GPU Accelerated Bipartite Graph Collaborative Filtering

1. Progress Summary

By now, I have finished the host side implementation of naive Collaborative filtering and full collaborative filtering algorithms with boost graph library and thrust graph library respectively. In the following sections, I will firstly give a brief introduction to some basic graph terminology and these two different graph libraries.

2. Basic graph terminology and graph libraries

Figure 1 present a graph model for a network of internet routers. Each router is represented by a vertex labeled by a to e and each connection is represented by an edge. Each edge is associated with a weight which is the average transmission delay here. A graph G typically consists of a vertex set V and an edge set E. Thus, we write G = (V, E). The size of the vertex set (the number of vertices in the graph) is expressed as |V| and the size of the edge set as |E|. An edge is written as an ordered pair consisting of the vertices connected by the edge. The ordered pair (u, v) indicates the edge that connects vertex u to vertex v. A graph can be directed or undirected, meaning the edge set in the graph consists respectively of directed or undirected edges.

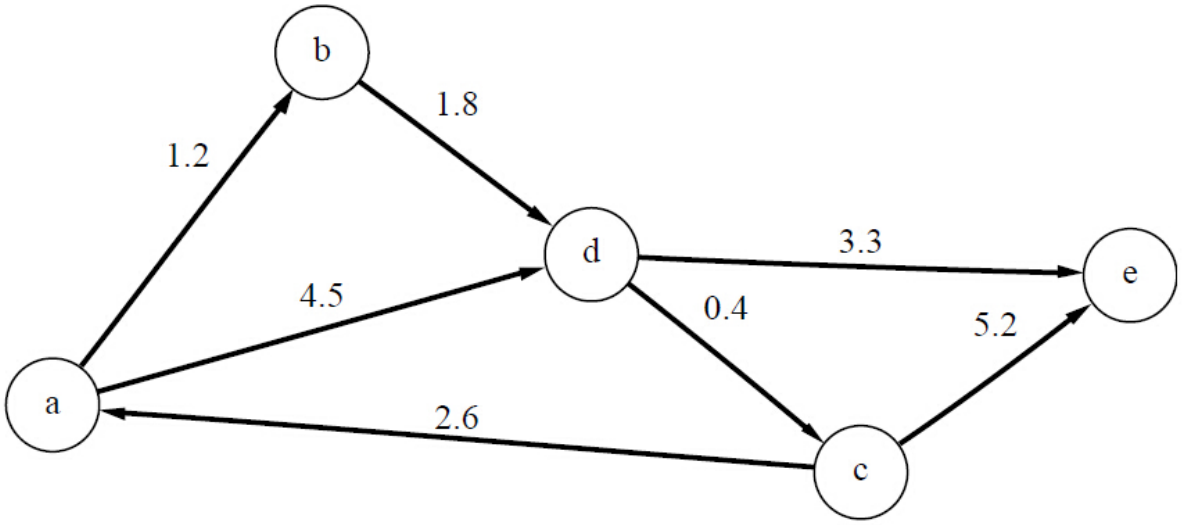

Figure 1 A network of Internet routers

The graph abstraction consists of several different kinds of collections: the vertices and edges for the graph and the out-edges, in-edges, and adjacent vertices for each vertex. Similar to the STL, the BGL uses iterator to provide access to each of these collections. There are five kinds of graph iterators, on for each kind of collection:

1. A vertex iterator is used to traverse all the vertices of a graph. The value type of a vertex iterator is a vertex descriptor.
2. An edge iterator is used to traverse all the edges of a graph. The value type of this iterator is an edge descriptor.
3. An out-edge iterator is used to access all of the out-edges for a given vertex u. Its value type is an edge descriptor. Each edge descriptor in this iterator range will have u as the source vertex and a vertex adjacent to u as the target vertex (regardless of whether the graph is directed or undirected).
4. An in-edge iterator is used to access the in-edges of a vertex v. Its value type is an edge descriptor. Each edge descriptor is this iterator range will have v as the target vertex and a vertex that v is adjacent to as the source.
5. An adjacency iterator is used to provide access to the vertices adjacent to a given vertex. The value type of this iterator is a vertex descriptor.

Thrust is a C++ template library for CUDA based on the Standard Template Library (STL). Thrust allows you to implement high performance parallel applications with minimal programming effort through a high level interface that is fully interoperable with CUDA C. Thrust provides a rich collection of data parallel primitives such as scan, sort, and reduce, which can be composed together to implement complex algorithms with concise, readable source code. But the standard thrust library doesn't provide any graph interface, so thrust graph library was developed by a third party. Although claimed can provide graph library for GPU side, but after check the source code and run the given examples, only the host side graph library works, while the GPU side graph library doesn't work.

3. Implementation

Figure 2 depicts the flow chart of whole process. First step is to build the bipartite graph from the bipartite graph trace files. Then we need to input the u_start and k. We will count relations for all vertices in part U with u_start and sort the relations in a descending order to get the k most related vertices. The full collaborative filtering is

based on the naive collaborative filtering, we count relations in part U for the most k related vertices respectively. The final stage is to sort all the relations in a descending order and remove the repeated vertices. We implemented this whole algorithm with both boost graph library and thrust graph library on the host side. The implementation is almost the same, except that the thrust graph library can't support undirected graph, but for the other aspects it is nearly the same.

