Efficient Probabilistic Latent Semantic Indexing using Graphics Processing Unit

Eli Koffi KOUASSI † Toshiyuki AMAGASA ‡ Hiroyuki KITAGAWA ‡

Graduate School of Systems and Information Engineering, University of Tsukuba

1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, JAPAN

E-mail: † keli@kde.cs.tsukuba.ac.jp, ‡ {amagasa, kitagawa}@cs.tsukuba.ac.jp

Abstract

In this paper, we attempt to accelerate the Probabilistic Latent Indexing (PLSI) exploiting the high parallelism of Graphic Processing Unit (GPU). Our proposal is composed of three methods. The first method is to accelerate the Expectation-Maximization (EM) computation by applying GPGPU matrix-vector multiplication. The second method uses the same principles as the first method but deals with the sparseness of co-occurrence of words and documents. The third method is to use the concurrent kernel execution, which is available on NVIDIA Fermi architecture, in order to speed up the second method. We compare the results to the most recent parallel execution of PLSI which combines a method of parallelization by OpenMP with the Message Passing Interface (MPI) for distributed memory parallelization. The experiments show that our method could be more than 100 times faster than the previous results. By dealing with the sparseness of the data, we could not only process more documents and words using GPU, but we could also keep more data on the device memory so that we can avoid massive data copy transfer between the host and the device susceptible to reduce the execution performance.

Keyword Graphics Processing Unit (GPU), Algorithms, Data Mining, Probabilistic Latent Semantic Indexing (PLSI), Expectation Maximization (EM)

1. Introduction

The importance of text analysis has been increasing due to the growing needs of information retrieval and text mining from large text databases. In such processes, dimension reduction is often used to project a document from a high-dimensional vector to a lower one in order to speed up the process and/or improve the accuracy.

LSI (Latent Semantic Indexing) is a commonly used technique of dimension reduction. It is based on the mathematical principle of SVD (Singular Value Decomposition), and has successfully been used in many applications. However, it is known that LSI has some drawbacks: 1) the resulting (reduced) dimensions are often difficult to interpret; 2) the input is a bag-of-words, which do not capable of incorporating structural information from the text; 3) ambiguous terms create noise in the vector space; and 4) SVD is computationally expensive.

To cope with these problems, Hofmann [1] proposed PLSI (Probabilistic Latent Semantic Indexing). PLSI is a statistically funded method which is based on the co-occurrence of terms and documents with a latent class. Consequently, it has a more robust statistical foundation, and is able to provide a proper generative data model. In addition, it can deal with domain specific synonymy and polysemous words, while LSI cannot. PLSI has proven to be effective and has been used in many applications. However since PLSI is based on Expectation-Maximization (EM) algorithm [2], it still suffers from long execution time when dealing with large datasets.

Many approaches have been trying to solve this drawback. Among them, the most recent approach is proposed by Wan et al. [3]. They executed the PLSI in a multi-CPU and distributed shared memory by the means of Message Passing Interface (MPI) across network environment while reducing the memory required by redesigning the EM principle applied to PLSI.

Meanwhile, for the last several years, GPGPU (General-purpose computation on graphics processing units) has been gaining much public attentions as a new computational platform. The idea is to exploit GPUs for not only image processing, but also general purpose computation. GPGPU now covers diverse range of applications, such as physical simulation, audio processing, video processing, cryptography processing, and so on [4][5][6][7]. However, text processing on GPGPU has not yet been studied very well, because

available memory on GPGPU is generally limited, while text processing requires much memory spaces.

In this paper, we propose a scheme for processing PLSI using GPGPU. As mentioned earlier, the technical challenge is twofold: how to deal with massive text data using limited memory on GPGPU and how to speedup PLSI using the functionality of GPGPU. Our contribution can be summarized as following: 1) reinterpretation of EM the algorithm using matrix vector multiplication to make the best use of GPGPU instructions; 2) reorganization of steps in the EM algorithm to save the space and to reduce the complexity; and 3) exploiting a dedicated data structure for sparse matrixes called occurrence matrix to cope with data sparseness. The above approach helps to process thousands of documents with thousands of unique terms in seconds. The feasibility of the proposed scheme is demonstrated by experiments.

