Task Parallelism in GPU libraries for Optimizing Betweenness Centrality

Bharath Shamasundar, Shubham Pandey
Department of Computer Science and Engineering, Missouri University of Science and Technology
Rolla, MO
Email: bbst59@mst.edu, spdgf@mst.edu

Abstract—**Betweenness centrality is a parameter which helps us to find the centrality of a particular node in a graph. It is measured by finding out how many times a vertex appears in the shortest paths of other vertex pairs. This parameter is widely used in networks to find the importance of nodes in the graph and is a better measure of centrality than closeness centrality. However, the high computational cost of these analyses as well as it's complexity prevents computation of betweenness centrality in large graphs. Furthermore, the changing nature of the networks invalidate the values calculated previously and thus leads to new analyses being done to find the variation of centrality metrics over with time. Although there have been optimized versions of Betweenness Centrality on the single core machines, as well as multi-core machines including various GPU implementations,the difficulty of programming GPUs, along with the data access irregularity and dynamic control flow, have presented significant challenges to developing an optimized version of this calculation on the GPU. One of the main issues while programming on the GPU is parallelizing tasks that happen in succession asynchronously.In this paper we try to address the issue of task parallelism on the GPU and present a high level abstraction on how to efficiently model the programming structure to optimize betweenness centrality calculation on GPUs with efficient load balancing and workload mapping for small world graphs.**

I. INTRODUCTION

With the dawn of the information era and the ever-growing need for new forms of data structures to handle data, graphs have always been the go-to salvager when dealing with huge networks of data. The simple yet effective portrayal of graphs and the ease with which one can maneuver them has made them widely conventional. Graph algorithms have become important problems to research in the era of multi-core and many-core computing. Because of their irregular memory access pattern, graph algorithms need special addressing for optimizing its performance on parallel architectures. Graph Algorithms infuse computer science, and Algorithms for working with them are fundamental to the field. Hundreds of interesting computational problems are represented in terms of graphs. We try to address one of the major issues that dampens effective utilization of the GPU cores on one the best performing GPU library[13] and how effective task parallelism modeling can lead to the betterment of such libraries. We use Betweenness Centrality as the algorithm for which the library will operate on,as leveraging task parallelism in Betweenness Centrality is an optimization problem where we have to minimize the running time.Betweenness Centrality is a popular graph analytic metric that has found application in social networks by figuring out the most followed person, studying protein-protein interactions, analyzing power grids along with significant applications in community detection, analyzing the contingency of power grid, and the study of the human brain. Section 2 deals with how graphs are represented on the multi-core platforms namely the Central Processing Unit and the Graphics Processing Unit including a brief overview of the CUDA programming model.Section 3 deals with the related work. Section 4 deals with Gunrock and it's programming model.Section 5 deals with the results which show why there is necessity for an optimization while doing a Full Pass Betweenness Centrality. In Section 6 we give an high overview of our algorithm which incorporates task parallelism with [13].The paper ends with Conclusion and Future which is in Section 7.

II. GRAPHS ON MULTI-CORE PLATFORMS

Many of the modern applications today that are making use of graphs rely on the fact that it is able to handle huge amounts of edges and vertices. While there are implementations of sequential algorithms with improved speedup, they cannot suffice to handle large amounts of data. In order to overcome these limitations there have been several attempts made to implement the parallel versions of these graph algorithms, which yield positive results but at a high hardware cost. With the advent of the GPU for data intensive computation massive parallelism is possible at relative low costs. Problem solving on the Graphical Processing Unit is highly potent and a low cost substitute to the modern multi core units. General Purpose Programming on the GPU to solve a problem is nothing but a graphics rendering problem as a result of which the range of solutions that can be ported onto the GPU is stunted. There are several problem domains that have gained significant speed up when ported onto the GPGPU model. Analysis of Graphs on the GPU has immense potential to exploit the architecture of the GPUs for constructive graph computation.

