Massively Parallel Expectation Maximization Using Graphics Processing Units

Muzaffer Can Altinigneli
University of Munich
Munich, Germany
can.altinigneli@lmu.de

Claudia Plant
Helmholtz Zentrum München
Technische Universität München
claudia.plant@helmholtz-muenchen.de

Christian Böhm
University of Munich
Munich, Germany
boehm@ifi.lmu.de

ABSTRACT

Composed of several hundreds of processors, the Graphics Processing Unit (GPU) has become a very interesting platform for computationally demanding tasks on massive data. A special hierarchy of processors and fast memory units allow very powerful and efficient parallelization but also demands novel parallel algorithms. Expectation Maximization (EM) is a widely used technique for maximum likelihood estimation. In this paper, we propose an innovative EM clustering algorithm particularly suited for the GPU platform on NVIDIA's Fermi architecture. The central idea of our algorithm is to allow the parallel threads exchanging their local information in an asynchronous way and thus updating their cluster representatives on demand by a technique called Asynchronous Model Updates (Async-EM). Async-EM enables our algorithm not only to accelerate convergence but also to reduce the overhead induced by memory bandwidth limitations and synchronization requirements. We demonstrate (1) how to reformulate the EM algorithm to be able to exchange information using Async-EM and (2) how to exploit the special memory and processor architecture of a modern GPU in order to share this information among threads in an optimal way. As a perspective Async-EM is not limited to EM but can be applied to a variety of algorithms.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data mining
; I.3.1 [**Hardware Architecture**]: Graphics processors

Keywords

Expectation Maximization; Graphics Processing Unit; CUDA; Fermi

1. INTRODUCTION

Leading experts from the data mining community voted the EM algorithm to be among the top ten algorithms having most impact on data mining research [1]. The EM algorithm repeats two operations in a loop: The expectation operation (E) gradually assigns each sample to the clusters according to its current likelihood, and the maximization operation (M) updates the cluster representatives such that they are optimal for the currently assigned samples. Thus we can say that EM belongs to the paradigm of alternating least squares (ALS) algorithms: The E-step determines an optimal assignment provided that the cluster representatives are fixed, and the M-step determines an optimal set of cluster representatives under the condition that the sample-to-cluster assignment is fixed. The classical variant of the EM-algorithm (called Batch-EM) performs a complete iteration where the E-operation is executed for every sample. After this E-phase the cluster representatives are determined in a separate M-phase. An alternative algorithm called *EM with incremental updates* or short *Incremental-EM* performs the E-step for a single sample. If the sample changes its cluster assignment to a sufficiently large degree, the affected cluster representatives are immediately changed before considering the next sample. In a non-parallel environment, this modification of the algorithm does not cause much computational overhead. But since the new set of cluster representatives is usually better than the old set, this algorithm takes considerably fewer iterations until convergence [2].

However, in a massively parallel environment like a GPU with hundreds of processors, we have to consider that the update of a cluster representative causes an access to a centralized data structure. This is undesirable for two reasons: (1) Global exchange of information is possible only through the device RAM memory which has a limited bandwidth and (2) the processes have to be synchronized when concurrently accessing such global information. This observation leads us to the idea of Asynchronous Model Updates (Async-EM). Rather than updating the global cluster representatives upon every considerable membership change, the single processes should collect a certain number of these updates and update the global cluster representatives often enough to speed up the convergence but rarely enough to spare memory bandwidth and avoid synchronization overhead. We additionally exploit the hierarchical structure of processors (which are grouped into *multiprocessors* having more efficient access to the so-called *shared memory*), and exchange more often the updates of the processes (called *thread group*) that run on the same multiprocessor.

Contributions

In this paper, we propose a massively parallel variant of the EM algorithm suitable for GPU environments. Our main idea is to make incremental model updates using the expectations obtained in parallel from mini-batches of the whole data set and to merge all these updates into a global model update which is used to calculate the expectations in the next set of parallel mini-batches. To summarize the benefits:

- **Fast Convergence.** We demonstrate that asynchronous model updates considerably speed up the convergence of the EM algorithm.
- **Intelligent usage of memory bandwidth.** We combine the idea of asynchronous model updates with an efficient algorithm for model consolidation which exploits the special characteristics of the memory hierarchy of a modern GPU: very fast registers, fast shared memory allowing for coalescing accesses within a thread group, and slower global memory.
- **Leading to substantial speed-up.** Our experiments demonstrate that the combination of both ideas provides 720 times speed-up over a single-core CPU implementation of Batch-EM and still 2 times speed-up over a state-of-the-art GPU implementation of Batch-EM.

Notations

Clusters are modeled by the Gaussian probability density function. The conditional probability of an object x given a cluster C is provided by:

$$P(x|C) = \frac{1}{\sqrt{(2\pi)^d \cdot |\Sigma|}} \cdot \mathbf{e}^{-\frac{1}{2}(x-\mu)^T \cdot \Sigma^{-1} \cdot (x-\mu)}, \tag{1}$$

with mean $\mu = (\mu_1, ..., \mu_d)^\intercal$ and $d \times d$ covariance matrix Σ. Each cluster is defined by μ, Σ and the weight w which it has in the overall model. The model $\mathcal{C}$ consists of a number of k such clusters $C_1, ..., C_k$. The overall probability of a sample x to be generated by the k clusters of the model $\mathcal{C}$ is provided by:

$$P(x) = \sum_{(w,\mu,\Sigma)\in\mathcal{C}} w \cdot P(x|(w, \mu, \Sigma)). \tag{2}$$

The degree (probability) to with which an object x belongs to a cluster $C = (w, \mu, \Sigma)$ is provided by the Bayes theorem:

$$P(C|x) = w \cdot P(x|C)/P(x). \tag{3}$$

In the maximization phase, the model parameters (w, μ, Σ) of each cluster C are updated according to the formulas:

$$w=\frac{1}{n}\sum_{x\in D}P(C|x), \mu=\frac{1}{nw}\sum_{x\in D}xP(C|x), \Sigma=\frac{1}{nw}\sum_{x\in D}xx^\intercal P(C|x). \tag{4}$$

The objective function of the EM algorithm to maximize the log-likelihood of the data D with respect to the model parameters $\mathcal{C}$, denoted by $LL(D, \mathcal{C})$ is provided by:

$$\max LL(D, \mathcal{C}) = \sum_{x\in D} \log(P(x)). \tag{5}$$

Batch-EM performs a loop until convergence which determines first the cluster memberships $P(C|x)$ of all objects $x \in D$ according to Eq. 3 (E-phase) and after that the model parameters (w, μ, Σ) according to Eq. 4 (M-phase).

