# CS293: Report on efficiency of regularization on Graph algorithms

{Alon Albalak, Koa Sato, Mayuresh Anand}

**Abstract**

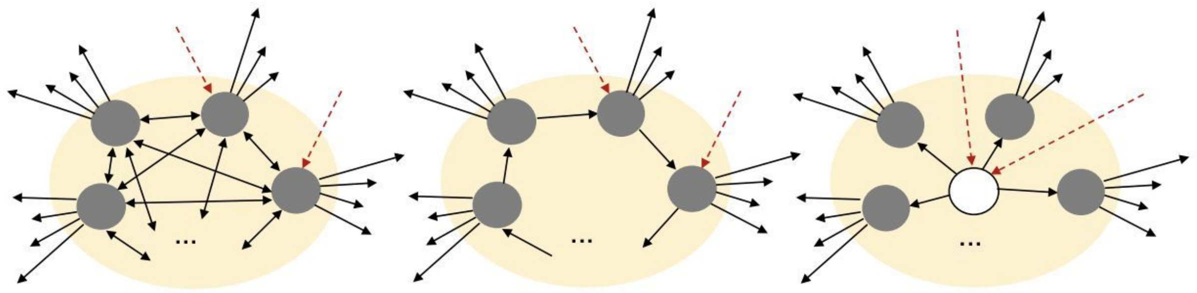
Graph algorithms such as BFS and SSSP process through a large number of vertices with uneven out-degree leading to inefficient use of GPU resources. If these graphs are regularized to a constant out-degree it is possible to optimize usage of GPU resources leading to efficient computation.

## Introduction

Graphical Processing Units (GPU) are composed of several cores which can run multiple threads. For efficient processing threads are organized into blocks which can be further divided into smaller collections of threads called warps. All threads in a warp work on the same piece of instruction and perform computation in tandem. To optimize scheduling a work one must consider the fact that warps must be scheduled such that there is very low waiting time among threads of same warp and also between different warps. This makes modern GPUs very efficient than CPUs as their highly parallel structure makes them more efficient at processing large blocks of data in parallel.

Various graph algorithms such as Breadth first search (BFS) and Single Source Shortest Path (SSSP) needs to process data at multiple vertices each with their own constraints. These vertices can have different outdegrees where some nodes may have a larger number of neighbours than others. Here, we can use GPU to process data for different paths from same or different vertices in parallel thereby reducing overall computation time. The issue that arises in parallel computing is that if some nodes have a very large degree than other nodes then a larger amount of computational resources is used by them. Also, as the distribution of work is not symmetric for vertices some warps may have larger waiting time than others leading to inefficient computation.

An idea that promises to resolve the issue is Graph regularization. We can regularize the graph by fixing the outdegree to a predetermined value (also called part-size) and to regularize a vertex we break it into (outdegree/partsize) many duplicate vertices and link the edges correspondingly. There can be many ways to orient the duplicate vertices some of which are shown below:



(a) $\mathcal{T}_{cliq}$ (b) $\mathcal{T}_{circ}$ (c) $\mathcal{T}_{star}$

The above methods of regularization suffer from the issue that they are not optimal in introducing new connecting edges. Hence, we use Uniform Degree Transformation to regularize the graph. This method is discussed further in section: **Methods**.

## Related Work

- **CuSha:** In Vertex-Centric Graph Processing on GPUs, Khorasani et al. [1] explain that although the compressed sparse row format of graphs is space efficient, it is very inefficient for graph analytics processing due to potentially irregular memory accesses. Especially for GPUs, where nearly 3 orders of magnitude can be seen in speed differences between on-chip register memory accesses and on-host memory accesses [3], memory access in a regular fashion is extremely important for efficiency. The authors offer 2 methods which improve graph analytics efficients, G-Shards and Concatenated Windows. For G-Shards, the authors draw inspiration from a general database method, sharding, which breaks a database into partitions, where each partition can be processed simultaneously without knowledge of the other partitions. Sharding allows for data to be processed in parallel. One of the major contributions of CuSha is that they create their own version of sharding which works well on GPUs. The second major contribution, Concatenate Windows, is a data representation which works synergistically with G-Shards to solve data locality issues inherent in sparse graphs.

