Measure Computation and Patterns Detection in Social Networks with GPU Acceleration

Gonçalo Sousa
Instituto Superior Técnico
Email: goncalo.sousa@tecnico.ulisboa.pt

***Abstract*—Applying known graph metrics to social network graphs have high execution times due to the large graph size. Using GPUs it is possible to improve the execution of these metrics, but the GPU memory is limited. The main focus of this work will be to use the GPUs to process graphs larger than the GPU memory**

GPUs have a complex hardware that is capable of running thousands of concurrent threads. Understanding the architecture of a GPU, the thread and memory organization is necessary to develop efficient GPU kernels. This work discusses the implementation of two node centrality metrics, the PageRank and beteweenness centrality, on a CUDA platform. It also studies the possibility of applying these metrics on large, graphs inside a GPU.

The PageRank results prove that it is possible to process large graphs on a GPU, using graph partitioning and CUDA Streams. Another interesting result was found by caching the rank transfer for each node, in each iteration. In the PageRank $9\times$ speedups where achieved for a graph with $7.0GB$ of size.

Keywords*—graphs, GPU, CUDA, PageRank, Betweenness centrality

I. Introduction

As online social networks have grown in size the last decade, current networks reach the tens of million users, with each user having, on average, hundreds of connections to other users. In order to keep users engaged, companies behind these networks use several metrics to customize the content served to each user. Graphs are best suited to represent this networks, as each user can be represented by a graph node and the connections between users are mapped to the graph edges. Processing these large graphs helps to better understand these new social interactions between users. Reducing the execution time of network metrics allows to better understand the evolution of the network topology and dynamics.

Parallel systems, can be roughly divided into: clusters, multiprocessors and multicore systems. GPUs belong to the latter group. GPUs have a special instruction set to run the thousands of parallel cores, and generally every core runs the same instruction on different data. One limitation of GPUs is that they don't have direct access to host memory, relying instead on their own internal memory. A solution to work with graphs whose size is larger than the GPU memory is to divide in smaller partitions that can fit inside the GPU memory.

This work will be focused on implementing an efficient solution capable of processing large social networks using a single cuda GPGPU card. The implementation will take into consideration the latest features availabe in the CUDA API. Two metrics have been choosen to demonstrate how to take advantage of the highly parallel architecture of a GPGPU. These two metrics, PageRank and Betweenness Centrality, have distinct access patterns to the graph. This creates different oportunities to explore the limits of the GPGPU.

II. Graphs

A graph $G = (V, E)$ is a representation of connected objects, where V is the set of vertices, the objects, and E the set of edges. An edge is defined as a connection between any two vertices. In the context of this work n will refer to the number of nodes in a graph, or $|V|$ and m will denote the number of edges in a graph, or $|E|$. In directed graphs, edges are replaced by directed edges that connect one vertex to another, these are respectively the source and the destination vertices.

A. Shortest Paths

A path in a graph is a sequence of edges that connect a sequence of vertices, where all vertices are distinct. In the case that the first and last vertices are the same it is considered a walk. On unweighted graphs the path has a length equal to the number of traversed edges.

Many of the network analysis measures use the shortest path between nodes. The shortest path is the path between any two vertices s and t that has the lowest length. The choice of algorithm to compute the shortest path depends on graph and measure characteristics. A detailed description of this algorithms, related data structure, and other graph related algorithms can be found in Skiena, 2008 [1].

1) Breadth First Search - BFS: The BFS is a search algorithm for unweighted graphs, it can also be used to discover the SSSP from a source vertex s. It uses an auxiliary queue Q and vertices are marked as $discovered$ as they are added to the it. The queue is initialized with the source vertex s. In every iteration the top element of the queue is removed, and all his neighbors that are $undiscovered$ are added to queue and set as $discovered$. The algorithm stops when the target element is removed from the queue.

B. Graph storage

A graph is usually stored in one of two ways: an adjacency matrix or adjacency list. But there are other storage formats that have a small space requirement and provide fast access times[2]. Plus they also increase the data locality, that is an important factor in GPU applications.

1) Compressed Sparse Row, CSR: It uses an array, *rowStart*, with $n + 1$ elements to store the index of the first nonzero element in each column. The last element of the array, $rowStart[n]$ stores nnz. Where nnz refers to the number of nonzero elements in the matrix.

III. Parallel programming in the CUDA platform

The speedup is the ratio between the execution times of the parallel program, run on p processors, and the sequential version of the same program. The ideal speedup value is equal to the number of parallel processors N. The reason is that to achieve this value it is necessary a fully parallel program without any kind of overheads. In some rare cases it is possible to achieve a superlinear speedup, meaning that the speedup is bigger than p.

The actual speed values usually are much lower than N, due the existence of communication overheads between parallel processors, and also the initialization cost. More information on parallel systems and performance metrics is available in Quinn, 2003 [3].

