International Journal of Networked and Distributed Computing, Vol. 4, No. 2 (April 2016), 96-105

Accelerated Diffusion-Based Recommendation Algorithm on Tripartite Graphs with GPU Clusters

Jingpeng Wang [1], Jie Huang [2] *, Mi Li [3]

[1] *School of Software Engineering, Tongji University,*
4800 Cao'An Highway
Shanghai, 201804, China
E-mail: jingpeng.wang@outlook.com

[2] *School of Software Engineering, Tongji University,*
4800 Cao'An Highway
Shanghai, 201804, China
E-mail: huangjie@tongji.edu.cn

[3] *School of Software Engineering, Tongji University,*
4800 Cao'An Highway
Shanghai, 201804, China
E-mail: limisky@gmail.com

Abstract

Exorbitant computation cost hinders the practical application of recommendation algorithm, especially in time-critical application scenario. Although experiments show that recommendation algorithm based on an integrated diffusion on user-item-tag tripartite graphs can significantly improve accuracy, diversification and novelty of recommendation, it is also very time-consuming. Therefore, a parallel solution is frequently needed to improve the execution speed of the algorithm. This paper explicitly presents the parallel implementation and optimizations of diffusion-based recommendation on weighted tripartite graphs algorithm using Compute Unified Device Architecture and related optimization solutions including accelerated memory access with shared memory, stream scheduling and GPU clusters optimization. Compared to the algorithm running on a single CPU core, the unoptimized GPU kernel can achieve 153.9 speedup on average with the input dataset consists of 30000 records on GTX 980. With shared memory applied, the time cost on device memory access saves about 50% on dataset of 90000 records and with 2-way streams scheduling, the kernel's performance improves about 7% ~ 13%. Based on the optimized GPU kernel, we evaluate the performance of the recommendation algorithm with customized socket communication mechanism on GPU clusters. And compared to a single GPU node, we achieve 7.55 speedup on clusters of 9 GPUs when recommending for 8000 users. Besides this, the speedup of GPU clusters is also 26.1 times of the speedup of our CPU clusters of 9 nodes and 1586.28 times of serial algorithm on one CPU core. It proves that GPU technology can dramatically improve the algorithm's performance.

Keywords: Diffusion-Based Recommendation Algorithms on Tripartite Graphs; CUDA; Stream Scheduling; Shared Memory; GPU Clusters.

---

[*] Corresponding author.

Published by Atlantis Press

1. Introduction

Recently, the exponential growth of the Internet and World Wide Web confronts us with the problem of information overload. Evaluating all these alternatives by ourselves is not feasible at all. Recommender systems[1], appearing as a natural solution to overcome this problem, could help users discover relevant information in large datasets[2]. With a personalized recommender system, websites are also able to better predict and rank content of interest to users by using historical user interactions[3]. Recommender system is now considered to be the most promising way to efficiently filter out the irrelevant data. So far, it has been successfully used in e-commerce[4] and many other fields.

The core of recommender systems lies in recommendation algorithms, whose input is usually a user-item rating matrix and output is a recommendation list. Whereas the number of items is often much larger than that of users, which leaves the matrix to be sparse, it would seriously diminish the quality of recommendation.

Web 2.0 has been ballooning in the last few years, thus user generated content is playing a more and more important role in recommender system. Tags, a non-hierarchical keyword or term assigned to a place of information[5], has drawn the attention of researchers.

The diffusion-based recommendation algorithm takes advantage of the relation among user, item and tag. It starts by constructing a tripartite user-item-tag graph using input data, then through a two-step diffusion process on both user-item and item-tag tripartite, recommendation list is generated. Experimental results on the Del.icio.us[6], MovieLens[7] and BibSonomy[8] datasets demonstrate that exploiting user-item-tag relation, this algorithm can significantly improve accuracy, diversification and novelty of recommendation.[9]

However, the computational overhead of this method is quite high. Considering that there exist lots of arithmetical operations in the algorithm, so it is very suitable for GPU acceleration.

During the past few years, Graphical Processing Unit (GPU) has evolved into a flexible platform with highly parallelization for general computing. General Purpose GPU (GPGPU)[10] researchers have achieved over an order of magnitude speedup over modern CPUs on some non-graphics problems. Initially, GPUs were programmed by low-level languages which restrict its application as computing workhorse. But with the release of the Compute Unified Device Architecture (CUDA)[11] of NVIDIA and the advent of GPUs consisting of multi-core processor with tremendous computational horsepower, programmers can use its parallel computing capabilities in real applications.

The main contribution of this paper is using CUDA and optimization strategies to accelerate the diffusion-based recommendation algorithm on tripartite graph. The architecture of this paper is as follows: Section two mainly introduces some related works and Section three concentrates on the introduction and complexity analysis of the algorithm. Section four mainly introduces the parallelization and optimization for the algorithm with CUDA. In Section five we conducted several experiments, collected related results and analyzed the corresponding results. The last section is conclusion and future work.

