ROP299 Status Report

Ruijian An

July 5, 2017

Introduction

Given a weighted graph G = (V, E) and a source vertex s, single source shortest path problem(SSSP) asks the shortest distance from the source vertex to all vertices in the graph[2]. SSSP is a fundamental part of graph theory and has broad applications in road networks[1], DNA microarrays[3] and many other problems which can be abstracted as graphs. Many algorithms, such as Dijkstra's algorithm and Bellman-Ford algorithm, are proposed to solve SSSP. To minimize the running time, Dijkstra's algorithm utilizes a priority queue, so it is sequential in itself. On the other hand, Bellman-Ford algorithm does much more work but contains a lot parallelism. Confronted with huge graphs like social network and road network, it is almost impossible to process vertices one by one using Dijkstra's algorithm. However, modern GPGPU provides a brand new solution to SSSP. In the report, I present my current work on parallelization of Bellman-Ford algorithm.

Related work

In 1950s, Bellman-Ford algorithm is proposed by Bellman and Ford. In the algorithm, each vertex in the graph has an additional attribute d to record the tentative distance from source vertex to itself. The algorithm also employs a technique called relax, which is used to process edges. If we relax an edge (u,v) whose weight is $w(u,v)$, we update the $v.d$ with $min(v.d,\ u.d + w(u,v))$. The Bellman-Ford algorithm relaxes each edge for $|V|$ - 1 times, so the running time is O($|V||E|$).

With the development of modern GPU, many parallel versions are proposed. The main parallelism is embedded in the step which relaxes all edges in the graph. P.Harsih proposes to first visit all vertices in parallel and relax outgoing edges[6]. However, many threads might read and write $v.d$ for some vertex v concurrently, so atomic operation is necessary to avoid race conditions. Instead of relaxing leaving edges, G.Hajela relaxes all incoming edges[7]. In this way, atomic operation is avoided, but either a loop or a nested kernel is introduced.

To optimize the performance of the parallel version, several techniques are employed to increase work efficiency. A.Davidson tests performances of *workfront sweep, near-far, bucketing*[9]. Among them, *near-far* achieves best result. It partitions the vertices into different buckets so that it can process the vertices with smaller value first, reducing the number of iterations of outer loop. F.Busato uses frontier propagation to largely decrease unnecessary work[10]. Frontier propagation takes advantage of a queue which only contains active vertices.

Parallel implementation

In my implementation, I combine the existing techniques and make some improvement. Most parallel versions above chooses to map a thread to each vertex and then relaxes incoming/outgoing edges. However, such way causes many potential inefficiency. First, the degree of vertices is different, which means each thread might face the imbalanced work problem. Second, parallelism hidden in the algorithm is not fully exploited. After visiting vertices in parallel, either a loop or a nested kernel is necessary to relax edges related to it[10]. Instead, I choose to map threads to edges. Usually, general or dense graphs with huge number of vertices have many more edges(which is $O(|V|^2)$ than vertices(which is $O(|V|)$). This method solves the imbalanced work problem and introduces more parallelism.

First, we need to choose a proper graph representation. Traditionally, graph is represented by either adjacency matrix or adjacency list. Also, compact adjacency list is commonly used in parallel graph algorithms[6][9]. CUDA can directly map threads to 1D, 2D arrays due to the kernel call, so adjacency matrix is the best choice for mapping to edges. Also, adjacency matrix is to generate and to test. Contrarily, it is hard to form a one-to-one correspondence between edges and threads for adjacency list due to different degree of vertices. For compact adjacency matrix, additional technique like scan are needed, which complicates the problem.

Algorithm 1: MAIN_ BELLMAN_FORD

```
   Input: dist, src_idx, matrix
 1 flag = 0;
 2 mask = [0, ..., 0];
 3 mask = [0, ..., 0];
 4 CUDA_INITIALIZATION(dist, src_idx, mask, mask1);
 5 for i ← 1 to |V| − 1 do
 6 |   CUDA_RELAXATION(dist, matrix, mask, mask1, flag);
 7 end
 8 if flag == 0 then
 9 |   break;
10 end
11 CUDA_SWAP(mask, mask1, flag);
```

The input of main function is *dist, src_idx, matrix*. The length of *dist* is equal to

the number of vertices in the graph and dist[i] represents the tentative shortest distance from source vertex to vertex *i*. *Src_idx* is the index of the source vertex. *Matrix* is an adjacency matrix where *matrix*[*i*][*j*] stores the weight of edge *(i, j)*.

This parallel version utilizes three additional structures *mask, mask1, flag*. First, *mask, mask1* aims to saves unnecessary relaxations. *Mask* records whether a vertex is active, so the length of both arrays is equal to $|V|$. If vertex i is active/inactive, then $mask[i] = 1/0$. In algorithm 3, we first check whether the starting vertex of an edge, and we only relax the edges related to active vertices, which saves much unnecessary work. A vertex i is marked as active once $dist[i]$ is updated. However, we cannot simply set $mask[i]$ to be 1 and set its starting vertex back to be inactive. As a result, *mask1* is needed to avoid problems and conflicts. We read from *mask*, update information in *mask1* and swap (algorithm 4) *mask* with *mask1* after one iteration

Second, *Flag* aims to break the outer loop once all vertices have found their shortest path. *Flag* is initialized to be 0 and would be set to 1 in algorithm 3 if any vertex's distance is updated. After each iteration, if *flag* is still 0, then the main algorithm can terminate early.