A. CUDA Programming Model

CUDA is a programming interface which is used to utilize GPU for general purpose computing. The programing can be done as an extension to the C programming language. The CUDA device is seen as a multi-core co-processor by the CPU. The memory restrictions present in GPGPU is not present in CUDA due to its design properties. Although the time to access the memory changes for different types of available

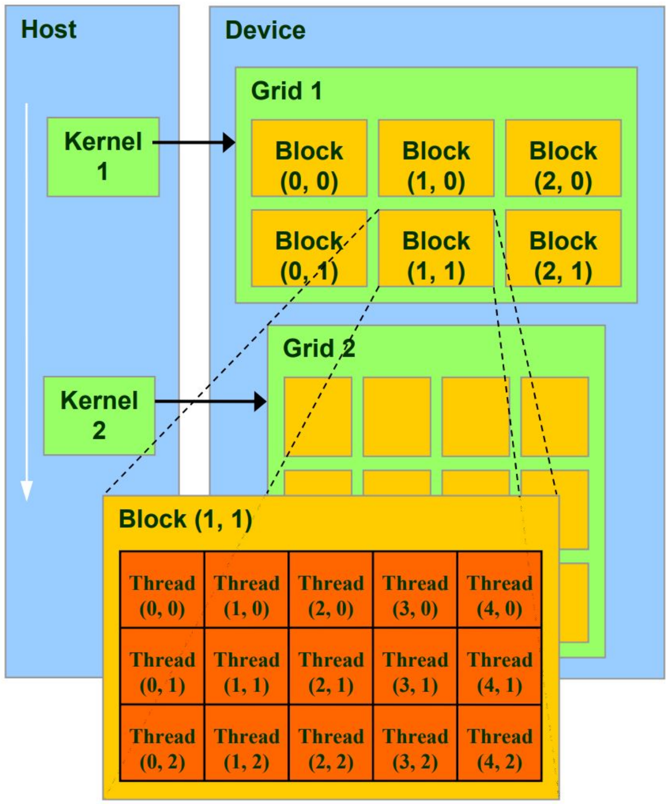

Figure 1. CUDA Programming Model

memory, the memory available on a particular device can be completely accessed by using CUDA with no limitation on its representation.

The hardware model of CUDA consists of multiprocessor having its own shared memory. This model performs the same instructions along a range of data at any particular time which makes it an SIMD processor. All the processors can access the device memory at any given instance which helps in making the communication among different processor feasible. Figure 1 shows the hardware interface of CUDA. The programming model of CUDA can be abstractly seen as a parallel computation of a task in a collection of threads.

A collection of threads which can run at the same time on a multiprocessor are called Warps. The size of a warp is determined by the programmer and remains constant during the entire process. Blocks, which can be seen as a collection of threads, runs on a multiprocessor simultaneously. Multiple blocks can be assigned to a single multiprocessor and their execution is time-shared. A unique ID is provided to every thread and block which helps to access them during the entire execution period. Each thread is assigned a core code for execution which is called the kernel. The kernel task can be performed by each thread on different data set by making use of thread and block IDs. Moreover, any memory location can be accessed at any particular instance as the device memory is available to all the threads.

B. Graph Representation on the CPU

Graphs can be represented in a variety of ways with every representation having it's own advantages and disadvantages. There are basically three criteria upon which a representation must be chosen and they are as follows:

- Memory required to accommodate the graph.
- Time required to determine whether an edge is present in the graph.
- Time required to find the neighbors of a vertex.

A graph can be represented by an edge list which as the name suggests consists of a list or array of E edges. Each edge is represented by storing the vertex ID's of the vertices that are incident to the edge in question. These values are stored as an array of objects and since most of the edges have just two or three numbers, the space complexity for an edge list is (E). For a graph with V vertices an adjacency matrix of V*V matrix of 0s and 1s where there is an corresponding 1 for row i and column j if there is an edge for the vertices (i, j) .With the help of an adjacency matrix it is easy to spot an edge in a graph by just looking at the matrix for an entry of 1 for the vertices (i,j).