2. Related Works

GPU acceleration, including OpenCL and CUDA technology, contributes a lot in massive calculations and elaborate algorithms, especially in scientific computing area. Bai, T. et al.[12] use three strategies to accelerate Nth-degree truncated polynomial ring with GPUs and achieve high improvement on performance. Hung, C. et al.[13] develop two efficient GPGPU-based parallel packet classification approaches to filter packets by leveraging thousands of threads.

But there are quite limited related works on using GPUs to accelerate recommendation algorithms. Kato and Hosino[14] proposed a CUDA implementation of user-user k-nearest neighbor search for user-based Collaborative Filtering. R. Li et al.[15] proposed a social network-aware Top-N recommender system using GPUs. This approach also uses user-based Collaborative Filtering and requires social network information to provide recommendations. Nadungodage, C.H.[16] proposes two parallel, item-based recommendation algorithms implemented using the CUDA platform and utilize two compression techniques to reduce the required number of passes and increase the speedup accounting on the high sparsity of the user-item data. The experimental results on synthetic and real-world datasets show that CUDA based algorithms outperform the respective CPU implementations.

The proposed accelerated algorithm in this paper use parallel and optimize techniques to process large-scale datasets in GPU, thereby increasing the possible speedup and the methods applied are based on tripartite graphs have been proven to be more accurate and fast.

3. Diffusion-based Recommendation Algorithm on Tripartite Graph

We start by mapping data of users' behavior into graph models. In general, a social tagging system consists of three sets: users $U = \{U_1, U_2, ..., U_M\}$, items $I = \{I_1, I_2, ..., I_N\}$ and tags $T = \{T_1, T_2, ..., T_K\}$. The tripartite graph representation can be described by two adjacent matrices, A and A' representing user-item and item-tag relations respectively. If U_m has collected I_n, we set $a_{mn} = 1$, otherwise $a_{mn} = 0$. Analogously, we set $a_{nk}{}' = 1$ if I_n has been assigned by tag T_k, and $a_{nk}{}' = 0$ otherwise. Figure 1 shows an illustration that consists of three users, five items and four tags.

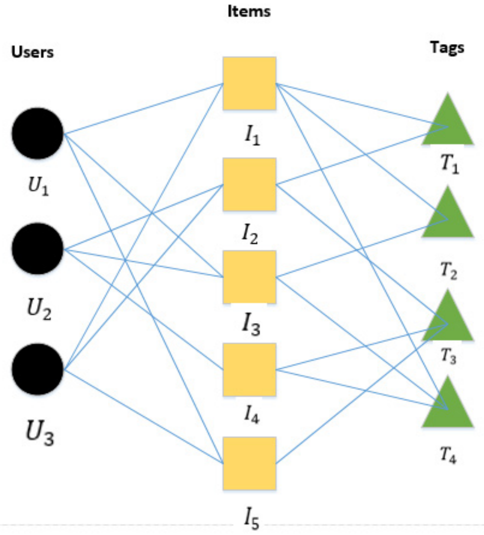


Fig. 1. A tripartite graph consists of three users, five items and four tags.

3.1. Diffusion on left bipartite graphs

In the diffusion process, we suppose that some kind of resource is initially located on items. Denote $\vec{f}$ as the initial resource vector on items, for example, f_n is the amount of resource located on I_n, given a user U_m, we set the initial resource vector $\vec{f}(U_m)$ as:

$$f_n(U_m) = a_{mn}, n = 1, 2, ..., N; a_{mn} = 0 \, or \, 1. \quad (1)$$

and as we can see in the Eq. (1), the initial resource is a given unit, which can be understood as a capacity for user U_m on item I_n. In addition, the initial resource vector varies from different users, thus capturing the personalized preferences.

Next, each item will averagely distribute its resource to all neighboring users, and then each user will redistribute the received resource to all his/her collected items. Denote $\vec{f}'$ as the resource vector after two-step diffusion, then $\vec{f}'$ can be calculated as follows:

$$f_n' = \sum_{m=1}^{M} \frac{a_{mn}}{C(U_m)} \sum_{k=1}^{N} \frac{f_k}{C(I_k)}, n = 1, 2, ..., N. \quad (2)$$

where $C(U_m)$ is the number of collected items for user U_m, $C(I_k)$ is the number of neighboring users for item I_k. This algorithm is originally motivated by the resource-allocation process on graphs[17], and has been shown to be slightly more accurate than the collaborative filtering based on MovieLens dataset.[18]

3.2. Diffusion on right bipartite graphs

As mentioned above, different collections of items reflect personalized preferences, and even for exactly the same items, different users may assign different tags to them. Therefore, the significance of collaborative tags in a recommender system is twofold.[9] Firstly, tags enrich the items' information and two items sharing many common tags are probably closely related in content. Secondly, the personalized preferences are naturally embedded in the different usages of tags.

