GPU Accelerated Bipartite Graph Collaborative Filtering

Qianbin Xia, xiaq2@vcu.edu

1. Introduction

A bipartite graph is one kind of graph in which all the vertices could be divided into two disjoint groups U and V. Vertices in U and V are connected through edges. For a bipartite graph, there is no edge within the same group U or V. Bipartite graph can be widely used to model problems in real life. For example, if you are boss of a company and one of your important employees has retired recently. Now you need to find some candidate from the LinkedIn to fill this opening position. How can you find the proper candidate from so many members in LinkedIn? Here we can use bipartite graph to model this problem. The members of LinkedIn compose the group U, while different kinds of skills are in group V. If there is an edge between one member in U and one kind of skills in V, that means the member has this kind of skill. The retired employee should have some specific skills to be good worker in his field. The problem to find proper candidate for the retired employee turned to find the members from the LinkedIn who share the largest number of same skills with the retired employee.

Another example is marriage problem. There are x men and y women desire to get married. Participants indicate who among the opposite sex would be acceptable as potential spouse. If you are one of them, how can you choose several potential spouses from all the opposite sex? To solve this problem, we can first model this problem with bipartite graph. All the participants compose group U, while group V consists of all the properties of the participants including: ages, appearance, job, salary, hobbies and so on. Every participant should have their own expectation for their other half. Now the problem is simple, one just need to find out who meets the most of his or her expectations and that one would be a good candidate.

There two algorithms: naive graph collaborative filtering and full graph collaborative filtering which can help us to make a good decision for problems like personal and marriage issues. Figure 1 shows the definition of the naive graph collaborative filtering and full graph collaborative filtering. The naive graph collaborative filtering is to find out k most related or matched vertices for a given

start vertex in group U. This algorithm can be applied directly to our previously proposed personal and marriage problems. The full graph collaborative filtering algorithm is based on the naive graph collaborative filtering. After finding out k most related vertexes for a given start vertex, the full graph collaborative filtering will find out k most related vertexes to each of the k vertex getting from naive graph collaborative filtering. This can has very important real applications in social network like Facebook. For friend recommendation systems in social networks, if we can find out k best friends from all Eric's friends, then we search k best friends for each of Eric's k best friends respectively. The meaning behind the full collaborative filtering algorithm is that, if two people are best friends, there is a high possibility that they will know each other's best friends through birthday party or some other activities. So if they are not friends on Facebook, we can recommend them to each other.

Definition 1.1. (Naive Graph Collaborative Filtering): Given a bipartite graph $G((U,V),E)$ and a vertex $u \in U$, find top k relevant vertices to u, denoted by $u'_1, u'_2, \cdots, u'_k$ according to:

$$u'_i = \arg \max_{x \in U \setminus \{u'_1, \cdots, u'_{i-1}\}} |\Gamma_u \cup \Gamma_x| \quad (1)$$

where $\Gamma_u = \{v | v \in V, (u,v) \in E\}$.

Definition 1.2. (Collaborative Filtering for Visualization): Given a bipartite graph $G((U,V),E)$ and a vertex $u \in U$, find top k relevant vertices to u, denoted by $U' = \{u'_1, u'_2, \cdots, u'_k\}$; then, for each $u'_i \in U'$, find the top k' vertices relevant to u'_i, denoted by $U''_i = \{u''_1, \cdots, u''_{k'}\}$. Return $U' \cup \{U''_i\}_{\forall i=1,\cdots,k}$.

Figure 1 Naive and full graph collaborative filtering

Although bipartite graph with these two filtering algorithms can help to solve problems in real life, unacceptable long latency due to mass number of vertexes and edges in a graph is an issue need to be resolved. A graphics processing unit (GPU) has highly parallel structures which can run large number of kernels concurrently. Hence, GPU could provide better performance over CPU for algorithms where processing of large blocks of data is done in parallel. GPU as an accelerator is very suitable for Naive and full graph collaborative filtering since each individual vertex can be coped with independently for partial operations of the whole algorithms. In this project, we aim to reduce the computing latency

through assigning part of the whole job to GPU. CUDA, which stands for Compute Unified Device Architecture, is a parallel computing platform and programming model created by NVIDIA. CUDA gives developers direct access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs. Another choice is OpenCL, but as CUDA is more concise, it is will be used in my project.

