the CPU and some synchronization that was needed to be done between the GPUs. The timeline that marked as A emphasize the work that we did not parallelize. This work contains sampling components, merge and split clusters. Consequently, we disabled that option by default in the `cudaKernel::init` function using the following line: `numGPU = 1`. On a different hardware and different GPU types, however, it is possible that this option will yield better results. Thus, the user may want to experiment with `numGPU > 1`, depending on the available hardware. Further the user might want to improve the performance by developing a way to decrease the C areas and parallelize A area in Figure 5.

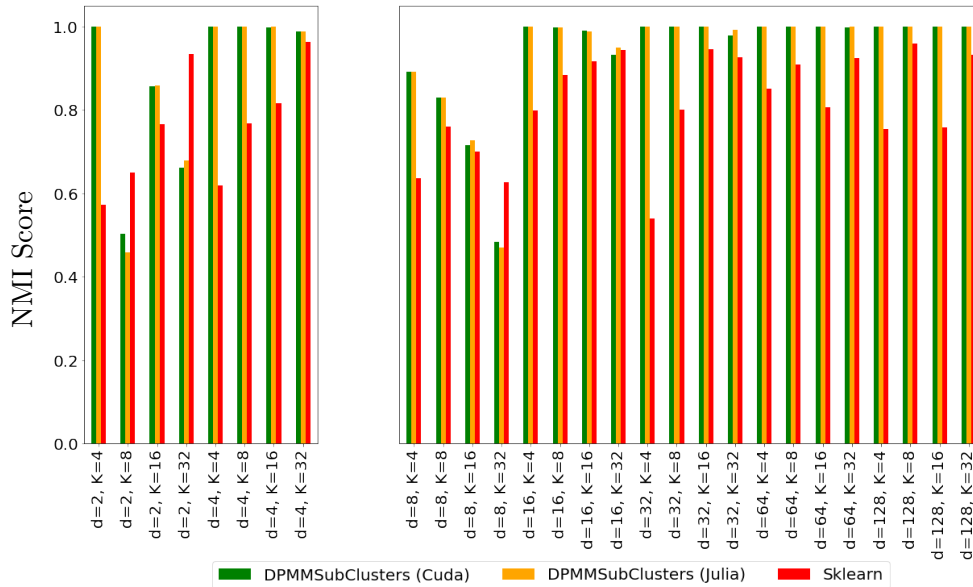


Figure 7: DPGMM synthetic data, NMI, $N = 10^6$. In the right panel, due to sklearn's slowness, we had to give it the unfair advantage of using the true K as the upper bound of the number of clusters.

5.4. Runtime complexity

We now go throughout the CUDA/C++ runtime complexity step by step based on the algorithm. The symbol G below denotes the number of GPU cores.

For each iteration of restricted Gibbs sampling

Sampling the cluster and sub-cluster parameters (including weights) takes constant time for each cluster. In Julia it is parallelized over P processes on the master. This takes $O(K \times d/P)$ time. In CUDA/C++ we observed that using "omp parallel for" (for K) when sampling the cluster parameters is slower than performing it sequentially. The root cause is the fact that Eigen (which we rely on for matrix manipulation) already uses multiple CPU cores and it also uses the GPU efficiently in each one of its threads. Thus, running over the K clusters in parallel is slower than sequentially. Therefore the complexity of CUDA/C++ is $O(K \times d)$. Sampling the cluster weights is also done in constant time. Copying cluster and sub-cluster weights and parameters from host to device is parallelized over CUDA streams over K clusters. The process is parallelized over K clusters and over G GPU cores so it takes $O(K \times d/G)$. Sampling the cluster assignments takes $O(N \times K \times T/G)$ time, where T depends on the observation model, e.g., for multivariate Gaussian observation model with NIW prior $T = d^2$, wherein for a multinomial observation model with Dirichlet prior $T = d$. Sampling the sub-cluster assignments takes $O(N \times T/G)$. Updating the cluster and sub-cluster sufficient statistics can be split up into 3 steps. The first step is for all streams to calculate the sufficient statistics for the data it is in charge of which is common to all kinds of distributions. This step takes $O(N/G)$ time. The second step is to calculate the sufficient statistics which is unique per the distribution. It is done in parallel to each sub-cluster (l, r) and to the cluster. This