Algorithm 2: CUDA_INITIALIZATION

```
   Input: dist, src_idx, mask, mask1
1  idx = getThreadIDx;
2  dist[idx] = INFINITY;
3  mask[idx] = 0;
4  mask1[idx] = 0;
5  if idx == src_idx then
6  |  dist[idx] = 0;
7  |  mask[idx] = 1;
8  end
```

Algorithm 3: CUDA_RELEXATION

```
   Input: dist, matrix, mask, mask1, flag
1  i = getThreadIDx;
2  j = getThreadIDy;
3  if mask[i] == 1 then
4  |  temp = atomicMin(dist[j], [i] + matrix[i][j]);
5  |  if temp < dist[j] then
6  |  |  mask1[j] = 1;
7  |  |  flag = 1;
8  |  end
9  end
```

Algorithm 4: CUDA_SWAP

```
Input: mask, mask1, flag
1 i = getThreadIDx;
2 temp[i] = mask1[i];
3 mask[i] = temp[i];
4 mask1[i] = 0;
5 flag = 0;
```

Discussion and Performance

In the initial Bellman-Ford algorithm, relaxation of all edges in the graph is the most expensive step, so the parallelization of the step determines overall performance. However, there are many ways to parallelize it. First, we have two choices: mapping threads to edges or mapping threads to vertices. In a general or dense graph, the number of edges is probably larger than than of vertices. As a result, mapping threads to edges introduces more parallelism and eliminates imbalanced work problem. However, the number of streaming multiprocessors and maximum number of threads on a SM is limited, which directly limits the size of the graph. Once the limit is reached, huge amounts of blocks launched by kernel must wait in order to be processed. To solve this problem, it is necessary to process a bunch of edges per thread. On the other hand, mapping thread to vertex can easily handle a much larger graph.

Second, an edge *(u,v)* can be considered as either an incoming edge or an outgoing edge. If we maps a thread to an edge, this problem makes little difference. Nevertheless, if a thread is mapped to a vertex, two ways has big impact on the implementation. I present the pseudocode of two implementations.

Algorithm 5: CUDA_RELEXATION_INCOMING

```
Input: dist, matrix, mask, mask1, flag
1  idx = getThreadIDx;
2  minimum = dist[idx];
3  for i ← 0 to |V| − 1 do
4      if mask[i] == 1 then
5          if dist[i] + matrix[i][idx] < minimum then
6              minimum = dist[i] + matrix[i][idx];
7              mask1[i] = 1;
8              flag = 1;
9          end
10     end
11     dist[idx] = minimum;
12 end
```

Algorithm 6: CUDA_RELEXATION_OUTGOING

```
   Input: dist, matrix, mask, mask1, flag
 1 idx = getThreadIDx;
 2 if mask[idx] == 1 then
 3 |   for i ← 0 to |V| − 1 do
 4 |   |   temp = atomicMin(dist[i], dist[idx] + matrix[idx][i]);
 5 |   |   if temp < dist[i] then
 6 |   |   |   mask1[i] = 1;
 7 |   |   |   flag = 1;
 8 |   |   end
 9 |   end
10 end
```

It is hard to see which method would have better performance. In the following week, I would implement these variant kernels to see the results.

Future work

Many details in the implementation can still be improved. First, Cormen proves that any subpaths of shortest paths are shortest paths[2]. In a dense graph, there are many paths from one vertex to another, which means the heavy edge is unlikely to be part of the shortest path. Also, A.Davidson's experiment shows that partitioning vertices into near piles and far piles achieves remarkable improvements[9]. In my implementation, I relax edges in parallel, so I think partitioning edges into light and heavy buckets may improve the performance, which I would test in the next week.

Second, the outer loop runs at most $|V|$ - 1 times. In last several iterations, the number of active vertices is small, so it might not be efficient to launch kernels on GPU to process them. Instead, CPU might do better in processing small number of vertices. As a result, a hybrid implementation utilizing both CPU and GPU probably outperforms the existing parallel versions.

Third, I choose to use adjacency matrices as the input. Such representations works well for dense or complete graph, but it is not a good match for sparse or general graph. To better process sparse graphs, additional data type which stores edges would be introduced.

Reference

1. F. B. Zhan, and C. E. Noon, Shortest Path Algorithms: An Evaluation Using Real Road Networks, Transportation Science, 1996.

2. Cormen, T. H., Cormen, T. H. (2001). Introduction to algorithms. Cambridge, Mass: MIT Press.

3. K. Lee, J. H. Kim, T. S. Chung, B. S. Moon, H. Lee and I. S. Kohane, Evolution strategy applied to global optimization of Clusters in Gene expression data of DNA Microarrays, Proc. 2001 IEEE Congress on Evolutionary Computation, May 2001, vol. 2, pp. 845-850, doi:10.1109/CEC.2001.934278.

4. R. Bellman, On a Routing Problem, Quarterly of Applied Mathematics, vol. 16, pp. 8790, 1958.

5. L. A. Ford, Network Flow Theory, Report P-923, The Rand Corporation, 1956.

6. P. Harish, V. Vineet and P. J. Narayanan, Large graph algorithms for massively multithreaded architectures, Technical Report IIIT/TR/2009/74, International Institute of Information Technology Hyderabad, India, 2009.

7. Hajela, G. and Pandey, M. (2014) Parallel Implementations for Solving Shortest Path Problem Using Bellman-Ford. International Journal of Computer Applications (0975-8887), 95, 1-6.

8. S. Kumar, A. Misra, and R. Tomar, A modified parallel approach to single source shortest path problem for massively dense graphs using cuda, in Computer and Communication Technology (ICCCT), 2011 2nd International Conference on, sept. 2011, pp. 635-639.

9. A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel GPU methods for single source shortest paths. In Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium, pages 349-359, May 2014.

10. Busato, F. Bombieri, N. (2016). An Efficient Implementation of the Bellman-Ford Algorithm for Kepler GPU Architectures.. IEEE Trans. Parallel Distrib. Syst., 27, 2222-2233.