Incremental-EM performs a loop until convergence which does the following combined EM-operation for each object $x \in D$: Determine the difference between the current cluster membership $P^{(i)}(C|x)$ and that in the previous iteration $P^{(i-1)}(C|x)$ and if required update the model parameters (w, μ, Σ) immediately according to Eq. 4.

The remainder of this paper is organized as follows: Section 2 summarizes the related work. Section 3 introduces the NVIDIA's Fermi GPU Architecture. Section 4 elaborates our technique Async-EM and describes the GPU implementation. Section 5 contains an experimental evaluation and Section 6 concludes the paper.

2. RELATED WORK

The algorithm Newscast EM [3] specified for the peer-to-peer scenario avoids a centralized M-step. Approximate decentralized updates improve parallelism and reduce communication costs. Randomly selected pairs of nodes exchange their local parameter estimates and combine them by weighted averaging.

In [4], the authors propose a technique for estimating GMMs for multimedia indexing in a peer-to-peer network. Each node stores a considerable amount of data. In a first step, each node estimates local parameters by running EM on its own data. The resulting models are then combined minimizing the Kullback Leibler Divergence which is implemented by transmitting the model parameters only.

From the area of sensor networks also some variants of distributed EM algorithms have been proposed [5–7]. The power supply of sensor nodes is often limited. Algorithms minimizing communication and computation costs are thus required. Most related to our work, the approaches [6, 7] exploit the fact that in EM local information can be collected independently and then be combined for obtaining the global parameter estimates, but with a different focus: While [6, 7] focus on reducing the communication costs in a sensor networks, we additionally focus on asynchronous model updates as a strategy to speed up convergence.

GPU clusters are organized like CPU clusters. A GPU cluster node has one or more powerful data parallel computing GPUs, which are composed of hundreds of lightweight computing cores. Parallelization is possible by distributing the work over the computing cores for independent computation of expectation and later computing the model parameter updates locally or in a centralized way similar to the P2P computing. The two major challenges we focus on are the synchronization of cores and the organization of the communication over the shared and main memory. Some approaches proposed implementations of the EM algorithm on GPUs, e.g. [8–10]. In all of these works, the E-Step and the M-step are separated and split into multiple kernel calls and all of them are GPU implementations of the classical Batch-EM algorithm. In our work, we have combined E- and M- steps into a single kernel call.

To summarize, we can distinguish between approaches closely following the classical EM paradigm sequentially iterating E- and M-step until convergence and focusing on certain important goals in a distributed environment, such as reducing the communication cost in [6, 7] or adapting the algorithm to the SIMD-environment of the GPU [8–10]. To further reduce communication costs, the classical EM algorithm has been modified by approximate updates [3, 5] or merging of local models [4].

In this paper, we investigate, if and to which extent the strategy of asynchronous model updates accelerates the convergence of the EM algorithm. Furthermore, we propose Async-EM, an efficient algorithm exploiting the memory hierarchy of modern GPUs.

3. FERMI GPU ARCHITECTURE

The Expectation phase of Async-EM is fully parallelizable, because each sample can be associated to each cluster independently. We demonstrate that the calculations in Maximization phase can be also effectively parallelized using reduction techniques. The parallel nature of Async-EM at the data-level motivates us to implement it on the NVIDIA Fermi Architecture. Detailed information about the Fermi Architecture can be found in NVIDIA's web-site and in [11].

The Compute Unified Device Architecture (CUDA) has a heterogeneous execution model in which the CPU is called host, whereas the GPU is called device. The developer can use an extension of the standard programming language C, called CUDA-C, for application development. CUDA gives the developer the opportunity of utilizing the computational elements and the memory of the GPUs. CUDA has two levels of abstraction for parallel computation: *Threads* are organized into *thread blocks* and thread blocks are organized into a *grid*. The code that works for the whole grid is called *kernel*. The whole work is divided into manageable sizes by using thread blocks.

Thread scheduling is handled by the GPU hardware. In Fermi architecture, the thread block scheduler called the GigaThread engine assigns thread blocks (maximum 8 thread blocks) to a multi-threaded SIMD processor called *Streaming Multiprocessor* (SM). A simplified block diagram of the SM is shown in Figure 1. Each Fermi SIMD processor has two SIMD warp schedulers and 32 SIMD lanes called *Streaming Processors* (SP) or CUDA cores, 16 load-store units (LSUs) and 4 special function units (SFUs). Each floating/integer SIMD instruction can be dispatched to 16 SIMD lanes, therefore it takes 2 clock cycles per warp to complete a SIMD integer/float instruction. Fermi SM has two instruction dispatch units, which can dispatch at least 2 instructions per clock cycle, therefore the effective Instructions Per Clock (IPC) is equal to 32 floating/integer operations per clock per SM. Similarly, it takes 8 clock cycles to complete a special instruction per warp in a SFU. Each SIMD thread running on a SIMD processor has its own program counter and scheduled by the warp scheduler. The warp scheduler can keep track of up to 48 threads of SIMD instructions. Hence, at most 1536 CUDA threads can be scheduled on a multi-threaded SIMD processor. A SIMD instruction is executed for all CUDA threads in the same warp, therefore all threads in the same warp are implicitly synchronized. Branch instructions will be serialized, in case branch divergence occurs inside a warp. Not all threads are active during execution of divergent branches, which means idle clock cycles for some of the SIMD lanes and performance penalty. Therefore, branch divergence inside warps

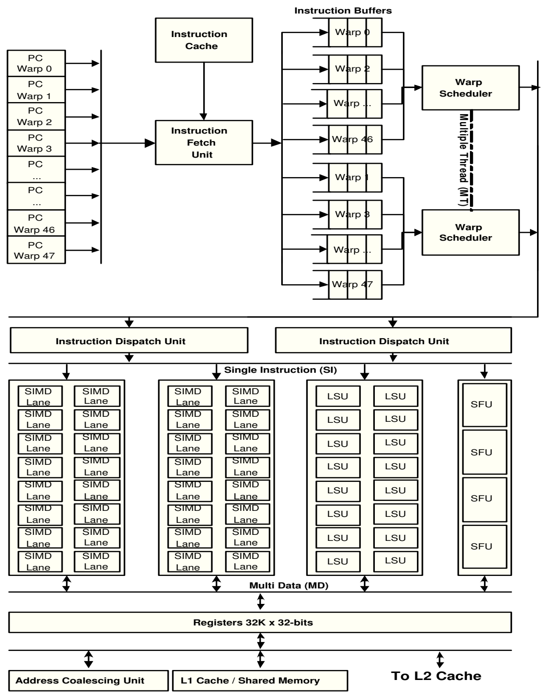

Figure 1: The SIMD warp scheduler has 48 independent threads of SIMD instructions with 48 PCs. Each Instruction Dispatch Unit can issue a SIMD instruction to 16 SIMD Lanes or 16 Load/Store Units (LSUs).