There are two major disadvantages with respect to this storage, firstly it takes up a space of (V2), even if the graph is sparse, that is the adjacency matrix has majority of 0s and there is lots of space used to represent only a few edges. Secondly to find a neighbor of a particular vertex i there is a need to look at V entries in row i even though there are only a few vertices adjacent to vertex i. Yet another way to represent graphs is the adjacency list which is nothing but the combination of adjacency matrix and edge lists. For every vertex i there is an array of vertices stored for every vertex that is adjacent to the vertex i. To find out if there is an edge (i, j) present we just have to traverse the corresponding lists of the vertices to find the vertex i and j. In the worst case scenario it takes (d) for the adjacency list where d is the degree of the vertices.

C. Graph Representations on the GPU

For computation be carried out on the GPU, the most basic challenge will be to maintain the bandwidth of the data, which is pretty high, to the GPUs memory and inherently utilizing the parallelism provided by the GPU compute units. The key here would be to access large chunks of data (coalesced access) and maximizing the ability of threads within warps to maintain the same control path through the code. Since most of the graph algorithms will run in series with some other algorithms the transfer cost to and from the CPU need not be taken into consideration for the running time. There is no point in having graph elements on the GPU if its performance is not remarkable when compared against top-tier CPU nodes.

The performance of graph algorithms on the GPU also depends on the datasets being used (i.e. dynamic or static graphs). Other factors to be noted are how the GPU will perform on graphs which takes more space than the GPU memory and how well does it scale beyond a single GPU. A graph G (V, E) is usually represented as an adjacency matrix. But for sparse graphs the problem is too much of space is taken up. As a result of this it is represented in a compact adjacency list. Each vertex points to its own array in the adjacency list (shown in Figure 2). There are two sets of arrays respectively one each for the vertices (V) and the edges (E). Each entry in the edge array refers to the vertex array. Another way to store the graph in the adjacency list format is the Compact Sparse Row (CSR) matrix format. During the various phases of graph processing, these representations helps us to restructure the sparse and uneven workload in a uniform and dense ones by making use of parallel primitives like prefix sum. These prefix sum is used by parallel threads to assemble a global edge set from the expanded neighbors and to store the unique vertices which are not yet visited into global vertex set in a parallel BFS computation.

The other representation is the edge list wherein threads are assigned to edges represented in the graph. Instead of using two uneven length of arrays as used in the CSR format two arrays of equal length are used to store the vertex pair for each edge. Since this is edge centric representation it will require more GPU memory when scaled onto large graphs. Blelloch proposes the v-graph (vector graph) for graph data representation. In this representation, segmented vectors are used to store the topology of the graph. A single segmented vector is used to the information about any edge in an undirected graph. Each vertex has a corresponding vector and each edge incident to the vertex corresponds to an element within a segment. However, for directed graphs, an incoming edge will have a different segmented vector than the outgoing edge. Each element in the outgoing edges vector is a pointer to a position in the incoming edges vector. Other informations such as weights are stored in additional vector in directed as well as undirected graph.

III. Related Work

In [1] the authors worked to generalize the geodesic centrality measures proposed by Freemans to find the betweenness in an undirected graphs to a directed graph. The authors have achieved this in four steps. In the first step, the generalization of point centrality measure is performed. In the second step, the authors have defined a unique maximally centralized graph for a directed graph while keeping constant the number of points having reciproctable arc as compared to unreciproctable arcs. The focus is then shifted to find the value of the maximally central arrangement of these arcs under the constraints defined. Normalization on the number of arcs can be performed as an alternative. This leads to the third step in which the authors define the relative betweenness centrality of a point regardless of the total number of the points in the graph. According to Gould, the centrality measure in a directed graph is not interpretable as they lack maximality standard. This objection is removed with the help of the normalization step. Convergence of Freeman's measure of betweenness in an undirected graph having no isolation with the relative directed centrality is achieved. In the last step the authors define the dominance of the most central node using the measures of this concept of graph centralization.