A. GPU architecture

A GPU is a SIMD system, meaning single instruction multiple data. In this time of system all threads execute the same instructions, but each thread is dealing with different information. There are specialized high-level APIs that allows developers to use compatible GPUs for general purpose processing.

The CUDA API was developed as part of the CUDA platform and programming model. Because was developed alongside the NVIDIA GPU architecture, it is not available for other platforms. But as a result CUDA applications usually has better performance than the OpenCL implementation for NVIDIA GPUs.

The CUDA GPU is built around an array of Streaming Multiprocessors. Each SM is independent from each other, this means that different kernels can be run at the same time in the GPU. This architecture also allows for the applications to scale with the available number of SMs. When a program invokes a GPU kernel, it is distributed through the available Streaming Muliprocessors.

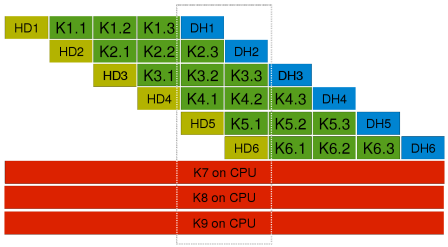

Figure 1. CUDA Streams parallel operations

1) CUDA Streams: The usage of CUDA Streams allows for faster computation times by hiding the memory operation with kernel invocations. It also allows to run different kernels at the same time in the GPU, as long as there are enough available resources. Streams also allow for multiple types of syncrionization, like global syncronization across all streams with $cudaDeviceSyncronize()$, a single stream syncronization with $cudaStreamSyncronize(stream)$, or using events to syncronize across multiple streams. When using streams it is important to consider times of both kernel invocations and memory operations, as well the invocation order of each operation.

IV. Network Metrics

There are several properties relevant to network analysis [4] that theoretically should have a great speedup when implemented in a GPU, like the PageRank and Betweenness Centrality. The first is the original Google algorithm to compute the importance of a web page based on the number and quality of connections to other webpages [5]. The second was first proposed in [6] as a measure of a node's centrality. The Betweenness Centrality for a node is defined as the number of times messages between to other nodes, have to pass through that node.

A. PageRank

The PageRank was developed to rank websites according to their relevance. It considers the number and quality of connections to pages to rank each page, this measure was created assuming that more important websites are more likely to be liked from other sites. Each page rank is computed as the sum of the contributions of all pages that link to it. Also each page contribution is calculated as the page rank equally divided by all links from that page.

Due to the difficulty in finding an exact solution, the PageRank is solved for an aproximate value. This approximation is most commonly computed using the Power Method, an iterative algorithm that computes the PageRank value with an error lower than ϵ. In each iteration a SpMV, a sparse matrix-vector multiplication, is performed between the graph matrix and the PageRank vector, as seen in Algorithm 1.

Algorithm 1 Power Method for PageRank

1: Initialize R randomly to be R_o, and let $k = 0$
2: **repeat**
3: Compute $Y = qAR_k$
4: $d = ||R_k||_1 - ||Y||_1$
5: $R_{k+1} = Y + dE$
6: $k \leftarrow k + 1$
7: **until** $||R_{k+1} - R_k||_1 < \epsilon$

The PageRank also considers the possibility that at some point a random surfer stops following links on a page, with the the $dampening factor$, p.

There is already several studies in implementing the PageRank in a GPU system. One study done using the AMD platform [7], uses the CSR format to store the graph. Their implementation involves sorting the graph by row length, and then proceding to divide the graph into three partitions, according to the number of non zeros in each row. Other works focused instead on using hybrid formats [8], for the SpMV computation. Both CSR and ELL-pack formats are used to store graph partitions.

When the datasets do not fit inside the GPU memory, it is suggested to implement a pipelining mechanism to transfer

data from the host to the GPU [7], [8]. The only drawback of this method is that both implementations present, throughputs between $16GB/s$ and $40GB/s$. Comparing these values to the $15.75GB/s$ maximum transfer speed of the PCI-E bus, indicate a possible bottleneck on the data transfer between the host and the GPU.

B. Betweenness Centrality

Betweenness Centrality[6] is a measure of a node's centrality. The betweenness centrality measures how much a network is dependent on a node to pass information from one side to another. This measure is usefull to determine what nodes are responsible for the flow of information inside a network, like for example how much a user is responsable for allowing interactions between other two users.

The betweenness centrality for a node v is defined as the number of paths that contain the v out of total number of shortest paths between s and t.

For a graph with n vertices and m edges, the current fastest known algorithm to compute the betweenness of all vertices[9], has a time complexity of $O(nm)$ for unweighted graphs and $O(nm+n^2 \log n)$ for weighted graphs. The Brandes' algorithm [9] implementation introduces the concept of dependency of a vertex s on any other vertex v. Taking σ_{st} as the number of shortest paths between s and t, and $\sigma_{st}(v)$ as the number of shortest paths that pass through v, the dependency can be defined by:

$$\delta_{s*}(v) = \sum_{w, v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_{s*}(w))$$

The implementation of Brandes' algorithm follows two main steps, repeated for each node as the source, they are: the graph traversal to find all shortest paths and computation of the centrality score for all vertices. The graph traversal is based on the BFS algorithm, with the adition of the predecessor sets and the number of shortest paths of each node. These two structures are used to compute the score for each node

There are already some works in implementing the betweenness centrality in GPUs with some degree of success [10], [11], that present a few interesting solutions on how to optimize the betweenness centrality on a GPU.