On the item-tag bipartite graph, analogously, supposing that a kind of resource is initially located on items, each will equally distribute its resource to its neighboring tags, and then each tag will redistribute the resources it gets to the items that have associations with it. Given the initial resource vector $\vec{f}$, the final resource vector $\vec{f}''$, after two-step diffusion is:

$$f_n'' = \sum_{k=1}^{K} \frac{a'_{nk}}{C(T_k)} \sum_{n=1}^{N} \frac{f_n}{C'(I_n)}, n = 1, 2, ..., N. \quad (3)$$

where $C(T_k)$ is the number of neighboring items for tag T_k, $C'(I_n)$ is the number of neighboring tags for item I_n.

Finally, we combine the diffusions on user-item and item-tag bipartite graph in a linear way:

$$f^* = \lambda \vec{f}' + (1-\lambda)\vec{f}''. \tag{4}$$

where $\lambda \in [0,1]$ is a tunable parameter, $\vec{f}'$ is the vector obtained from Eq. (2), $\vec{f}''$ is the vector derived from Eq. (3), and both of the two vectors are attained for the same target user. In the extreme cases $\lambda = 0$ and $\lambda = 1$, the algorithm becomes pure diffusion on item-tag and user-item bipartite graphs, respectively.

3.3. Pseudo code and complexity analysis

Algorithm: Diffusion-Based Recommendation Algorithm on Tripartite Graphs

```
Input: Target user u; target item i;
       three matrices user_item, item_tag and user_tag,
       representing the relations between user and item,
       item and tag, user and tag respectively;
       four hashtables adUserItem, adItemUser,
       adTagItem and adTagUser, in adUserItem, its
       key is user and value stores number of item the
       user rated, the other hashtables are similar;
       M is the number of users in the dataset, N, K are
       the numbers of items and tags; $\lambda$ is a parameter.
Output: Predicted score finalScore
finalScore = 0;
for m = 1 to M do
    for n = 1 to N do
        a = user_item[$user_m$][i] / adUserItem[$user_m$]
        b = user_item[u][$item_n$] / adItemUser[$item_n$]
        leftScore += a * b
    end
end
for k = 1 to K do
    for n = 1 to N do
        a = item_tag[i][$tag_k$] / adTagItem[$tag_k$]
        b = user_item[u][$item_n$] / adItemUser[$item_n$]
        rightScore += a * b
    end
end
finalScore = leftScore * $\lambda$ + rightScore * (1 - $\lambda$)
```

The pseudo code of the algorithm is as above and obviously from the pseudo code we can see that the computational cost of the diffusion steps on user-item (left score) and item-tag (right score) bipartite graphs could be very heavy, especially when the dimension of user-item and item-tag matrices is extremely high. The complexity of this algorithm is analyzed as follows.

First, we need to calculate the corresponding scores of each item for users when recommending items to one user through the algorithm. The scores are combined by the left part and the right part, on the left part, according to Eq. (2), for every item, it needs to distribute its initial resources to users that have collected them, so the computing complexity is $M \times N$ (M and N are the number of users and items, respectively). In the second diffusion process, every user needs to redistribute its received resources to neighboring items, as analyzed before, the computing complexity is still $M \times N$. Since the two processes are sequential, the final computing complexity for the left part is $M \times N$. While for the bipartite graph on the right part, according to Eq. (3), the diffusion processes are similar to the left part, so the computing complexity for the right part is $K \times N$ (K and N are the number of tags and items, respectively).

Finally, the linear combination nearly doesn't cost any time, therefore, to make recommendations for all users, the time complexity of this algorithm is about $M \times N \times K$ and the computational overhead is heavy.

4. Parallel Implementation and Optimization with CUDA

With complexity and bottleneck analysis with Intel VTunes Amplifier XE, we can find the diffusion on left and right bipartite graphs is time-consuming procedures. Since the diffusion approach is independent in each loop of the left and right diffusions, therefore, it has intrinsic parallelism. The program is also mainly involved with arithmetic operation, so the program is a very suitable case for GPU acceleration.

Then, we conduct our experiments through CUDA thread/block model (allocating threads and blocks in GPU with CUDA APIs). Here we use JCuda[19] wrapper to encapsulate our CUDA kernels.

4.1. GPU kernel implementation

Here we implement the GPU kernel with thread/block model provided by CUDA. Before building thread/block model, we firstly copy corresponding data from the host onto GPU device with CUDA memory allocation and copy API. These data include the row to target user *u* and the column to target item *i* in *user_item* matrix, *adUserItem* array and *adItemUser* array in left bipartite graph diffusion, the row to target item *i* in *item_tag* matrix and row to target user *u* in *user_item* matrix, *adTagItem* array and *adItemUser* array in right bipartite graph diffusion. Besides, we also allocate the output array on GPU devices.

After that, we mapped these data to CUDA threads with CUDA thread/block model. To fully utilize the GPU device, we allocate enough threads in GPU blocks as much as possible to ensure that the theoretical occupancy can almost reach 100 percent. According to the result in experiments, we find the thread/block model in Figure 2 and Figure 3 is reasonable. These data arrays: the row to target user *u* in *user_item* matrix, *adUserItem* and output array use way of block mapping in Figure 2. These data arrays: the column to target item *i* in *user_item* matrix, the row to target item *i* in *item_tag* matrix, *adUserItem* and *adTagItem* use way of thread mapping in Figure 3.