Bader et.al [2] used small-world networks having massive size to present a new algorithm which executes in a parallel lock-free way to compute the betweenness centrality. Their algorithm also achieves better spatial cache locality compared to previous approaches by making small changes in the way the graph data is structured. In fact, Bader et.al [3] were the first ones in implementing the parallel versions for centrality based algorithms used in social analytics. In their results they were able to perform a rigorous analysis of networks which were larger by the order of three magnitude than the graphs which can be handled by the current network analysis (SNA) software packages. In their implementation, they use the CRAY-MTA 2 whose memory bandwidth is very high and the memory access is uniform. They also use a use a cache-friendly adjacency array representation. By altering the Degree Centrality, Closeness Centrality and the Stress for Betweenness Centrality according to the CRAY computers architecture they observe high-performance gains. The the authors of [4] utilized the best known sequential algorithm to present a new algorithm which can perform in a parallel way and having low spatial complexity. Their storage complexity of their algorithm is O (V +E) while it also enables parallel execution in an efficient manner. Also, the algorithm can be implemented using the coarse-grained parallelism which makes it well suited for processes in distributed memory.

Bader et.al in [5] performed experiments to check the performance of a graph algorithm. The authors choose small world networks to compute the betweenness centrality of the vertices by making use of 2 NVIDIA Tesla and Fermi GPUs and performed a comparison compared with a parallel open source implementation which made use of an Intel multi-core CPU. They observed parallelism at three different level of granularity by executing the betweenness centrality algorithm. At the coarse grain level, the authors found parallel execution for each source vertex during every iteration. The vertices in the same frontier can be processed in parallel which exploits the medium grain parallelism. The neighbors of each vertex can be processed in a parallel way which helps to make use of the fine grain parallelism. Due to this, the abstraction method of CUDA groups the hierarchy of threads into blocks and grids which helps to exploit the parallelism at the medium and fine grain level as different blocks in a grid are assigned vertices present in the same frontier. The neighbors of vertices are then processed in parallel by each thread block which are delegated to it. The performance and scalability were limited in the previous implementations due to inefficient graph traversals and large data structures that are stored locally. Bader et.al [6] provided a work-efficient algorithm to calculate the betweenness centrality on GPU. The algorithm performs especially well for graphs having a large diameter. Two different algorithms that are alternating in nature are proposed which utilizes either the memory bandwidth of the GPU or the asymptotic efficiency of the work on the basis of the graph structure. The decision of the first method is based on how significant is the change of length of the frontier during various iterations while the second method utilizes a small amount of work performed initially by the algorithm which helps in finding the best method of parallelism to process the assigned work.

Our main focus for GPU comparison is based on [6] where their hybrid approach differs from previous implementations in that they discard the predecessor array needed for the calculation of the betweenness score. This change helps in reducing the space complexity of the local data structures from O(m) to O(n). Additional computation cost is incurred due to the removal of space but the overall computational complexity of the algorithm is not changed. The neighbors of a vertex are traversed instead of direct traversal of the predecessors during the dependency accumulation stage. The shortest path calculation is also optimized by eliminating the read-write discrepancy by using the atomic operations. This leads to the dependency accumulation which they solve by giving the work-efficient dependency algorithm. They checked the successor of every vertex instead of the predecessors which helped them to eliminate the use of atomics. Since vertices at the end of the BFS tree by definition have no successors, the dependency accumulation starts one level closer to the root of the treeIn recent publications, there has been significant importance given to Data Structures in implementing these algorithms [7][8].

IV. GUNROCK

In the previous section we discussed some of the work that has been done with low-level optimizations on the GPU. Often such optimization require complex and in-depth knowledge of handling each and every data-structure mapped to the GPU.High Level libraries mask this and focus more on the operations being performed rather than how the operation is performed. Gunrock[13] is one such high-level programming library that makes use of High performance GPU computing primitives along with a High Performance framework. The main focus of this library was to optimize some of the basic graph algorithms like Breadth First Search(BFS), Single Source Shortest Path(SSSP) and Betweenness Centrality(BC). The superiority of this library comes from the fact that there is a data-centric abstraction present in the library that is specifically designed for the GPU. Along with this there is the added benefit of flexible programmability which allows simple and flexible interface to allow user-defined operations. The framework and optimization details are hidden from the user but it is automatically applied when there is need for a particular operation. There are four main primitive operations in Gunrock and they are:

1) Advance
2) Filter
3) Compute
4) Segmented Intersection

In Gunrock every step in an algorithm operates on only those nodes that are present in the Frontier set. The advance operation in Gunrock puts vertices in this frontier set by traversing vertices at each level of the Graph. Once operations on a set of vertices are performed, the next frontier set is calculated by making use of the advance operation. The next operation that Gunrock performs is the filter operation. Once the frontier set is calculated, it is essential to evaluate as to which vertices in this set is the most compute intensive and those vertices must be handled first leaving out the other vertices in the set to be handled later. This is done with the help of the filter operation which ignores those vertices mostly having low edge degree. The segmented intersection operation filters those nodes that are common when multiple frontier vertices are computed. This is common when neighbors are being visited in Bread First Search or Single Source Shortest Path traversal.The compute operation is the main operation in Gunrock which computes on each of the vertices present in the frontier set. These operations can either be computing the shortest distance, calculation scores for betweenness centrality or just distance scores for each level of the graph.

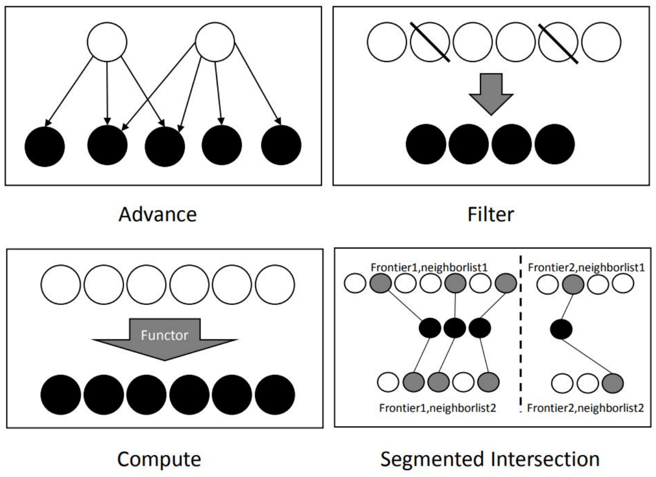

Figure 2. Gunrock Primitive Operations

V. RESULTS

In this section we speak about the initial results to show that task parallelism is indeed needed for better optimization of Betweennness Centrality on the Gunrock library. We have conducted two experiments for Betweenness Centrality, one being the Full Pass BC, where the BC score is calculated for each and every node present in the graph, and the second is the Single Pass BC where the score is calculated for only a single node and not the entire graph. We compare the sequential implementation of Betweenness Centrality which utilizes the boost library with Gunrock implementation of BC for a single pass BC whereas we do a three-way comparison involving the sequential implementation of Betweenness Centrality which utilizes the boost library, Gunrock which implements Betweenness Centrality and Adam et.al [6] implementation of GPU optimized version of BC. All our implementations have been performed on the Amazon EC2 cloud infrastructure and utilized the Intel i7 microprocessor having 8 cores in total for the sequential implementation of BC. We make use of the g++ compiler with -03 optimization for the sequential version.For Adam et.al's [6] implementation as well as the Gunrock implementation we utilize the Tesla K80 GPU with compute capability 3.7 using CUDA 9.0 along with the nvcc compiler. The dataset used for the full and single pass BC is given in Table 1 and Table 2 respectively. All our graphs have been taken from the 10th DIMACS challenge[16]. The

Table I
FULL PASS BC DATASET

Dataset	No. of Vertices(thousands)	No.of Edges(thousands)
Luxemberg-osm	114	119
CaidaRouterLevel	119	600
Rgg-n15	32	160
Delaunay-n16	65	190

Table II
SINGLE PASS BC DATASET

Dataset	No. of Vertices(million)	No. of Edges(million)
RoadNet-CA	1.9	5.5
Delaunay-n21	2	12.5
Webbase-1M	1	6.2
Belgium-osm	1.4	3

graphs for the single and full pass BC are present in Figure 3 and 4 respectively. In Figure 3 which is the single pass