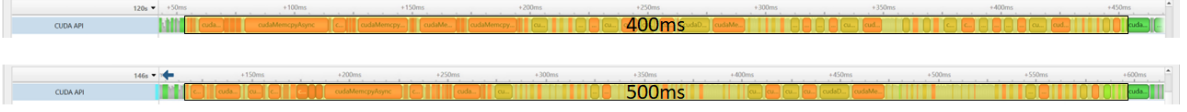


Figure 8: CUDA memory allocation by time for the sufficient statistics of DPGMM synthetic data, $N = 10^6$, $d = 64$. In the upper panel $K = 8$, in the lower panel $k = 16$.

step takes $O(N \times T/G)$ time. The third step is to aggregate across all streams. This step takes $O(K \times d)$ time. Overall, this takes $O(N \times K \times T/G) + O(N \times T/G) + O(K)$ time. So total: $O(N \times K \times T/G)$.

Splits

Proposing splits by looking at each cluster is $O(K)$, noting that we can drop the T here as we use previously calculated values. Processing all the accepted splits requires updating the sufficient statistics which could take at the worst case $O(N/G) + O(K)$ if all clusters are split.

Merges

Proposing merges by inspecting each cluster pair is $O(K^2)$, as in the splits, we can drop the T , by using the previously calculated values. Processing all the accepted merges also requires updating sufficient statistics. The worst case (i.e., if all clusters are merged) is thus $O(N) + O(K)$.

To summarize, the total runtime complexity is $O(K) + O(N \times K \times T/G) = O(N \times K \times T/G)$

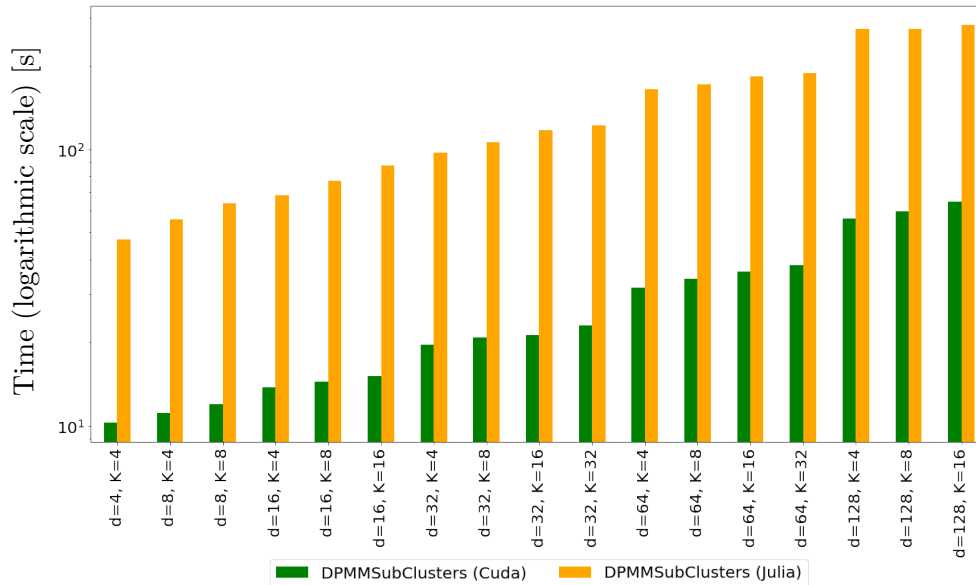
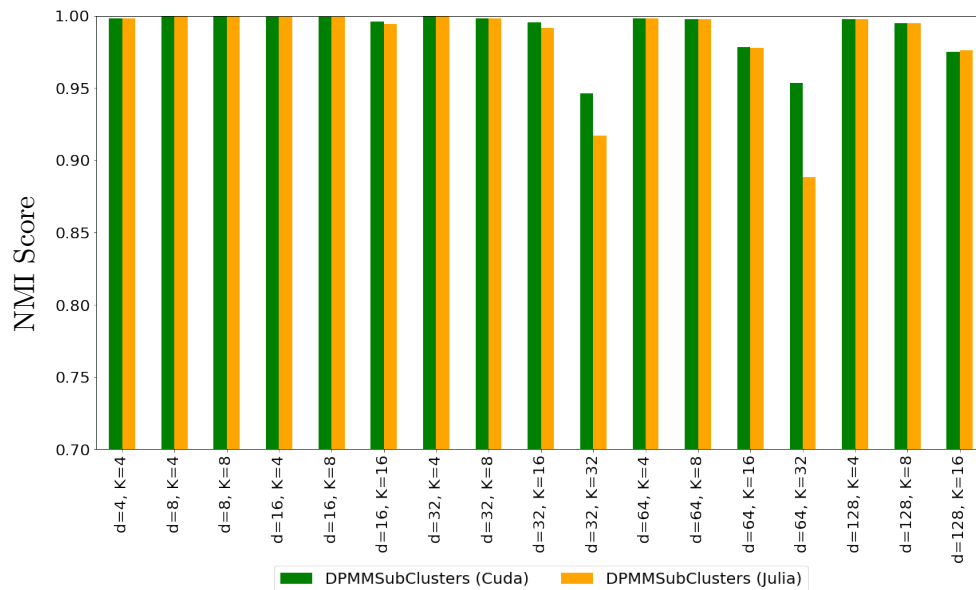
5.5. Memory complexity