One of the solutions introduced [11] was dividing the source vertices through the available GPU Stream Multiprocessors, and then parallezing the graph traversal inside each SM. Another improvement was to remove the need of the predecessor vertex set, by traversing all neighbors on the centrality computation step.

The other solution [10] focus instead on improving the data access patterns and preprocessing the graph in order to achieve the best speedups. One of their solutions is the virtual-CSR, a format where vertices with a hight degree are replaced by several virtual vertices, each one with at most $ndeg$ edges. One other improvement made was to compress the graph by to removing vertices with degree 1.

V. PageRank Implementation

A. Sequential implementation

Looking into the SNAP implementation of PageRank, [12], the parallelization on a GPU is straightforward, and does not need any kind of manipulation of the graph. A simplified sequential implementation, using less complex data structures, was used to create the first parallel implementation.

Besides the initialization of the rank arrays, there are three main steps that can be parallelized. They are: the actual sparse matrix vector multiplication, SpMV, the sum of the new vector and the computation of the difference in the PageRank vector between the previous iteration vector and the new vector. These two last steps, add the probability of a used stop clicking links, when this addition is made this step also ensures that the sum of the vector is always 1. The computation of rank difference in the PageRank vector is also responsible of stopping the execution of the program when the difference between the newly computed rank vector and the old one is smaller than the specified $error$.

B. Parallel SpMV

Parallelizing the SpMV step from the SNAP implementation can be easily achieved from assigning each node to a different thread. This requires launching as many threads as many nodes there are in the graph. For each node obtain all the outgoing edges, and for each one compute the percentage of the node rank that is going to transferred.

Writing the rank transfer on the destination node will create write conflicts when two different threads try to write into the same position. To solve the conflict it is necessary to use atomic writes for this operation.

1) Removing atomics by transposing the graph matrix: The need of atomic operations in the previous implementation came from the mapping of a source node to each thread. Considering that no data is written back to the source node, it is not relevant if it is the same thread that processes all the node neighbors. Assigning instead each thread to a destination node and then obtaining the set of incoming edges, removes the need for the atomic operation. Using this mapping each thread will only write to its assigned node position, thus having no write conflicts.

In the adjacency matrix format it is pretty straight forward to discover what edges arrive to the n node by looking into the n column of the matrix. The same is not possible when the same graph is represented in the CSR format, as it is in 3.

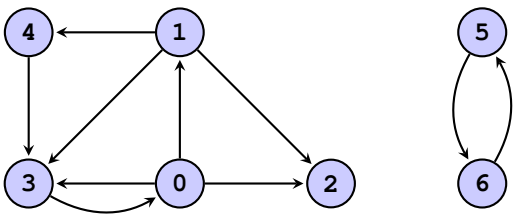

Figure 2. Example graph

The number of outgoing edges for each node is also required, therefore that information is kept in the $values$ array.

$$RowStartIndices = [0 \quad 3 \quad 6 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10]$$
$$ColumnIndices = [1 \quad 2 \quad 3 \quad 2 \quad 4 \quad 3 \quad 0 \quad 3 \quad 6 \quad 5]$$

Figure 3. Example graph in CSR format

This representation is very similar to the CSR format, and even more to the CSWG format [13], that also has an array equivalent to $values$.

2) SpMV without atomic operations: Using the transposed matrix allowed to improve the the SpMV kernel, removing atomic operations and increasing the locality of the writing operations. With this implementation each thread the warp will perform the write operations into the continuous memory positions, reducing the number of memory accesses.The main advantage of this implementation is the absence of atomic operations, that were creating several extra memory accesses as well stalling several threads until the atomic operation ended.

Algorithm 2 CUDA PageRank SpMV without atomic operations

```
1: for all node of graph do in parallel
2:     for edge = rowStart[node] to rowStart[node+1] do
3:         srcNode = columnIndices[edge]
4:         rankChange = R_k[srcNode] * dampening/values[srcNode]
5:         R_k + 1[node] = R_k + 1[node] + rankChange
6:     end for
7: end for
```

3) SpMV kernels for different node sizes: By splitting the graph into partitions that have only nodes of the same size it is possible to largely improve the thread occupancy, and also to have a better workflow distributions. One possible way to do this division is to split the graph into partitions, one with small nodes, other with medium nodes and a third with the largest nodes, [7], and for each of these partitions a slightly different SpMV kernel was implemented.