The remainder of the paper is organized as follows: In Section 2 we present the preliminaries. In Section 4 we expose our proposal after presenting some related work in section 3. In Section 5 we show the results of experiments conducted. Section 6 concludes this paper and mentions some future works.

2. Preliminaries

2.1. GPU Programming with CUDA

In order to help programmers to concentrate on their algorithm instead of wasting time on learning graphics principles, NVIDIA has released an interface called Compute Unified Device Architecture (CUDA). CUDA is a C/C++ like language which exposes the hardware internal memory architecture to the programmer. Programming a GPU with CUDA makes it able to run very high number of threads in parallel.

The CUDA program runs in a unit called kernel. A kernel is composed of a single grid. A grid is composed of multiple threads blocks. A block is composed of multiple threads which can cooperate among them through the shared memory; a thread is the smallest execution unit.

So fare programming with CUDA requires understanding the memory architecture. CUDA memory architecture is composed of six different levels which are: per-thread registers and local memory, per-block shared memory, per-grid global, constant and texture memory. While constant and texture are read-only, the other are read-write memories.

2.2. Probabilistic Latent Semantic Indexing

Probabilistic Latent Semantic Indexing (PLSI) is a technique to reduce dimensions of a set of documents. The core of PLSI is the aspect model, which is a latent or "hidden" model associating the hidden class $z \in \{z_1, z_2, ..., z_K\}$ with the co-occurrence of document $d \in \{d_1, d_2, ..., d_D\}$ and word $w \in \{w_1, w_2, ..., w_W\}$. Suppose that the terms are chosen independently of the documents for each hidden class, the probability form of this model can be expressed as:

$$P(d,w) = \sum_{z \in Z} P(z)P(d \mid z)P(w \mid z) \qquad (1)$$

One of the common ways to solve (1) is maximizing the log likelihood function as expressed in (2):

$$L = \sum_{d \in D} \sum_{w \in W} n(d,w) \log(P,w) \qquad (2)$$

where *n(d, w)* is the number of occurrence of the word *w* in the document *d*. A standard procedure to solve the likelihood is the Expectation-Maximization (EM) algorithm [2]. The EM is an iterative method composed of two steps. While the values of the log likelihood are estimated in the Expectation (E) step based on the values of the parameters in (1), these parameters are updated in the maximization (M) step. The EM stops when a predefined condition is reached. In our work we defined the number of iterations. In the case of PLSI, the E step defines the probability a word in a particular document is explained by the hidden class z, as and the M step updates the probabilities.

(E): $$P(z \mid d,w) = \frac{P(z)P(d \mid z)P(w \mid z)}{\sum_{z'} P(z')P(d \mid z')P(w \mid z')} \qquad (3)$$

(M): $$P(w \mid z) = \frac{\sum_d n(d,w)P(z \mid d,w)}{\sum_d \sum_{w'} n(d,w')P(z \mid d,w')} \qquad (4)$$

$$P(d \mid z) = \frac{\sum_d n(d,w)P(z \mid d,w)}{\sum_{d'} \sum_w n(d',w)P(z \mid d',w)} \qquad (5)$$

$$P(z) = \frac{\sum_{d,w} n(d,w) P(z \mid d,w)}{\sum_{d,w} n(d,w)} \qquad (6)$$

3. Related Work

Many researchers are now interested in GPU because of its attractive performance benefit. One of the most famous categories is physical simulation on GPU, such as fluid dynamics and water condensation [6]. In the area of signal and image processing, many works has been presented, such as segmentation [7], image thresholding [8]. GPU has been successfully applied to problems in databases and data mining. Bakkum et al. proposed to accelerate SQL queries using CUDA [9]. Zhan et al. accelerated text mining using CUDA [10]. Govindaraju et al. proposed a good frame work called GPUTeraSort, which is to provide a fast sorting mechanism for large databases using CUDA [11].