We now look at the amount of memory used on the GPU. The data is stored as one array, so we have $O(d \times N)$ on the GPU. The amount of data for the labels and sub labels is $O(N)$. Each stream also has a copy of the cluster and sub-cluster parameters which takes $O(K)$ space. We also have to aggregate sufficient statistics for each cluster after sampling the assignments, taking $O(K \times T)$. Hence, we have a total memory usage of $O(d \times N)$. Since usually $N \gg K$, the memory overhead is insignificant in comparison to the data itself.

6. Illustrations

6.1. Computational details

We ran our tests on two different hardware. One with strong CPU, more RAM and GPU from GTX family: i7-6800K CPU, 3.40GHz with 6 cores, 64GB RAM, GeForce GTX 1070. Second configuration with slower CPU, less RAM and GPU from RTX family: E5-2620 v3 CPU, 2.40GHz with 6 cores, 32GB RAM, Quadro RTX 4000. In both cases we used the same code. We used CUDA driver version 11. We tested on Windows 10 and Linux Ubuntu 18.04 and 21.04. The Python version that we used for the wrapper was 3.8. In the CUDA/C++ package for windows we used **OpenCV** version 4.5.2 to demonstrate visualization in 2D of the clustering process iteration by iteration. For simplicity all the figures that appear in this section were taken from the same system which is the second system E5-2620 v3.

Figure 9: DPMNMM synthetic data, time, $N = 10^6$.Figure 10: DPMNMM synthetic data, NMI, $N = 10^6$.

6.2. Synthetic data: The Dirichlet process Gaussian mixture model

We tested our implementation on small and large synthetic GMM datasets. We wrote a class `data_generators` to generate the synthetic datasets. We ran 112 tests with different parameters: N ($10^3, 10^4, 10^5, 10^6$), d (2, 4, 8, 16, 32, 64, 128) and K (4, 8, 16, 32). For each test we ran 100 iterations (sufficed for convergence in all methods). We repeated each test 10 times with the same random Gaussian synthetic data that we generated and then averaged

