Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing*

Amir Hossein Nodehi Sabet
University of California, Riverside
anode001@ucr.edu

Junqiao Qiu
University of California, Riverside
jqiu004@ucr.edu

Zhijia Zhao
University of California, Riverside
zhijia@cs.ucr.edu

Abstract

Graph analytics delivers deep knowledge by processing large volumes of highly connected data. In real-world graphs, the degree distribution tends to follow the power law – a small portion of nodes own a large number of neighbors. The *high irregularity of degree distribution* acts as a major barrier to their efficient processing on GPU architectures, which are primarily designed for accelerating computations on regular data with SIMD executions.

Existing solutions to the inefficiency of GPU-based graph analytics either modify the graph programming abstraction or rely on changes to the low-level thread execution models. The former requires more programming efforts for designing and maintaining graph frameworks; while the latter couples with the underlying architectures, making it difficult to adapt as architectures quickly evolve.

Unlike prior efforts, this work proposes to address the above fundamental problem at its origin – the *irregular graph data* itself. It raises a critical question in irregular graph processing: *Is it possible to transform irregular graphs into more regular ones such that the graphs can be processed more efficiently on GPU-like architectures, yet still producing the same results?* Inspired by the question, this work introduces *Tigr* – a graph transformation framework that can effectively reduce the irregularity of real-world graphs with correctness guarantees for a wide range of graph analytics. To make the transformations practical, *Tigr* features a lightweight *virtual transformation* scheme, which can substantially reduce the costs of graph transformations, while preserving the benefits of reduced irregularity. Evaluation on *Tigr*-based GPU graph processing shows significant and consistent speedup over the state-of-the-art GPU graph processing frameworks for several graph algorithms on a spectrum of irregular graphs.

*Here, *irregularity* refers to the power-law degree distribution.

ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173180

CCS Concepts • **Computing methodologies → Parallel computing methodologies**;

Keywords Vertex-Centric Graph Processing, Irregularity, Power-law Graph, Graph Transformation, GPU, SIMD

ACM Reference Format:
Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems (ASPLOS '18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3173162.3173180

1 Introduction

Graph analytics is fundamental in unlocking key insights by mining large volumes of highly connected data. Unlike the traditional analytics based on "one-to-one" or "one-to-many" relationships, graph analytics allows more complex reasoning by exploring "many-to-many" relationships, such as identifying influencers in social networks [46], spotting frauds in bank transactions [57], optimizing supply chain distribution [66], and developing recommendations [12] and more effective medical treatments [7]. There is a growing need for accelerating graph analytics by taking advantages of modern parallel architectures.

Packed with up to thousands of computing units, GPUs have emerged as an attractive computing platform for large graph processing. Recent work [32] has shown orders of magnitude efficiency improvement over traditional CPU-based graph processing, such as GraphLab [37]. Despite the promise, existing GPU-based graph processing suffers from low efficiency due to the highly irregular degree distribution in real-world graphs. By nature, the degree distribution of real-world graphs tend to follow the *power law* (known as *power-law graphs*) – a small portion of nodes [1] own a large number of neighbors (i.e., one-hop nodes) while most nodes are connected to only a few neighbors. Such a highly skewed degree distribution makes these graphs ill-suited to GPUs' single-instruction multiple-data (SIMD) execution, which is primarily designed for accelerating computations with more regular data structures [39].

In the popular vertex-centric graph programming where the nodes of a graph are distributed across threads for processing, graph irregularity results in severe load imbalance among threads. On GPU architectures, threads are organized in *warps* and threads in the same warp proceed in an SIMD execution fashion – threads that finish their tasks earlier

[1]We use node and vertex interchangeably in the context of graph structures.

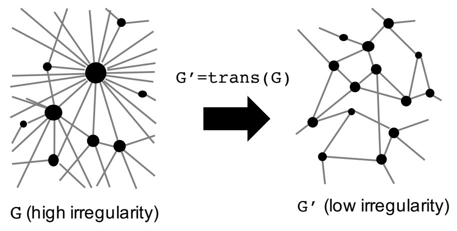


Figure 1. Illustration of Graph Irregularity Reduction.

have to wait until other threads in the same warp finish their computations, before swapping in the next warp of threads. In this case, the load imbalance among threads can lead to inefficiencies at both intra-warp and inter-warp levels. As a result, the GPU utilization drops to merely 25.3%-39.4% for commonly used graph analytics [32].

State of The Art. To address the above barrier, research so far either modifies GPU thread execution models [20, 23, 30] or changes the graph programming paradigm [16, 31, 32, 69]. For instance, warp segmentation [30] and maximum warp [23] improve the GPU efficiency by decomposing a warp of threads into a group of sub-warps. With enhanced flexibility, changes like these tightly couple with underlying GPU architectures, making them harder to adapt as GPU architectures quickly evolve. By contrast, CuSha [32] and Gunrock [69] propose new graph representations and new programming abstractions, respectively, which often require extra programming efforts to adopt.

Different from prior efforts, this work proposes to address the irregularity issue at its origin by *transforming irregular graphs into more regular ones*, namely ***Tigr***, as illustrated in Figure 1. Note that this is radically different from changing graph representations [32, 36] (e.g., CSR to CSR5 format) or partitioning graphs [17, 37]. *Tigr* allows to change the topology of a graph (i.e., structural transformations) while does not generate any graph partitions, thus there is no need for explicit partition synchronization.

To achieve this goal, there are three key challenges:

- *Effectiveness*. How should graphs be transformed such that their irregularity can be effectively reduced?
- *Correctness*. How can we ensure that the processing of transformed graphs still yields the same results?
- *Efficiency*. How can we minimize the transformation cost while preserving its effectiveness?

First, to reduce the graph irregularity, this work proposes a class of structural transformations based on a simple yet effective idea – *node splitting*. Basically, the transformations first identify nodes with high degrees, then iteratively split the nodes until their degrees reach a predefined limit. We refer to these transformations as *split transformations*.

The design complexity of split transformations lies in the *connection* among the split nodes. Different connections may lead to different extent of irregularity reduction. Even more complex, they may affect the convergence rate of graph analytics and alter the final results. In general, there exists a basic tradeoff between graph irregularity reduction and the convergence rate of graph algorithms. More importantly, this work identifies a promising type of split transformations that is able to achieve a good balance between irregularity reduction and convergence speed, while preserving the result correctness for a wide range of graph algorithms, called *uniform-degree tree transformation* or *UDT*.