Nodes classified as small if their size is less than 8 incoming edges and medium nodes have at least 8 but less than 96 edges. The associated kernels for these partitions are described as follows: for small node no changes are made into the previous implementation; for medium nodes, the warp is divided into 4 parts and each part processes a different node; large nodes are processed by an entire warp. Not only this division reduces most of the unbalanced workloads of each thread present in the previous implementation but it also improves the access to global memory.

C. Optimizing the CUDA kernels

After further observation of the SpMV kernels, it was noted that the computation performed of the transferred rank is repeated for each edge on the srcNode but the resulting value is always the same, line 4 of Algorithm 2. This value does not change during each SpMV computation, and computing requires two accesses to global memory, one to read the rank vector and another to read the values vector.Considering that some nodes have a very large number of edges, and only a few have exactly one edge, computing this values before the SpMV and storing them in a temporary array should also improve the overall performance of the SpMV computations.

Also when computing the new rank of each node it is possible to delay the writing of the new rank until all node incoming edges are processed. The write to the PageRank vector, on global memory, can be done at the end of the SpMV computation of that node.

The kernel also performs a warp reduce to get the sum of $result$ across all threads on the warp that are processing the same node. This way only one thread per node performs the write. The reduce operation is done by the $\_\_shfl\_down$ instruction. This instruction is capable of accessing a value from another thread in the same warp, as illustrated in Figure 4.

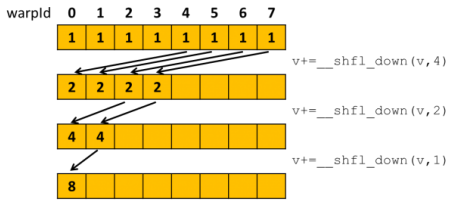

Figure 4. Shuffle down operation

1) Supporting graphs larger than the GPU memory: The GPU memory is far smaller than some of the larger graphs available on our datasets. A partition scheme was implemented by dividing the graph in rows.

To apply a row partitioning scheme to a graph in CSR format is only necessary to know the number of nodes in each partition. There are two ways to determine number of nodes. n_{part}, in each partition, the simpler one is to divide the nodes equally among the partitions, the other ways is to balance the total number of edges across all partitions. These partitions can be either applied to the entire graph, or after splitting nodes as described in Section V-B3, apply this partition scheme to each one of the node classes. Considering that nodes of the same class have a similar number of edges, dividing the node equally among the partitions will provide a satisfactory balance of the edges per each partition.

Knowning the number of nodes in each partitions, it is possible to copy to the GPU memory only the fraction of the CSR data structure relevant to the current partition. The copy of the partitiion is done imediatly before invoking the kernel that is going to process it. When the kernel is finished the partition information on the GPU is replaced by the next partition.

2) Concurrent copies to GPU memory and computations: Lastly, when dealing with multiple partitions that are swapped in and out of the GPU memory, it is necesary to also consider the memory transfer time. This transfer can have a large impact in the overall performance. It was this step that was pointed as a limitation in previous work [7], [8], as the data transfer would take longer than the actual computation.

As refered in Section III-A1, using CUDA Streams it is possible to perform more concurrent operations in a CUDA architecture. Using CUDA Stream is possible to perform a

Algorithm 3 CUDA SpMV kernel for large nodes

```
__global__ void multMVWarp(prankPrecision* prank,
    int nNodes, int nodeOffset,
        int edgeOffset, int* rowStart, int* edgeList
    , int* nodeIds,
        prankPrecision* output) {
    int threadNode = (blockIdx.x * blockDim.x +
    threadIdx.x) / warpSize;
    double result = 0.0;
    if (threadNode < nNodes) {
        int processingNode = threadNode + nodeOffset
    ;
        int start = rowStart[processingNode] +
    threadIdx.x % warpSize;
        int end = rowStart[processingNode + 1];
        for (int edge = start; edgethreadNode end;
    edge += warpSize) {
            result += prank[edgeList[edge-
    edgeOffset]];
        }
        __syncthreads();
        result += __shfl_down(result, 16);
        result += __shfl_down(result, 8);
        result += __shfl_down(result, 4);
        result += __shfl_down(result, 2);
        result += __shfl_down(result, 1);
        __syncthreads();
        if (!(threadIdx.x % warpSize)) {
            output[nodeIds[processingNode]] +=
    result;
        }
    }
}
```

copy to the GPU memory and invoke a kernel at the same time. In effect hidding the communication costs between CPU and GPU behind the computation.

D. Parallel Reduce

Not only reduction operation can be implement in a very efficient way in a CUDA architecture, achieving $O(\log N)$ complexity on a block-wide reduce operation, but such implementation for a block-wide reduce is provided and documented in the CUDA framework [14] as part of their collection of sample programs.

The PageRank requires two reduce operations: one to compute the leaked rank, the other to get the convergence of the PageRank vector between two iterations. By reducing several blocks at the same time and using the kernel invocations as syncronization points, it is possible to use the previous reduce implementation to vector larger than the block size, as show in Figure 5.

