Optimizing exact computation of Betweenness Centrality for CUDA

Aakriti Gupta
Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore, India
aakriti@cadl.iisc.ernet.in

***Abstract*—Betweenness centrality is an important metric in the study of network analysis. This report discusses the problem of exact computation of betweenness cenrality index in network analysis. BC is an important metric in small world network analysis which is expensive to compute. A new strategy is presented to parallelize the best known serial algorithm for computing BC on CUDA architecture exploring parallelism at all the different levels of granularity offered in the algorithm. Further optimizations are made into this strategy by exploiting CUDA specific notions of coalesced memory accesses, warping, shared memory etc.**

I. Introduction

Network analysis is currently an area of active research with applications ranging from social network analysis (friendship circles, organizational networks), phylogeny reconstruction and bio-informatics (protein interaction networks) to the Internet (web link analysis) etc.

One of the problems in network analysis is to determine how important a given node is relative to other nodes in the network. For example, in a social network, one is interested in finding how important a person is. Quantifying centrality of nodes is a well studied problem and several metrics have been proposed for the same. Betweenness centrality is an important metric in the study of network analysis. The philosophy behind this metric is: "An important node will lie on a high proportion of shortest paths between other nodes in the network."

Several algorithms for computing this metric exist in the literature. In this project I have tried to come up with an efficient CUDA implementation of the same.

II. Motivation

Betweenness centrality is applicable to many fields. Applications that use betweenness centrality as a building block include finding communities within a graph representing information flow, detecting communities in social networks, analyzing brain network and deploying detection devices in communication networks. Evaluating betweenness centrality for a graph G = (V, E) is computationally demanding step of various applications.

III. Problem Description

Betweenness centrality is a measure of a node's centrality in a network equal to the number of shortest paths from all vertices to all others that pass through that node. The betweenness centrality of a node v is given by the expression:

$$BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{1}$$

where σ_{st} is the total number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of those paths that pass through v.

Calculating the betweenness of all the vertices in a graph involves calculating the shortest paths between all pairs of vertices on the graph. This takes $\Theta(|V|^3)$ time with the FloydWarshall algorithm. It is assumed that graphs are undirected and connected with the allowance of loops and multiple edges.

The algorithm by Brandes [1] is considered the state of the art serial algorithm for computing betweenness centrality. Brandes algorithm for computing betweenness centrality in a graph is a key breakthrough beyond the naive cubic method that computes explicitly the shortest paths in a graph. It requires O(n+m) space and run in O(nm) and O(nm+n^2logn) time on unweighted and weighted networks,respectively.

A. Brandes Algorithm: Basis for parallel BC algorithms

Brandes algorithm calculates dependency of a source vertex s on any vertex v ∈ V as follows:

$$\delta_s(v) = \sum_{t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{2}$$

The betweenness centrality of a vertex v is then expressed using the following equation:

$$BC(v) = \sum_{s \neq v} \delta_s(v) \tag{3}$$

The key insight is that $\delta_s(v)$ satisfies the following recurrence, where $pred(s, w)$ is a list of immediate predecessors of w in the shortest paths from s to w.

$$\delta_s(v) = \sum_{w:v \in pred(s,w)} \frac{\sigma_{sv}(v)}{\sigma_{sw}}(1 + \sigma_s(w)) \tag{4}$$

Using this, Brandes algorithm works as follows: Each s ∈ V is considered as source of shortest paths and the contribution of s to $BC(v)$ for all v is computed in two phases. In the first phase, a shortest-path computation is performed from s, that computes $pred(s, v)$ and $\sigma_s(v)$ for all nodes. In the second phase, the predecessor list is traversed backwards, in non decreasing distance order with the help of Stacks and

Queues. And for each v $\in$ V, $\delta_s(v)$ is computed based on equation 4. The contribution to BC vector is computed using equation 3.

B. Parallelism in Brandes algorithm

The Brandes algorithm has parallelism at multiple levels.