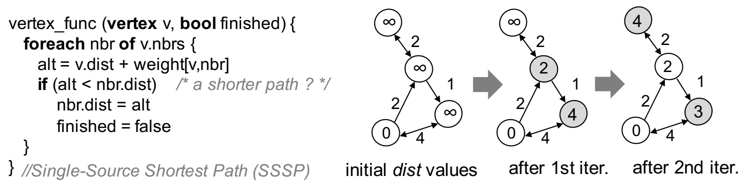
As the name suggests, UDT transforms a high-degree node into a tree structure with nodes of identical degrees. This special design leads to two important properties. First, it ensures that the distances (i.e., #hops) among split nodes only increase logarithmically as the degree of the to-split node increases. This minimizes the negative impact of split transformations on the convergence rate of graph algorithms. Second, it preserves basic graph properties, like *connectivity*, *paths*, and *degrees*, which in turn supports the correctness of UDT for a variety of graph analytics.

Physically transforming graphs may incur substantial costs in time and space. To address it, this work proposes *virtual split transformations*, which add a virtual layer on top of the original irregular graph, making it "look more regular". Essentially, virtualization separates programming abstraction from the physical graph. The separation allows computation tasks to be scheduled at the virtual layer (on the transformed graph) while the actual value propagation is carried at the physical layer (on the original graph). In this way, it eliminates the needs of physical graph transformations while preserving the benefits of reduced graph irregularity. Moreover, the virtual transformation simplifies the correctness enforcement by preserving the original value propagation pattern at the physical layer.

Finally, we integrate the proposed graph transformations *Tigr* into a lightweight GPU graph processing framework. Thanks to the data-level transformations, its code base is much smaller than other GPU graph processing solutions. Evaluation on six important graph analytics confirms the effectiveness and efficiency of the proposed transformations with substantial speedups over existing solutions.

Contributions. This work makes a four-fold contribution.

- This work directly targets the irregular graph data for addressing the fundamental efficiency issue in irregular graph processing, complementary to existing techniques.
- It proposes a class of novel structural transformations that can effectively reduce the irregularity of real-world graphs while guaranteeing the correctness.
- To make the graph transformations practical, it designs a *virtual transformation* scheme, which eliminates the needs of expensive physical graph transformations.
- Finally, it implements the proposed transformations and compares with the state-of-the-art GPU graph processing frameworks. The results confirm both the effectiveness and correctness of the transformations. (Github repository: https://github.com/amirnodehi/Tigr)



```
vertex_func (vertex v, bool finished) {
  foreach nbr of v.nbrs {
    alt = v.dist + weight[v,nbr]
    if (alt < nbr.dist)   /* a shorter path ? */
      nbr.dist = alt
      finished = false
  }
} //Single-Source Shortest Path (SSSP)
```

Figure 2. Example of (Push-based) Vertex-Centric Programming.

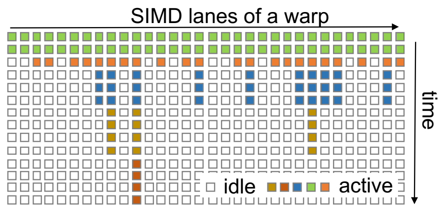


Figure 3. SIMD Execution.

2 Background and Motivation

This section briefly introduces the parallel programming model for graph analyses, the SIMD execution model on GPU architectures, and the special challenges for GPU-based graph processing.

2.1 Vertex-Centric Programming

Graph analytics is notoriously difficult to parallelize due to its inherent dependencies [39]. In response to the challenge, *vertex-centric programming model* has quickly established its popularity in recent years for its simplicity, high scalability and strong expressiveness. Since implemented by Google Pregel [41], this model has been widely adopted by many parallel graph engines, including Apache Giraph [1], GraphLab [37], PowerGraph [17], MaxWarp [23], CuSha [32], and many others. The model is based on a simple paradigm "thinking like a vertex" – computations are defined from the view of a vertex rather than a graph. In specific, a vertex function is first defined, and then applied on each vertex. Based on the Bulk Synchronous Parallel (BSP) model [65], computations of different vertices are synchronized at the graph level, iteration by iteration, until a certain number of iterations or a convergence property is met.

Example. Figure 2 provides a vertex-centric programming example for finding the shortest path from the source node to the other nodes iteratively. Initially, each node in the graph has an infinite distance to the source node (`dist=∞`), except the source node (`dist=0`). By invoking the vertex function `vertex_func`, each node attempts to update its neighbors' distance values, based on its own value from the last iteration and the distances to its neighbors (`weight[v,nbr]`). The updates of different nodes are synchronized iteration by iteration. The whole computation halts when all node values stop changing – the algorithm converges.

In the above scheme, node values are propagated by updating neighbors' values through outgoing edges. This scheme is known as *push*-based. By contrast, the node values can also be propagated by gathering values from neighbors through incoming edges and updating the node's own value, which is known as *pull*-based. Both schemes have been used by some prior graph frameworks. In the following, we assume a *push*-based vertex-centric programming on *directed* graphs, but similar ideas can also be applied to pull-based scheme and undirected graphs – which actually are special cases of directed graphs with each edge having both directions.

2.2 GPU and SIMD Execution

On GPU architectures, computing units (i.e., GPU cores) are organized by a number of streaming multiprocessors (SM). Typically, GPU applications distribute computation tasks to thousands of parallel threads. These threads are grouped into *warps*. In NVIDIA's GPU architectures, a warp typically contains 32 threads. Threads in the same warp are assigned to a single SM, and proceed in an SIMD fashion [2]. That is, threads execute the same instructions (or nothing), but on different data. Following the SIMD execution model, even though some threads have finished their computations earlier, their occupied computing units (also called *SIMD lanes*) cannot be released for other computations, until all the threads in the warp have finished, as illustrated in Figure 3.

Though offering massive threads for parallel executions, whether the tremendous computing power of GPUs can be utilized effectively depends on the computation regularity.

2.3 Challenges of GPU-based Graph Processing

Real-world graphs, like social networks and the web, are highly irregular. For example, a basic graph characteristic profiling on three popular real-world graphs (`LiveJournal`, `Higgs Twitter` [35], and `Hollywood` [5]) reveals that over 90% of nodes have degrees less than 20 while less than 2% of nodes have degrees around 1000, up to 14,000.