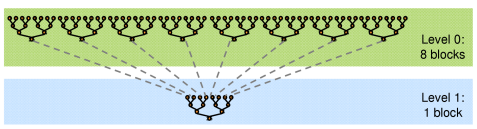

Figure 5. Global reduce operation

E. Computing the rank difference

This last step is responsible for stopping the execution of the program when the difference between the newly computed rank vector and the old one is smaller than the specified $error$. It is also responsible to add the probability the a user stops following links, while ensuring that the sum of the vector is always 1.

When this kernel is done, a reduce operation is applied to the new rank vector, that is now storing the change in rank for every node. This will provide the overrall change on rank vectors, if this value is smaller than the $error$ parameter, the PageRank execution of will finish. Otherwise the new rank value is reset and the PageRank computation will proceed.

F. PageRank with graphs larger than GPU memory

The usage of streams in the SpMV computation is already described in Section V-C2. Partitions of nodes of similar size are computed in concurrent streams, this serves to hide the communication cost behind the computation of another partition of similar size. The reduce and difference computation is syncronized so it only starts after all partitions are processed. The reset for the next iteration is done by using one stream to compute the next PageRank values while the other stream resets the $newRankV$ to 0. Lastly the device is syncronized again, to ensure that both streams have completed before starting the new iteration.

VI. Betweenness Centrality

The best betweenness centrality algorithm is the Brandes' [9], explained in Section IV-B. The Brandes' algorithm computes the betweenness centrality scores by calculating all the shortest paths from a $source$ node to every other node, and then calculating a partial centrality score for each node. This process is repeated for every node as a $source$ node. The final centrality score is obtained by summing all partial scores.

A. Parallelizing the Brandes' algorithm

There are three possible parallelization possibilities in the Brande's algorithm:*Source node* Each $source$ can be processed individually as there are no dependencies between the associated computations. *Nodes at the same distance* The computation of shortest paths, using the BFS is done by traversing the graph as a tree. Nodes at the same depth, or level, in the tree all have the same distance to the $source$ node. *Neighbors of the same node* On the BFS when processing a node it is necesary to traverse all his edges.

In this implementation each $source$ node is processed by a different Stream Multiprocessor. The processing of a each $source$ node is done by a single block, with nodes at the same distance from $source$ being processed in parallel by the block threads. This approach is a combination of the first two parallelization options described before. The parallel implementation requires only small changes to the Brandes' algorithm in order to allow parallel threads and reduce the space requirements.

1) Shortest Paths: The computation of the shortest paths, using the BFS algorithm is slightly modifed from the original Brande's algorithm. It no longer needs to keep the predecessor vertex as the centrality accumulation step will traverse each node individually.

Table I. USED DATASETS AND RESPECTIVE SIZES

Graph	Nodes	Edges	CSR Size (MB)
soc-Slashdot0811	77 360	905 468	20
soc-LiveJournal1	4 847 571	68 993 773	282
com-Orkut	3 072 441	117 185 083	459
com-Friendster	65 608 366	1 806 067 135	7140

To enable the parallel computation of nodes at the same distance from $source$, two queues are used. One with the set of nodes that are being processed in the current iteration and another with the nodes that are going to be processed in the next iteration.

One other change to enable the parallel processing inside the same block is the to track the iteration where nodes where processed. This information is associated to the stack of processed nodes by using an auxiliary array that works like the $rowStart$ in the CSR format.

2) Dependency acumulation: This step is more straight forward as it uses the previous stemp results to compute the partial centrality scores for each $source$ node. In this step nodes at the same distance can be processed at the same time by different threads. For each node all edges are traversed once again to find the predecessor nodes, this not only greatly reduces the mmmory requirements, but is also reported to provide speed ups.

B. Parallel source node processing

Due to the several syncronization points necessary in the BFS kernel, this step can only be assigned to one block. In CUDA each block is mapped to a single Stream Multiprocessor, and each card has several SM's, using only one block is not optimal. The best solution is to have each SM process a different $source$ node. This effect can be obtained using CUDA Streams, where each Stream only processes one $source$ node at a time.

The major limitation of this implementation is that it is not capable to work with graphs larger than the GPU memory. Considering the number and size of auxiliary data structures, like the $queue$, $stack$ and $dependency$ vectors, these will occupy a significant part of the GPU memory, further reducing the limit on the graph size.

VII. TEST ENVIRONMENT

The test hardware is system wih a i7 4790K running at 4.0GHz, 16Gb of RAM and a GTX 970 with 4Gb. The used datasets are obtained from the SNAP Large Network Dataset Collection [15], the datasets where chosen based on their size and the represented network type. Details on each of the datasets are on table I.

To guarantee the validity of the results, each test case was run five times. The presented results are the average of all five runs. Plus the computing times for both sequential and CUDA implementations will only consider the actual execution time. The speedups presented in the evaluation are obtained by comparing the CUDA implementation to a single-core sequential implementation.