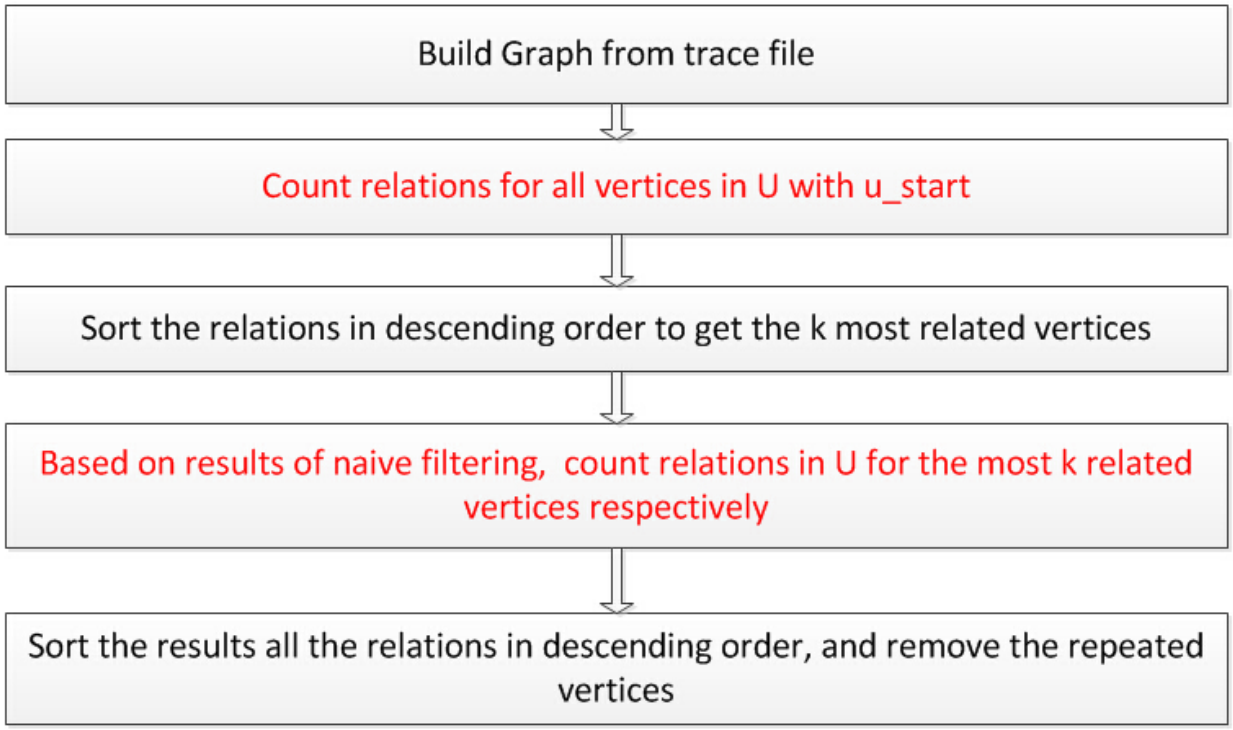

Figure 2 Flow chart for the implementation of the whole algorithm

Table I lists the bipartite graph trace files used in our experiments. The first column is the trace file names. The second column is the number of vertices in party U, while the third column is the number of vertices in party V. The last column is the number of edges in the graph. All these parameters will affect the execution time.

Table I 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

4. Results

Currently, I only finished the host side implementation. For the GPU side, as the boost graph library doesn't provide GPU side library, we can't make it with BGL. Even the thrust is CUDA supportive, but the thrust graph library doesn't work. The input parameters are as following:

Trace file name: the trace file used to build the graph

u_start: the start vertex in party U

k: number of most related vertices need to be found

Since we don't have the device side implementation right now, we just want to get performance bottleneck of our whole algorithm through the host side experiments. 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. During our experiment, we will show the execution times for each stage separately. We could roughly classify the bipartite trace file into two categories: intensive graph and sparse graph. Intensive graphs mean they have a higher average vertex degree that vertex in party U will have more connections with vertex in party V. Connectus is an example of intensive graph. While all the others can be treated as sparse graph. For intensive graph like connectus, the relation count stage will contribute to the main latency of the whole program. While for sparse graphs, the sorting stage will be the bottleneck. For the course project we will focus on the relation count part and utilize the GPU to reduce the latency. Table II shows the result for connectus and NotreDame_actors.

Table II 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

RCN means relation counting for naive collaborative filtering. SN means sorting for naive collaborative filtering. RCF and SF are for full collaborative filtering. RR means repeated

vertices removing. From the result, we find for connectus which is an intensive graph the relation counting is major contributor of the whole latency. While for sparse graph like NotreDame_actorsmt, the sorting party will be the bottleneck and the relation counting part could be negligible.