However, we found that few works exploit GPU for text analysis. This is due to the fact that processing text data using GPU gives rise to some technical challenges. One major challenge is that text data are in many cases voluminous, while the available memory on GPU is quite limited. One of related works in this category is presented by Wu et al. They proposed to cluster large data point using CUDA [12]. Many other fields are available in the survey on GPU published by Owens et al. [4].

3.1. Parallel Latent Semantic Indexing using GPU

Cavanagh et al. proposed to accelerate LSI using GPU [13]. Their idea is to speed up the most time-consuming part of LSI, which is the singular value decomposition (SVD), by parallelization using GPU. Specifically, they use the Lanczos algorithm because of its efficiency. Also, they use CUBLAS, which is a GPU version of the famous numerical computation library called Basic Linear Algebra (BLAS) [14]. As a result, they can process big matrices such as 4000-by-4000 in seconds. Our work is different from this work, because our target is PLSI. In addition, their algorithm assumes that the number of latent classes is divisible by 16, which is not flexible when we apply it to real-life applications. Our approach, on the other hand, can deal with any number of latent classes, though the underlying indexing model is different.

3.2. Parallel and Distributed Latent Semantic Indexing

Regarding research works on parallelizing PLSI, to the best of our knowledge, there have been only few works. In [3] Wan et al. attempt to cope with the drawbacks of PLSI, that is, it considerably consumes computing resources in terms of both execution time and internal memory. Specifically, their approach is based on a former work by Hong et al. [15], which proposes a shared-memory parallelization using OpenMP. Wang et al., in addition, combine distributed memory parallelization using MPI. They also consider the sparseness of the co-occurrence of terms and documents and retouch the EM algorithm in order to save memory for computation. Our work is different from those approaches, because we exploit GPU for speeding up PLSI.

4. Parallelization of PLSI

4.1. Expectation-Maximization as Matrix-vector Multiplication

Our approach reinterprets the PLSI equations as matrix vector multiplication. Many optimized libraries such as BLAS exist already so we could take advantage of such libraries to get the work done in less time. Our approach is summarized in the Figure 1. In the process of EM implementation, we can distinguish two properties from the expression. One part which stays constant for each iteration and shared by all latent class and the other part which depends on each latent class. This analysis helped us to separate the computations and combine some steps of the EM in order to save time and memory space. We also think the equations at a matrix and vector level so that computation can be done once a time, making the EM more efficient. Thus, we adopt the matrix-vector multiplication including transpose of vector and matrix. In Equation (3), the most right parameters can be written as shown in (7).

$$\Gamma(d,w) = [P(z)P(d \mid z)]^T P(w \mid z) \qquad (7)$$

where A^T is the transpose matrix A and P(z) is a K-by-K diagonal matrix. The denominator $\Gamma(d,w)$, is persistent through all latent classes for each iteration and is a D-by-W matrix. For any value $i \in \{1,2,...,K\}$ with K the maximum number of classes, the numerator specific to each class, can be computed as shown in equation (8), where $P(z_i)$ is the probability of the latent class i, $P(d|z_i)$ and $P(w|z_i)$ are vectors of size D and W representing the probability of d and w knowing z respectively. These vectors can easily be extracted from the Z-by-D matrix of P(d|z) and Z-byd-W matrix of P(w|z).

$$\Psi_{z_i}(d,w) = P(z_i)[P(d \mid z_i)]^t P(w \mid z_i) \qquad (8)$$

Finally, E step can be computed from Γ and Ψ. As we can see from (4)~(6) and shown in Figure 2, we implicitly hide computation complexity by multiplying each pair (d,w) with its occurrence coefficient n(d,w) when computing the final step of the expectation because the probability calculated in (3) is never used alone but always multiplied with the occurrence n(d,w). Equation

