Exploiting GPGPUs for Determining Inherent Privacy in Large Social Graphs

Daniel Sawyer
University of South Florida
Tampa, Florida 33620
danielsawyer@mail.usf.edu

Hunter Morera
University of South Florida
Tampa, Florida 33620
hmorera@mail.usf.edu

Sathyanarayanan Aakur
University of South Florida
Tampa, Florida 33620
saakur@mail.usf.edu

ABSTRACT

Anonymization of data for research purposes is essential for preserving the privacy of those involved; this is especially true when dealing with social networks. If the anonymization is insufficient, then a smart attacker would be able to extract sufficient information and the privacy would be compromised. This can lead to disastrous consequences depending on the nature of privatized data. In this paper, we propose the use of structural information to identify the nodes which are prone to such attacks and hence strengthen the anonymization techniques. Conversely, the same can be used to exploit such weaknesses in the anonymization techniques to recover data. We introduce two metrics - safety and risk metrics that are representative of the level of susceptibility of the social graph to adversial attacks based on structural properties. To this end, we use powerful parallel processing techniques in the form of General Purpose Graphical Processing Units (GPGPUs) to scale up the process to larger graphs. We show results on real-life datasets that are more than 100,000 nodes in size and more than 1 million edges.

CCS CONCEPTS

•Security and privacy → Usability in security and privacy;
•Computing methodologies → Massively parallel algorithms;
Vector / streaming algorithms;

KEYWORDS

Privacy, Structural Clustering, Social Graphs, GPGPUs, CUDA

ACM Reference format:
Daniel Sawyer, Hunter Morera, and Sathyanarayanan Aakur. 2017. Exploiting GPGPUs for Determining Inherent Privacy in Large Social Graphs. In *Proceedings of Programming on Massively Parallel Systems, Tampa, Florida USA, July 2017 (CIS6930),* 5 pages.
DOI: 10.475/123_4

1 INTRODUCTION

Structural anonymization of social graphs is integral in preserving its privacy, as it decouples node identities from malicious attacks. Structural anonymization aims to perturb the topology of the graph such that individual nodes cannot be recognized based on local information that might be available to the attacker, such as node degree, clustering coefficient or neighborhood information, to name a few. The importance of structural anonymization is particularly highlighted by the cases of "Jefferson High" [1] and "Tastes, Ties, and Time: Facebook data release" [2] where the anonymized versions of the social networks were de-anonymized by exploiting its structural properties such as neighborhood information of popular nodes.

One measure of anonymity is how much effort it takes to break the secret. As computations on large graphs are inherently expensive: the larger the graph, the more computational effort is needed to de-anonymize the anonymized version of the graph. We thus ask: is it true that large real graphs are more private than small real graphs? If this were true, graph anonymization efforts could be tailored for the size of the graph: smaller graphs would need more sophisticated anonymization procedures, larger graphs would require less significant anonymization effort to provide the same guarantees. To answer this question, we need a way of defining the inherent privacy of a graph. In this work, we focus on the privacy of node identity and ignore the issue of edge anonymity.

We refer to the inherent privacy of a graph as the property of the graph to conceal identifiable topological information for a large set of its nodes. In this respect, a clique, a ring, a star, a lattice graph are all example of inherently private graphs: based on topological properties, all nodes are identical, thus filost in the crowdfi. This intuition is at the core of k-anonymity anonymization techniques [8, 38]. In this paper, we define the inherent privacy of a graph based on linkage covariance as defined by Aggarwal et al. [5]. It has been shown that linkage covariance varies insignificantly in the processes of anonymizing the graph by swapping edges. Consequently, they showed that simple edge-swapping anonymization techniques are insufficient for providing anonymity, as the signature of nodes remains intact.