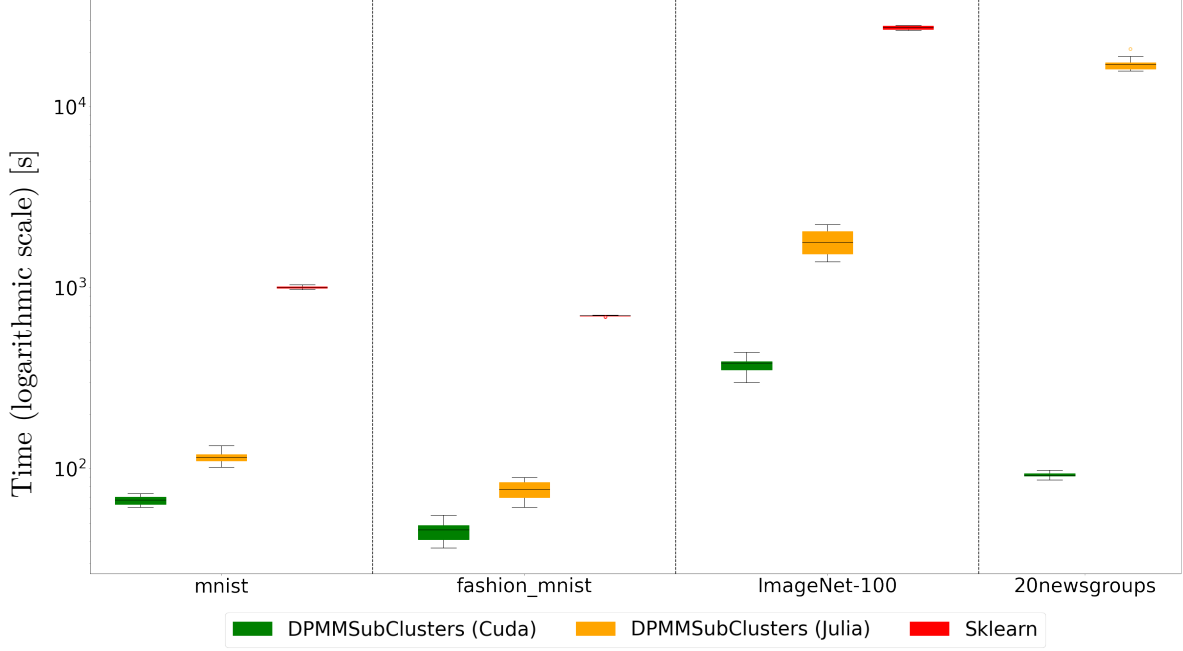


Figure 11: DPMM on real data: running times. In the 20newsgroup dataset (where the data dtype is discrete) the components are Multinomials. In the other datasets the components are Gaussians.

the resulting NMI scores and running times. In each test we compared CUDA/C++, Julia, and a Bayesian Gaussian mixture from **sklearn** (Pedregosa *et al.* 2011). The latter, like our implementations, infers K with the rest of the model. However, unlike our implementations, this **sklearn**'s model requires an upper bound on K .

During our testing we observed that, on a single machine, our CUDA/C++ implementation is faster than our Julia implementation in most of the cases. The cases in which Julia was faster were when running with a low N (up to 10K) and a low d (up to 32). When the data is larger (e.g., when $N = 10^6$ or $d = 128$) the CUDA/C++ solution is 2.5 times faster than Julia on average. In general, when $N \times d \times K$ is high the CUDA/C++ solution is faster than the Julia solution. Both Julia and CUDA/C++ are faster on average than **sklearn** solution: on average, Julia was 2.6 times faster and the CUDA/C++ was 5.3 times faster. Moreover, in a few cases the CUDA/C++ was 35 times faster than **sklearn**.