1) Coarse grained parallelism: We can process multiple source nodes in parallel. In this parallelization strategy, each thread picks an arbitrary graph node and computes its contribution to the betweenness values of other nodes. Each of these computations is independent.
2) Medium grained parallelism: We can do parallel processing of all vertices that are in same frontier while expanding any given source vertex in the outer coarse grained loop.
3) Fine grained parallelism: We can explore the neighbors of each vertex in the same frontier in parallel while building up queue for a given outer coarse grained iteration.

The coarse grained parallelization strategy is simple and effective, but each outer loop iteration that is performed in parallel requires its own storage, so the space overhead of this scheme is substantial. The medium and fine grained parallelism approach is more space-efficient since we only need to maintain a single graph instance, but poses a more challenging goal for parallelization due to non-trivial data dependencies and requirement of atomic updates. The updates on each $BC(v)$ has to be atomic and can be done using using either atomicAdd or by simple reduction operation.

IV. Related Work

In the paper [2] Bader et al. came up with the first parallel implementations of some widely-used social network centrality analysis metrics including betweeness centrality. They have shown scalable performance for up to 40 processors.

In the following paper [3] by Bader et al. lock free parallel algorithm is developed for CRAY XMT system.

The common approach that has been explored in the literature targets medium and fine grained parallelism and focuses on a distributed memory environment like [4]. Fast and scalable parallel algorithms have been proposed but they have mostly targeted architectures like the CRAY MTA-2.

In a conference poster, [5] Pande et al. have discussed one approach towards parallelizing this for GPUs. They have explored the fine and medium grained parallelism on GPU. I have developed a different strategy which exploits parallelism at all the 3 levels of granularity.

Due to the expensive computation of BC for large graphs, the trend in the literature is to approximate the computation rather than computing exact betweenness centrality for nodes as discussed in [6] or incrementally building from smaller graphs as discussed in [7].

In a recent paper [8] authors have modified Brandes algorithm to by applying some heuristics and have explored parallelism on this new version. Exploring this for CUDA can be one of the things in the future work.

V. Methodology

A. Early Experiments

Initial experiments were done by exploring the fine and medium grained parallelism by running the main outer loop on CPU and within each iteration making small kernel calls to parallelize the small inner loops. This approach failed because:

1) Not very scalable
2) Too many tiny kernel calls instead of one huge call turns out to be expensive
3) Over head of a kernel call is much more than is being compensated
4) Starts to show in results

Then parallelizing of the bigger loop was explored. This is not straightforward because of the huge memory requirements it imposes. If we naively try to give one thread for one independent iteration of the outer loop, there's too much work to be done by one thread. And also, there's not enough memory. Instead I thought of a hierarchical approach to address all three levels of parallelism.

B. Building hierachical strategy

The basic Brandes algorithm [1] is shown in figure 1. The strategy developed to parallelize this problem is breifly described as follows:

1) For relatively small graphs, with n number of nodes, n blocks are created each consisting of n cuda threads. Since there are n iterations of the outer loop and within each such iteration we can explore at most n-way parallelism. Threads within a block cooperate for computing betweenness centrality contributions from a given source vertex. The source vertex for a given block is the block ID.
2) For larger graphs, CPU-GPU overlap is exploited wherein few iterations of the outer loop run on CPU simultaneously with rest of the iterations running on GPU. Results are accumulated on CPU. This is fairly simple approach, the only tricky part is to decide how many nodes should be computed on CPU and how many on GPU. I have not come up with a uniform solution that fits all graphs alike, instead I have experimented and converged on a solution iteratively.
3) Coarse grained parallelism is addressed with the help of thread blocks. Each thread block represents one iteration of the outer loop.
4) Medium and fine grained parallelism are explored by the threads within a block. They collaborate on initializing the various arrays, exploring the neighbors and frontier vertices simultaneously. Basically each thread is assigned a node or a fixed number of nodes depending upon the graph size in the problem.
5) Tapping into the medium and fine grained parallelism is not straightforward like the outer coarse grained loop. Although the tasks are fairly independent of each other they are many race conditions that arise if we are not careful. For example, while exploring neighbors of a frontier node