Table II. LIVEJOURNAL PAGERANK EXECUTION (WALL) TIME WITH DIFFERENT RANK PARAMETERS

Dampening	Error	Iterations	CPU time (s)	CUDA time (s)	SpeedUp
0.850	$1{\cdot}10^{-4}$	24	6.4489	1.2697	5.0791
0.850	$1{\cdot}10^{-5}$	36	9.5676	1.8353	5.2131
0.850	$1{\cdot}10^{-10}$	105	27.3441	5.0987	5.3629
0.850	$1{\cdot}10^{-15}$	175	45.3952	8.4079	5.3991
0.900	$1{\cdot}10^{-4}$	35	9.2886	1.7880	5.1950
0.990	$1{\cdot}10^{-4}$	309	80.0007	14.7421	5.4267
0.999	$1{\cdot}10^{-4}$	2,313	597.4695	109.4677	5.4580
0.990	$1{\cdot}10^{-15}$	2,804	740.5829	130.2306	5.6867

VIII. PAGERANK

First it is not only important to check if the PageRank implementation described in the previous chapter is capable of achieving any speedup compared with the sequential execution. Aditional tests with different PageRank parameters where also run to understand and exemplify how these affect the overall execution times, a constant speedup over all all test cases was expected as the only difference is the number of iterations..

Each one of these node classes was divided into 12 partitions. Although a smaller number of partitions could be used for graphs that fit inside the GPU, after inspection of the execution of CUDA programs it was found to have no negative effects on the performance.

With the soc-livejournal1 graph it was achieved a speedup of about 5,7. This low value for speedup can be attributed to the cost on initializing the GPU and lauching the multiple kernels. It is also noteworthy that the speedup has a slight increase in test cases that have a higher number iterations. This behaviour can be explained by the small impact of the preprocessing step to splitting the nodes by their size.

Using with different values of the dampening factor and the approximation error lead to a different number of iterations of the Power Method Algorithm. As expected the achieved speedup is independent of the number of iterations. The difference in the Cuda Total Time and Cuda Execution Time results from transposing the matrix in the step.

A. Partitioning graphs

The friendster graph has over 1800 Million edges, storing the graph needs around 7GB. The graph alone can not be copied entirely to the GPU memory, and it is also necessary to have space to store the other structures necessary for the PageRank computation.

While each of the node classes can be seen as graph partition, these are not enough. Each node class was further divided into twelve paritions with the same number of nodes. The total GPU memory necessary was reduced to around 3.5GB. This size includes all data structures as well two partitions, one being copied to the GPU and another being processed.

Using more partitions than the strictly necessary to the GPU computation can improve the performance. This is a result of the usage of CUDA Streams, where the first copy to GPU memory is the only one not hidden behind computation. With more partitions, the copy operation times are smaller, leading to a slightly increase in performance in each iteration.

Table III. FRIENDSTER PAGERANK EXECUTION (WALL) TIME WITH DIFFERENT RANK PARAMETERS

Dampening	Error	Iterations	CPU Total Time (s)	CUDA time (s)	SpeedUp
0.850	$1{\cdot}10^{-4}$	20	351.0906	39.3774	8.9160
0.850	$1{\cdot}10^{-5}$	26	393.4610	50.0138	7.8670
0.850	$1{\cdot}10^{-10}$	55	811.0283	100.2703	8.0884
0.850	$1{\cdot}10^{-15}$	84	1,230.2332	149.0471	8.2540
0.900	$1{\cdot}10^{-4}$	23	347.5900	44.7532	7.7668
0.990	$1{\cdot}10^{-4}$	33	492.5164	62.1141	7.9292
0.999	$1{\cdot}10^{-4}$	34	507.2943	63.9134	7.9372
0.990	$1{\cdot}10^{-15}$	135	1,967.8159	239.5791	8.2136

Table IV. EXECUTION (WALL) TIME WITH DIFFERENT PARTITIONING PARAMETERS

Parameters			CUDA execution time (s)	
Warp Division	Small nodes	Medium nodes	com-friendster	soc-LiveJournal1
8	8	96	39.0518	1.2224
8	8	64	38.2691	1.1587
8	8	16	37.5193	1.1359
8	4	32	38.4340	1.1467
8	4	16	37.8621	1.1166
8	4	8	38.4340	1.1167
4	8	16	37.8953	1.1011
4	4	16	37.7561	1.1047