The high irregularity in real-world graphs poses a major challenge to efficiently utilizing GPU's processing power for many graph analytics. In vertex-centric graph programming, each node communicates with its neighbors to update their values (see Section 2.1). The higher number of neighbors a node has, the more computations it has to perform. When mapping nodes to GPU threads, a highly biased node degree distribution would lead to severe load imbalance across GPU threads. At intra-warp level, some threads may finish earlier, leaving their SIMD lanes idle. At inter-warp level, this leads to some GPU SMs underused while others being busy.

Next, we will describe how to address this basic issue with graph transformations.

3 Graph Transformations

This section first introduces the general ideas of a class of novel structural transformations – *split transformation*, then focuses on a promising type of split transformation with desired properties – *uniform-degree tree transformation*.

[2]Single-instruction multi-thread (SIMT) model in NIVIDA's term.

3.1 Split Transformations

Graph irregularity can be reflected by the highly skewed node degree distribution. To reduce such irregularity, we consider transforming nodes with high degrees into sets of nodes with lower degrees. In (push-based) vertex-centric programming (Section 2.1), values are propagated through outgoing edges. Therefore, we focus on the *outdegree* – the number of outgoing edges of a node [3]. For simplicity, we refer to *outdegree* as *degree*, unless otherwise noted. Formally, we define high-degree nodes in a graph as follows.

Definition 1. *Given a directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, and a predefined degree threshold K ($K \geq 1$), Node v, $v \in V$, is a **high-degree node** if and only if it has an outgoing degree $d(v)$ such that $d(v) > K$. The threshold K is called **degree bound**.*

To transform high-degree nodes, our strategy is to split each high-degree node into a set of split nodes and evenly distribute its (outgoing) edges to some split nodes based on the degree bound K, as illustrated in Figure 4. We refer to this process as *split transformation*. Assume the neighbor set of a node v via outgoing edges is denoted as N_v. Similarly, the neighbor set of node set S is denoted as N_S, $N_S = \cup_{v \in S} N_v$. Then, split transformation can be formally defined as below.

Definition 2. *Given a high-degree node v and the degree bound K, a **split transformation** of node v is a mapping*

$$\mathcal{T} : (v, N_v) \mapsto (I \cup B, N_I \cup N_B) \quad (1)$$

*where (i) I is the **internal split node set** i.e., $N_I \cap N_v = \emptyset$; (ii) B is the **boundary split node set** i.e., $B = \{v' | N_{v'} \cap N_v \neq \emptyset\}$; (iii) $N_B \supseteq N_v$ and $|B| = \lceil |N_v|/K \rceil$.*

Condition (iii) ensures the original outgoing edges are evenly distributed based on degree bound K. Together, we refer to $I \cup B$ as a ***family***. The *degree of a family* equals to the highest degree of all nodes in the family. Different families form disjoint sets of nodes. For a split node with degree less than K, we name it a ***residual node***.

Though the basic idea of split transformation is intuitive, the concrete designs of splitting is non-trivial, due to the complexities in *connecting the split nodes*, that is, designing internal split node set I and its outgoing neighbor set N_I.

Design Tradeoffs. In general, there are various *topologies* to connect the split nodes of a family. We illustrate the tradeoffs in designing connection topologies with three representative transformations that are based on a clique connection ($\mathcal{T}_{\text{cliq}}$), a circular connection ($\mathcal{T}_{\text{circ}}$), and a star-shaped connection with a "hub" node ($\mathcal{T}_{\text{star}}$), respectively (see Figure 5). For $\mathcal{T}_{\text{cliq}}$ and $\mathcal{T}_{\text{circ}}$, the incoming edges of the original node (red dashed arrows) are randomly assigned to the split nodes; For $\mathcal{T}_{\text{star}}$, the incoming edges all connect to the hub node.

Table 1 summaries the impacts of the three designs on graph size, degree, and the maximum number of hops to propagate a value to the split nodes. In terms of graph size,

[3]Similarly, *indegree* should be used in a pull-based scheme.

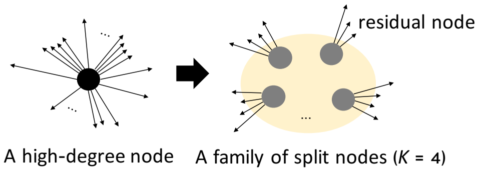


Figure 4. Illustration of Split Transformation.

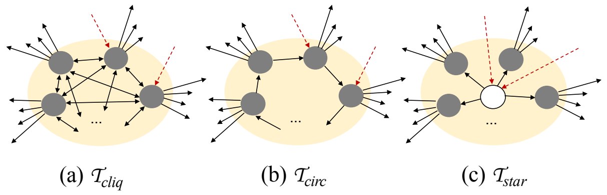


Figure 5. Three Example Connections.

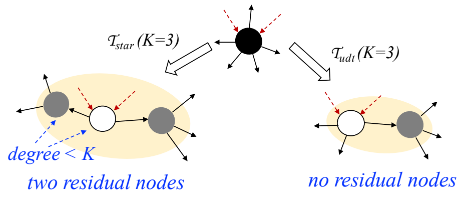


Figure 6. Comparison between $\mathcal{T}_{\text{star}}$ and $\mathcal{T}_{\text{udt}}$.

$\mathcal{T}_{\text{cliq}}$ introduces the highest space cost, for its quadratic increase of extra edges. As to the irregularity reduction, $\mathcal{T}_{\text{circ}}$ wins – the new degree only depends on degree bound K. Besides space cost and irregularity reduction, a less obvious yet critical effect of split transformation is its influence on value propagation speed, that is, how fast node values are propagated through the graph. This directly affects the convergence rate of graph algorithms. The influence can be estimated by the maximum number of hops needed to propagate a value within a family. As shown in the fifth column of Table 1, $\mathcal{T}_{\text{circ}}$ performs the worst as it needs up to $\lceil d/K \rceil - 1$ hops to propagate a value from one split node (with an incoming edge) to another. By contrast, $\mathcal{T}_{\text{cliq}}$ and $\mathcal{T}_{\text{star}}$ only need one hop to cover all the split nodes.

The above analysis indicates *a general tradeoff among space cost, irregularity reduction, and value propagation rate*. Weighing the advantages and disadvantages, $\mathcal{T}_{\text{star}}$ shows relative superiority for its low space cost and fast value propagation. The only downside is the relatively high family degree caused by the adding of the hub node. Next, we show a promising type of split transformation that shares benefits with $\mathcal{T}_{\text{star}}$ while without the hub node issue.