A node's linkage covariance vector is considered to be the structural signature of a node and hence can be used to describe the structural similarity of nodes within the graph. This, naturally, allows us to use linkage covariance as part of the metric to define inherent privacy. We define the inherent privacy of a graph as a function of the percentage of nodes in the graph that have unique linkage covariance signatures. Alternatively, we can reason about the inherent privacy of a graph by analyzing the percentage of nodes that have at least k other nodes with identical signatures (like k-anonymity).

However, the algorithm, defined in Section 2.1, is highly complex and resistant to scaling to large social graphs. In this paper, we describe the techniques used to scale the algorithm on massively parallel heterogeneous computing platforms like GPGPUs. We begin with a brief introduction of the algorithm in Section 2 followed by the implementation details for various modes of implementation in Section 3.

CIS6930, Tampa, Florida USA
2017. 123-4567-24-567/08/06…$15.00
DOI: 10.475/123_4

2 INHERENT PRIVACY

In this section, we describe the algorithm for determining the inherent privacy of a social graph. We begin with discussion of the concept of linkage covariance as defined by Aggarwal et al. We, then, follow with discussion on approximation techniques that make it conducive to scaling.

2.1 Linkage Covariance

We use the definition of linkage covariance as proposed by Aggarwal et al[5]. Formally, for two nodes p and q, the definition of linkage covariance $LinkCov(p, q)$ is equal to the covariance between the two random variables :

$$\begin{aligned} LinkCov(p,q) &= E[\hat{X}^p \cdot \hat{X}^q] - E[\hat{X}^p] \cdot E[\hat{X}^q] \\ &= \sum_{k=1}^{N} x_{pk} \cdot x_{qk}/N - (\sum_{k=1}^{N} x_{pk}/N \cdot \sum_{k=1}^{N} x_{qk}/N) \end{aligned} \quad (1)$$

For a given node p, let $\hat{X}^p$ represent the random 0-1 variable, which takes on the value 1, if node p is linked by an edge to any potentially adjacent node q and 0 otherwise. W e have instantiations of this random variable for all possible (potentially) adjacent nodes q, and the corresponding instantiation is denoted by $\hat{X}^q$. The value of is 1, if an edge does indeed exist from node p to node q. The advantage of linkage covariance is that it is robust to edge additions and deletions for massive and sparse graphs. The linkage covariance for a given node does not change very easily. Hence, they can be used to define a signature or characteristic vector for that node. There are several ways of defining this signature or characteristic vector.

- When the mapping between two different graphs are completely unknown, we can create a vector of linkage covariances, which are sorted in decreasing values. The most natural form of the characteristic vector of node p is defined as the ranked linkage characteristic vector.
- When the mapping between two different graphs are approximately known for all nodes, we can create a sort order of nodes in the two graphs corresponding to this mapping. The sort order is used to define the signature vector. This provides more accurate results, when we match the signatures between the two graphs.
- In some cases, an approximate mapping is known for some of the nodes, but not others. In such cases, if the mapping is known, we use a sort order on the nodes, and to define the remaining part of the signature, we use a sort order on the magnitudes of the link covariances.

The algorithm for computing the linkage covariance signature for all nodes in the graph is illustrated in Algorithm 1. Using the linkage covariance signatures, we can identify nodes whose linkage covariance signature is identical. This gives us the nodes who are structurally identical within the graph and hence find out how many identical groups are present for all the nodes. This allows us to evaluate inherent privacy of a social graph.