(9) gives the E step for a given class i.

$$Ez_i(d,w) = \frac{P(z_i)[P(d \mid z_i)]^T P(w \mid z_i)}{[P(z)P(d \mid z)]^T P(w \mid z)} n(d,w) \qquad (9)$$

```
1: ComputeTotalOccurrence
2:      For each iteration
3:          ComputeExpectationDenominator
4:          For each latent class
5:              FinalizeExpectationForCurrentClass
6:              ComputeWordScoreForCurrentClass
7:              ComputeDocScoreForCurrentClass
8:              UpdateProbabilitiesForCurrentClass
9:      End latent class Loop
10: End iteration Loop
```

Fig. 1: Pseudo-code of proposal algorithm of EM computation

$$\begin{array}{cccccc}
\frac{\Psi_{11}(d,w)}{\Gamma_{11}(d,w)} n(d_1,w_1) & \frac{\Psi_{12}(d,w)}{\Gamma_{12}(d,w)} n(d_1,w_2) & \dots & \dots & \dots & \frac{\Psi_{1W}(d,w)}{\Gamma_{1W}(d,w)} n(d_1,w_W) \\
\frac{\Psi_{21}(d,w)}{\Gamma_{21}(d,w)} n(d_2,w_1) & \frac{\Psi_{22}(d,w)}{\Gamma_{22}(d,w)} n(d_2,w_2) & \dots & \dots & \dots & \frac{\Psi_{2W}(d,w)}{\Gamma_{2W}(d,w)} n(d_2,w_W) \\
\dots & \dots & \dots & \dots & \dots & \dots \\
\frac{\Psi_{(D-1)1}(d,w)}{\Gamma_{(D-1)1}(d,w)} n(d_{D-1},w_1) & \frac{\Psi_{(D-1)2}(d,w)}{\Gamma_{(D-1)2}(d,w)} n(d_{D-1},w_2) & \dots & \dots & \dots & \dots \\
\frac{\Psi_{D1}(d,w)}{\Gamma_{D1}(d,w)} n(d_D,w_1) & \frac{\Psi_{D2}(d,w)}{\Gamma_{D2}(d,w)} n(d_D,w_2) & \dots & \dots & \dots & \frac{\Psi_{DW}(d,w)}{\Gamma_{DW}(d,w)} n(d_D,w_W)
\end{array}$$

Fig. 2 Final Step of Expectation multiplied by the occurrence

In our implementation, we adopt the array reduction method in the M step. Belloch [16][17] has proposed a method optimized for CUDA and publicly available as library named CUDPP. We used the library to compute this part of our proposal. Then we update the values of P(z), P(d|z) and P(w|z).The process of array reduction consists of three steps. The first scan, which consists of computing the whole matrix reduction in order, be able to normalize the probabilities. This method is more efficient than the atomic add which could produce the same result during the final step of expectation computation. In effect the atomic add operation with CUDA makes all threads waiting. Simple experiment showed our method is better than atomic add for large matrix. The second step consist of operating the "row multi scan" feature available on CUDPP to compute the probabilities of documents and the last step is the "row multi scan " of the transpose of Expectation in order to compute the probabilities of words. More details about CUDPP are available on the CUDPP website [18]. After these steps the update process for maximization starts. The update process is particularly easy and fast on GPU because no loop is required.

4.2. Exploiting the Sparseness of the Occurrence Matrix

4.2.1. Exploiting the Data Sparseness

The co-occurrence of terms and document from which we build the occurrence matrix is a very sparse matrix. Table 1 shows the sparseness level of some well-known training sets we used for our experiments.