Algorithm 1: Betweenness centrality in unweighted graphs

```
C_B[v] ← 0, v ∈ V;
for s ∈ V do
    S ← empty stack;
    P[w] ← empty list, w ∈ V;
    σ[t] ← 0, t ∈ V;   σ[s] ← 1;
    d[t] ← −1, t ∈ V;   d[s] ← 0;
    Q ← empty queue;
    enqueue s → Q;
    while Q not empty do
        dequeue v ← Q;
        push v → S;
        foreach neighbor w of v do
            // w found for the first time?
            if d[w] < 0 then
                enqueue w → Q;
                d[w] ← d[v] + 1;
            end
            // shortest path to w via v?
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v];
                append v → P[w];
            end
        end
    end
    δ[v] ← 0, v ∈ V;
    // S returns vertices in order of non-increasing distance from s
    while S not empty do
        pop w ← S;
        for v ∈ P[w] do δ[v] ← δ[v] + (σ[v]/σ[w]) · (1 + δ[w]);
        if w ≠ s then C_B[w] ← C_B[w] + δ[w];
    end
end
```

Fig. 1. Brandes Algorithm

in parallel, each thread independently decides whether the particular neighbor it is dealing with fits the criteria and hence should be enqueued or not. Since threads in a block share queue which is implemented using shared memory, updates into the queue have to be atomic. For this I tried two approaches, first was to use syncthreads function call to ensure the updates are done atomically. This is analogous to AtomicAdd approach. But this resulted in poor speedup. Next approach was to let each thread set a flag to indicate this needs to be enqueued. At the end of the iteration, one thread enqueues them all avoiding any race conditions.

6) Within a block, threads cooperate using shared memory.
7) Implemention of stack and queue operations are done through device functions. They have the property that CUDA converts any device function call into inline function call.
8) Finally, to ensure atomic updates into the BC vector, kernel is split into two. First kernel computes the BC vector on a per block basis and then stores the result in the persistent global memory. Second kernel does a reduction operation on all the BC vectors, adds them up to generate one final BC vector. This is added to the computations done by CPU and the final result is stored.

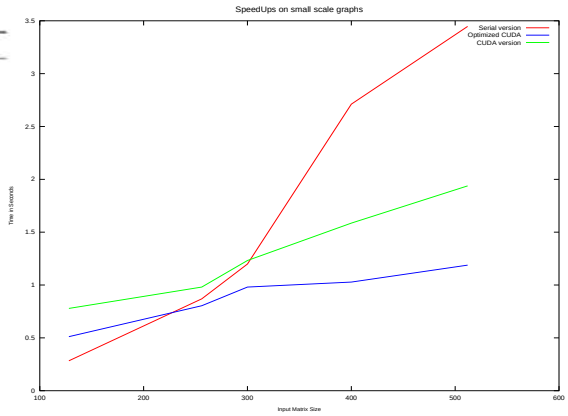

Fig. 2. Speed Up small graphs

9) Further optimizations were made by making maximum use of shared memory and making coalesced memory accesses. For example, accesses made into a 2D array residing in global memory is accessed column wise by every thread. So in one memory access, all the threads in a warp end up getting their required elements. As opposed to accessing the array row wise which will cause each thread to make a separate memory access into the expensive, high latency global memory. Similary, optimum use of the limited shared memory is made by storing intermediate, repeatedly used results into the shared memory. Example, stack and queues and partial dependency arrays are stored in shared memory.

VI. Experiments and Results

A. Experiment Setup

The versions were tested for random synthetic graphs of variable sparsity for different number of nodes on Tesla architecture. The results are average of at least 5 independent runs of the code.