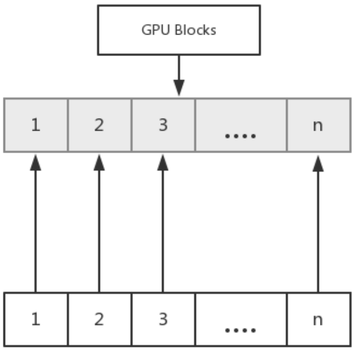


Fig. 2. GPU blocks mapping in a grid, variable n represents the size of the array variable, which is need to copy onto the GPU device.

4.2. Shared memory optimization

According to NVIDIA's documentation, on-chip shared memory is much faster than local and global memory. Its latency is roughly 100x lower than uncached global memory latency. Shared memory is allocated per thread block, so all threads in the block have access to the same shared memory. Threads can access data in shared memory loaded from global memory by other threads within the same thread block. Hence we optimized our program with shared memory to accelerate the access of GPU memory in a GPU kernel.

In GPU kernel experiments, we find out that the arithmetical operation on output array involves lots of global memory access. Thus we use shared memory to collect arithmetical results and reassign to the variable array at the end of kernel execution. By this optimization with intermediate variable, we expect to reduce the time delay on memory access and achieve corresponding improvements.

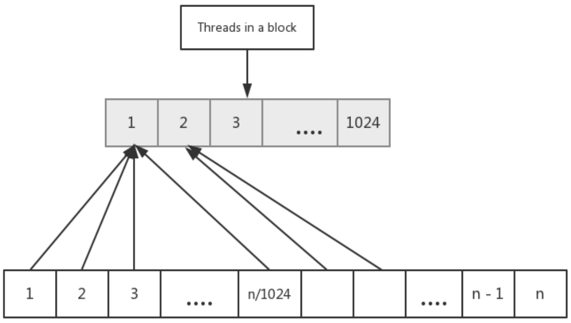


Fig. 3. GPU threads mapping in a block, variable n represents the size of the array variable, which is need to copy onto the GPU device. (Block dimension is 1024).

4.3. Stream scheduling

CUDA provides a programming model called CUDA stream with the ability to schedule multiple CUDA kernels simultaneously. One CUDA stream can encapsulate multiple kernels, and they have to be scheduled strictly following a particular order. However, kernels from multiple streams can be scheduled to run concurrently.

The main purpose of using CUDA streams is to hide the memory latency: when kernel A is loading/writing data, kernel B can occupy the cores for computation. As a result, the cores in the Multiprocessor reach better utilization.

In our previous program with CUDA shared memory optimized, we can find the time cost on the data copy operation from GPU device to host can be optimized with streams to improve the performance of serial kernels when the dataset contains about 8000 - 20000 records.

This indicates that with CUDA stream scheduling applied, we can overlap a certain amount of time cost on kernel execution like a pipeline, meanwhile we can hide the memory latency and implements kernel concurrency. The snapshot of NVIDIA Visual Profiler on multiple serial kernels execution with default stream can be seen in Figure 4, the size of dataset in the experiment on serial kernels is about 7500 records.

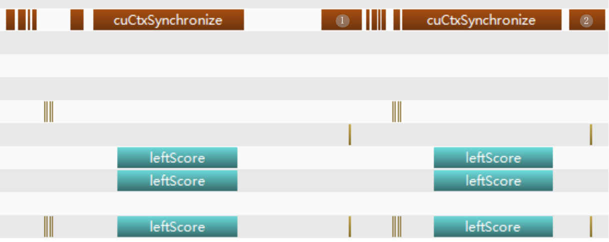


Fig. 4. The snapshot of GPU serial kernels on left bipartite diffusion from NVIDIA Visual Profiler (dataset size is 7500 records). Marker 1 and 2 represent memory copy operation from device to host.

From the snapshot, we can see memory latency exists during the process of data transmission from device to host. But because of that I/O time cost from device to host is much shorter than kernel execution, the improvement of CUDA stream scheduling will not be very much

We allocate and register a double buffer on host as pinned memory to store the GPU results temporarily through Java NIO APIs, with this channel, GPU device can communicate with host by Direct Memory Access (DMA), when one stream is executing kernels, the other one can write data to the buffer. In this way, we implement a two-way stream to accelerate the GPU kernels (left bipartite and right bipartite are similar). Detail pseudo code can be seen as follows:

Pseudo Code: CUDA Stream Scheduling Optimization for Left Bipartite

```
Input: Target user $u$; target item $i$;
       matrix user_item, representing the relations between user and item;
       two hashtables adUserItem, adItemUser, in adUserItem, its key is user and value stores number of item the user rated, the other hashtable are similar;
       M, N are the numbers of users and items.
Output: Predicted score finalScore
Begin:
finalScore = 0; STREAM_NUMBER = 2;
prepare compiled PTX file
initialize CUdevice, CUcontext, CUmodule, CUfunction
create 2 CUstreams as an array variable streams
cudaHostAlloc a byte buffer
cuMemAlloc input array variables :
$user\_item[user_m]$, $user\_item[:][item_n]$,
$adUserItem[user_m]$, $adItemUser[item_n]$
for i = 1 to STREAM_NUMBER do
    cuMemcpyHtoDAsync (streams[i % 2])
end
for i = 1 to STREAM_NUMBER do
    cuLaunchKernel (streams[i % 2])
end
for i = 1 to STREAM_NUMBER do
    cuMemcpyDtoHAsync (ByteOffset, streams[i % 2])
end
finalScore = sum (result)
End
```