shall be avoided. We have seen that several levels and types of parallelism are possible using different levels and types of abstractions. Multiple instructions can be scheduled to different SIMD multiprocessors using multiple data (MIMD). Multiple threads of SIMD instructions can run on SIMD processors. Independent instructions from the same warp can be scheduled to the SIMD lane pipelines which is called instruction level parallelism. All of these different types of parallelisms are called Single Instruction Multiple Level (SIMT) thread model by NVIDIA.

It is important to understand the Fermi memory model to effectively parallelize EM. The host can read or write to off-chip DRAM, but not to on-chip memories used only by the device. Each multi-threaded SIMD processor has its own on-chip *Shared Memory*/L1 Cache in total 64 KB used by all blocks assigned to the SIMD processor. The threads within a block can use shared memory together by taking care of synchronization issues. 32 SIMD lanes are fed by in total 32K four byte on-chip *registers*, which are not shared by the threads. Hence, if all 1536 threads are active (at 100% occupancy), each thread can have at most 21 registers. A thread can not have more than 63 registers (achieved at 33% occupancy). Other than this, a thread has its own not shared section on off-chip RAM which is called the *local memory*. Local memory contains the stack frame, spilling registers and arrays which do not fit into registers. The difference of this private off-chip RAM section from other RAM sections is that it is cached into L1. Fermi does not have huge caches like CPUs or deep pipelines like vector architectures. Instead, it amortizes the memory latency mainly by using thread and instruction level parallelism extensively. Caches help to decrease memory demand to off chip RAM during local memory accesses or function calls and improve the effective memory bandwidth in case of uncoalesced memory accesses.

4. ASYNCHRONOUS EM ON GPU

The E-operation can be perfectly parallelized on GPUs as well as on any other parallel architecture by distributing the samples to different processors. However, when taking a closer look at Incremental-EM, its strict sequential philosophy becomes obvious: The cluster models (w, μ, Σ) have to be updated *before* the cluster membership of the *next* object is determined. Therefore, Incremental-EM is in general not suitable for this kind of parallelization. To also have the advantage of faster convergence through more frequent model updates, our idea is to weaken the strict sequential policy and to allow parallel processes to perform their E-operations independently. But rather than waiting until the E-operations of all objects have been processed, we allow the processes to collect their private changes locally, to use them immediately, and to exchange them from time to time in an asynchronous way. When these local model updates are exchanged, the different sets of cluster representatives have to be merged together, which we call the *consolidation*. The rate of the exchange and consolidation can be adapted to the bandwidth with which the processors are connected and to the conflict rate of synchronization. CUDA-based GPUs are composed of a set of streaming multiprocessors (SM) each of which consists of several processors. In analogy to this hierarchy, the threads are also grouped into thread-groups. All threads from one thread group are executed on the same SM, and can exchange data through the fast shared memory (cf. Section 3). Threads of different thread groups can exchange information only through the much slower global memory. We will show how this hierarchy of processes, processors, and memory units can be exploited to share the cluster models in an optimal way. Figure 2 displays the algorithm Async-EM in Pseudocode.

4.1 Parameter Representation

First we show how the model parameters can be represented in a way that facilitates consolidation. Since the vector $\mu = (\mu_1, ..., \mu_d)^\intercal$ and the covariance matrix $\Sigma = [\sigma_{i,j}]$ are defined by fractions, it is inefficient and numerically unstable to directly update these parameters. It is easier to update the following parameters instead:

$$W = \sum_{x \in D} P(C|x), \tag{6}$$

$$X_i = \sum_{x \in D} x_i \cdot P(C|x), \qquad 1 \le i \le d, \tag{7}$$

$$Q_{i,j} = \sum_{x \in D} x_i \cdot x_j \cdot P(C|x), \qquad 1 \le i, j \le d. \tag{8}$$

Besides, Fermi Architecture uses a new standard implementation *Full IEEE 754-2008 32-bit and 64-bit precision*, which improves the accuracy considerably compared to previous architectures. Our incremental method is relatively insensitive to numerical inaccuracies, because we do not use batch update in which numerical errors accumulate quickly with massive amounts of data. Obviously, the two original parameters $w = W/n$ and $\mu_i = X_i/W$ can be easily derived from our new consolidated parameters. But also the more difficult $\Sigma = [\sigma_{i,j}]$ can be efficiently derived as we can see in the following:

$$\begin{aligned}\sigma_{i,j} &= \frac{\sum_{x \in D}(x_i - \mu_i)\cdot(x_j - \mu_j)\cdot P(C|x)}{\sum_{x \in D} P(C|x)} \\ &= \frac{1}{W}\left(\sum_{x \in D}(x_i x_j - \mu_i x_j - \mu_j x_i + \mu_i \mu_j)\cdot P(C|x)\right) \\ &= \frac{Q_{i,j}}{W} - \frac{X_i X_j}{W^2}.\end{aligned}$$

Since all parameters of a cluster model $\mathcal{C} = (W, X_i, Q_i)$ are now defined in terms of a sum, the consolidation simply reduces to summing up the different local parameter sets. The computation of the inverse and determinant of the covariance matrix Σ is facilitated by the Cholesky decomposition $\Sigma = L \cdot L^\intercal$ where L is a left triangular matrix. Cholesky decomposition is the least expensive method for the inversion of a symmetric and positive definite matrix. It can be empirically shown even for non-parallel versions of EM clustering [12] that doing Cholesky decomposition in an asynchronous way clearly improves the overall runtime provided that the two extremes (decomposing almost only once per iteration and decomposing almost at every update) are avoided. The performance curve yields an extremely wide and stable minimum (*bathtub curve*) as long as these extremes are avoided. Therefore, our method performs Cholesky decomposition at every asynchronous update operation which naturally avoids the inefficient extremes. We show next how consolidation between different processors of a streaming multiprocessor and between different multiprocessors works.