3.2 Uniform-Degree Tree (UDT) Transformations

One straightforward solution to the hub node issue of $\mathcal{T}_{\text{star}}$ is recursively applying $\mathcal{T}_{\text{star}}$ to the hub node until its degree drops to K. As a consequence of the recursive splitting, a hierarchy of families would be created, where the height of the hierarchy equals to the depth of the recursion. However, this recursive $\mathcal{T}_{\text{star}}$ may introduce more residual nodes, as

Table 1. Properties of Split Transformations (K: degree bound; d: degree of original high-degree node).

	#new nodes	#new edges	new degree	max #hops	space cost	irreg. reduction	value prop.
$\mathcal{T}_{\text{cliq}}$	$\lceil d/K \rceil - 1$	$(\lceil d/K \rceil - 1) \cdot \lceil d/K \rceil$	$K + \lceil d/K \rceil - 1$	1	high	low	fast
$\mathcal{T}_{\text{circ}}$	$\lceil d/K \rceil - 1$	$\lceil d/K \rceil - 1$	$K + 1$	$\lceil d/K \rceil - 1$	low	high	slow
$\mathcal{T}_{\text{star}}$	$\lceil d/K \rceil$	$\lceil d/K \rceil$	$\max\{K + 1, \lceil d/K \rceil\}$	1	low	varies	fast

shown in Figure 6-(a). Applying $\mathcal{T}_{\text{star}}$ to a node of degree five results in two residual nodes. Situations like this not only compromise the irregularity reduction, but also introduce unnecessary split nodes. To avoid such issues, we propose another transformation scheme, called *uniform-degree tree transformation*, or *UDT* ($\mathcal{T}_{\text{udt}}$), which ensures at most one residual node in the generated family.

Algorithm 1 illustrates the how UDT works. Instead of creating a hub node at each splitting, UDT introduces new split nodes on demands. This is achieved by maintaining a queue of split nodes to connect. Initially, the queue contains all neighbors of the high-degree node. If the queue has more than K (degree bound) nodes, a new node is created and connected to K nodes popped from the queue. After that, the new node is appended to the queue. This process iterates until the queue has no more than K nodes. The remaining ones are assigned to the original node.

Figure 6-(b) shows a UDT example on a node of degree five. After the transformation, the new structure has no residual nodes, comparing to the two residual nodes in $\mathcal{T}_{\text{star}}$.

Properties of UDT. The output of Algorithm 1 forms a tree structure where the degree of each node (or except the root) equals to K. We refer to this tree structure as *uniform-degree tree*, hence the name of UDT transformation.

Besides the uniform degree property, UDT also features the following important properties:

- **P1**: UDT is a type of split transformation (Definition 2).
- **P2**: After the transformation, there exits a unique path connecting the incoming edges of the original node to each of its outgoing edges. Because (i) the original node with all incoming edges becomes the tree root (see Lines 12-13 in Algorithm 1) and (ii) each outgoing edge of the original node is only connected to one node in the tree (i.e., pushed once into the queue at Line 4 in Algorithm 1).
- **P3**: The number of hops to propagate a value through the split nodes (i.e., tree height) only increases logarithmically $O(log_K d)$ to the degree of the original node d.

Since the transformation at most traverses each node and each edge once, the time complexity of UDT for the entire graph is linear to the graph size $O(|V| + |E|)$.

Similar to the side effects of other split transformations, UDT increases the size of the graph. However, our analysis indicates that, with the benefits of reducing degree d to a constant K, the graph size only increases linearly $O(d/K)$ in terms of both nodes and edges. As to the graph diameter D, the increase is at most $O(D \cdot log_K(|E|/d))$ [4].

[4]For space limit, the details will be available on the project website.

Algorithm 1 UDT Transformation

```
1:  if degree($v$) > $K$ then                         ▷ for each high-degree node
2:      $q$ = new_queue()
3:      for each $v_n$ from v's neighbors do
4:          $q$.add($v_n$)                             ▷ add all original neighbors
5:          $v$.remove_neighbor($v_n$)
6:      while $q$.size() > $K$ do
7:          $v_n$ = new_node()
8:          for $i$ = 1..$K$ do
9:              $v_n$.add_neighbor($q$.pop())
10:         $q$.push($v_n$)                            ▷ add a new node
11:     $S$ = $q$.size()
12:     for i = 1..$S$ do                              ▷ connect to the root node
13:         $v$.add_neighbor($q$.pop())
```

Next, we discuss how UDT can preserve the correctness for a diverse set of graph algorithms.

3.3 Enforcing the Correctness for $\mathcal{T}_{\text{udt}}$

As discussed above, UDT, like other split transformations, may substantially change the structure of the original graph. In this case, *will graph analyses still yield the same results as processing on the original graphs? If not, how can we enforce the correctness for this type of transformations?*

It is obvious that the correctness of UDT depends on graph analyses, in particular, the graph properties that various graph analyses rely on. Hence, instead of discussing the correctness for each graph analysis, we first present the important graph properties that UDT preserves. Base on that, we can infer what kinds of graph algorithms can yield correct results and what cannot.

We define a path $P(v_i, v_j)$ as the set of edges on the path from node v_i to node v_j.

Theorem 1. *Given a graph G(V, E), let v_1, $v_2 \in V$, then there exists a path $P(v_1, v_2)$ in G iff there exists a path $P'(v_1, v_2)$ in the UDT-transformed graph G'. Furthermore,*

$$P'(v_1, v_2) = P(v_1, v_2) \cup E_{new} \tag{2}$$

where E_{new} is a set of new edges, that is, $E_{new} \cap E = \varnothing$.

Proof. If none of the nodes on original path $P(v_1, v_2)$ are high-degree nodes, then $P'(v_1, v_2) = P(v_1, v_2)$ and $E_{new} = \varnothing$. Otherwise, assume p_i is a high-degree node, then p_i will be transformed into a uniform-degree tree, as illustrated in Figure 7. Assume p_{i-1} and p_{i+1} are the nodes before and after p_i on path $P(v_1, v_2)$, then based on the **P2** property of UDT, there exists a unique path from p_{i-1} to p_{i+1}. Assume p_{i-1} and p_{i+1} connect to m and n in the tree, respectively. Then we have $P'(v_1, v_2) = P(v_1, v_2) \cup E_{new}$, where $E_{new} = P'(m, n)$. On the other hand, by removing the edges in $P'(m, n)$ from

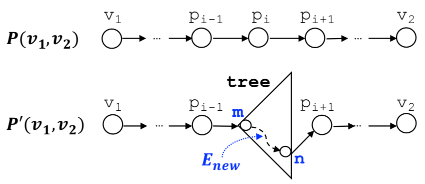


Figure 7. A Path Before and After UDT.