The linkage covariance function begins with the edge list of a graph G. Using the edge list information, for each vertex, we get the neighborhood of with every other node in the graph G. For all the vertices V in the graph G, we then calculate the linkage covariance of the vertices and as the set difference of the neighbors of vertices and respectively. Since the linkage covariance value is associative, the linkage covariance between vertices and is also the same value. This is shown Algorithm 1 from lines 5 through 9. In Line 10, the process of ranking the linkage covariance signatures is done to order the signatures in monotonically decreasing order. This process is repeated for all the vertices in the graph G.

```
computeLinkageCovariance (G);
E ← EdgeList(G)
∀V_i ∈ E : Construct N(V_i) − where N(V_i) is the neighborhood
  of vertex V_i
LC ← ∅
foreach V_i ∈ E do
    V_s ← V(E) \ V_i
    foreach V_j ∈ V_s do
        LC(V_i, V_j) ← |{N(V_i) ∩ N(V_j)}|
        LC(V_i, V_j) ← LC(V_i, V_j)
    end
    LC(V_i) ← Sorted vector of {LC(V_i, )}
end
UniqueLC ← ∅
IdenticalGroup ← ∅
foreach V_i ∈ E do
    V_s ← V(E) \ V_i
    foreach V_j ∈ V_s do
        if LC(V_i) == LC(V_j) then
            if LC(V_i) ∉ UniqueLC then
                UniqueLC ⋃{LC[V_i]}
            end
        end
    end
end
N_group ← ∅
foreach LC ∈ UniqueLC do
    N_LC = ∑_{k=1}^{N} [LC[V_i] == LC]
    N_group ⋃{N_LC}
end
foreach n ∈ N_group do
    N_l = ∑_{k=1}^{N} [LC[V_i] == LC]
    IdenticalGroup[n] = {N_LC}
end
return IdenticalGroup
```

Algorithm 1: Algorithm for Computing Linkage Covariance for a given graph G.

We then take the linkage covariance signature of each vertex and compare it with the linkage covariance signature of every other vertex in the graph. If it is unique, we count the number and categorize it appropriately. For all non-unique linkage covariance signatures, we group them based on the size of the group and the number of groups of each size that is identical. This is shown from lines 11 through 27 of the algorithm. We return the identical groups that are found from the algorithm for evaluating the inherent privacy of the graph.

The algorithm for computing the linkage covariance introduced in Algorithm 1 has the computational complexity of $O(|V|^2)$, where

V is the set of nodes in the graph. Although Aggarwal et al reckon that the negative term in Equation 1 can be ignored in the case of large and sparse graphs (which is the case for social networks), this time complexity remains $O(|V|^2)$.

2.2 Scaling

Aggarwal et al have formulated the value of linkage covariance as mentioned in Equation 1. In graph parlance, the first term in the equation represents the common neighbors between nodes p and q. The second term translates to the product of the degrees of nodes p and q respectively. Considering large social graphs, the second term in the Linkage Covariance calculation can be omitted since it will give rise to a very small value. Thus, the equation for calculating linkage covariance reduces to just considering the common neighbors between nodes p and q, over the number of nodes N as shown below.

$$LinkCov(p,q) = \sum_{k=1}^{N} x_{pk} \cdot x_{qk}/N \tag{2}$$

This contributes to the reduction in memory footprint. In addition to this, the value of the number of nodes N for large social graphs will be very high. But large social graphs are sparse, so the common neighbors between a pair of nodes will be a small number. Hence, Equation 2 will give rise to a floating-point value. To make computation easier, since the linkage covariance value is going to be uniform for all nodes, we get rid of the division by N, the number of nodes. This reduces memory footprint considerably since there will be $O(|V|^2)$ different values of linkage covariances. After this level of optimization, the equation for computing linkage covariance becomes as shown below.

$$LinkCov(p,q) = \sum_{k=1}^{N} x_{pk} \cdot x_{qk} \tag{3}$$

Additionally, we improve the time complexity of the algorithm by exploiting the symmetric nature of the adjacency matrix of a graph. This means that the linkage covariance of node (p,q) is equal to the linkage covariance of node (q,p). Hence this associative property allows us to process only one half of the adjacency lists and obtain the linkage covariance of all nodes in the graph.

3 IMPLEMENTATION DETAILS

In this section, we describe the implementation details of the algorithm. We begin with description on the specifications of the computing platform on which the experiments were conducted. We follow with discussion on the various implementation methodologies of the algorithm that we conducted and discussion on their performance.