4.4. Implementation on GPU clusters

Furtherly, based on previous optimized GPU kernel, we implement the algorithm on our GPU clusters to accelerate the speed of the recommendation algorithm. We adopt socket with customized data packet to handle the communication between GPU nodes. The data packet consists of corresponding input data. Detail architecture of the communication model among different nodes can be seen in Figure 5.

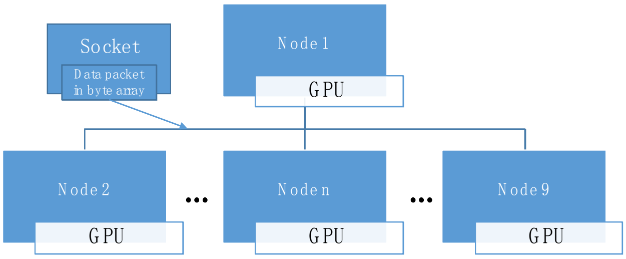


Fig. 5. The architecture of the communication model among 9 different nodes. And n is a number of node and it ranges from 3 to 8. The customized data packet send/receive is based on socket between nodes.

In Figure 5, we setup a node of the clusters as master node, other 8 nodes are responsible for communicating to it as slave node. Although customizing data packet to communicate between GPU nodes is not a universal method to handle communication between different processes, it can save a certain amount of time cost than some parallel communication frameworks such as MPI et al in experiments. Besides, we also use different sizes of datasets to observe the performance of the recommendation algorithm on clusters.

For each GPU node, we dispatch comparatively large amount of users for recommendation with our customized data packet so that we can decrease the time cost on communication between processes. In this way, we expect to maximize the speedup of the program.

We also implement the parallel program on our CPU clusters. Each node in CPU clusters have a multi-core CPU, we use Java Fork/Join to parallelize our programs on thread-level and we use our communication model between nodes is built by customized data packets. We hope to compare the performance of CPU clusters based parallel program and the performance of our implementation on GPU clusters to observe their differences.

5. Experiment and Result Analysis

5.1. Experimental environment settings

The experiments are conducted on CPU-GPU clusters, which are consist of 9 nodes. Each node in the cluster has the following configuration:

- CPUs in clusters: Intel(R) Core™ i7 4790 (4 cores, 8 threads in all), frequency is 3.60GHz.
- GPUs in clusters: The GPU clusters consists of 2 GTX 980 and 7 GTX 970. The clock frequency of GTX 980 is 1279MHz and that of GTX 970 is 1253MHz.
- The version of Java virtual machine runtime is Java 8 Update 60.
- The version of CUDA runtime is 7.5.
- The MovieLens dataset used in the experiment is from GroupLens.[20] GroupLens Research has collected and made available rating data sets from the MovieLens website. The data sets were collected over various periods of time, depending on the size of the set.
- The operating system: Windows 10 x64 Professional.

5.2. Experiment results and related analysis

5.2.1. Recommendation algorithm results evaluation

To evaluate the recommendation algorithm, experiments are conducted on MovieLens dataset. Meanwhile, the area under the ROC curve[21], recall[22], diversification[22] and novelty[23] are employed as metrics. For a particular user, the area under ROC curve, abbreviated by AUC, is the probability that a randomly selected removed item is given a higher score by this algorithm than a randomly selected uncollected item, and the AUC for whole system is the average of all users. Recall is used to quantify the accuracy of recommended items, diversification measures the uniqueness of different users' recommendation lists and novelty quantifies the capacity of an algorithm to generate novel and unexpected results.

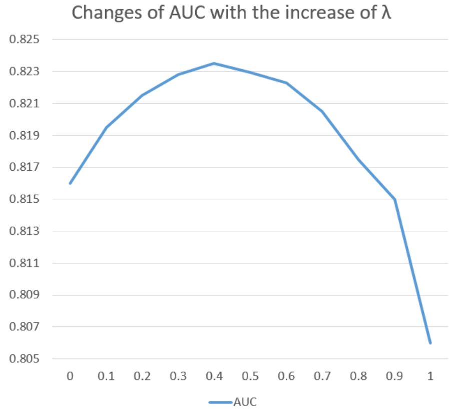


Fig. 6. The changes of AUC with the increase of λ, Y axis represents AUC and X axis represents λ.