$P'(v_1, v_2)$, we can recover the original path $P(v_1, v_2)$, except a node notation difference, that is, edges $p_{i-1} \rightarrow p_i$ and $p_i \rightarrow p_{i+1}$ become $p_{i-1} \rightarrow m$ and $n \rightarrow p_{i+1}$. □

Based on Theorem 1, we have three corollaries.

Corollary 1. *UDT preserves graph connectivity.*

Proof. By the definition of connectivity and Theorem 1. □

Corollary 1 ensures the correctness of UDT for connected components (CC), by preserving both the inter and intra connectivities of all the connected components in a graph.

Dumb Weights. For some graph analyses, like finding the shortest path, the calculation also involves the edge weights. Here, we show that, by carefully assigning weights to the newly introduced edges, UDT can preserve some even more interesting graph properties.

The key to the weight assignment is to make *the new edges contribute nothing to the calculation*. We can achieve this by assigning "*dumb weights*" to the new edges. We next present two such cases (Corollary 2 and Corollary 2).

Corollary 2. *UDT preserves the distance between any pair of nodes in a weighted graph by assigning weight zero to all UDT-introduced edges.*

Proof. See Equation 2 in Theorem 1. By assigning weight zero to all edges in E_{new}, $P'(v_1, v_2)$ and $P(v_1, v_2)$ will have the same total weight. By preserving the total weight on every path, the distances between pairs of nodes remain. □

According to Corollary 2, it is easy to find that UDT can preserve the results for single-source shortest path (SSSP) and between centrality (BC), for which the calculations are only based on the distances between node pairs. Since breath-first search (BFS) is equivalent to SSSP on graphs with all edge weights of 1, UDT can also preserve the results for BFS.

Figure 8 shows the UDT with dumb weights for SSSP. The distance between A and B remain six after transformation.

Corollary 3. *UDT preserves the minimal edge weight in a path by assigning weight infinity to all UDT-introduced edges.*

Proof. See Equation 2 in Theorem 1. By assigning weight infinity to all edges in E_{new}, $P'(v_1, v_2)$ and $P(v_1, v_2)$ will have the same minimal edge weight. □

Corollary 3 confirms that UDT can preserve the results for single-source widest path (SSWP), for which the calculation is purely based on the minimal edge weight along a path.

Finally, we have a corollary for *degree*-based analyses.

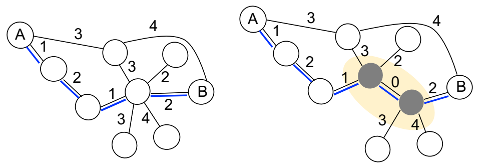


Figure 8. Example of UDT with dumb weights for SSSP.

Corollary 4. *For push-based and pull-based vertex-centric programming, UDT preserves the indegrees and outdegrees, respectively, for all nodes in the original graph.*

Proof. By definition, UDT keeps all the incoming edges of the original node unchanged, as values are only propagated along the outgoing edges in a push-based scheme. Similarly, for a pull-based scheme, UDT keeps the outgoing edges of the original node unchanged. □

Corollary 4 ensures the correctness for graph analyses that rely on indegrees or outdegrees for node value calculation, such as PageRank (PR). Since PR depends on outdegrees only, its correctness can be ensured by using a pull-based vertex-centric programming model.

Applicability Discussion. Together, UDT can preserve the correctness for a spectrum of *connectivity-based*, *path-based*, and *degree-based* graph analyses, including the widely used CC, SSSP, SSWP, BC, BFS, and PR.

Despite the promises, there are graph analyses for which UDT or other split transformations may fail to preserve the results. These include analyses that require preserving the *neighborhood* of nodes, such as graph coloring (GC), triangle counting (TC), clique detection (CD), and some others. By checking the graph property requirements, the applicability of UDT or other split transformations for a specific graph analysis can be determined.

Note that physically transforming irregular graphs takes extra time and space. Furthermore, the transformed graphs may take more iterations to process due to the slowdown of value propagation (caused by splitting). To address these issues, we propose to *virtually* apply split transformations, without physically changing the graphs.

4 Enabling Virtual Graph Transformations

This section discusses how to apply the split transformations *virtually*, such that the benefits of physical split transformation – reduced graph irregularity – can be preserved, while without suffering from its practical issues.

4.1 Virtual Split Transformations

To avoid physical graph transformations, we propose to add a *virtual layer* on top of the original graph (*physical layer*), then perform split transformations only at the virtual layer, leaving the original graph intact, as shown in Figure 9. We refer to this scheme as *virtual split transformation*. The nodes at physical and virtual layers are called *physical nodes* and *virtual nodes*, respectively.

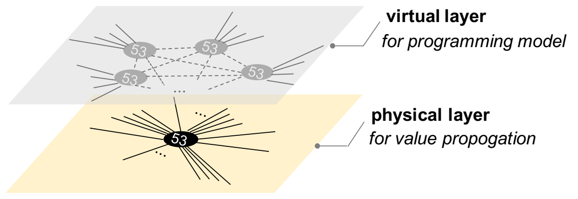


Figure 9. Illustration of Virtual Split Transformation.

The key to virtual split transformation is to *separate the programming model from the physical graph data*:

- First, by exposing the virtual layer to the vertex-centric programming model, node value computation tasks are scheduled at the virtual layer.
- Second, computed (virtual) node values are propagated at the physical layer, hidden from the programming model.

From the view of vertex-centric programming model, the graph has been transformed and become more regular; while physically, it is still the original irregular graph.

Next, we discuss the design of virtual split transformation and explain how it ensures the correctness.

Virtualization Design. Essentially, virtualization is about constructing a mapping between the physical layer and the virtual layer. In the context of graph virtualization, it needs to define a *node mapping* map_v and a *edge mapping* map_e.

$$v = map_v(v'), e = map_e(e')$$

In fact, split transformations do not specify the edge assignment from a high-degree node to the split nodes, except that the edges should be distributed evenly (Section 3.1). This flexibility allows edge mapping to be implicitly defined based on the node mapping, the order of edges in the storage, and the degree bound K (see an example shortly). That is, a node mapping itself is sufficient to define the virtualization.