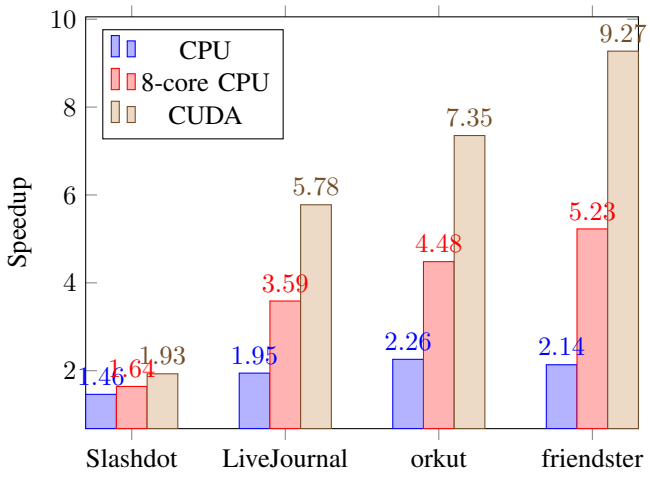

Figure 6. PageRank speedup increase with rank cache

B. Node Classes

In Table IV possible partition parameters where tested for both soc-livejournal and com-friendster. The combination of 8 divisions per warp, 8 edges as the limit for medium nodes and 16 edges as the limit for large nodes was found to have the best performance on the larger graphs, and was similar to the best performing parameters in the soc-livesocial1.

C. Caching the rank values

As described in V-C the implementation of the PageRank was changed. The rank transference done througn each node is computed before the begining of the SpMV operation. This processed, called rank cache, as in effect it caches the multiplication of the previous rank with the dampening factor and diving by number of edges of the source node. When testing the performance increase of this step, Table V, it was found to not only increase the speedup across all graphs but also being more relevant on larger graphs, Figure 6. This performance increase can be explained by reducing the number of memory accesses required by each node as well the number of floating poing operations. With larger graphs it becomes less likely that for the same memory access to read many source nodes for incoming edges of the current node, and with more nodes it also becomes less likely that node information is still on the GPU cache.

D. CUDA Streams

The usage of Streams to perform a memory operations and computation concurrently was described in V-F. Experimental results show that, excepting the first memory copy from host to the device and a few others, all memory operations happen at the same time that a kernel is running.

It is also interesting to note that in the majority of the pairs of memory copy and kernel invocations there is a lapse between the end of the copy and the begining of the corresponding kernel. This lapse happens because the previous kernel is still being run. Meaning that the memory operations will not represent a bottleneck even with faster kernels.

IX. BETWEENNESS CENTRALITY

Looking in how the graphs are prcessed in the betweenness centrality and the necessary data structures, no aparent partition scheme would allow for graphs that are not completely inside the GPU memory to be processed. In section VI-B it is described a way to process each $source$ in a different Stream Multiprocessor, that while scalable across multiple GPU's[11], it limits the graph size. The soc-livejournal1 graph only has around 282MB, but with 20 simultaneous $source$ nodes it is necessary to have 3.2GB of memory available on the GPU. This solution doesn't seem practical with the current hardware.

In spite of the low speedups achieved with the betweenness centrality, using a parallel approach for this problem might still be benefitial. This is evident on Table VII, where the CPU execution would take about two months to complete, a parallel approach to the same problem would take just over two weeks. Using the a more efficient approach it should be possible to further reduce the parallel execution times [10].

X. CONCLUSIONS

This work focus on processing graphs on a GPU, with special emphasys on graphs larger than the GPU memory. One of the main problems with processing large on GPU is that the graph has to partitioned, to be processed each partition has to be first copied to the GPU.

Table V. EXECUTION (WALL) TIME OF THE PAGERANK, INCLUDING THES CACHE OPERATION

Graph	CPU (s)	8 core CPU (s)	CUDA (s)	CUDA Cached (s)	Speedup 8 core CPU	Speedup CUDA	Speedup CUDA Cached
Slashdot	0.0424	0.0290	0.0258	0.0220	1.4630	1.6424	1.9297
LiveJournal	6.4489	3.3148	1.7980	1.1164	1.9455	3.5867	5.7766
orkut	1.9838	7.9919	1.7830	1.0870	2.2594	4.4822	7.3525
friendster	351.0906	164.3603	67.1656	37.8621	2.1361	5.2272	9.2729

Table VI. BETWEENNESS CENTRALITY EXECUTION (WALL) TIME IN CUDA

Graph	CPU time (s)	8 core CPU time (s)	CUDA time (s)	Speedup
CA-HepTh	3.8621	1.4103	0.8886	4.3465
Soc-Slashdot	431.6054	291.7025	175.7743	2.4555
Soc-LiveJournal1(est.)	$4.9384 \cdot 10^6$	$1.9582 \cdot 10^6$	$1.5393 \cdot 10^6$	3.2081

Table VII. LIVEJOURNAL BETWEENNESS CENTRALITY EXECUTION (WALL) TIME, THE LAST VALUE IS ESTIMATED

nNodes	CPU time (s)	8 core CPU time (s)	CUDA time (s)
512.0000	537.3114	215.4100	1.6971
32,768.0000	33,329.2340	13,236.8417	10,402.7689
65,536.0000	66,782.4897	26,477.0303	20,916.9217
$1.3107 \cdot 10^5$	NaN	NaN	41,652.0177
$2.6214 \cdot 10^5$	NaN	NaN	83,225.0354
$4.8476 \cdot 10^6$	$4.9384 \cdot 10^6$	$1.9582 \cdot 10^6$	$1.5393 \cdot 10^6$