- **Tigr:** Many approaches to improving graph analytics efficiency involve altering the graph programming extraction or the thread execution models. In **T**ransforming **I**rregular **GR**aphs for GPU-Friendly Graph Processing, Sabet et al. [2] approach the problem of efficient graph computation on GPU from a more general viewpoint. They pose the problem as an issue of graph irregularity and propose regularizing algorithms as the solution. Their best performing solution is a "virtual" graph transformation where they regularize a given graph, but do not alter the original representation. Instead, they create a virtual layer on top of the graph which is pointed to by the original graph. This allows their method to benefit from graph regularization without creating any new nodes in the actual structure.

## Contributions

We made contributions in the following ways. First, we implemented the uniform degree tree (UDT) transformation algorithm. This idea was presented in the TIGR paper that we read, but the publically available repository did not have any code to perform this sort of transformation. They instead had a virtual transformation which was not conducive to running with CuSha. With UDT, we came up with the idea of creating an intermediate representation for graphs. This type of transformation reduced irregularity which potentially gave improvements when running on GPUs, which we would then feed into an optimizer which was CuSha. Another contribution we made was that we had to change code in CuSha so that it would recognize the concept of virtual nodes. In UDT, virtual nodes are nodes that are created by the user to reduce irregularity for easier processing for graph algorithms. However, these nodes should always have a weight of 0 between themselves and the parent node that it spawned from. As a result, an algorithm like BFS had to be changed so that it would look at the weights of the graph to treat the distance between a parent node and a virtual node as 0 instead of 1. Lastly, we produced results which are explained later and analyzed as to why some instances of our method were successful and not as successful.

## Methods

**Algorithm 1** UDT Transformation

```
if degree($v$) > $K$ then                          ▷ for each high-degree node
    $q$ = new_queue()
    for each $v_n$ from v's neighbors do
        $q$.add($v_n$)                             ▷ add all original neighbors
        $v$.remove_neighbor($v_n$)
    while $q$.size() > $K$ do
        $v_n$ = new_node()
        for $i$ = 1..$K$ do
            $v_n$.add_neighbor($q$.pop())
        $q$.push($v_n$)                            ▷ add a new node
    $S$ = $q$.size()
    for i = 1..$S$ do                              ▷ connect to the root node
        $v$.add_neighbor($q$.pop())
```

The Uniform-Degree Tree transformation works by splitting only nodes whose outbound degree is greater than some degree bound, K. Given a graph, and a node whose outbound degree is greater than K, the UDT algorithm (algorithm 1) creates new nodes as needed, such that each new node will have exactly K outbound edges, and the original node contains the remaining edges.

UDT was selected due to some key properties. Firstly, the solution to any graph algorithm remains unchanged. The UDT transformation is in some ways optimal, for example, there exists a unique path from any incoming edge of an original node to an outgoing edge of a transformed node. This path uniqueness guarantees that we aren't altering solutions for any graph algorithm. For example, by setting the weight of any new edge to 0, we ensure that single-source shortest path maintains the correct original solution. Second, the number of additional nodes created increases only logarithmically with the size of the original node being split and guarantees that we have at most 1 node with degree less than the chosen degree bound. While increasing the number of nodes or edges is not an ideal situation, by using UDT rather than a circle, star, or clique transformation we are minimizing the additional computations.

## Experimentation

In our experiments, we ran all of the experiments on a system with configuration:

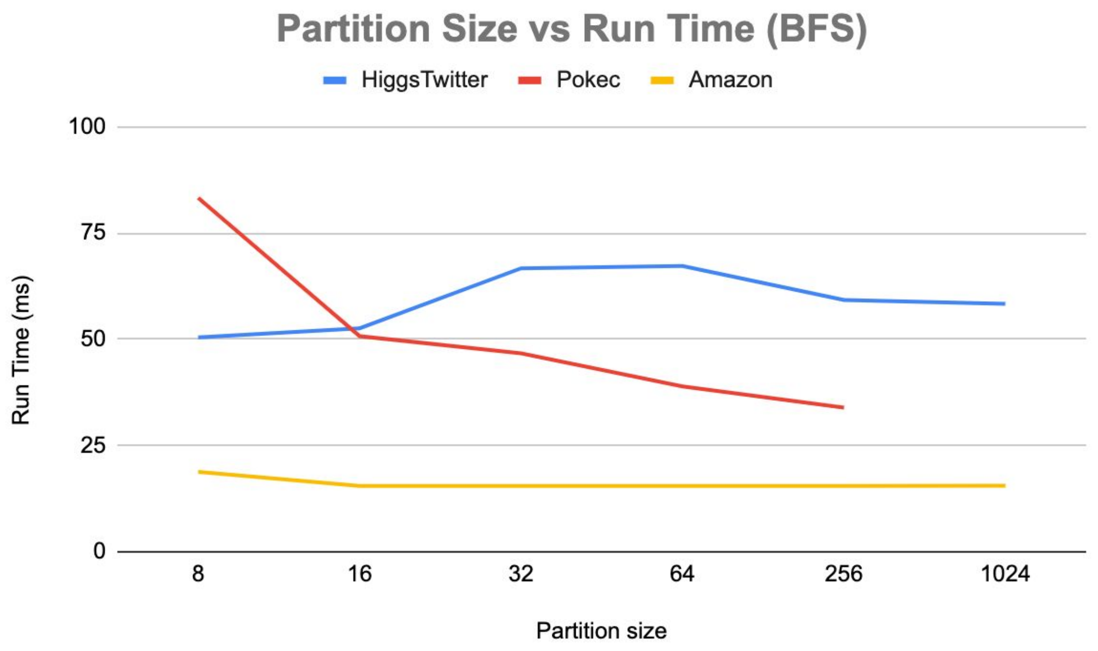
- i5 6600k @3.50 GHz,
- a GTX 1060 6GB,
- 16GB of memory,
- Ubuntu 18.04 with CUDA version 9.1.

We tested our method against three datasets:

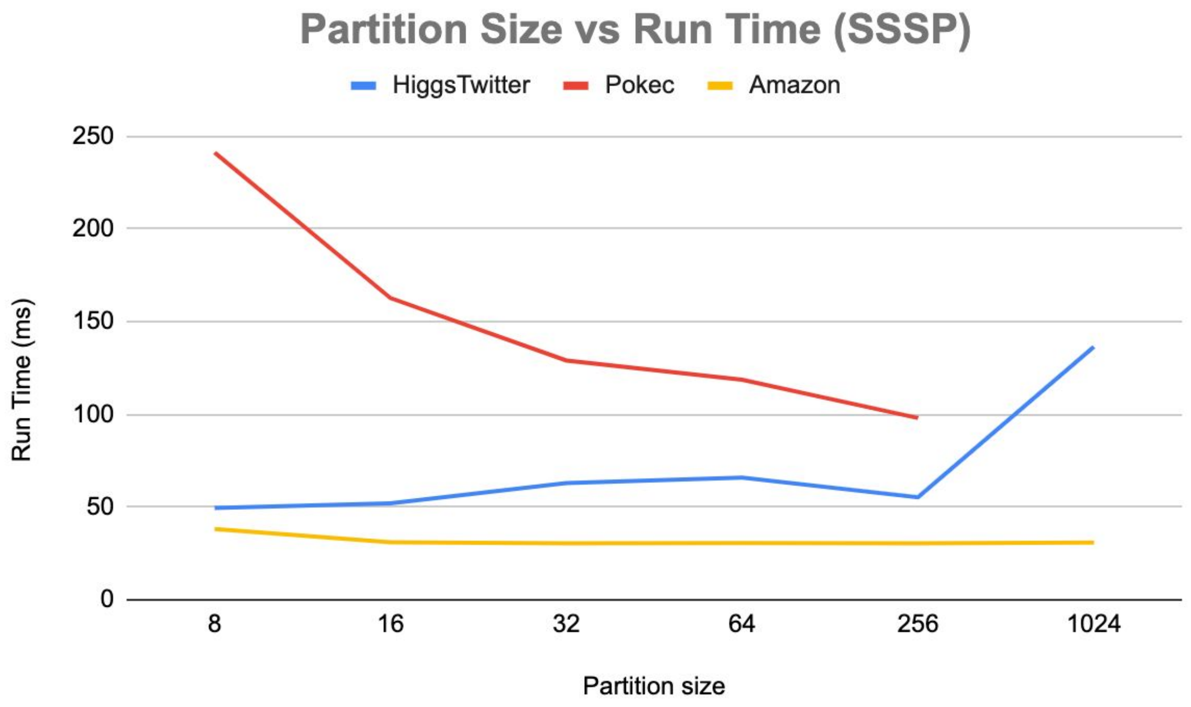
- **Higgs:** a Twitter social network graph with 456626 nodes and 14855842 edges
- **Pokec:** a graph from a social network in Slovakia with 1632803 nodes and 30622564 edges
- **Amazon:** an Amazon co-purchasing network from June 1, 2003 with 403394 nodes and 3387388 edges.