Depending on when the node mapping is generated, we propose two alternative virtualization designs: *virtual node array* and *on-the-fly mapping reasoning*.

- *Virtual Node Array*. This design creates a node mapping before graph processing and store it in a structural array, namely *virtual node array*. Each element in the array is a structure of two nodes $\{v, v'\}$, representing a mapping between physical node v and virtual node v'. This array can be effectively integrated into the popular compressed sparse row (CSR) graph representation. See the example in Figure 10, a high-degree node v_2 is split into two virtual nodes v_2' and v_2''. Node v_2's original edges to nodes v_5, v_4 and v_6 are *implicitly* mapped to virtual node v_2' based on their order in the edge array and the setting of K (i.e., 3), the rest are mapped to virtual node v_2'' (i.e., edge mapping). Note that any incoming edges to the original node (v_2) would be shared by split nodes (v_2' and v_2'').
  The space cost of virtual node array is bounded by the number of virtual nodes and controllable by tuning the degree bound K (more details in Section 6).

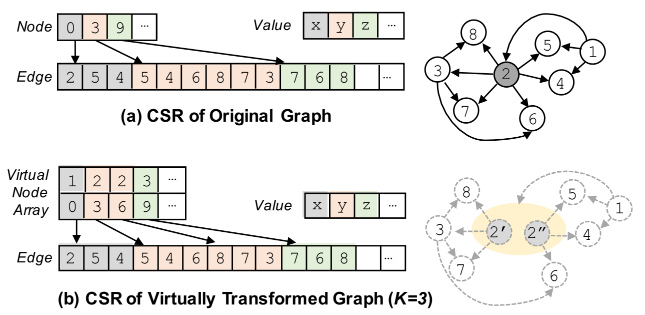


Figure 10. Integrating Virtual Node Array into CSR Format.

- *Dynamic Mapping Reasoning*. In certain scenarios, even allocating a little extra memory is undesirable. In this case, we can dynamically compute the mapping based on the node splitting logic (i.e., degree bound K). See the example in Figure 10. Before processing node v_2, a reasoning runtime finds its degree is 6, which is greater than K, hence splits it into two virtual nodes ($\lceil 6/3 \rceil$), each with three edges of v_2. In this way, we determine the node mapping dynamically, eliminating the needs for storing a mapping. Essentially, this design trades off computation cost for better memory efficiency.

As shown above, virtual split transformations are more lightweight compared to physical graph transformations. Next, we discuss how (virtual) nodes' values are propagated after the virtual split transformation.

Implicit Value Synchronization. As mentioned earlier, with virtualization, node values are propagated at the physical layer (i.e., on the original graph). Consider the *virtual node array* design [5] as shown in Figure 10. Despite the fact that a physical (high-degree) node is split into multiple virtual nodes, the values of these virtual nodes are all stored to *the same memory location* - the place for the value of the original physical node. Notice that, in Figure 10, the value array remains unchanged. This allows virtual nodes of the same family automatically synchronize their values.

The synchronization brings two key benefits:

- Faster value propagation comparing to that on a physically transformed graph. Consider the virtually transformed graph in Figure 10. A value from node v_1 can immediately reach both nodes v_2' and v_2'' without any extra hopping. By contrast, it may one or multiple hops to reach a split node on a physically transformed graph.
- Correctness enforcement for general vertex-centric graph analyses. We elaborate this benefit in the next subsection.

4.2 Enforcing Correctness

The correctness of virtual split transformations is enforced by a simple yet effective mechanism – *implicit value synchronization*, which relaxes the constraints for applying split transformations, leading to much stronger conclusions.

[5]Similar ideas are also applicable to *on-the-fly mapping reasoning*.

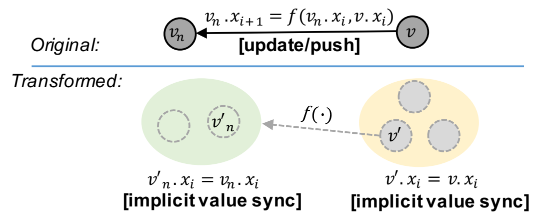


Figure 11. Correctness for Push-based Scheme.

Theorem 2. *Virtual split transformations preserve the results for all **push-based** vertex-centric graph analyses.*

Proof. In push-based vertex-centric programming, a node updates the value of each neighbor one by one, based on its own value and the neighbor's value obtained from the last iteration. Consider the node v and its neighbor v_n in Figure 11. Suppose both of them are high-degree nodes. After virtual split transformation, the values of virtual nodes at both ends, such as v' and v'_n, remain unchanged, thanks to the implicit value synchronization. By applying the same function $f(\cdot)$, the new value of the neighbor $v_n.x_{i+1}$ would also be the same as the one calculated on the original graph, that is, $v'_n.x_{i+1} = v_n.x_{i+1}$. Since the equality holds from initialization (i.e., $i = 0$), it will continue to hold for all the following iterations till convergence or termination. □

Theorem 3. *For pull-based vertex-centric graph analyses, to ensure the correctness of virtual split transformations, the vertex function needs to be associative.*

Proof. In pull-based vertex-centric programming, a node v uses the values of its neighbors $v_i.w$ to update its own value based on the vertex function $v.w = f(v.w, v_1.w, v_2.w, \cdots, v_n.w)$[6], $v_i \in v.nbrs$. In the transformed graph, a virtual node v' is only connected to a subset of the neighbors of the original node v (i.e., $v'.nbrs \subset v.nbrs$). Hence, the calculated value $v'.w$ may not equal to $v.w$. However, because of implicit value synchronization, virtual nodes of the same family will repeatedly update the same value at the physical layer, that is, $f(f(\cdots f(v.w, \cdots), \cdots))$. Since the neighbors of virtual nodes (of the same family) are disjoint, each of them appears exactly once in the nested function. If the vertex function f is associative, then nested function can be reduced to exactly the original vertex function with all neighbors included. □

Fortunately, many graph analyses once implemented in pull-based scheme are purely based on associative vertex functions [32], such as SUM, MIN, and MAX. These include popular ones like SSSP, BC, SSWP, BFS, and PR[7]. Besides associativity requirements, virtual split transformation for pull-based scheme further requires the updates to the value array are performed with atomic operations.