Table 1 Sparseness of some real world data sets

Dataset	Docs	Words	Occurrence	rate
CRAN	1398	7867	63813	0.58%
MED	1033	10062	28013	0.32%
CISI	1460	9765	34947	0.29%

Because the total memory size of GPU is strictly limited, it is crucial to exploit sparse matrix representation techniques to deal with large document dataset. There are many kinds of such approaches [19] for GPU. Here we have two main goals when dealing with the sparse matrixes; 1) efficiently use of the memory so that we can process more documents, 2) make the number of working threads as many as possible. Our approach in order to reach these goals combines the Compress Sparse Row (CSR) and the Coordinate (COO) format to represent sparse matrix on the device memory. These representations are shown in Figure 3. The coordinate (COO) format is a quite simple way of storing the sparse matrix. The matrix is represented with three different vectors name row, col, and data where a triplet (row, col, data) represents the row index, the column index and corresponding non-zero value in the matrix. The CSR on the other hand is just the optimization of the COO representation. In effect, the column indices and the data vectors are exactly the same. The only difference is in the representation of the row's information. Here, the notion of pointer appears. If we deal with an M-by-N matrix, the size of the pointer is *M+1*; pointer[i] stores the stores the offset of the non-zero element in the i^{th} row. The last element *pointer[M+1]* stores the total of non-zeros elements in the sparse matrix.

Using this approach, the generated number of threads is exactly the number of non-zero elements of the occurrence matrix, also by using both representations we reach an interesting point of our research: the processing depends on neither documents number nor terms number but only on the occurrences of terms in the co-occurrence matrix. During the computation each threads will be assign the task to compute only for one occurrence. Thus

all the threads have approximately the same amount work and we can keep all the threads busy during computations.

So far the maximum number of threads block that could be run at once on any NVIDIA GPU is 65565 blocks. In this research we use a Fermi architecture card which allows 1024 threads block. By assigning one thread to one occurrence we can easily determine the maximum size of the dataset our system will be able to process a dataset of about 67million occurrences. The number of element to be process is quite large and outpaces the reasonable training set we could get, allowing us to say our method can deal with very large datasets.

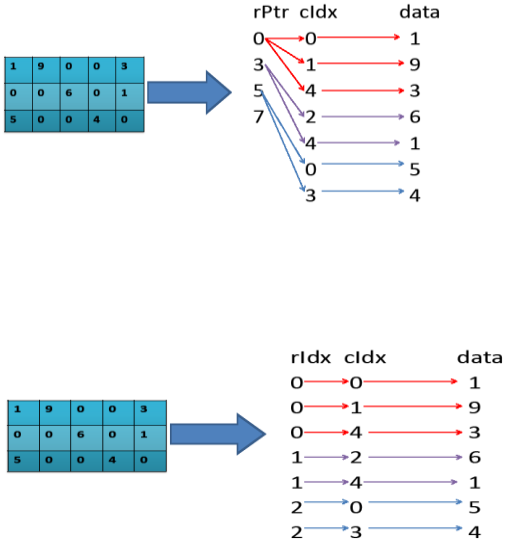

Fig. 3: Sparse matrix representation (a) Coordinate format (b) Compressed sparse row format

In the following paragraphs we show through a sample occurrence matrix, how exactly the computations of the PLSI on the device are done.

Let consider a small matrix of 3-by-5 dimension with 7 non-zero elements as shown in Figure 3. Following the CSR and COO representation, we generate 7 threads, the number of non-zero elements of the matrix. We compute the numerator of the expectation steps in Figure 4. Each thread is assigned the task to calculate the numerator of each element.