4.2 Model Consolidation

We decompose the data set into subsequences called *superchunks* and each superchunk into subsequences called *chunks*. A chunk is processed by parallel threads in one common thread group (on the same SM). Each thread is responsible for one or more samples. First, all threads read the most current set of cluster models from the global memory to the shared memory, read the next sample to be clustered from the global memory to the registers and determine the gradual cluster memberships and the differences of these memberships to the previous iteration. After updating their local cluster models they might also process further samples.

When the chunk is finished the threads cooperatively consolidate their cluster models by a technique called *parallel sum reduction*. Then the next chunk is processed, and this is repeated until no more chunk is associated to the respective streaming multiprocessor. All streaming multiprocessors have associated a set of chunks which they are working on. The cluster models are exchanged and consolidated inside a SM after every *chunk*, and the fast *shared memory* is used for this parallel sum reduction.

After all threads inside the current thread block have processed their chunks (which form together a superchunk), a single thread from each thread block consolidates its own cluster model with the global model with atomic operations. For this consolidation, the global memory is used. Since the global memory has by a factor of 8 times lower memory bandwidth than the shared memory, the consolidation between thread blocks should be executed much less frequently than the consolidation *inside* a thread block.

```
Algorithm Async-EM runs on all of the CPUs in parallel
CPU-Thread 1 only part {
01- Randomly initialize global models $W^g_{cpu}, Xi^g_{cpu}, Qi^g_{cpu}$
      resident in the shared memory visible to all CPU-Threads.
}
02- Synchronize CPU-Threads
03- counter:= 1;
  repeat
04-   Initialize local CPU models $W^l_{cpu}, Xi^l_{cpu}, Qi^l_{cpu}$
        using the global model parameters.
05-   Copy models to GPU memory $W^l_{gpu}, Xi^l_{gpu}, Qi^l_{gpu}$.
06-   Call Async-EM-Par. Kernel running on the GPU,
        each CUDA Thread Block responsible for a super-chunk.
07-   Copy local updates $\delta W^l_{gpu}, \delta Xi^l_{gpu}, \delta Qi^l_{gpu}$
        back to the CPU memory $\delta W^l_{cpu}, \delta Xi^l_{cpu}, \delta Qi^l_{cpu}$.
08-   Synchronize CPU-Threads.
09-   Increment counter.
    CPU-Thread 1 only part {
10-   Merge local updates $\delta W^l_{cpu}, \delta Xi^l_{cpu}, \delta Qi^l_{cpu}$ from
        CPU-Threads in global model $W^g_{cpu}, Xi^g_{cpu}, Qi^g_{cpu}$.}
  until counter = $Async.EMIter.Limit$;
end Algorithm Async-EM

Algorithm Async-EM-Par. GPU Kernel
  for each $chunk \in superchunk$ in parallel [CUDA Thr. Blk.]
01-   Load $W^{l-Thr.Blk}_{gpu}, Xi^{l-Thr.Blk}_{gpu}, Qi^{l-Thr.Blk}_{gpu}$
        into the GPU shr.mem. from $W^l_{gpu}, Xi^l_{gpu}, Qi^l_{gpu}$.
    for each sample $x_{[1..spt]} \in chunk$ in parallel [CUDA Thr.]
      for each cluster $c \in \mathcal{C}$
02-     Cholesky decomposition $\Sigma = [\sigma_{i,j}]$ at every
          $Stepsize.Cholesky$.
03-     Compute $P(x|C)$, $P(x)$ and $P(C|x)$ (cf. Eq.1-3)
04-     if $\sum_{C \in \mathcal{C}_{old}} P(C|x) - \sum_{C \in \mathcal{C}_{new}} P(C|x) > \epsilon$
05-       Update $W^{l-Thr.Blk}_{gpu}, Xi^{l-Thr.Blk}_{gpu}, Qi^{l-Thr.Blk}_{gpu}$
            in the GPU shr.mem. (cf. Eq. 6 - 8).
  GPU-Thread 0 only part in each Thr. Blk. {
06- Atomic update $\delta W^l_{gpu}, \delta Xi^l_{gpu}, \delta Qi^l_{gpu}$.}
end Async-EM-Par. GPU Kernel
```

Figure 2: Async-EM and parallel GPU Kernel Pseudocodes. $Async.EMIter.Limit$ **= 100,** ϵ **= 0.01,** $Stepsize.Cholesky$ **= 1,** spt **= 4 (samples per thread) are the optimal algorithm parameters.**

The parallel sum reduction, which is shown in Figure 3, is a technique in which the threads cooperatively consolidate their cluster models automatically avoiding shared memory access conflicts. Additionally, the access pattern is selected in a way that it facilitates efficient access to the different shared memory banks (the so-called coalesced accesses). The idea is that every thread knows an address where a fixed partner thread stores its local cluster models. The thread consolidates its model with that of the partner thread. In each step, the number of threads which are still alive decreases by half until only one thread is left and only *one* set $\mathcal{C}$ of k cluster models is known.

4.3 Comprehensive Example

Figure 4 gives a comprehensive example and visualizes our algorithm for a data set of $n = 2048$ samples. The dataset is decomposed into two superchunks each of which is de-

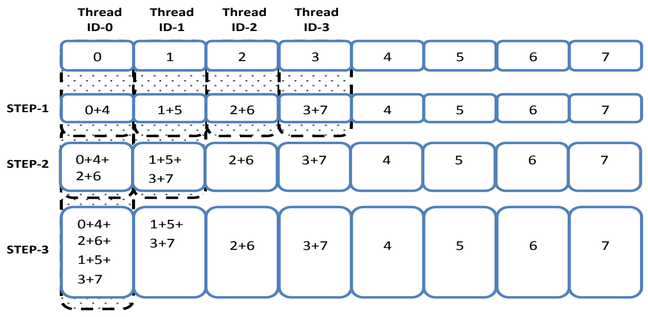

Figure 3: The Parallel Sum Reduction. Each of the thread is handling a single local parameter. Only half of the threads are active at each step. As an example, to consolidate the local statistics for 8 local parameters, 3 (= $\log_2 8$**) steps are necessary.**

composed into 8 chunks. We consider here a grid which is composed of two thread blocks. It is possible that both thread blocks are running on the same SM consecutively or they are scheduled to different SMs by the GigaThread engine and run completely in parallel. On each of the thread blocks, a total of 128 threads run effectively in parallel assigning samples to clusters and updating their local cluster models. After their completion, the 128 threads on each thread block consolidate their local updates using the parallel sum reduction technique on Eq. 6-8 in the fast shared memory. Then, the 128 threads on each thread block are dedicated to the samples of the next chunk which are clustered according to the consolidated model. When all chunks of Superchunk #1 on all thread blocks have been processed, the thread blocks consolidate their local models using atomic operations to the global memory. When all superchunks have been processed in this way, we start again with Superchunk #1. Convergence is achieved when there has been no update in both superchunks. It might also happen that convergence is achieved after the processing of Superchunk #1 (if Superchunk #2 has not caused any cluster update in the previous iteration).