2. Implementation

Figure 2 present the flow chart for the implementation of the whole algorithms. The whole process can be divided into five stages. The first is to build the bipartite graph from trace files. The second is relation counting for all vertexes in U with u_start. The next stage is sorting the relations in a descending order to get the k most related vertices. The fourth stage is based on the results of naive filtering to count relations in U for the k most related vertices respectively. The final step is to sort all the relations in a descending order, and remove the repeated vertices.

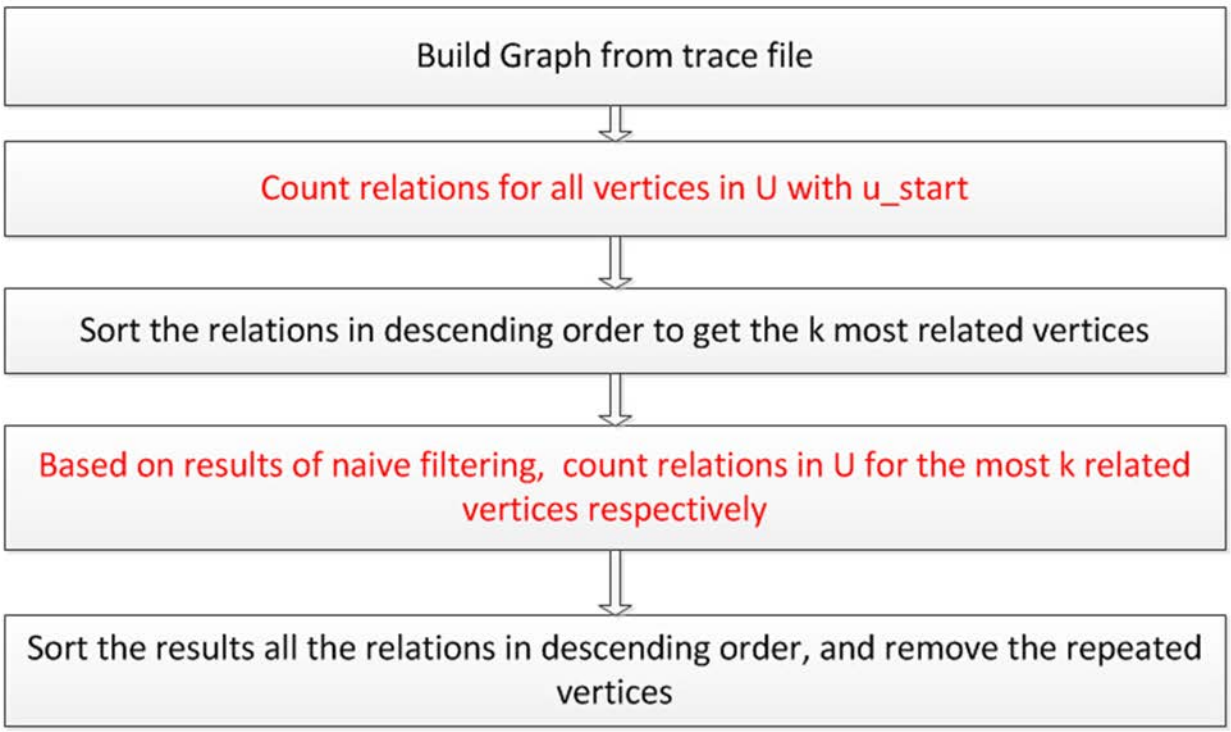

Figure 2 Flow chart for the implementation of the whole algorithms

On the hose side, we have implemented the whole algorithms with boost graph library and thrust graph library. Boost graph library only support the host side implementation. For thrust graph library, it is claimed to support CUDA graph library. However, thrust CUDA graph library doesn't work after testing. After the implementation of the host side filtering algorithm, we divide the whole algorithm into five stages: relation counting for naive filtering, sorting for naive filtering, relation counting for full filtering, sorting for full filtering, and repeated vertices removing. Execution time is measured for each stage. Table I shows the result. From the result we find, for intensive graph like Connectus, the relation counting will be the main contributor to the whole latency. While for sparse graph like NotreDame_actors the sorting part will be the performance bottleneck.