Together, Theorems 2 and 3 guarantee correctness for vertex-centric graph analyses in a broad sense.

[6]We assume vertex function includes the node value itself as a parameter.
[7]PageRank requires modifying the logic of its vertex function.

Algorithm 2 SSSP on Virtually Transformed Graph

```
 1: __global__ SSSP_Kernel(bool finished)
 2:     nodeId = virtualNodes[tid].physicalNodeId        ▷ main difference
 3:     d = distance[nodeId]                                   ▷ value array
 4:     start = virtualNodes[tid].edgePointer
 5:     end = virtualNodes[tid+1].edgePointer - 1
 6:     for i = start..end do                  ▷ push value to each neighbor
 7:         alt = d + edges[i].weight
 8:         if alt < distance[edges[i].nbr] then
 9:             atomicMin(&distance[edges[i].nbr], alt)
10:             finished = false
```

4.3 Example

Algorithm 2 shows an example of programming SSSP for the virtually transformed graph using *virtual node array*. Since threads are scheduled at the virtual layer, the virtual node ID is also the thread ID – `tid`, which is reflected by `vituralNodes[tid]`. At Line 2, a virtual node ID is mapped to the corresponding physical node ID. This is the main difference comparing to the vertex function for the original graph. The remaining code is the same as that in the original vertex function, except that `nodes[tid]` gets replaced by `vituralNodes[tid].edgePointer`.

4.4 Optimization for GPU Architectures

Data locality plays a critical role in the performance of GPU applications. Here, we examine the potential issues in the design of virtual split transformations that may harm the data locality, and address them with a memory access optimization, namely *edge-array coalescing*.

***Edge-array Coalescing*.** We assume the *virtual node array* design for the virtualization, but the idea is also applicable to the other design on-the-fly mapping reasoning.

With virtual split transformations, threads are scheduled based on the virtually transformed graph. Specifically, each thread is assigned to process a virtual node by propagating its value to its neighbors. This requires accessing the edges of this virtual node. In the default setting, the edges of a virtual node are stored consecutively in the edge array, as shown in Figure 10. Hence, from a single thread's view, the edge array accesses have good locality. However, GPU threads are grouped into warps (of 32 threads) and proceed in an SIMD fashion. From a warp's view, the access to the edge array is actually *strided*, where the stride length equals to the degree bound K. Consider the two virtual nodes in Figure 10, the first virtual node starts from index 3 of the edge array, while the second virtual node starts from index 6. Since the threads of the same warp share local caches, such a strided accessing pattern hurts the data locality.

To address the locality issue, we reorder the edges during the construction of CSR. Instead of assigning consecutive edges to a virtual node, the new assignment follows a strided pattern (see Figure 12). The *stride* and *offset* are the number of virtual nodes in the family and virtual node ID within the family, respectively. That is, the second virtual node is assigned with edges 1, 3, and 5 (index starts from 0). In this

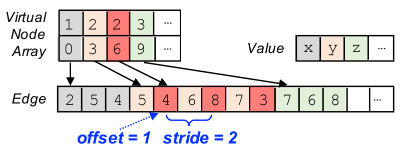


Figure 12. Edge-array Coalescing.

Algorithm 3 SSSP with Edge-array Coalescing.

```
1:  __global__ SSSP_Kernel(bool finished)
2:      nodeId = virtualNodes[tid].physicalNodeId
3:      offset = nodes[nodeId] + virtualNodes[tid].offset
4:      stride = virtualNodes[tid].stride
5:      d = distance[nodeId]
6:      for i = 1..K do                              ▷ K: degree bound
7:          e = offset + stride × i                  ▷ compute edge array index
8:          alt = d + edges[e].weight
9:          if alt < distance[edges[e].nbr] then
10:             atomicMin(&distance[edges[e].nbr], alt)
11:             finished = false
```

way, when the virtual nodes of the same family are scheduled to the same warp (as they are consecutive in the virtual node array), each time they access an edge, a consecutive chunk of memory will be loaded. We refer to this reordering technique as *edge-array coalescing*. Algorithm 3 describes SSSP with edge-array coalescing. The main differences happen at Lines 3, 4 and 7, which calculate the edge index.

5 Implementation

We implemented the proposed split transformations as a graph transformation framework – ***Tigr*** and integrated it into a lightweight GPU graph processing engine, written in C++ and CUDA. For physical graph transformation, *Tigr* implements *UDT* (Section 3.1); For virtual transformation, *Tigr* uses *virtual node array* (Section 4.1) for lower runtime cost. In addition to *edge-array coalescing* (Section 4.4), our GPU graph engine also implements *worklist* and *synchronization relaxation*. The former tracks a set of active nodes and only processes the active ones in each iteration; The latter allows to use values computed in the current iteration (along with values from the last iteration) for node value updates. Both optimizations are orthogonal to split transformations.

Selection of K. Degree bound K can be tuned based on graph algorithms and graph characteristics to maximize the benefits. However, for virtual graph transformation, we only observed marginal improvements by turning K. Hence, for simplicity, we empirically choose $K = 10$ for its overall best performance across settings.

By contrast, for physical graph transformation (UDT), we did observe substantial performance variations for different values of K. In fact, the best value of K primarily depends on the degree distribution. As more nodes are with higher degrees, the best K increases correspondingly. In practice, we use a simple heuristic that pre-defines a mapping between K and the maximum degree of a graph for selecting K.

6 Evaluation

This section evaluates the efficiency and effectiveness of split transformations for graph processing on GPUs.

6.1 Methodology

We compare *Tigr*-based GPU graph processing with three state-of-the-art general GPU graph processing frameworks: maximum warp [23], CuSha [32], and Gunrock [69]. Both implementations of CuSha and Gunrock are obtained from their public repositories. For maximum warp, we use an implementation from the CuSha repository. Table 2 lists the methods used in our comparison.

Besides, we compared with low-level implementations of some specific graph primitives, such as ECL-CC [25], Elsen and Vaidyanathan's PR [13], Davidson and others' SSSP [11], as well as the BFS by Merrill and others [44]. In fact, Gunrock [69] has systematically compared with several "hardwired" implementations and has shown performance superiority (except CC). Therefore, we choose to compare with Gunrock and leave the comparisons with these specific implementations to our project website [8].

Table 2. Methods in Evaluation.