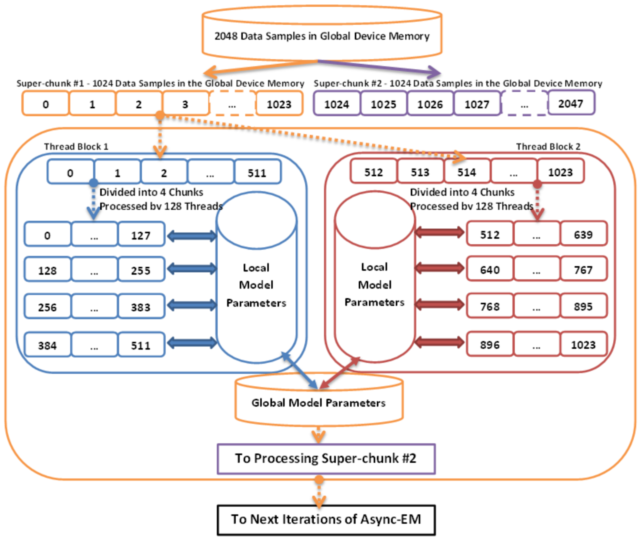

Figure 4: A Single Async-EM Iteration. All local parameters fit into the shared memory. Global model parameters updated at the end of the kernel.

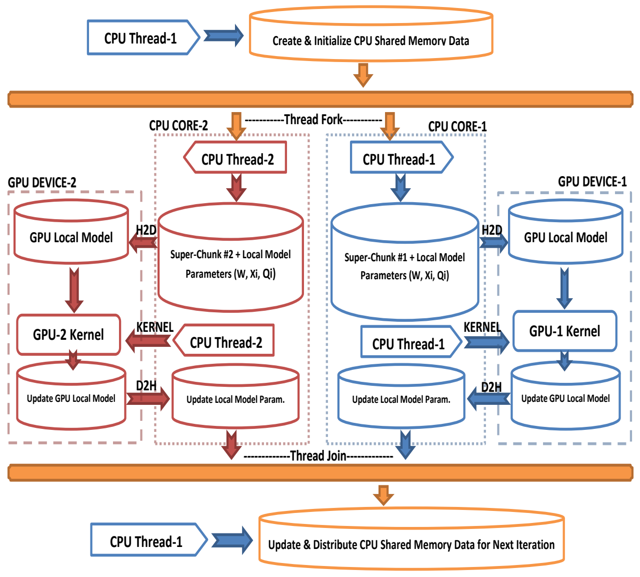

Figure 5: Two GPUs Extension of the Single Async-EM Iteration. Each CPU thread is responsible for host-to-device (H2D), device-to-host (D2H) memory transactions and kernel calls.

Multi-GPU Extension

We extend the Async-EM to two GPUs as shown in Figure 5. CPU Thread-1 is responsible to create the memory structures shared by the two CPU threads. Each CPU Thread calls its own GPU kernel and manages its own GPU memory transactions. The data samples in the Superchunk #1 (or Superchunk #2) are assigned and copied to GPU-1 (or GPU-2) and reside in the device during all of the Async-EM iterations. The master CPU thread (Thread-1) consolidates the final global model parameters into the memory shared by the two CPU threads for the next Async-EM iterations. Only consolidated model parameters, which have much smaller memory size compared to the superchunk data samples, are transferred to GPUs between Async-EM iterations.

Multi-Stream Extension

The local model parameters together with the Superchunk #1 (or Superchunk #2) do not fit to GPU memory for extremely large amount of data samples with high number of dimensions. If this condition is detected and if there is no additional GPU is available, the superchunk is divided into smaller chunks, which can fit into the device memory and transferred to the GPU part-by-part. Each smaller chunk transferred to the GPU separately and a kernel call is performed and the calculated model parameters are transferred back to the CPU Thread. But, sequential execution of memory-transactions and kernel calls (H2D-Kernel-D2H) is expensive and transfer rate on the PCI-Express bus is the bottleneck. The *CUDA Streams* are a sequence of GPU operations which are executing in the issue-order and CUDA operations from different streams can run concurrently or interleaved in the GPU. In this way, memory accesses are hidden and up to 2.4 times speed-up is obtained compared to sequential execution.

Handling Large Number of Clusters or Dimensions

The scenario shown in Figure 4 is optimistic in the sense that the local model parameters $(W, X_{1..D}, Q_{1..D})$ are copied to the shared memory on the device only once and reside there during all of the iterations of the Async-EM. Since the shared memory is limited in size, it is not feasible to store the whole local data on the shared memory for a large number of clusters or dimensions. If this condition is detected, the local parameters are copied from global device memory to shared device memory partially. For this purpose, a separate location for each thread block is reserved in the global device memory to store its own local parameters as shown in Figure 6. As an example, $X_{1..D}$ (or diagonal $Q_{1..D}$), which has originally 16 clusters, is not fitting to the shared memory. Therefore, in the first step, the local parameters of the first 8 clusters are cached to the shared memory. A second step is necessary to process the remaining 8 clusters. Finally, a single thread from each block consolidates the final local statistics into a single data structure with atomic operations both in the global device memory.