In Figure 6 we demonstrated the time comparison for $N = 10^6$. In the left side when $d \leq 4$ we see that our solutions were faster than **sklearn**. Due to the slowness of **sklearn** for the cases when $d > 4$ we set in the right side of the figure an easier target for **sklearn** and set **sklearn**'s upper bound for K to the true K (without doing it, **sklearn**'s running time for a high d was impractically slow, many orders of magnitude slower than our implementations). It is obvious from the figure that even when we gave a big advantage to **sklearn**, our implementations still beat it. For accuracy we measured the NMI for each case. NMI comparison is displayed in Figure 7. Again we split the test into two parts. The left side of the figure describes a fair comparison where all the methods were given the same conditions. The right side describes the case **sklearn** enjoyed the advantage of a reduced complexity achieved by using the true K

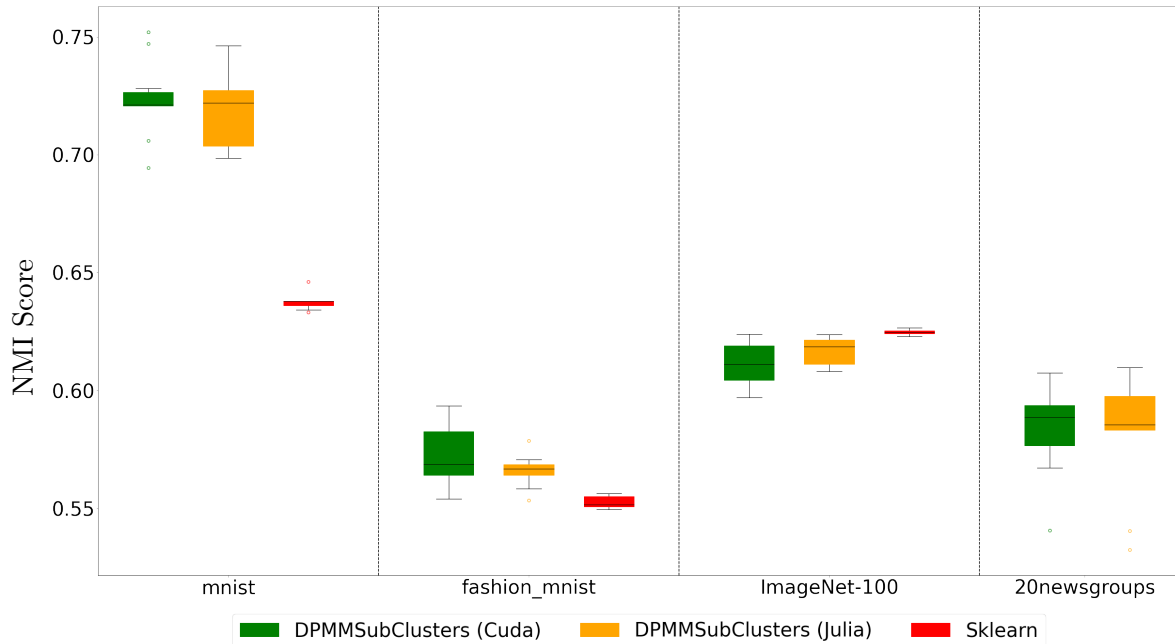


Figure 12: DPMM on real data: NMI scores. In the 20newsgroup dataset (where the data dtype is discrete) the components are Multinomials. In the other datasets the components are Gaussians.

as the upper bound. Despite that advantage, our solutions still yielded better results almost in all cases. Our CUDA/C++ implementation has an overhead which does not exist in our Julia implementation. The root cause of this overhead is due to the memory allocation that it is needed to be done from device to host and from host to device. When $N \times d \times K$ is higher the overhead will be higher. In Figure 8 we demonstrate the time which is needed for the memory allocation for the GPU. In this case as a sample we look into the sufficient statistics calculation. In case of $K = 8$ (in the upper panel) it takes 400 milliseconds to complete the operation. In case of $K = 16$ (in the lower panel) it takes 500 milliseconds. In both cases $N = 10^6$ and $d = 64$. All tests can be reproduced by running the `dpmmwrapper.ipynb` Jupyter notebook from <https://github.com/BGU-CS-VIL/dpmmpython>. Because of the slowness of `sklearn`, the notebook as is might take many days. You can consider to reduce the number of permutations that we tested for N , d and K or to reduce the number of repeats.

