CS516 : Project 3 :SSSP Optimizations on GPU: Implementation of a Hybrid approach using Workfront Sweep and Near-Far pile method

Bhupeshraj Senthilkumar (bs718), Sairama Rachakonda(sr1122)
Rutgers University

1 Introduction

Finding the shortest paths is one of the fundamental problems in graph theory. Several solutions exists in finding shortest paths from a single vertex to all other vertices in the graph. They are called single source shortest paths (SSSP). The two popular algorithms that are used to find SSSP are Dijkstra's algorithm and Bellman-Ford algorithm.

These two algorithms span parallel vs efficiency spectrum. Dijkstra's algorithm exposes no parallelism but does efficient computation, whereas Bellman-ford algorithm has parallelism but does redundant computation making it inefficient. Several methods are suggested that explores the space between efficiency and parallelism that are targeted for fine-grained massively parallel machines such as GPUs.

2 Related Work

A. Dijkstra's Algorithm : It is the most effective sequential algorithm that can be used to find SSSP in the graphs. The algorithm maintains a priority queue of vertices based on its distance from the source vertex. At each iteration the algorithm considers the top vertex in the queue, which is shorter to the source vertex, and considers all edges leaving that vertex. The distances of the vertices reached by these edges are updated in the priority queue. The running time of the algorithm is $O(vlogv + e)$. But Dijkstra's algorithm shows no parallelism, the only parallelism it shows is the edges leaving the vertex at the top of the priority queue. So Dijkstra algorithm is poorly suited for GPU architecture.

B. Bellman-Ford Algorithm : Bellman-Ford Algorithm consider all edges at each iteration when compared to only the edges leaving the top of the priority queue in the Dijkstra's algorithm. Also Bellman-Ford algorithm operates on all vertices and each vertex maintains the distance from the source vertex. So at each iteration this distance is updated and the number of iterations performed is equal to the number of vertices - 1. So the running time of Bellman-Ford algorithm is $O(ve)$. Since each vertex distance is updated independently in each iteration, Bellman-Ford Algorithm is very well suited for GPU architecture.

C. Work-Front Sweep : This is an improvement on Bellman-Ford Algorithm as suggested in [1]. The strategy is to select a list of active vertices whose computation is changed from the previous iteration. At the beginning, we have only a vertex queue containing only the source, and on each iteration this method visits all the associated edges in parallel for each vertex in the queue and updates the distances of the vertices attached to those edges. This algorithm uses atomicMin operation to ensure proper updation of vertex distances. On any iteration, if the vertex distance is updated it is put into the vertex work queue for the next iteration. This process is continued till the work queue becomes empty. In this method the amount of work done to organize the queue at each iteration is minimal, but leads to substantial savings in edges touched.

D. Near Far Pile : In this method additional overhead is added to organize the work queue to increase the work efficiency. In work-front sweep all the updated vertices are treated with equal priority. But rather than processing all the vertices in the work queue we can select the subset of vertices based on the distance heuristic. Simple distance heuristic is to split the vertices in the work queue based on the incremental weight (Δ). At any iteration, the vertices in the work queue falls between Δi and $\Delta(i+1)$. The near pile consists of vertices less than distance Δi and the far pile consists of vertices greater than distance Δi. After all vertices in the near pile are processed the vertices from the far pile are

taken and split into near and far pile again based on the distance $\Delta(i+1)$.

This method's efficiency is based on the fact that unprocessed work in the far Pile can be discarded as closer vertices in the near pile are processed thereby minimizing the number of times we touch the data in the far pile. Selecting the proper Δ is important in deciding the tradeoff between parallelism vs work-saved. Based on [2]. the value of $\Delta = \Theta(1/d)$ shows good tradeoff between work-efficiency and parallelism. In [1] the value of $\Delta = cw/d$.

3 Motivation

The above methods as described in [1], gives efficient parallelism vs work-efficiency trade-off. The paper tests its implementation against eight graphs grouped into 3 types 1. Road Networks, 2. Meshes, and 3. Scale-free graphs. Road networks have high diameter and low degree while scale-free graphs have relatively high degree and low diameter. Meshes lie somewhere in between.

For high-diameter road networks, without preprocessing, their implementations are unable to outperform an efficient serial implementation. This is the expected behavior due to lack of parallelism. Also in high-diameter road networks, the implementation takes more iterations thus increasing the per-iteration overhead.

The paper analyzes the step efficiency at each iteration of parallel Bellman-Ford, Workfront sweep and Near-Far Pile method. Bellman-ford has best step efficiency and worst work efficiency, because all nodes are processed at each iteration, allowing for the fastest propagation from source to the other nodes. Dijkstra algorithm has the worst step efficiency and best step efficiency since only one node is processed at any given iteration. The methods Workfront sweep and Near-far pile are built so that they have enough parallelism to fully occupy the GPU with work-efficiency closer to Dijkstra's algorithm and step-efficiency closer to Bellman-ford.