Table I Latencies for different computing stages

	RCN	SN	RCF	SF	RR
connectus	0.03	0	1.42	0.05	0.03
NotreDame_actors	0	240.35	0.03	12015.76	0.01

Since the boost graph library and thrust graph library can’t support the GPU side developing, I utilized an alternative solution to show our GPU accelerated implementation. We model the problem with many graph nodes which possesses multiple adjacent nodes. Here the graph node is the vertexes in group U, while the adjacent nodes are the connected vertexes in group V. For the host side implementation is similar to our boost and thrust graph library implementation.

Currently GPU is only utilized to accelerate the relation counting part of the whole algorithm. For naive filtering, each threads takes care of one vertex in group U and count how many same vertexes in group V they share with u_start. Shared memory is leveraged to reduce the global memory accessing latency. The grid and block dimensions are defined as following:

```
dimGrid(1, 1)
dimBlock(max_id_A, 1)
```

There is only one block, but the number of threads depends on the number of vertexes in group U in the x dimension.

For full graph collaborative filtering, shared memory is also utilized. The number of blocks depends on k. The number of threads per block depends on the number of vertexes in group U. The grid and block dimensions are defined as following:

dimGrid(1, k)

dimBlock(max_id_A, 1)

There are seven function units in the source code:

1) Main: read in the trace file and call the sub-functions.
2) FilterOnDevice: filtering on device.
3) naiveFilterKernel: kernel function for naive filtering.
4) fullFilterKernel: kernel function for full filtering.
5) filterOnHost: function for filtering on host side
6) selSort: sorting relations in a descending order.
7) Compare: compare the result between host side and gpu side to check correctness.

Table II lists the available bipartite graph traces. The second column is the number of vertexes in group U. The next column is the number of vertexes in group V. The last column is the number of edges in the graph.

Table II Bipartite graph traces

	Num of U	Num of V	Num of Edges
Divorce	50	9	225
Sandi_sandi	314	360	613
Connectus	512	394792	1127525
NotreDame_actors	392400	127823	1470404
IMDB	428440	896308	3782463

3. Results

The input for the program includes: trace file name, u_start, and k.

For Connectus with u_start = 3, the result shows in table III and figure 3. The result only include the latency from the naive relation counting, naive relation sorting, and full relation counting. For the GPU side, data transfer latency is also

included. As we have mentioned earlier, for intensive graph like Connectus, the relations counting is the bottleneck. Therefore, by utilizing GPU to accelerate the relation counting part can significantly improve the performance. From figure 3 we find the execution time on the GPU side keeps the same with the changing of k from 10 to 100. The reason is that the GPU has enough thread to parallelize the additional computing tasks. But on the CPU side, the execution time is linearly increasing with the increment of k due to serial execution property of CPU.

Table III Result for Connectus

k	10	20	30	40	50	60	70	80	90	100
CPU	6.30	12.15	18.00	23.24	29.14	35.00	40.86	46.71	52.55	58.43
GPU	1.01	0.98	0.955	1.02	1.07	1.06	1.13	1.04	1.06	1.145