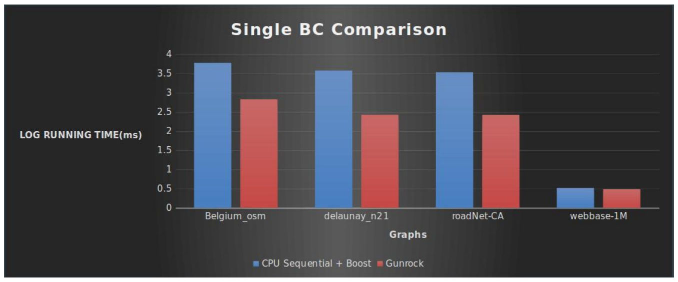

Figure 3. Single Pass BC comparison

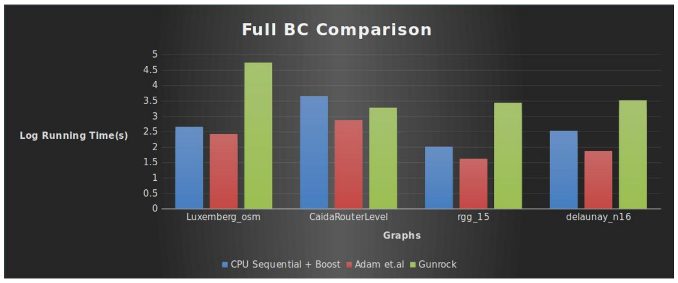

Figure 4. Full Pass BC comparison

BC comparison we can see that Gunrock performs extremely well compared with the CPU version of BC which has been enhanced with the boost library. All of the datasets except the webbase-1M graph give an average speedup of 2x with gunrock which goes to show that fine-grained granularity is not necessary for scale up when computing BC scores. In Figure 4 we have the full pass BC comparison of sequential BC with boost, Gunrock and Adam et.al's hybrid low-level implementation. Gunrock performs extremely poorly when the scenario is a full pass BC. This is because Gunrock although scales extremely well for a single pass where you need to compute the BC score for just a single node, it performs sequential tasks for computing a full pass BC. This means that when it shifts the computation from one node to the other in computing BC scores there is no task parallelism involved where it could compute multiple node BC scores in parallel. This results in it's performance going down for all the datasets shown in Figure 4. In some cases it performs even worse than the sequential version using boost.

VI. PROPOSED ALGORITHM AND COMPLEXITY ANALYSIS

In this section we give our proposed algorithm that involves task parallelism. We make use of the four main primitive operations that are involved in Gunrock but with the added flexibility of tasks being performed in parallel. There are three main parts to the algorithm namely Optimized frontier search, Shortest path calculation using the compute operation and the final stage is betweenness score calculation which involves dependency calculation of each node present in the graph with the help of the advance operator.

Algorithm 1 Optimized Frontier Search

```
procedure VISIT VERTICES($G(V), G(E)$)
    while $G(V) \neq empty$ do in parallel
        Allow multiple edge visits for the same node
        Select high ratio of undiscovered vertices
        $Frontier_Queue \leftarrow G(V)$
        Place high priority vertices at the start of the Queue
    return $Queue$
```

Algorithm 2 SSSP using Compute operation

```
procedure SHORTEST PATH($Frontier_Queue$)
    while $Queue \neq empty$ do in parallel
        v = Pop from Queue(has high priority)
        Use Atomic Operation to avoid Read-Write Conflict
        $SS_Queue \leftarrow Shortest_Path$(v)
    return $SS_Queue$
```

Algorithm 3 Dependency Calculation with Advance operation

```
procedure BC SCORE($SS_Queue, Frontier_Queue$)
    while $Frontier_Queue \neq empty$ do in parallel
        Calculate dependency(Front-Queue)
        BCScore(v)=Sum(Dependency(v)/Total(ShortestPath(v))
    return $BC_Score$
```