In addition, we have used synthetic data to evaluate our software on distributed CPU based inference, synthetic Gaussian data with a dimension of 32 and 16 clusters was generated for the experiment. The computations were carried out on e2-standard-8 GCP instances, each equipped with 8 cores. The results in Table 3 showcase the run-time measurements for different sample sizes and distributed configurations, with the number of processes and machines varying from 1 to 32. The Threads version utilizes a single process and takes advantage of BLAS multi-threading. The findings reveal that increasing the number of processes and machines generally leads to a reduction in run-time, while there is a diminishing return in performance improvement as the number of machines and processes increases, as the number of samples grow, the benefit from using more processes increases. This suggests that while parallelizing

Number of Samples	Processes	Machines	Runtime (seconds)
10000	Threads	1	4.7
	8	1	18.24
	16	2	29.13
	24	3	35.11
	32	4	44.12
100000	Threads	1	33.60
	8	1	27.41
	16	2	29.83
	24	3	37.87
	32	4	45.74
500000	Threads	1	149.32
	8	1	66.12
	16	2	46.41
	24	3	47.36
	32	4	52.14
1000000	Threads	1	309.22
	8	1	91.46
	16	2	63.61
	24	3	59.16
	32	4	58.52
2000000	Threads	1	619.16
	8	1	154.832
	16	2	96.30
	24	3	84.40
	32	4	77.02

Table 3: Run-time measurements for different sample sizes and distributed configurations. The data-sets are all with dimension of 32 and 16 clusters. Threads version uses a single process and utilize BLAS multi-threading.

the DPMM algorithm can yield significant performance gains, optimal resource allocation should be considered to avoid inefficient scaling.

6.3. Synthetic data: The Dirichlet process multinomial mixture model

We ran 72 tests with different parameters: N ($10^3, 10^4, 10^5, 10^6$), d (4,8,16,32,64,128) and K (4,8,16,32) while $d \geq K$. For each test we ran 100 iterations (sufficed for convergence). We repeated 10 times each test with the same random multinomial synthetic data and then averaged the NMI results and running times. In each test we compared our CUDA/C++ and Julia solutions to each other (**sklearn** does not support multinomial components when the number of components is unknown). Unlike in the case of Gaussian components, here, with multinomials, we observed that our CUDA/C++ solution was uniformly faster than our Julia solution. On average the former was 5 times faster than the latter. The results (using $N = 10^6$) are displayed in Figure 9. The corresponding NMI scores appear in Figure 10.

6.4. Real data

We also tested our solution on real datasets, where, as pre-processing, we used Principal Component Analysis (PCA) to reduce the dimensionality. For DPGMM we used the following datasets: *mnist* (LeCun *et al.* 1998) ($N = 60000, d = 32, K = 10$); *fashion mnist* (Xiao, Rasul, and Vollgraf 2017) ($N = 60000, d = 32, K = 10$) and *ImageNet-100* (Deng, Dong, Socher, Li, Li, and Fei-Fei 2009) ($N = 125000, d = 64, K = 100$). We compared our CUDA/C++ and Julia implementations with **sklearn** `BayesianGaussianMixture`. For DPMNMM we used *20newsgroups* (Lang 1995) ($N = 11314, d = 20000, K = 20$), and compared our CUDA/C++ and Julia implementations to each other (**sklearn** does not support multinomial components when the number of components is unknown). In the DPMNMM case, when the dimension was very high ($d = 20,000$) the CUDA/C++ package was 188 times faster than Julia. In Figure 11 we compared the running times on each dataset between the different packages. It is clear from the figure that our CUDA/C++ package is much faster than the two others, and that our Julia package is much faster than **sklearn**. In Figure 12 we displayed the NMI comparison of the different packages. On *ImageNet-100*, our CUDA/C++ and Julia packages are almost equal to **sklearn** (with a minor difference of 0.013). In all other cases, our solutions were more accurate than **sklearn**. Moreover, even though, on *ImageNet-100*, our solutions did not score higher NMI values than **sklearn**, the K value that we predicted was much more accurate. That is, while the true K was 100, **sklearn** predicted $K = 500$ (which was the value of the upper bound it was given). In contrast we predicted on average $K = 96.8$ (with a standard deviation of 17.78).