1- Load $P(z_k)$ from global memory
2- Load $P(d_i|z_k)$ from global memory
3- Load $P(w_j|z_k)$ from global memory
4- Calculate the numerator
5- Update the numerator in global memory

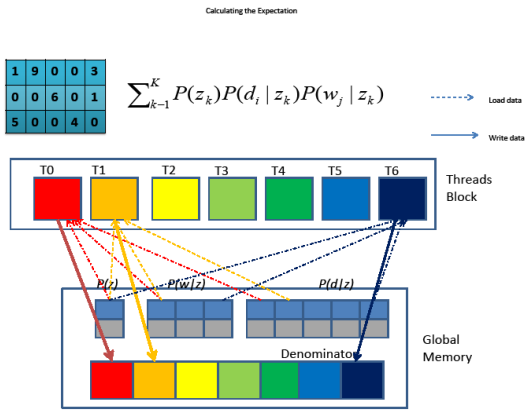

Fig. 4: Task assignment to threads

Almost the same process leads to the computations the denominator of the Expectation step. However, we need to iterate through all the latent classes as following:

1- Load $P(z_k)$ from global memory
2- Load $P(w_j|z_k)$ from global memory
3- Load $P(d_i|z_k)$ from global memory
4- Repeat for all classes
5- Calculate the denominator in the local memory
6- Update the numerator in global memory

All other steps of the expectation and maximization are computed though almost the same process that we don't describe in detail in this paper for space reason.

4.3. Optimization on Fermi Architecture

As explained Figure 1, the EM algorithm exposes the class-dependent and class-independent part of or the PLSI. We observed that computation of classes' expectations and updates do not interfere, thus is data independent. As the recent release of NVIDIA Fermi architecture offers the possibility to execute multiple kernels concurrently, we take advantage of this feature to compute the PLSI for each class per iteration. More details about CUDA concurrent kernel execution are available in The CUDA Programming Guide [20]. In order to avoid global memory being overwritten, we design the system so that the each kernel has its own write space. This is not memory efficient and limits its application to data set of modest size. In a near future we plan to fix this limitation.

5. Experimental Evaluation

We conducted numbers experiments in order to prove the effectiveness of our method. We first compare our results with the most recent work and then extend in order to show how efficient our algorithm is.

5.1. Experimental Design

We conducted numbers experiments in order to prove the effectiveness of our method. We first compare our results with the most recent work and then extend in order to show how efficient our algorithm is.

5.2. Dataset

We used both synthetic and real world data. We used the CRAN, MED, and CSI training set publicly available . We applied standard stemming and stopwords to the training set in order to reduce noise. The data set specifications are summarized in Table 1. We also generated synthetic dat. We predefined the number of words documents and the percentage of nonzero elements the data can hold. We also specified the number of class into which we could classify the document in order to measure the impact of classes to the processing time in both GPU and CPU environment

5.3. Implementation Environment

The hardware specification for the CPU version of our implementation is Intel Quad-core Xeon E5620 processors with 6.4GT/s (Intel QuickPath Interconnect) and up to 8MB shared cache. We used C++(gcc 4.4.1) to implement the algorithms. We executed on CentOS 5.0. The GPU card used is the Tesla C2050. It is Fermi architecture with 14 Multiprocessors and 32 cores per multiprocessors and 3GB memory, conned to the CPU memory with a PCIe up to 144Gb/s.

5.4. Evaluation Results

5.4.1. Real Data

We then executed both our proposal and ours in the same environment making the number of latent class varying from 8 to 32. The execution times are summarized in Tables 2, all times are in milliseconds. For space reason we show only one result in Table and two speed-up comparison result in Figure 3. However due to memory restriction, we could not execute our first proposal on real data. Also the optimization should at least have the same performance as method 2. The internal check specific to the optimization method creates overheads making it slightly slower than the sparse method. We show in the next section how to get better performance of our optimization.