In the first stage of the algorithm we perform the optimized frontier search to place those nodes that we are computing into the frontier queue from which high importance nodes are operated on. In Gunrock this is done through the advance step, but the important thing here is to perform multiple advances such that there is more than one frontier queue we are operating on.Also we use the filter operation here to find those nodes that require major computation resources. In the second stage we compute the shortest path for each of the nodes present in the frontier queue. The main difference here is previously Gunrock did not allow these two stages to be done in parallel, i.e. the frontier search and the shortest path calculation. We are proposing in our algorithm that by doing these two tasks in parallel it can speed up the process of Full Pass BC calculation as previously there was no task parallelism present in Gunrock. Within each of these stages we make use of atomic operation to handle read-write conflict.Once these two stages are completed then we perform the third stage which is the betweenness score calculation for each of the nodes. In order to perform the third stage it is necessary that the first two stages are at a complete halt. This is done on the GPU with the help of barrier synchronization available on the GPU. The dependency calculation in step three is done with the successor look ahead strategy that is present in [6]. Taking inspiration from their approach we also apply this strategy where rather than calculating dependencies looking backward we do this by looking ahead for untouched vertices. Pictorially our algorithm is as shown in Figure 5.

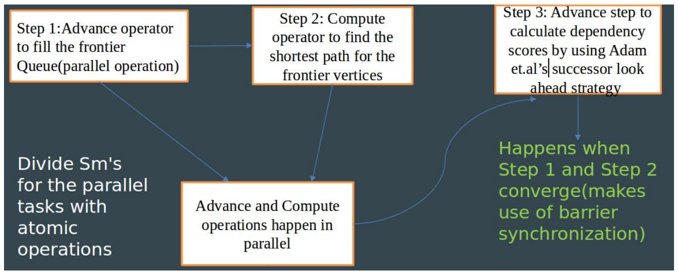
Step 1:Advance operator to fill the frontier Queue(parallel operation)
Step 2: Compute operator to find the shortest path for the frontier vertices
Step 3: Advance step to calculate dependency scores by using Adam et.al's successor look ahead strategy
Divide Sm's for the parallel tasks with atomic operations
Advance and Compute operations happen in parallel
Happens when Step 1 and Step 2 converge(makes use of barrier synchronization)

Figure 5. Pictorial Representation of the Algorithm

The complexity that is involved in calculating BC in the hybrid approach is O(n*m), where n is the number of vertices and m is the number of edges present in the graph.For our approach the two main operations that happen in parallel are the advance and compute operations in step 1 and 2. Since all our graphs have varying degrees present in them, the number of edges is certainly greater than the number of nodes in the graph.(m¿n). Since traversing the graph at each level and in parallel finding the shortest path is the most compute intensive stage, our algorithm will also takes an average of O(n*m) running time. The added advantage of our algorithm is that we do not need to worry about the low level details of managing data structures like in the hybrid approach of [6]

Our load balancing approach which involves dividing the parallel task into the appropriate Streaming Multi processors on the GPU as well as performing atomic operations within each of these stages is as shown in Figure 6.

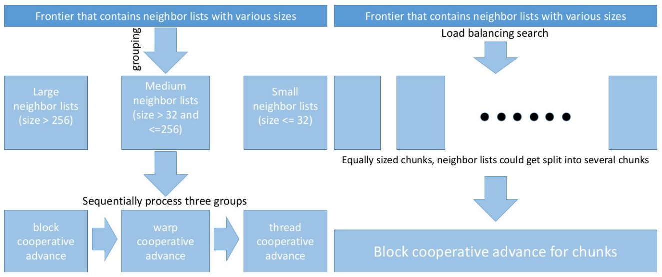
Frontier that contains neighbor lists with various sizes
Grouping
Large neighbor lists (size > 256)
Medium neighbor lists (size > 32 and <=256)
Small neighbor lists (size <= 32)
Sequentially process three groups
block cooperative advance
warp cooperative advance
thread cooperative advance
Frontier that contains neighbor lists with various sizes
Load balancing search
Equally sized chunks, neighbor lists could get split into several chunks
Block cooperative advance for chunks