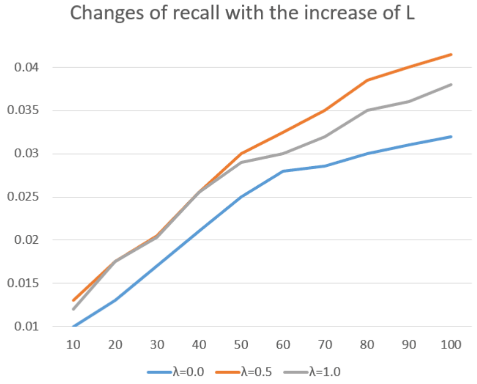


Fig. 7. The changes of recall with the increase of L, Y axis represents recall and X axis represents L (L is the length of recommendation list).

The experimental results conducted in the works are shown in the Figure 6, 7, 8 and 9 which are similar with that of Zhang et al[9]. From them we can see that when considering AUC, at the optimal values, the improvement for MovieLens is 2.1% compared with the algorithm without tag information[18] Furthermore, in

Figure 8 and Figure 9, we find that the algorithm depending more on tags can provide more personalized and higher novel recommendations.

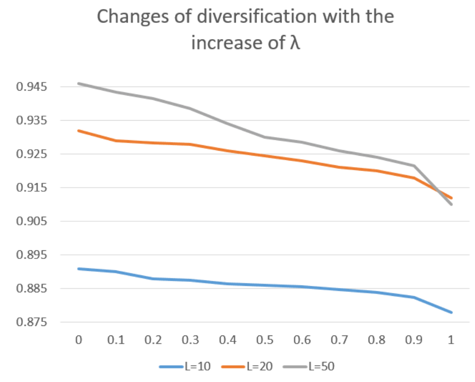


Fig. 8. The changes of diversification with the increase of λ, Y axis represents diversification and X axis represents λ.

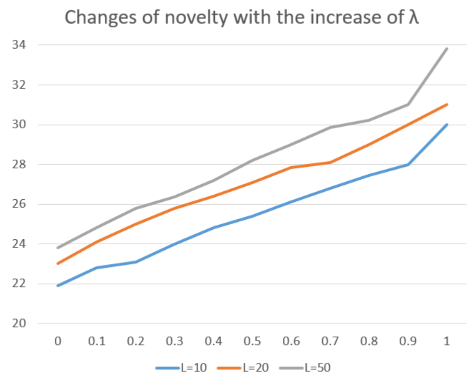


Fig. 9. The changes of novelty with the increase of λ, Y axis represents novelty and X axis represents λ.

5.2.2. *GPU kernel experiment*

We conduct a series of experiments on different size of datasets, the size of datasets ranges from 7500 records to 100000 records based on MovieLens dataset. These experiments aim at verification of GPU programs' performance improvement and speedup. Figure 10 is the average speedup of GPU programs on different size of datasets. Figure 11 is the properties of GPU programs on dataset of size = 7500 records.

The speedup result in Figure 10 shows the GPU kernel can launch multiple threads simultaneously to improve performance. The speedup of GPU program is very considerable. The speedup can reach 153.91 when the input dataset has 30000 records. Here the fluctuation of the speedup in Figure 10 should be this reason: when the size of dataset increase from 10000 to 30000, the number of user is increasing but is less than 1024 (GPU Block dimension is 1024). That means threads in a block are becoming more and more utilized, at the same time, the number of blocks allocated increases with the item number, thus the speedup is raising. But when the size of dataset increase from 30000 to 100000, the threads in a block have already been fully utilized and due to thread divergence, the speedup fluctuate under different sizes of input dataset.

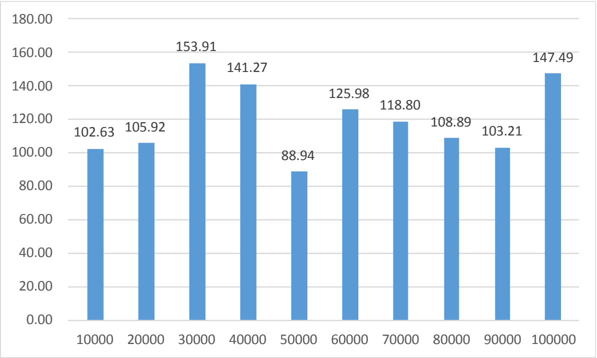
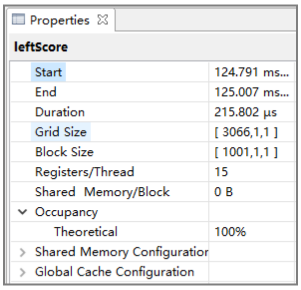


Fig. 10. The average speedup of GPU programs on different size of dataset. In the figure, the X axis represents the size of dataset (number of records) and the Y axis represents the speedup of GPU kernels.

And from Figure 11 we can see under this thread/block model (the number of threads in a block nearly reach 1024), the theoretical occupancy can reach 100%, which means the GPU device is fully utilized in the experiment.



leftScore

Start	124.791 ms...
End	125.007 ms...
Duration	215.802 μs
Grid Size	[3066,1,1]
Block Size	[1001,1,1]
Registers/Thread	15
Shared Memory/Block	0 B
Occupancy	
Theoretical	100%
Shared Memory Configuratior	
Global Cache Configuration	

Fig. 11. The properties of GPU kernel on dataset of size = 7500 records.