7. Summary and discussion

We extended the DPMM inference method from Chang and Fisher III (2013) by efficient and easily-modifiable implementations for high-performance distributed sampling-based inference. Our software can be adopted by practitioners in an easy way where the user is free to choose between either a multiple-machine, multiple-core, CPU implementation (written in Julia) and a multiple-stream GPU implementation (written in CUDA/C++). We also provide an optional Python wrapper which hides the CPU and GPU implementations as a single package with one interface. The proposed implementation supports both Gaussian and Multinomial components. However, it is also easy to extend the code to support other exponential families. We also tested the proposed implementation for learning models from real datasets as *mnist*, *fashion mnist*, *ImageNet-100* and *20newsgroups*. We compared our solution to **sklearn** and showed that for large and high dimensions our solution achieves the same, or better, accuracy while being much faster. Empirically, we found that when using high-dimensional Gaussians on a single machine our GPU package is faster than any other solution that is currently publicly available. For Multinomial components, our GPU package showed even better results which are by at least two orders of magnitude faster than any other existing solution. We provided a solution for cross operating systems platforms (Windows & Linux). Our tests can be easily reproduced via a **Jupyter** notebook included with our Python package.

7.1. Future directions

A promising avenue for future development in this field lies in the adaptation of DPMM inference methods to modern deep learning frameworks such as **JAX**, **PyTorch**, and **TensorFlow**.

Open source	Usage	Link
cnpy	Read and write models from npy format	https://github.com/rogersce/cnpy
zlib	Required by cnpy	https://github.com/madler/zlib
dirichlet_distribution	Dirichlet distribution	https://github.com/gcant/dirichlet-cpp
logdet	Log determinant for Eigen using Cholesky	https://gist.github.com/redpony/fc8a0db6b20f7b1a3f23
vcflib	Log(gamma) and multi-normal sampling	https://github.com/vcflib/vcflib
eigen	Matrix and vector operations	https://eigen.tuxfamily.org
stats	Sampling from an inverse-Wishart	https://www.kthohr.com/statslib.html
gcm	Required by stats	https://github.com/kthohr/gcm
jsoncpp	Read and write JSON files	https://github.com/open-source-parsers/jsoncpp
MIToolbox	Calculates NMI	https://github.com/Craigacp/MIToolbox
opencv	Drawing (for debugging)	https://opencv.org/

Table 4: C++ open-source packages used in the proposed implementation.

Open source	Usage	Link
Clustering.jl	Evaluation metrics	https://github.com/JuliaStats/Clustering.jl
DistributedArrays.jl	Distribute Computations	https://github.com/JuliaParallel/DistributedArrays.jl
Distributions.jl	Probability Distributions	https://github.com/JuliaStats/Distributions.jl
JLD2.jl	Saving and restoring checkpoints	https://github.com/JuliaIO/JLD2.jl
NPZ.jl	Compatibility with Numpy data	https://github.com/fhs/NPZ.jl
SpecialFunctions.jl	Log Gamma function	https://github.com/JuliaMath/SpecialFunctions.jl

Table 5: Julia open-source packages used in the proposed implementation.

This integration could serve as a catalyst for several significant advancements. By leveraging these frameworks, it would become possible to create abstractions for utilizing different types of hardware resources, including CUDA cores and Tensor cores, potentially leading to further performance optimizations across various GPU architectures. Furthermore, the distributed nature of these frameworks could be harnessed to enhance the scalability and

generalization capabilities of DPMM inference, enabling efficient utilization of clusters or cloud computing resources. This integration could also pave the way for improved usability through the development of higher-level APIs and seamless integration with existing deep learning workflows. Ultimately, such developments could result in a more versatile, powerful, and user-friendly approach to DPMM inference, capable of adapting to the evolving landscape of machine learning and data analysis.

Acknowledgments

In Table 4 we specify all the open sources that we use in the CUDA/C++ package. In Table 5 we specify all the open sources that we use in the Julia package.