3.1 Computing Platform

The experiments were conducted on a computer system, whose technical specification is described below:

(1) A 4 TB disk based storage system
(2) Intel i7-7700K @4.8GHz
(3) G.Skill 32GB DDR4 @3600MHz
(4) GPU: Nvidia Titan X Pascal: 3584 Cores, 12GB GDDR5X
(5) Ubuntu Desktop 16.04.2
(6) CUDA Version 8.0, sm35

3.2 Naive Implementation

The linkage covariance signature for the entire graph can be computed in a completely Naive manner by computing the auto-covariance of the adjacency matrix. An adjacency matrix is a $N \times N$ dimension matrix of 1's and 0's, where the element of the matrix at the location A_{ij} is 1 if there exists an edge between the two nodes i and j. The auto-covariance of the adjacency matrix is comparable to the adjacency matrix multiplied by itself and normalized by the number of vertices and hence can be equated to the Equation 3. This is referred to as the Naive implementation in this paper.

The issue with this implementation is the amount of memory required for processing. When scaling up to hundreds of thousands of nodes as is the case with real life social graphs, the memory requirements are massive and hence it is not pragmatic to expect this approach to be able handle graphs with large amounts of vertices. For example, consider a graph with 75,000 nodes. To accommodate a linkage covariance value in working memory, we would need $75000 \times 75000 \times sizeof(Element)$, which, in C, would be 22,500,000,000 bytes or 22.5 GB. It is to be noted that we would be needing at least twice this amount of memory as we would need to store the adjacency matrix as well as the resulting linkage covariance matrix. Hence, the memory footprint required for computation scales almost exponentially when dealing with increasingly larger social graphs.

3.3 CPU Optimization

To overcome the space complexity, we represented the graph as adjacency lists instead of adjacency matrix. This reduced the memory footprint of the algorithm as real-life social graphs are large (in terms of vertices) and sparse i.e. the number of edges is far less when compared to a fully connected network. For optimizing the CPU version for fair comparison with GPU, we made use of dynamic memory allocation in CPU for expanding and shrinking the linkage covariance signatures as well as for the representation of the social graph. An important fact to note is that we also used the various optimization techniques in the C programming language such as fast indexing and integer operations to ensure that the memory was managed efficiently. An advantage of the CPU version was the ability to dynamic management of memory as the number of non-zero linkage covariance values is not easily discernible at compile-time. Hence *realloc* and *malloc* were used extensively for jagged array representations. However, this was not conducive to parallelization at the CPU level (through either MPI or threads).

3.4 GPU Naive Implementation

For the GPU implementation we began with the GPU version of the Naive implementation - the computation of auto-covariance through generic matrix multiplication. It can be seen from Section 4.2 that the memory management issue prevailed in the GPU version as well while still managing to speed up the processing compared to the Naive CPU version. However, the efficiency of

Adjacency lists and dynamic memory management (from the Optimized CPU version) could not be beaten.

3.5 GPU Optimization

In this section, we explain the different optimizations that we performed to achieve better efficiency for computing the linkage covariance.

3.5.1 GPU Optimization 1. In this version, we used adjacency lists for representing the graph data structure in the compressed sparse row (CSR) format. This allowed us to compress the graph representation into a single one dimensional vector and a separate vector for representing the length of the adjacency for a specific node. We store only the non-zero elements in a row-wise order. Two arrays, values and columns, whose sizes are equal to the number of nonzero elements, store the value and column index of each element in increasing order of the column index in each row. This allows us to store the data in continuous blocks and store as less data as possible. However, as dynamic memory allocation was not possible in the kernel, we were handicapped with the availability of memory on the GPU, which, even at a heft 12GB, was unable to scale beyond graphs having more than 10,000 nodes. This is mainly due to the fact that the linkage covariance signature for the graph was still in the matrix representation of $N \times N$ dimension as the number of non-zero linkage covariance values was known at compile-time.