We also tested our method on all three datasets with varying part sizes of 8, 16, 32, 64, 256, and 1024. This way we were able to tell which parameters gave the best results and with which graphs.

Below is a visualization of run times for processing BFS on the three different datasets with varying part sizes. Note that because Pokec was such a large graph, we ran out of memory for transforming the graph.



Below is a similar graph, but with running the SSSP algorithm, which uses edge weights rather than the number of edge links like in BFS.



Below is a table of some run times required to process SSSP on the Higgs social network graph. Note that our times with almost all part sizes were better than the CuSha baseline, but not as good as Tigr's. Also, as our part size increases, we reach a time closer to that of CuSha, which makes intuitive sense since a larger part size means we did less to regularize the graph.

| Twitter | UDT-CuSha | UDT-CuSha | UDT-CuSha | UDT-CuSha | UDT-CuSha | UDT-CuSha | CuSha | Tigr |
|---|---|---|---|---|---|---|---|---|
| Part Size | 8 | 16 | 32 | 64 | 256 | 1024 | N/A | 8 |
| Time (ms) | **49.4730** | **52.0758** | **62.9978** | **65.9644** | **55.3284** | 136.479 | 132.789 | 6.7826 |

Below is another table of some run times to process SSSP, but on the Amazon graph. Note how our times actually get better as we increase part size rather than decrease part size like previously. Also note how our time gets closer to the run time of CuSha as we increase part size, similar to before, which adds to our belief that decreasing part size decreases our intervention in transforming the graph.

| Amazon | UDT-CuSha | UDT-CuSha | UDT-CuSha | UDT-CuSha | UDT-CuSha | UDT-CuSha | CuSha | Tigr |
|---|---|---|---|---|---|---|---|---|
| Part Size | 8 | 16 | 32 | 64 | 256 | 1024 | N/A | 8 |
| Time (ms) | 38.1308 | 30.966 | **30.4466** | 30.5708 | **30.4798** | 30.8584 | 30.5586 | 7.182 |

To add to why our method has different results on different graphs, we looked at the structures of the graphs. On the Higgs dataset, we had much better performance than on Amazon. We can possibly contribute this to the fact that Higgs had an average of about 32 edges per node and the Amazon dataset had an average of 4 edges per node, despite both graphs having about 400000 nodes each. This points out the possibility that our method only works on graphs that have a much larger irregularity. Also, the clustering coefficient, or the degree to which nodes are clustered together, is larger on the Amazon graph, at a value of 0.4198, compared to the value of 0.1887 of the Higgs dataset. Both metrics seem to indicate that some amount of graph analysis is required before deciding if a UDT transformation is necessary.

## Conclusion and Future Work

We presented an idea of performing a transformation of a graph and using it as an intermediate representation which would then be processed on another GPU graph optimizer. By utilizing UDT algorithm, we performed an efficient graph transformation which was shown to improve performance on GPUs. Then, we performed experiments using CuSha and our intermediate representation. Our results showed that the structure of the graph matters the most when performing a graph optimization.

In future work, we would hope to continue to explore the idea of graph transformation for optimization on GPUs by performing preliminary analysis to decide if other types of graph transformations would better suit the graph. This type of dynamic analysis would be something to look into as well as performing our method on many more types of graphs with different structures and clustering coefficients.

## REFERENCES:

[1] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. CuSha: Vertex-centric graph processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 239--252, June 2014.

[2] A. H. Nodehi Sabet, J. Qiu, Z. Zhao, "Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing", *Proc. ASPLOS*, 2018.

[3] Farber, Robert. (2011). CUDA Application Design and Development. 10.1016/C2010-0-69090-0. Chapter 6 Efficiently Using GPU Memory.