Table 2: Med data set experimental results

Latent Class	GPU Sparse	GPU Concurrent Kernel	MPI Multiprocessor
8	44.21	45.69	3999.60
16	86.95	89.51	9997.30
32	167.73	172.59	20006.80
50	262.04	270.00	31003.50

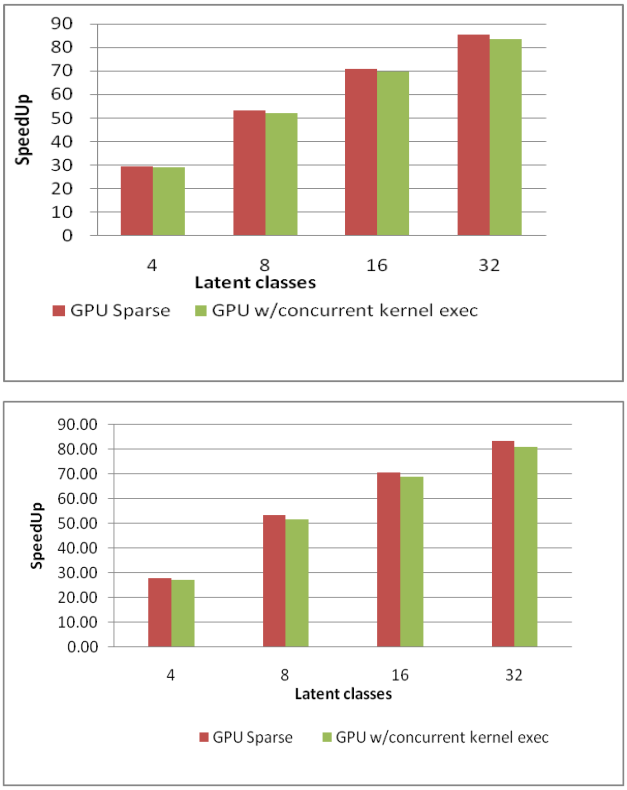

Fig. 5: PLSI Speed-up with GPU (a) med; (b) CISI

5.4.2. Synthetic Data

In this section we conducted three different experiments. First we tested the impact of the sparseness on the overall execution time by varying the non-zero elements in the occurrence matrix. Second we tested. We fixed the latent class to 32, the occurrence matrix is of size 1000-by-5000, and the concurrent kernels are 16. Figure 4(b) shows the output showing how the number non-zero elements affect execution time. Our second experiment was to show the performance of the concurrent kernel execution. NVIDIA allows at most 16 kernels to be executed concurrently. We show how parallelizing the class execution through concurrent kernel execution is advantage. Figure (b) shows the overall execution time for different kernels. Here 1 means no concurrent kernel execution thus the same as the method 2, we set the latent class to 32. The

occurrence matrix is of size 1000-by-5000 and the non-zero elements 1%.

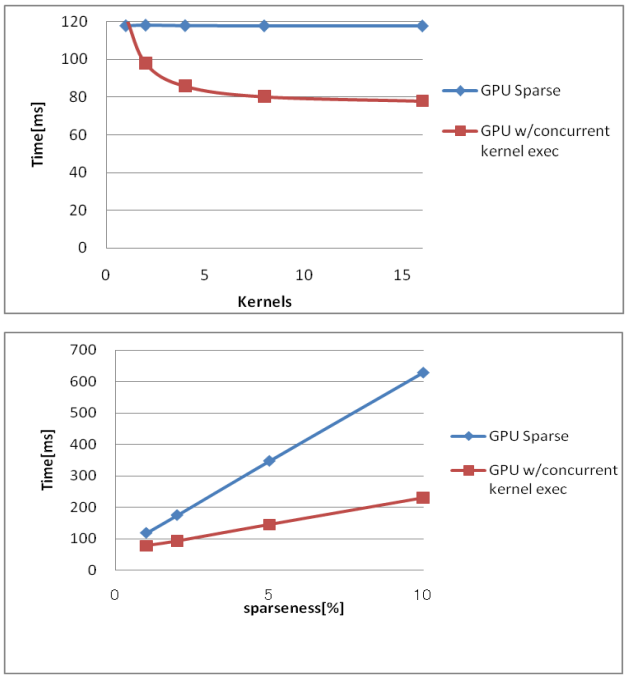

Fig. 6: 7 Synthetic data (a) concurrent kernels, (b) sparseness

6. Conclusion and Future Work

In this research we show how the parallel PLSI executed using GPU. Our implementation which uses the matrix-vector implementation of EM is more efficient both in memory and execution time. We also showed how the use of sparse algorithms can help save memory space for the storing the data on device memory. Our optimization using the concurrent kernels execution feature available on NVIDIA Fermi architecture is a method which can drastically reduce the execution time of PLSI of the improved version we proposed. However the memory inefficient of the method is a big problem we're working on now. We also plan to execute the folding-in operations to make a full and complete application of the Parallel PLSI of GPGPU.