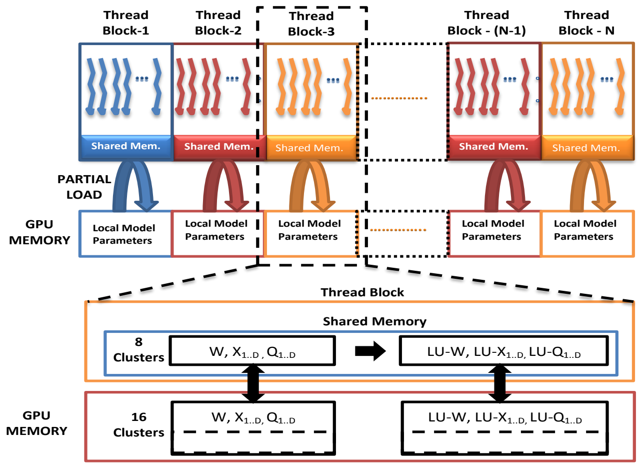

Figure 6: Partial loading and updating of the local parameters in case of large number of clusters or dimensions (Q is diagonal).

5. EXPERIMENTS

Firstly, we compare convergence, modeling error and execution time performances of the Batch-EM, Incremental-EM and Async-EM algorithms using synthetic data sets. We change the underlying characteristics of the data sets, e.g., model parameters and the overlapping of the clusters to observe the impact on the performance values. Following this point, we compare GPU performance against the CPU performance of the gmdistribution.fit function of the MATLAB Statistics Toolbox on synthetic data. We demonstrate how well the performance of the Async-EM is scaling up with an additional GPU. The execution time performance of GPU reference implementations of Batch-EM are given in [8–10] for two types (small and big) of data sets. Therefore, we create randomly two synthetic multivariate distributions with a similar cluster constitution and compare per EM iteration execution time of the Async-EM.

After that, we run the reference implementation of [8] on our GPU to make a hardware independent comparison of Batch-EM and Async-EM algorithms. We selected [8], because the source code is available and the authors demonstrated that [8] outperforms the other reference implementations. Moreover, [8] can work with non-diagonal covariance matrices. But, the most important reason is that [8] is more suitable to work with massive data sets. For example, [9] uses in M-Step $K \times D \times N$ matrix (K: number of clusters, D: number of dimensions and N: number of samples), which shall completely fit into GPU memory at once, which is not always feasible for the data sets used in this paper.

Finally, we report the convergence and the execution time performances of the Batch-EM and Async-EM GPU algorithms on two real data sets from the UCI machine learning repository [13]: We use the Statlog (Shuttle) data set and the Forest Covertype data set; Covertype is one of the biggest data sets in UCI repository.

We report the results of our Async-EM implementation on one and two GPU configurations of NVidia GTX480 Fermi GPUs. The software is developed with CUDA C Toolkit version 4.1 and is running on a PC with Intel i7-920 @2.6Ghz, 12GB RAM, Win7-64bit operating system. The CUDA experiments and their results with corresponding kernel and compiler settings are given in section 5.4.

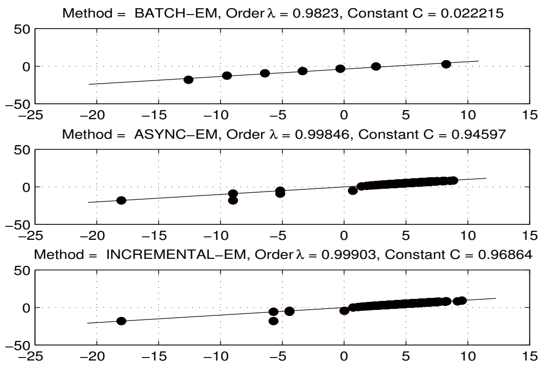

Figure 7: The Batch-EM has a $\lambda \approx 1$ and $C \approx 0$, which signs a super-linear convergence for non-overlapping clusters. Async-EM and Incremental-EM are converging sub-linearly where $\lambda \approx C \approx 1$

5.1 Convergence

Given the sequence of the target value evolution of an iterative algorithm as X = $\{x_1, x_2, x_3, ..., x^*\}$ which converges to x^*, the convergence speed is defined as: $\lim_{i\to\infty}(x^{i+1} - x^*)/(x^i - x^*)^\lambda = C$ where C is a positive constant, i is the iteration number and λ is the convergence rate. An algorithm has a super-linear convergence if $\lambda = 1$ and $C = 0$ and sub-linear convergence if $\lambda = 1$ and $C = 1$. In case of $1 < \lambda$, we have different kinds of super-linear converges, e.g. quadratic convergence if $\lambda = 2$.

By taking the logarithm of both sides of the equality, we obtain:

$$\log(x^{i+1} - x^*) = \lambda \cdot \log(x^i - x^*) + \log(C) \qquad (9)$$

The convergence rate λ and the positive constant C, in Eq.9, are calculated from the least squares fit of the likelihood function as depicted in Figure 7. Here, the X-Axis values are determined by the equation $\log(x^i - x^*)$, where x^i represents the likelihood value at ith iteration and x^* is the final converged likelihood value. Similarly Y-axis values are determined by the equation $log(x^{i+1} - x^*)$. We observe in our experiments that the convergence of Batch-EM starts behaving sub-linearly, if we increase the overlapping of the clusters.

We analyze firstly the *Log-likelihood* of different EM algorithms for the randomly created distributions initialized exactly the same way. The Async-EM algorithm slightly outperforms Incremental-EM in terms of required number of samples to be observed before convergence. On the other hand, the Batch-EM methods requires considerably more iterations or to observe the same samples several times to converge. This observation is intuitive, because more frequent updates leads faster convergence, hence the Async-EM stops earlier. The Incremental-EM has an improved convergence rate as proposed and demonstrated in [14] compared to Batch-EM, with the guarantee of the convergence to a local optimum. The Async-EM provides additional stability by treating the mini-batches (with *chunk* size) as a single sample and has the guarantee of convergence similar to the Incremental-EM. The log-likelihood does not necessarily improve at every Async-EM step.

As the next step, we focus on how fast the desired high accuracies (or undesired low *modeling-error*) for the model parameters can be reached. The EM Algorithms described in this paper are all unsupervised methods, which means that they are only targeted to optimize *log-likelihood.* The *modeling-error metric* shows quantitatively the squared error of current and desired memberships of the data samples. $P(C_{desired}|x)$ can be calculated *exactly* for synthetic data by using the model parameters from which the input data distribution is created. $P(C_{real}|x)$ is obtained at the end of the clustering. We repeat our tests several times with different initializations and cluster constitutions. The Async-EM outperforms in majority of the cases both Incremental-EM and Batch-EM methods in terms of required number of iterations to obtain accurate model parameters especially for clusters with much overlap. The Async-EM is more immune against being trapped in a local minima. We summarize that Async-EM combines the best of Batch-EM and Incremental-EM algorithms. The log-likelihood of Async-EM converges faster than the Incremental-EM. The estimated model parameters reach accurate values in a stable manner but much faster than in Batch-EM.