The CUDA Streams are used in this work to hide memory copy operations with the execution of the kernels, processing other graph partitions already on the GPU memory. This approach has not been refered in any the previous works that focused the graph processing on GPUs, but it is a common practice in CUDA environments [14]. This work shows that, for the PageRank, it is possible to process graphs larger than the GPU memory in a GPU by partitioning the graph, it also shows that is possible to improve current PageRank implementation by optimizing the GPU kernels. Regarding the betweenness centrality the implementation described in this work no only achieves small speedups against a single CPU core but also is not capable to process graphs larger than the GPU memory.

XI. PAGERANK

Splitting the nodes in different classes according to their size allows for specialized SpMV kernels with more coalesced memory operations and less idle threads. Regarding the implementation of the SpMV kernels a mechanism to reduce non coalesced memory accesses was implemented. In the PageRank the each node transfers equal portitions of its rank through all of its outgoing edges. It makes sense to compute this transfer only once and cache the value to be used when visiting each node edges. Just by using this optimization it was possible to increase the GPU implementation $1.77\times$.

The resulting CUDA implementation is capable of dealing with graphs that are bigger than available GPU memory. The communication cost between host and GPU is hidden by the kernel computations, with the exception of the initial transfer in each iteration. This partition scheme works as long all auxiliary data structures, with a space complexity of $O(n)$, are fully stored inside the GPU memory, plus enough space for two partitions.

XII. BETWEENNESS CENTRALITY

The betweenness centrality traverses a large part of the graph several times, each time starting from a different node. This behaviour doesn't allow for partition schemes as used in the PageRank. It is necessary to have the entire graph accessible in memory, limiting the graph size to the GPU memory. With each block processing a different source node, it is necessary to have at least as many blocks as SM in the GPU chip, but wWhen processing several nodes at the same time, the required memory space quickly exceeds the available GPU memory for graphs with just a couple million nodes.

With this betweenness centrality implementation only small speedups where achieved. Considering that execution time of the betweenness centrality can take a few of months for graphs with just a couple of million nodes, a speedup of three or four times significantly reduces the execution time to just some weeks.

XIII. FUTURE WORK

There are some possible improvements to increase the GPU performance of both measures. In the PageRank it should be possible to use better balanced partitions with similar number of edges, as well presorting the graph by node size. This two steps should increase the kernel efficiency, by having similar workloads for threads in the same warps. Sorting the graph by node size will also reduce the need for a redirection array used in SpMV kernels. Applying these optimizations should not interfere with the proposed solution regarding the hiding the memory transfers behind kernel computations. These transfers can also be further improved by using pinned memory on the host to improve transfer speeds on large memory operations.

Relating to achieve better performance with the betweenness centrality possible improvements [10] are balancing the number of edges on each node by using virtual nodes and also removing node with only one edge from the graph. To allow graphs larger than the GPU memory is first necesary to allow the processing of each source to use more than just one block.

REFERENCES

[1] S. S. Skiena, *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd ed., 2008.

[2] NVIDIA Corporation, *NVIDIA CUDA cuSPARSE Library*, August 2014.

[3] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[4] L. da F. Costa, F. A. Rodrigues, G. Travieso, and P. R. V. Boas, "Characterization of complex networks: A survey of measurements," *Advances in Physics*, vol. 56, pp. 167–242, Jan. 2005.

[5] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," Technical Report 1999-66, Stanford InfoLab, November 1999.

[6] L. C. Freeman, "Centrality in social networks: Conceptual clarification," *Social Networks*, vol. 1, no. 3, pp. 215–239, 1979.

[7] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu, "Efficient pagerank and spmv computation on amd gpus," in *Parallel Processing (ICPP), 2010 39th International Conference on*, pp. 81–89, Sept 2010.

[8] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus: Implications for graph mining," *Proc. VLDB Endow.*, vol. 4, pp. 231–242, Jan. 2011.

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

[10] A. E. Sariyüce, K. Kaya, E. Saule, and U. V. Çatalyürek, "Betweenness centrality on gpus and heterogeneous architectures," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, (New York, NY, USA), pp. 76–85, ACM, 2013.

[11] A. McLaughlin and D. A. Bader, "Scalable and high performance betweenness centrality on the gpu," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (Piscataway, NJ, USA), pp. 572–583, IEEE Press, 2014.

[12] J. Leskovec and R. Sosič, "SNAP: A general purpose network analysis and graph mining library in C++." http://snap.stanford.edu/snap, June 2014.

[13] A. Rungsawang and B. Manaskasemsak, "Fast pagerank computation on a gpu cluster," in *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pp. 450–456, Feb 2012.

[14] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, August 2014.

[15] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." http://snap.stanford.edu/data, June 2014.