5.2.3. *Shared memory optimization*

Based on GPU kernel in the previous experiment, we optimize the memory access of the kernel with shared memory. We allocate shared memory for storing result array to decrease the time cost on memory access. Figure 12 is a detail time cost comparison on different sizes of datasets.

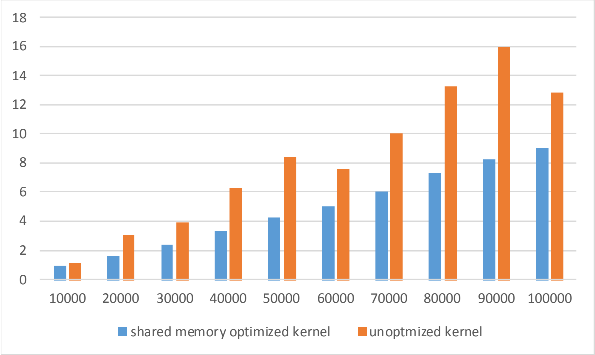


Fig. 12. The comparison of time cost between shared memory optimized kernel and unoptimized kernel of GPU programs on different size of dataset. In the figure, the X axis represents the size of dataset (number of records) and the Y axis represents the time cost of GPU kernels (units in millisecond). The blue bar graph is the time cost of GPU kernels optimized with shared memory and the orange bar graph is the time cost of unoptimized GPU kernels.

From these results we can see that the shared memory optimization can dramatically improve the performance of kernels which contain lots of global memory access. In our experiments, shared memory optimized kernels can improve about 50% on performance when the size of dataset is 90000 records.

5.2.4. *Stream scheduling optimization*

In the experiments with input dataset containing 8000 - 20000 records, the time cost on memory copy operation from GPU device to host takes up a certain proportion of the total execution time of GPU kernels. Therefore, for these cases, it is necessary to optimize these kernels with CUDA streams to implement kernel concurrency.

In Figure 13, we implement the CUDA stream optimized GPU kernel of left bipartite graph on dataset of 10000 records. The kernel in the other stream can overlap the memory copy (from GPU device to host) of previous kernel in the stream. Although the memory latency does not take much proportion in one kernel's period, the CUDA stream optimized kernel still brings improvement to program's performance.

For the dataset of size ranges from 7500 - 20000 records, the improvement of stream optimized kernel could achieve 7% ~ 13% on average, while for the dataset of size larger than 20000 records, the improvement can only achieve less than 7% on average. In all, CUDA stream scheduling mechanism does can improve the performance of GPU kernels.

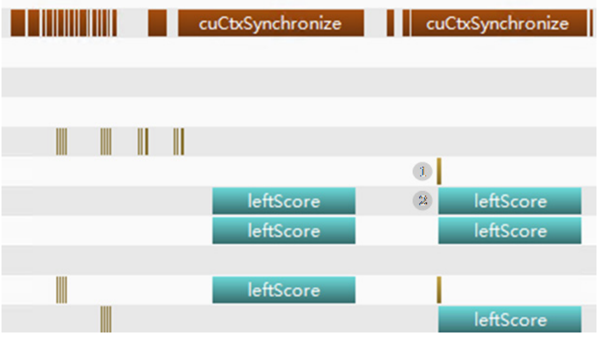


Fig. 13. The snapshot in NVIDIA Visual Profiler on stream optimized kernel of left bipartite graph on dataset of 10000 records. Marker 1 represents the memory copy operation from device to host, and marker 2 represents the GPU kernel.

5.2.5. *GPU clusters experiment*

In cluster experiment, the GPU kernels are optimized with shared memory and stream scheduling. The experiment results are collected on recommending for 8000 users with dataset of 100000 records. Firstly, we tallied the average time cost on a single CPU core for recommending 8000 users with diffusion-based recommendation algorithm on tripartite graph, and it will take 30336 seconds completing recommendation. Then we use our cluster-based GPU program instead, the total time cost is 19.124 second. We achieved 1586.28 speedup on GPU cluster. Compared to a single GPU node, we achieve 7.55 speedup on a cluster of 9 nodes on average.

The paralleled algorithm with Java Fork/Join, we dispatch our recommendation tasks on 8 Java threads and achieve 7.9 speedup on the same size of dataset. Based on this threads configuration, we ported the algorithm to clusters of 9 nodes, the program achieved 60.8 speedup compared to serial algorithm on a single CPU core. From the performance results of our CPU and GPU clusters, we can find out that GPU technology does bring a very considerable improvement in performance.

6. Conclusion and Future Work

In this era of information overload, recommender system has been playing a more and more important role in all sorts of areas. Many algorithms have been proposed to improve the quality of recommendation. The usage of tag information helps tripartite algorithm significantly improve its accuracy, diversification and novelty. According to the experiments results, we can conclude that GPU technology can improve the performance of recommendation algorithm dramatically. Shared memory, streams scheduling and GPU clusters can bring a further improvement on performance. The speedup can achieve 1586.28 on clusters recommending for 8000 users. Compared to a single GPU node, we also achieve 7.55 speedup on a cluster of 9 nodes. The speedup of GPU clusters is also 26.1 times of the speedup of CPU clusters. It indicates that the powerful platform with GPUs and multiple optimization strategy provides an effective method with high performance to recommend for multiple users.