5.2 Comparison on Synthetic Data

The speed-up values are calculated with respect to gmdistribution.fit function of MATLAB Statistics Toolbox (ver.7.3), which is very efficiently coded and widely used. Figure 8 shows the speed-up of Async-EM CUDA implementation with respect to the gmdistribution.fit function for different data set sizes. The kernel time increases linearly proportional to number of samples as expected, because the computational complexity of EM is $O(NKD)$ for a diagonal covariance matrix. The speed-up obtained for 2 Mega (2^{21}) samples with $K = D = 8$ on a single GTX 480 GPU is approximately 137 and the kernel execution time is nearly 51 msec. We observe that the Async-EM is scaling well with an additional GPU for the same problem size as plotted in Figure 9. The speed-up obtained for 2 Mega-samples with

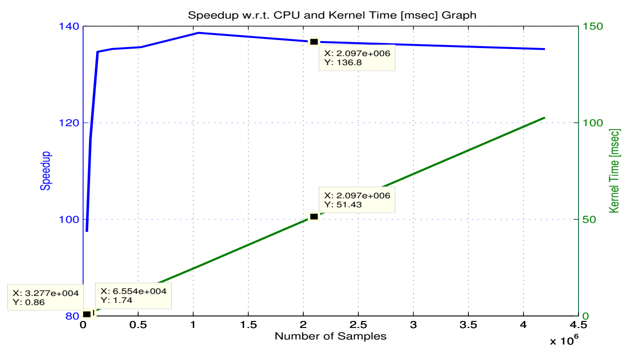

Figure 8: Speed-up of the Async-EM running on a single GPU w.r.t. MATLAB impl. of the Batch-EM on the CPU (blue Y-axis) and Kernel Execution Time plot (green Y-axis)

$K = D = 8$ on two GPUs is approximately 272, which is nearly double of the single GPU performance. The kernel execution time is nearly half of the single GPU performance and is 26 msec.

As the next step, we compare the execution time of our Async-EM single GPU implementation (marked by '*') with other Batch-EM GPU implementations published previously. Our Async-EM implementation runs nearly 13 times faster than [9] for a relatively small data set as shown in Table 1. In addition to that, we obtain nearly 15 times speed-up for a larger data set compared to [10] as shown in Table 2. For both of these data sets, [8] performs comparable to our performance. Therefore, to make a better hardware independent comparison, we obtained the source code from the authors of [8]. We changed only the data reading module of the code and use the rest of the code as it is. We compile the code in Release mode and run it on our CPU and GPU platform with a big synthetic data set. The results are given in Table 3. We obtain nearly 720 times speed-up compared to single core CPU implementation. The speed-up compared to GPU implementation of the Batch-EM is nearly 2 times, which is very similar to GTX 260 result given in Table 2. Please note that, the execution times are given as per-kernel and total execution time of an algorithm depends on how many times the kernel is called. We observe that the Async-EM algorithm requires *an order of magnitude less*

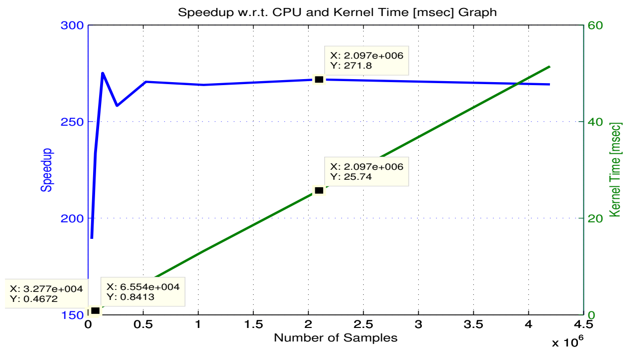

Figure 9: Two GPUs speed-up of the Async-EM w.r.t. MATLAB impl. of the Batch-EM on the CPU. The Async-EM is scaling well with an additional GPU.

kernel calls compared to Batch-EM GPU implementation with synthetic data sets.

Table 1: Comparison of the Async-EM with other Batch-EM GPU implementations for small data set

Method	N, K, D	Time/Speed-up	Ref.
Batch EM	$46800, 8, 8$	$16.2ms/1$	[9]
Batch EM	$46800, 8, 8$	$1.6ms/10$	[8]
Async-EM	$46800, 8, 8$	$1.26ms/13$	*

Table 2: Comparison of the Async-EM with other Batch-EM GPU implementations for large data set

Method	N, K, D	Time/Speed-up	Ref.
Batch EM	$2^{20}, 10, 8$	$450ms/1$	[10]
Batch EM	$2^{20}, 10, 8$	$62ms/7$	[8]
Async-EM	$2^{20}, 10, 8$	$30ms/15$	*

Table 3: Comparison of the Async-EM with Batch-EM CPU&GPU implementations of [8] on the setup used in this paper

Method, Ref	Hw.	N,K,D	Time/Speed-up
Batch EM, [8]	i7-920	$2^{20}, 10, 8$	$21.6s/1$
Batch EM, [8]	GTX480	$2^{20}, 10, 8$	$61.4ms/352$
Async-EM, *	GTX480	$2^{20}, 10, 8$	$30.0ms/720$

5.3 Comparison on Real Data

We use two real data sets from the UCI repository [13] to compare performance values of the MATLAB CPU Batch-EM and the GPU Async-EM implementations. We focus on the performance values: *Mean negative log-likelihood per sample* which is minimized in the EM, the number of the *EM iterations* until convergence and *speed-up* values obtained with a single and two GPUs. We run a preliminarily K-means clustering with 10% of the randomly selected data to initialize initial centroids. After that, the result of the K-means is used to initialize both of the EM algorithms. We obtain the statistical results given in Table 4 for the Statlog data set after 100 and for the much larger Covertype data set after 25 independent runs. The Async-EM has a slightly better log-Likelihood for Statlog and requires nearly 6 times less iterations to convergence. CPU uses the benefit of a bigger cache for a small data sets like Statlog, therefore the speed-up values are relatively lower compared to big data sets like Covertype. On the other hand, the Batch-EM has a slightly better log-Likelihood for Covertype data set. But, Batch-EM also needs nearly 4 times more iterations to convergence. The speed-up values are nearly 2.5 times higher than the smaller data set. In summary, we observe similar performance characteristics as on synthetic data.