3.5.2 GPU Optimization 2. In this optimization technique, we used two kernels - one for estimating the number of non-zero linkage covariance values and the other for getting the actual linkage covariance values themselves. The first kernel is an abbreviated version of the actual linkage covariance kernel which merely checks the number of non-zero values for each vertex's linkage covariance signature and hence provides an exact number of linkage covariance values. We used this information along with the second kernel for using the same CSR representation for the linkage covariance signatures as we did using the adjacency lists for the representation of the graph data.

We did not parallelize the computation of the identical groups as there existed a serial dependence on the different linkage covariance vectors - the part of sorting the linkage covariance signature for each node. This part of the computation was optimized in terms of a histogram on the CPU side of the implementation.

3.5.3 GPU Optimization 3. In this optimization technique, we used 4 kernels - one for estimating the number of non-zero linkage covariance values, second for getting the actual linkage covariance values themselves, third for summing up the lcm adjacency matrix for the offsets, and the final one to calculate the histogram. The main difference between Optimization 4 and this kernel is that we use one block to calculate lcm of a single vertex instead of a single thread calculating each vertex, it also uses thrust::sort inside the kernel. This kernel is faster in certain scenarios but Optimization 4 was the fastest overall.

3.5.4 GPU Optimization 4. In this optimization technique, our best performing one, we were able to parallelize the histogram computation. We used three kernels - one for estimating the number of non-zero linkage covariance values, one for getting the actual linkage covariance values themselves and the final one for computing the histogram.

A very important drawback for the second optimization technique - the sorting portion, was overcome by using an in-built functionality on the CPU side of the algorithm for QuickSort. Then the updated linkage covariance signature was transferred back to the kernel in the CSR format where it was compared against each of the other nodes for identity.

4 EXPERIMENTAL RESULTS

In this section, we describe the results of the experimental evaluation of our approach. We begin with a short description of the datasets used for evaluation followed y quantitative results on the increase in efficiency obtained.

4.1 Datasets

We used 10 publicly available social network datasets for evaluating our approach. Taken from the Stanford Network Analysis Project, these are anonymized, real-life social networks and web networks that were used for evaluation. Ranging from 63 nodes to more than a million nodes and even more edges, we evaluated each of the described methodologies on these datasets. The details of the datasets is shown in Table **??**

Name	Nodes	Edges
TerrorNet	63	308
BlogCatalog3	10,308	330,983
p2p-Gnutella30	36,678	88,328
soc-Epinions1	75,879	508,837
Slashdot0811	77,360	905,468
Douban	154,906	654,188
Email-EuAll	265,213	420,045
com-amazon.ungraph	548,458	925,872
Youtube Ungraph	1,157,806	2,987,624
Youtube Edges	1,138,478	2,990,443

Table 1: Description of the properties of each of the datasets used in the experiments.

4.2 Quantitative Results

In this section the comparison of the approaches on each of the datasets is shown. All results are in seconds.

Note: If results were not present due to memory constraints, then it is marked with "***Too large***".

It can be seen that the methods with any portion on the CPU was unable to complete. **CPU Naive vs. CPU Optimized**

CPU Optimized vs. GPU Naive

Dataset	CPU Naive	CPU Opt	Increase
TerrorNet	0.000657	0.000133	0.000524
BlogCatalog3	2658.3	21.7	2636.6
p2p-Gnutella30	Too Large	21.3	21.3
soc-Epinions1	Too Large	139.8	139.8
Slashdot0811	Too Large	170.1	170.1
Douban	Too Large	313.9	313.9
Email-EuAll	Too Large	2644.0	2644.0
com-amazon.ungraph	Too Large	2081.8	2081.8
Youtube Ungraph	Too Large	Too Large	0.0
Youtube Edges	Too Large	Too Large	0.0