B. Results

The optimized CUDA version fares almost 2 times better than unoptimized CUDA code. Overall for small scale graphs the speed up is promising (Fig. 2). This is shown for graph sizes 128-600. For large scale graphs, CPU-GPU version is executed (Fig. 3) this is shown for graph sizes 600-1000. In this case speed up is not that good. This is because of limited shared memory, which causes irregular data accesses into the global memory which hampers the speedup. Shared memory latency is a few 100 times lower than global memory hence we see the difference.

C. Observations

Although showing massive scope of parallelization, betweenness centrality computation by parallelizing Brandes algorithm have certain characteristics that limit the scope of the parallelization. Such as:

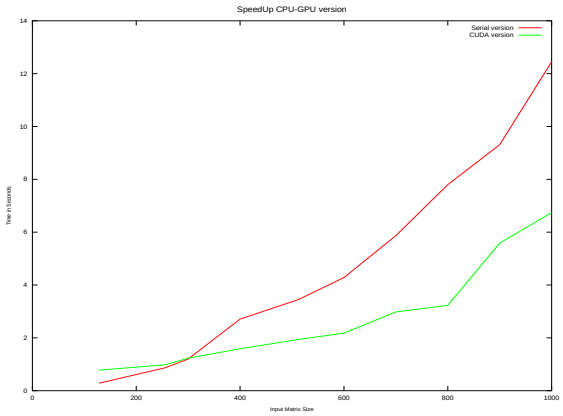

Fig. 3. Speed Up

1) Memory intensive: Large memory footprint, and a significant number of non contiguous memory accesses to global data structures.
2) Low arithmetic intensity: Betweenness centrality doesn't have any computations. Memory latency is visible and doesn't get shadowed by computations.
3) Multiple parallel traversals need more memory as we need to store more copies of the intermediate arrays and lists. This limits the amount of parallelism that can be exploited.

VII. Conclusions

Parallel computation of betweenness centrality can be exploited at different levels of granularity:

1) Coarse grained: Iterations from each source vertex s can be done in parallel
2) Medium grained: Parallel processing of all vertices that are in same frontier
3) Fine grained: Exploring neighbors of each vertex in parallel

Here all three have been exploited using CUDA for small scale graphs. For largescale graphs, exact computation of BC is not computationally viable.

There is load imbalance among CUDA threads, since the algorithm is basically an extension of BFS wherein each thread takes up a vertex and checks if it belongs to the frontier or not. It is a little better since it may be the case thread will be checking conditions for more than one vertex but in a random graph, this still can cause load imbalance. Load is dependent on the inherent graph structure.

VIII. Future Work

1) Addressing the load balancing problem. We can look into preprocessing the graph and then dividing the graph nodes to different threads optimally.
2) Optimizing the computation of betweenness centrality for larger graphs on GPU.
3) Improving the CPU-GPU version of the implementation.

References

[1] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, pp. 163–177, 2001.

[2] D. A. Bader and K. Madduri, "Parallel algorithms for evaluating centrality indices in real-world networks," in *Proceedings of the 2006 International Conference on Parallel Processing*, ser. ICPP '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 539–550. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2006.57

[3] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarra-miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," Tech. Rep., 2009.

[4] N. Edmonds, T. Hoefler, and A. Lumsdaine, "A space-efficient parallel algorithm for computing betweenness centrality in distributed memory," in *International Conference on High Performance Computing*, Goa, India, 12/2010 2010, to appear.

[5] D. B. P. Pande, "Computing betweenness centrality for small world networks on a gpu," Lexington, Massachusetts, Tech. Rep., 2011.

[6] K. Jiang, D. Ediger, and D. Bader, "Generalizing k-betweenness centrality using short paths and a parallel multithreaded implementation," in *Parallel Processing, 2009. ICPP '09. International Conference on*, 2009, pp. 542–549.

[7] O. Green, R. McColl, and D. Bader, "A fast algorithm for streaming betweenness centrality," in *Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Confernece on Social Computing (SocialCom)*, 2012, pp. 11–20.

[8] D. Prountzos and K. Pingali, "Betweenness centrality: algorithms and implementations," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 35–46. [Online]. Available: http://doi.acm.org/10.1145/2442516.2442521