Future work includes but is not limited to: re-configure thread/block model to adapt large size of dataset (records number ranges from 100000 to 1000000); implement the algorithm on GPU arrays for performance comparison experiments; try to apply other heterogeneous solutions, such as Field Programmable Gate Array (FPGA) or other hardware accelerators.

References

1. P. Resnick and H. Varian, Recommender systems, *Communications of the ACM*, **40**(3) (1997), pp. 56-58.
2. A. Q. Macedo, L. B. Marinho, and R. L. Santos, Context-Aware Event Recommendation in Event-based Social Networks, in *Proceedings of the 9th ACM Conference on Recommender Systems*, (2015), pp. 123-130.
3. X. Yi, L. Hong, E. Zhong, N. N. Liu, and S. Rajan, Beyond clicks: dwell time for personalization, in *Proceedings of the 8th ACM Conference on Recommender systems*, (2014), pp. 113-120.
4. J. B. Schafer, J. A. Konstan, and J. Riedl, E-commerce recommendation applications, in *Applications of Data Mining to Electronic Commerce* (Springer US, 2001), pp. 115-153.
5. Wikipedia, *Tag (metadata)*, (2016), [online] Available at: https://en.wikipedia.org/wiki/Tag_(metadata) [Accessed 5 Apr. 2016].
6. Delicious.com, *Delicious*, (2016), [online] Available at: http://delicious.com/ [Accessed 5 Apr. 2016].
7. GroupLens, *MovieLens*, (2013), [online] Available at: http://grouplens.org/datasets/movielens/ [Accessed 5 Apr. 2016]..
8. H. KDE Group, *BibSonomy :: home*, (2016), [online] Bibsonomy.org. Available at: http://www.bibsonomy.org [Accessed 5 Apr. 2016].
9. Z. Zhang, T. Zhou and Y. Zhang, Personalized recommendation via integrated diffusion on user–item–tag tripartite graphs, *Physica A: Statistical Mechanics and its Applications*, **389**(1) (2010), pp. 179-186.
10. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, et al., A Survey of General-Purpose Computation on Graphics Hardware, *Computer Graphics Forum*, **26**(1) (2007), pp. 80-113.
11. NVIDIA Developer, *CUDA Zone*, (2015), [online] Available at: https://developer.nvidia.com/cuda-zone [Accessed 5 Apr. 2016].
12. T. Bai, S. Davis, J. Li, Y. Gu and H. Jiang, Accelerating NTRU Encryption with Graphics Processing Units, *International Journal of Networked and Distributed Computing*, **2**(4) (2014), pp. 250-158.
13. C. Hung and S. Guo, Fast Parallel Network Packet Filter System based on CUDA, *International Journal of Networked and Distributed Computing,* **2**(4) (2014), pp. 198-210.
14. K. Kato and T. Hosino, Solving k-Nearest Neighbor Problem on Multiple Graphics Processors, in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, (IEEE Computer Society, 2010), pp. 769-773.
15. R. Li, Y. Zhang, H. Yu, X. Wang, J. Wu, and B. Wei, A social network-aware top-N recommender system using GPU, in *Proceedings of the 11th annual international ACM/IEEE joint conference on Digital libraries*, (ACM, 2011), pp. 287-296.
16. C. H. Nadungodage, Y. Xia, J. J. Lee, Gpu accelerated item-based collaborative filtering for big-data applications, in *Big Data, 2013 IEEE International Conference on*, (IEEE, 2013), pp. 175-180.
17. Q. Ou, Y. Jin, T. Zhou, B. Wang and B. Yin, Power-law strength-degree correlation from resource-allocation dynamics on weighted networks, *Physical Review E*, **75**(2) (2007).
18. T. Zhou, J. Ren, M. Medo, Bipartite network projection and personal recommendation, *Physical Review E*, **76**(4) (2007).
19. Jcuda.org, jcuda.org - Java bindings for CUDA, (2016), [online] Available at: http://jcuda.org/ [Accessed 5 Apr. 2016].
20. GroupLens, *GroupLens*, (2016), [Online]. Available: http://grouplens.org/. [Accessed 5 Apr. 2016].
21. A. Clauset, C. Moore and M. Newman, Hierarchical structure and the prediction of missing links in networks, *Nature*, **453**(7191) (2008), pp. 98-101.
22. J. Herlocker, J. Konstan, L. Terveen and J. Riedl, Evaluating collaborative filtering recommender systems, *ACM Transactions on Information Systems*, **22**(1) (2004), pp. 5-53.
23. T. Zhou, L. Jiang, R. Su and Y. Zhang, Effect of initial configuration on network-based recommendation, *EPL (Europhysics Letters)*, **81**(5) (2008), p. 58004.