5.4 Parallel NSight Experiment Results

All of the experiment results are given for the settings: 128 thread blocks with 128 threads per block, '-maxregcout=63', '-use_fast_math=Yes' compiler flags are set and L1/Shr.-Mem. is configured as (48/16) KB. **Occupancy:** Achieved occupancy is 35%. Theoretical occupancy is 42% limitted by registers per thread (50), which means that five Thread

Table 4: Comparison of the Async-EM with 'gmdistribution' implementation of the Statlog(Shuttle) and Covertype data sets

Method	Hw.	Dataset	N,K,D	NLogL,Sigma	Iter.,Sigma	Speed-up(1/2GPU)
Batch EM,Matlab	i7-920	Statlog	$58000, 7, 9$	$21.28, 0.48$	82.6, 58.1	1
Async-EM	GTX480	Statlog	$58000, 7, 9$	$21.08, 0.41$	14.18, 9.42	28.07/49.62
Batch EM,Matlab	i7-920	CoverType	$581012, 7, 54$	$-5.67, 0.12$	43.2, 14.6	1
Async-EM,	GTX480	CoverType	$581012, 7, 54$	$-5.63, 0.09$	21.76, 6.43	79.16/125.64

Blocks can be assigned to each SM. **Instruction Statistics:** The issued and executed instructions are nearly the same with 0% of instruction serialization. The executed Instruction Per Clock (IPC) is equal to 1.21 (upper limit is 2). SM are active at least with one warp 96% of the time and workload is distributed fairly between SMs. **Branch Statistics:** There is no divergent branch. Branch and control flow efficiencies are nearly equal to 100%, which means that our kernel will not suffer from serialized branch execution. **Issue Efficiency:** Our kernel has at least 2 eligible warps per clock cycle per SM as desired. The consecutive instructions do not have long execution dependency cycles, therefore maximum dependency $IPC = 2$. **Achieved Flops:** All floating point operations are single precision. Instruction mix is 50% ADD, 30% MUL, 15% Special and 5% Fused MUL/ADD. Achieved floating operations per second is nearly equal to $100 GFLOPS$. **Memory Statistics:** There is nearly no-overhead in external/shared/local memory operations, which means that the kernel operations are coalesced without shared memory bank conflicts and with high cache hit ratios (80% L1 hit and 50% L2 hit). Each memory request is performed in a single transaction.

6. CONCLUSION

We highlighted with our experiments that *substantial speed-up values* can be obtained with CUDA Async-EM implementation over the multi-core Batch-EM CPU implementation. Moreover, state-of-the-art Batch-EM GPU implementations are outperformed by our Async-EM GPU implementation in terms of *kernel execution time, required number of iterations to converge and to obtain accurate models parameters.*

We applied distributed EM principles to the GPU Environment, which are widely used in P2P computing and sensors networks. We regarded the streaming multiprocessors (SMs) of the GPU like the nodes of a CPU cluster. Each SM is working independently on its local data which is stored in its shared memory. Similarly, the CPU nodes distributed inside a clusters are working on their own data in their local memory space. The global model parameters are updated by each SM independently on the GPU's global device memory. Similarly, the nodes of a cluster update the global parameters in the shared memory space located on a single node or on the peer node. We extended our idea by increasing the abstraction level in our architecture. We have added another layer on top and let multiple GPUs to communicate over the shared memory space (i.e. DDRAM on the main board) and assign a CPU thread to each GPU. In future work we connect multiple-GPU nodes together with an additional layer (e.g. Open MPI) on top of our abstraction layers and form a multi-GPU cluster. Furthermore, we aim at transferring these ideas to other alternating least squares algorithms, to enable e.g. high-performance non-negative matrix factorization or inference in Bayes nets.

7. REFERENCES

[1] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, and etal., "Top 10 algorithms in data mining," *Knowledge and Information Systems*, vol. 14, no. 1, pp. 1–37, 2007.

[2] P. Liang and D. Klein, "Online em for unsupervised models," in *Proceedings of Human Language Technologies*, ser. NAACL '09. Stroudsburg, PA, USA: Association for Computational Linguistics, 2009, pp. 611–619.

[3] W. Kowalczyk and N. A. Vlassis, "Newscast em," in *NIPS*, 2004.

[4] A. Nikseresht and M. Gelgon, "Gossip-based computation of a gaussian mixture model for distributed multimedia indexing," *IEEE Transactions on Multimedia*, vol. 10, no. 3, pp. 385–392, 2008.

[5] T. Mensink, W. Zajdel, and B. Kröse, "Distributed em learning for multi-camera tracking," in *IEEE/ACM International Conference on Distributed Smart Cameras*, 2007.

[6] R. D. Nowak, "Distributed em algorithms for density estimation and clustering in sensor networks," *IEEE Transactions on Signal Processing*, vol. 51, no. 8, pp. 2245–2253, 2003.

[7] D. Gu, "Distributed em algorithm for gaussian mixtures in sensor networks," *IEEE Transactions on Neural Networks*, vol. 19, no. 7, pp. 1154–1166, 2008.

[8] A. D. Pangborn, "Scalable data clustering using gpus. master thesis, rochester institute of technology, 2010," http://apangborn.com/ms-thesis/#Download.

[9] N. S. L. P. Kumar, S. Satoor, and I. Buck, "Fast parallel expectation maximization for gaussian mixture models on gpus using cuda," in *HPCC*, 2009, pp. 103–109.

[10] A. Harp, "Em of gmms with gpu acceleration with cuda," http://andrewharp.com/gmmcuda, lastly updated on 21/05/2009.

[11] J. L. Hennessy and D. A. Patterson, *Computer Architecture A Quantitative Approach, 5th Edition.* Morgan Kaufmann, 2011.

[12] C. Plant and C. Böhm, "Inconco: interpretable clustering of numerical and categorical objects," in *KDD*, 2011, pp. 1127–1135.

[13] A. Frank and A. Asuncion, "UCI machine learning repository," 2010. [Online]. Available: http://archive.ics.uci.edu/ml

[14] R. Neal and G. E. Hinton, "A view of the em algorithm that justifies incremental, sparse, and other variants," in *Learning in Graphical Models.* Kluwer Academic Publishers, 1998, pp. 355–368.