Road Networks tend to have low parallelism-to-work-efficiency tradeoff. The Near-Far pile is nearly as work-efficient as the serial implementation, the amount of parallelism at every iteration is extremely low. The amount of parallelism is too low to saturate the GPU, and therefore is unable to outperform an efficient serial implementation. But when graph grows to a substantial size, the near-far pile method reduces unnecessary work while maintaining enough parallelism to saturate the machine.

Scale Free graphs tend to have high parallelism-to-work-efficiency tradeoff. So implementing Work-front sweep method for scale free graphs lead to unnecessary work reducing the step-efficiency.

So we need a hybrid approach that can deal with all types of graphs. The hybrid approach switches between Workfront sweep and Near-Far.

4 Solution

In the beginning of the SSSP algorithm in both Workfront sweep and Near-Far Pile method, there is often lack of parallelism and also after most of the graph is explored there will be a lack of parallelism.

Initially, during the start of SSSP methods, we don't have parallelism. Hence, we start with Workfront sweep method, to increase the parallelism. This way we increase the step efficiency. Once the step efficiency is increased and the work queue has sufficient parallelism, we need to start increasing the work efficiency. This can be done by prioritizing the work we do. To prioritize we switch to the Near-Far pile.

The hybrid approach will make sure that there is always sufficient parallelism while also making sure that the work is done on vertices which have higher priority.

The engineering challenge in our implementation would be, to come up with a good heuristic, which guides us to switch between the WorkFront sweep and the Near-Far pile method. We are working to come with a good heuristic.

5 Implementation

Our implementation of the hybrid approach that switches between the workfront sweep and the near far pile method is implemented as follows which uses maximum parallelism of the GPU at all times. The pseudocode of our parallel implementation is as follows:

1. Base Case : Initial Updated Vertices = source vertex
2. Maintain priority queue for the updated vertices based on the distance.
3. Calculate the number of edges needed to processed based on the updated vertices.
4. Based on the number of edges collected, we decide whether to implement near-far pile or workfront sweep method.

5. Once we decided whether to use near-far pile or workfront sweep method we implement parallel bellman-ford algorithm on the edges collected. The near-far pile is divide into near and far pile such that there is enough parallelism in the near pile vertices edges.
6. Get the updated vertices from the parallel bellman-ford algorithm and append the far pile vertices to the updated vertices for the next iteration
7. Loop steps 2 to 6 till there are no updated vertices.

5.1 Algorithm Analysis

The above algorithm makes sure that the parallelism is exploited in the GPU at all times. Often problem with selecting a delta in Near-Far pile is that sometimes there wont be enough parallelism in the near pile for every iteration.

The decision to choose between near-far pile and the workfront sweep method is decided based on the number of threads run by the kernel. We take the user given input and if the number of edges to be processed based on the updated vertices is greater than some integer multiple of total number of threads, then we have enough parallelism so we switch to near-far pile method by dividing the vertices such that near pile has edges close to that parameter we defined. Else we don't have enough parallelism and we use work-front sweep method.

So the hybrid approach will overcome the drawbacks of workfront sweep and the near-far pile method by making sure that we have enough parallelism in the GPU for computation and also making sure that no extra amount of work is done.

6 Implementation Issues & Results

The Near Far and Hybrid Implementation are implemented as part of the project. The graphs amazon0312, webgoogle, msdoor are working correctly for the near-far pile and the work front sweep method.

Other graphs like roadNet-CAL and roadNet-CA are giving segmentation fault due to high number of vertices. We feel like its due to memory constraints in the machine.

The Hybrid method implemented is still having issues which we couldn't debug within the time, so we couldn't get the results required for this project.

The results obtained for the workfront sweep are mentioned in the Project 2 Report. The kernel running time results obtained for the near-far pile method is mentioned below for the three graphs that we are able to run :

1. msdoor : 768ms (10x faster than sequential and 1.5x faster than workfront sweep)
2. amazon0312 : 24.51 ms(5x faster than sequential and 2x slower than workfront sweep)
3. web-google : 32.541 (11x faster than sequential and 1.5x slower than workfront sweep)

Graphs	Workfront Sweep	Near-Far Pile
RoadNet-CA	29	———————
msdoor	1033	768
roadNet-CAL	3032	———————
WebGoogle	23	32.541
Amazon0312	12	24

So by implementing the hybrid approach our goal was to have no particular method faster or slower for certain types of graphs and that all graphs have similar running time maximizing parallelism and reducing redundant computations.

We also experimented with changing different delta for different graphs. And for msdoor we found that by changing the delta we get different running times due to parallelization and redundant work tradeoff. This is shown by the table below :

Delta	Running Time (ms)
1000	678
2000	701
3000	770
5000	764
10000	867
20000	832

7 Conclusion:

We have implemented both near-far pile and the hybrid approach to find SSSP as a part of this project. We hope that the hybrid approach will deliver better results for all kinds of graphs by switching between near-far pile and workfront sweep method making sure that enough parallelism is maintained without doing redundant computations.

8 References

1. Work Efficient parallel GPU Methods for Single-Source Shortest paths by Andrew Alan Davidson, Sean Baxter, Michael Garland, John D Owens

2. U. Meyer and P. Sanders ”Δ- Stepping a parallelizable shortest path algorithm”.