Dataset	CPU Opt	GPU Naive	Increase
TerrorNet	0.000133	0.00409	-0.004
BlogCatalog3	21.7	112.3	-90.6
p2p-Gnutella30	21.3	Too Large	-21.3
soc-Epinions1	139.8	Too Large	-139.8
Slashdot0811	170.1	Too Large	-170.1
Douban	313.9	Too Large	-313.9
Email-EuAll	2644.0	Too Large	-2644.0
com-amazon.ungraph	2081.8	Too Large	-2081.8
Youtube Ungraph	Too Large	Too Large	0.0
Youtube Edges	Too Large	Too Large	0.0

CPU Optimized vs. GPU Opt 1

Dataset	CPU Opt	GPU Opt 1	Increase
TerrorNet	0.000133	0.0003	-0.000167
BlogCatalog3	21.7	126.4	-104.7
p2p-Gnutella30	21.3	Too Large	-21.3
soc-Epinions1	139.8	Too Large	-139.8
Slashdot0811	170.1	Too Large	-170.1
Douban	313.9	Too Large	-313.9
Email-EuAll	2644.0	Too Large	-2644.0
com-amazon.ungraph	2081.8	Too Large	-2081.8
Youtube Ungraph	Too Large	Too Large	0.0
Youtube Edges	Too Large	Too Large	0.0

CPU Optimized vs. GPU Opt 2

Dataset	CPU Opt	GPU Opt 2	Increase
TerrorNet	0.000133	0.00376	-0.0036
BlogCatalog3	21.7	16.8	4.9
p2p-Gnutella30	21.3	4.4	16.9
soc-Epinions1	139.8	78.6	61.2
Slashdot0811	170.1	72.3	97.8
Douban	313.9	135.0	178.9
Email-EuAll	2644.0	254.5	389.5
com-amazon.ungraph	2081.8	348.9	1732.9
Youtube Ungraph	Too Large	Too Large	0.0
Youtube Edges	Too Large	Too Large	0.0

CPU Optimized vs. GPU Opt 3

Dataset	CPU Opt	GPU Opt 3	Increase
TerrorNet	0.000133	0.00274	-0.0026
BlogCatalog3	21.7	16.2	5.5
p2p-Gnutella30	21.3	1.8	19.5
soc-Epinions1	139.8	32.2	107.6
Slashdot0811	170.1	58.8	111.3
Douban	313.9	25.7	288.2
Email-EuAll	2644.0	252.6	2391.4
com-amazon.ungraph	2081.8	181.4	1900.4
Youtube Ungraph	Too Large	Too Large	0.0
Youtube Edges	Too Large	Too Large	0.0

CPU Optimized vs. GPU Opt 4

Dataset	CPU Opt	GPU Opt 4	Increase
TerrorNet	0.000133	0.00274	-0.0026
BlogCatalog3	21.7	10.3	11.4
p2p-Gnutella30	21.3	0.8	20.5
soc-Epinions1	139.8	60.1	79.7
Slashdot0811	170.1	54.3	115.8
Douban	313.9	16.5	297.4
Email-EuAll	2644.0	232.5	2411.5
com-amazon.ungraph	2081.8	52.7	2029.1
Youtube Ungraph	Too Large	Too Large	0.0
Youtube Edges	Too Large	Too Large	0.0

5 CONCLUSION

We showed results on several real-life datasets and several methods for performing the structural clustering on GPGPUs and achieved really significant optimizations.

REFERENCES

(1) Bearman, Peter S., James Moody, and Katherine Stovel. "Chains of affection: The structure of adolescent romantic and sexual networks 1."American journal of sociology110.1 (2004): 44-91.

(2) https://cyber.harvard.edu/node/94446

(3) Sala, Alessandra, et al. "Sharing graphs using differentially private graph models."Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference. ACM, 2011.

(4) Zheleva, Elena, and Lise Getoor. "Preserving the privacy of sensitive relationships in graph data."Privacy, security, and trust in KDD. Springer Berlin Heidelberg, 2008. 153-171.

(5) Aggarwal, Charu C., Yao Li, and S. Yu Philip. "On the hardness of graph anonymization."Data Mining (ICDM), 2011 IEEE 11th International Conference on. IEEE, 2011.