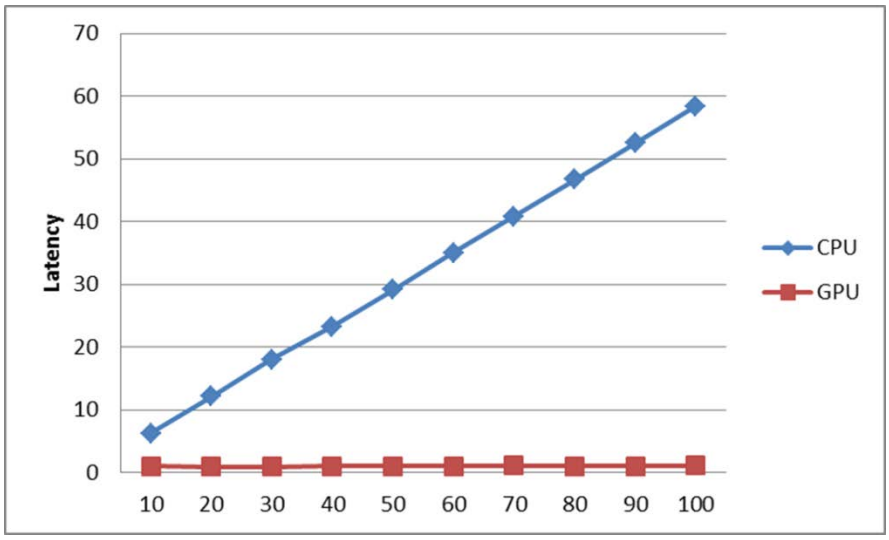

Figure 3 Result for Connectus

Table IV and figure 4 show the result for NotreDame_actors with u_start equals 211. NotreDame_actors is a sparse graph which means the relation sorting is the big head of the execution time. Since our result includes the naive relation sorting latency, so the improvement is not as remarkable as Connectus.

Table IV Results for NotreDame_actors

k	10	20	30	40	50	60	70	80	90	100
CPU	276	291	305	321	333	346	361	370	380	396
GPU	259	259	259	259	259	259	259	259	259	259

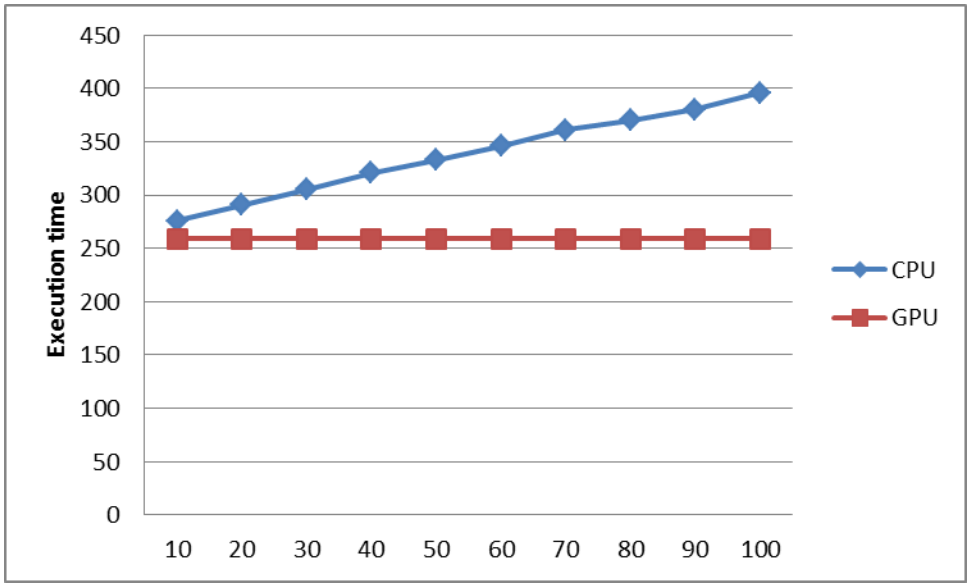

Figure 4 Execution time for NotreDame_actors

4. Conclusions

Graphs can be roughly divided into intensive graph and sparse graph depends on the ratio of number of vertexes and edges. For intensive graph, the relation counting will be the major contributor of the whole execution time. While for a sparse graph, the relation sorting will be the bottleneck. In our project we focus on the relation counting part and utilize GPU to accelerate this process. From the result, we find GPU can improve the performance for both intensive and sparse graph collaborative filtering. The execution times for naive relation counting, naive relation sorting, and full relation counting on the GPU side keeps the same with the increasing of k thanks to the high parallel computing power. While on host side, the execution time increases linearly with the increment of k for intensive sparse as the naive relation sorting process is negligible. The execution time for sparse graph will also increase due to the increase of k, but since most of the execution time is on relation soring part which we have not yet leverage GPU to accelerate, the result is not a linear line.

5. Work in the future

First, we can find a usable CUDA graph library to test the benefit under the real case. What's more, in our project we improve the performance of relation counting part, but leave the relation sorting part. Since in a sparse graph filtering process, the sorting part will takes much longer latency compared with the relation counting part, in order to get a universal solution, we need to leverage GPU to accelerate the sorting part in the future.