Acknowledgement

The authors gratefully acknowledge the funding support of Grant-in-Aid for Scientific Research on Priority Areas by MEXT (#21013004).

7. References

[1] Thomas Hofmann, Probabilistic Latent Semantic Indexing, Proceeding of the Twenty-Second Annual International SIGIR Conference on Research and Development in Information Retrieval (1999).

[2] A.P. Dempster, N.M. Laird, D.B. Rubin, Maximization likelihood from incomplete data via EM algorithm, Journal of the Royal Statistical Society. Series B(Methodological), Vol. 39, No. 1 (1977),pp.1-38

[3] R. Wan, V.N. Ahn, and H. Mamitsuka, Efficient Probabilistic Latent Semantic Analysis through Parallelization, AIRS (2009), LNCS 5839, pp. 432-443,2009

[4] John D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger J. Lefohn et al.(2007), A Survey of General-Purpose Computation on Graphics Hardware, Computer Graphics Forum, 26: 80-113.

[5] Harris M.J., BaxerⅢ W., S.T., Lastra A., Simulation of cloud dynamics on graphics hardware, Graphics Hardware 2003, pp.92-101 (2003)

[6] Lefohn A.E., Kniss J. M., Hansen C.D., Whitaker R.T. A stream narrow-band algorithm: Interactive computation and visualization surfaces. IEEE Trans. Visual. Comput. Graph. !0(4): 422-433 (2004).

[7]N.K. Govindaraju, Jim Gray, Ritesh Kumar, Dinesh Manacha, GPGPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management, ACM-SIGMOD (2006)

[8] Yan R., Welch G., Fast Image segmentation and smoothing using commodity graphics hardware, J. Graph Tools. 7(4):91-100.

[9] Peter Bakkum and Kevin Skadron, Accelerating SQL Database Operations on a GPU using CUDA, 94-103. (2010)

[10] Yongpeng Zhang, Franck Mueller, Xiaohui Cui, Thomas Potok, GPU-Accelerated Text Mining, EPHAM'09 (2009).

[11] N.K. Govindaraju, Jim Gray, Ritesh Kumar, Dinesh Manacha, GPGPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management, ACM-SIGMOD (2006)

[12] Ren Wu, Bin Zhang, M. Hsu, Clustering Billions of Data points Using GPUs. UCHPC-MAW (2009)

[13] J. M. Cavanagh, T. E. Potok, Xiaohui Cui, Parallel Latent Semantic Analysis using a Graphics Processing Unit, GECC'2009, pp.2505-2509 (2009)

[14] CUBLAS, The NVidia SDK linear algebra http://www.nvidia.com/content/cudazone/cuda_sdk/Linear_Algebra.html

[15] Hong C., Chen W., Shan J., Chen Y., Zhang Y. Parallelization and characterization of probabilistic latent semantic analysis. In: Pro. 37th International Conference on Parallel Processing, pp. 628-635(2008)

[16] Guy E. Belloch, Vector Models for Data-Parallel Computing. The MIT Press (1990)
[17] Guy E. Blelloch. Prefix Sums and Their Applications,"Synthesis of Parallel Algorithms", Edited by John H. Reif, Morgan Kaufmann (1991).
[18] http://gpgpu.org/developer/cudpp
[19]N.Bell, M. Garland, Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report (2008)
[20] www.nvidia.com, NVIDIA CUDA C Programming Guide version 3.1 (2010)