References

- Antoniak CE (1974). “Mixtures of Dirichlet Processes with Applications to Bayesian Nonparametric Problems.” *The Annals of Statistics*, **2**(6), 1152–1174. doi:10.1214/aos/1176342871.
- Chang J (2014). *Sampling in Computer Vision and Bayesian Nonparametric Mixtures*. Ph.D. thesis, Massachusetts Institute of Technology.
- Chang J, Fisher III JW (2013). “Parallel Sampling of DP Mixture Models Using Sub-Cluster Splits.” In *Advances in Neural Information Processing Systems*.
- Dempster AP, Laird NM, Rubin DB (1977). “Maximum Likelihood from Incomplete Data via the EM Algorithm.” *Journal of the Royal Statistical Society B*, **39**(1), 1–38. doi:10.1111/j.2517-6161.1977.tb01600.x.
- Deng J, Dong W, Socher R, Li LJ, Li K, Fei-Fei L (2009). “Imagenet: A Large-Scale Hierarchical Image Database.” In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255. IEEE.
- Dinari O, Yu A, Freifeld O, Fisher III J (2019). “Distributed MCMC Inference in Dirichlet Process Mixture Models Using Julia.” In *IEEE CCGRID Workshop on High Performance Machine Learning*, pp. 518–525.
- Gelman A, Stern HS, Carlin JB, Dunson DB, Vehtari A, Rubin DB (2013). *Bayesian Data Analysis*. Chapman & Hall/CRC. doi:10.1201/b16018.
- Ghosal S, Van der Vaart A (2017). *Fundamentals of Nonparametric Bayesian Inference*. Cambridge University Press.
- Hastings WK (1970). “Monte Carlo Sampling Methods Using Markov Chains and Their Applications.” *Biometrika*, **57**(1), 97–109. doi:10.1093/biomet/57.1.97.
- Jain S, Neal RM (2004). “A Split-Merge Markov Chain Monte Carlo Procedure for the Dirichlet Process Mixture Model.” *Journal of Computational and Graphical Statistics*, **13**(1), 158–182. doi:10.1198/1061860043001.

- Lang K (1995). “Newsweeder: Learning to Filter Netnews.” In *Proceedings of the Twelfth International Conference on Machine Learning*.
- LeCun Y, Cortes C, Burges CJC (1998). “The MNIST Database of Handwritten Digits.” URL <http://yann.lecun.com/exdb/mnist/>.
- Müller P, Quintana FA, Jara A, Hanson T (2015). *Bayesian Nonparametric Data Analysis*. Springer-Verlag. doi:10.1007/978-3-319-18968-0.
- Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011). “**scikit-learn**: Machine Learning in Python.” *Journal of Machine Learning Research*, **12**, 2825–2830.
- Pitman J (2002). “Combinatorial Stochastic Processes.” *Technical Report 621*, Department of Statistics, UC Berkeley.
- Robert C, Casella G (2013). *Monte Carlo Statistical Methods*. Springer-Verlag. doi:10.1007/978-1-4757-4145-2.
- Smith WD (2002). “How To Sample from a Probability Distribution.” *Technical Report 17*, NEC Research.
- Sudderth EB (2006). *Graphical Models for Visual Object Recognition and Tracking*. Ph.D. thesis, Massachusetts Institute of Technology.
- Wang R, Lin D (2017). “Scalable Estimation of Dirichlet Process Mixture Models on Distributed Data.” In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*.
- Xiao H, Rasul K, Vollgraf R (2017). “Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms.” *arXiv 1708.07747*, arXiv.org E-Print Archive. doi:10.48550/arXiv.1708.07747.

Affiliation:

Or Dinari, Raz Zamir, Oren Freifeld
Department of Computer Science
Ben-Gurion University
Beer-Sheva, Israel

E-mail: dinari@post.bgu.ac.il, razzam@post.bgu.ac.il, orenfr@cs.bgu.ac.il

John W. Fisher III
Computer Science & Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts, United States of America
E-mail: fisher@csail.mit.edu