Figure 6. Load Balancing Model on the GPU for our approach

VII. Conclusion and Future Work

In this paper we try to incorporate task parallelism and efficient load balancing for betweenness centrality calculation for Gunrock which is the best performing GPU library till date for graph computation. Our procedure is based on efficient workload mapping involving parallel tasks. Although we do not have implementations our algorithm we believe parallel task mapping can be a major help in improving the efficiency of the library. As part of future work we intend to move towards the implementation of our algorithm on the GPU. Although there are better approaches for finding betweenness centrality by exploiting structural properties in community graphs [14], in which the algorithm proposed performs better than Brandes algorithm [15] which is the basis of the hybrid GPU approach, there has been no implementation of it on the GPU and could be an interesting work for the future.

VIII. Acknowledgement

We would like to thank Professor Sajal Das for his valuable help and guidance. Also we would like to thank Professor John Owens and his team from UC Davis for providing suitable inputs for Gunrock.

References

[1] 1. White, D.R., Borgatti, S.P., 1994. Betweenness centrality measures for directed graphs. Social Networks 16, 335346

[2] Madduri, K., Ediger, D., Jiang, K., Bader, D. A., Chavarria-Miranda, D. (2009, May). A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In Parallel and Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on (pp. 1-8).IEEE.

[3] Bader, D.A., Madduri, K.: Parallel algorithms for evaluating centrality indices in real-world networks. In: ICPP 2006. Proceedings of the 2006 International Conference on Parallel Processing, pp. 539550. IEEE Computer Society Press, Los Alamitos (2006).

[4] N. Edmonds, T. Hoefler, and A. Lumsdaine, A space efficient parallel algorithm for computing betweenness centrality in distributed memory, in High Performance Computing (HiPC), 2010 International Conference on, dec. 2010, pp. 1 10.

[5] P. Pande and D. A. Bader. Computing betweenness centrality for small world networks on a GPU. In 15th Annual High Performance Embedded Computing Workshop (HPEC), 2011.

[6] A. McLaughlin and D.A. Bader, Scalable and High Performance Betweenness Centrality on the GPU, The 26th IEEE and ACM Supercomputing Conference (SC14), New Orleans, LA, November 16-21, 2014.

[7] Fan, Rui, Ke Xu, and Jichang Zhao. ”A GPU-Based Solution to Fast Calculation of Betweenness Centrality on Large Weighted Networks.” arXiv preprint arXiv:1701.05975 (2017).

[8] ariyce, Ahmet Erdem, et al. ”Graph manipulations for fast centrality computation.” ACM Transactions on Knowledge Discovery from Data (TKDD) 11.3 (2017): 26.

[9] Beamer, Scott, Krste Asanovi, and David Patterson. ”The GAP benchmark suite.” arXiv preprint arXiv:1508.03619 (2015).

[10] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 12). 117128

[11] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. 2014. Work-Efficient Parallel GPU Methods for Single Source Shortest Paths. In Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2014). 349359.

[12] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge, ser. Contemporary Mathematics, vol. 588, 2013.

[13] Wang, Yangzihao, et al. ”Gunrock: A high-performance graph processing library on the GPU.” Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, 2016.

[14] Das, Sima, and Sajal K. Das. ”Leveraging network structure in centrality evaluation of large scale networks.” Local Computer Networks (LCN), 2015 IEEE 40th Conference on. IEEE, 2015.

[15] Brandes, Ulrik. ”A faster algorithm for betweenness centrality.” Journal of mathematical sociology 25.2 (2001): 163-177.

[16] David A. Bader, Henning Meyerhenke, Peter Sanders, Dorothea Wagner (eds.): Graph Partitioning and Graph Clustering. 10th DIMACS Implementation Challenge Workshop. February 13-14, 2012. Georgia Institute of Technology, Atlanta, GA. Contemporary Mathematics 588. American Mathematical Society and Center for Discrete Mathematics and Theoretical Computer Science, 2013.