Abbr.	Framework
MW	Maximum warp w/ warp size range: 2~32 [23]
CuSha	CuSha w/ G-Shards or Concatenated Windows [32]
Gunrock	Gunrock graph processing library [69]
Tigr-UDT	UDT split transformation-based graph processing
Tigr-V	Virtual split transformation-based graph processing
Tigr-V+	Virtual split transformation w/ edge-array coalescing
baseline	Our lightweight GPU graph engine w/ Tigr disabled

The hardware platform is a Linux workstation equipped with an Intel Xeon E3-1225 v6 CPU (4 cores, 3.30GHz), 32GB memory, and an NVIDIA Quadro P4000 GPU with 8GB memory and 1792 cores. All GPU code is compiled with CUDA 8.0 using the highest optimization level. The timing results reported are the average of 10 repetitive runs.

Table 3. Datasets in Evaluation

d_{max}: maximal outdegree, d: diameter, K_{udt} and K_v: degree bounds

Dataset	#Nodes	#Edges	d_{max}	d	K_{udt}	K_v
Pokec social [35]	1.6M	31M	8.8K	11	500	10
LiveJournal [35]	4.0M	69M	15K	13	1K	10
Hollywood [5]	1.1M	114M	11K	8	1K	10
Orkut [35]	3.1M	234M	33K	7	1K	10
Sinaweibo [56]	59M	523M	278K	5	10K	10
Twitter2010 [56]	21M	530M	698K	15	10K	10

Table 3 lists the graph datasets used in our experiments, all of which are real-world power-law graphs. The evaluation includes six widely used graph analyses: *breath-first search* (BFS), *connected components* (CC), *single-source shortest path* (SSSP), *single-source widest path* (SSWP), *between centrality* (BC), and *PageRank* (PR).

[8]https://github.com/amirnodehi/Tigr.

Table 4. Performance Comparison.
execution time: *ms*; the best performance is bolded

Alg.	Dataset	MW	CuSha	Gunrock	Tigr-V+
BFS	pokec	60.32	21.73	28.23	**14.64**
BFS	LiveJournal	149.6	57.62	51.47	**27.76**
BFS	hollywood	89.4	142.26	24.54	**15.9**
BFS	orkut	276.13	129.93	227.83	**77.73**
BFS	twitter	1514.44	1060.85	344.06	**178.53**
BFS	sinaweibo	1160.01	OOM	OOM	**299.24**
SSSP	pokec	94.37	44.49	73.34	**40.77**
SSSP	LiveJournal	228.39	115	127.54	**62.21**
SSSP	hollywood	180.16	331.46	85.49	**44.84**
SSSP	orkut	538.99	279.33	452.89	**159.85**
SSSP	twitter	1670.21	OOM	533.47	**269.75**
SSSP	sinaweibo	1529.09	OOM	1297.46	**699.35**
PR	pokec	20.81	**2.06**	30.67	22.1
PR	LiveJournal	30.63	**4.61**	33.04	34.25
PR	hollywood	16.73	20.35	**12.7**	15.09
PR	orkut	135.65	**16.59**	171.7	156.32
PR	twitter	**216.21**	OOM	243.07	221.49
PR	sinaweibo	445.8	OOM	**444.02**	463.06
CC	pokec	54.94	**17.94**	37.44	42.32
CC	LiveJournal	133.98	49.42	59.54	**47.4**
CC	hollywood	71.08	98.87	89.36	**21.38**
CC	orkut	221.67	**132.37**	170.51	207.93
CC	twitter	1427.73	979.03	683.89	**573.53**
CC	sinaweibo	928.45	OOM	772.52	**579.13**
SSWP	pokec	111.44	52.29	-	**36.86**
SSWP	LiveJournal	353.02	163.58	-	**65.67**
SSWP	hollywood	141.2	239.13	-	**22.63**
SSWP	orkut	479.12	211.38	-	**121.48**
SSWP	twitter	1546.68	OOM	-	**240.48**
SSWP	sinaweibo	1527.14	OOM	-	**635.23**
BC	pokec	-	-	87.09	**42.86**
BC	LiveJournal	-	-	109.56	**73.61**
BC	hollywood	-	-	55.77	**39.21**
BC	orkut	-	-	399.96	**207.58**
BC	twitter	-	-	732.28	**475.23**
BC	sinaweibo	-	-	1507.25	**1033.97**

6.2 Comparison With Existing Methods

Table 4 reports the performance results of tested methods (we will discuss Tigr-UDT and Tigr-V in Section 6.3). For MW with varying virtual warp sizes, the best performance is chosen. Similarly, for CuSha, we report results of the better one between G-Shards and Concatenated Windows. Some results on SSWP and BC are missing as the corresponding frameworks are lack of such graph primitives.

Memory Requirements. Table 4 indicates that some graph processing frameworks require larger memory space in order to accommodate their special graph representations or their growing runtime memory consumption. When the memory requirement exceeds the GPU memory limit, an error of out of memory (OOM) is thrown. This happened to both CuSha and Gunrock when running on relatively large datasets such as `sinaweibo` or `twitter`. In comparison, Tigr-V+ did not encounter any OOM issue in all tested datasets and algorithms, thanks to its limited space cost (see Section 4). Besides our method, MW is also free from OOM issues, since it is based on the modifications to GPU thread execution model which does not introduce significant space cost.

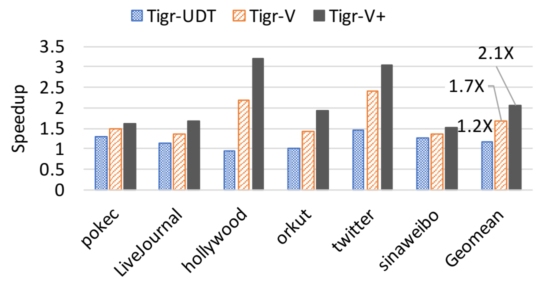


Figure 13. Speedups of *Tigr* over baseline (SSSP).

Performance. On one hand, there is no such a single method that always performs the best in all tested cases. On the other hand, the results clearly show that Tigr-V+ achieves substantial performance improvements over the existing methods for most datasets and algorithms, thanks to its capability in graph irregularity reduction. In particular, Tigr-V+ achieves up to 5.43X speedup over MW method on `LiveJournal` dataset when running SSWP algorithm. It also outperforms CuSha by 10.4X on the same algorithm with the `hollywood` dataset. Comparing with Gunrock, Tigr-V+ achieves around 3X speedup when running BFS and SSSP algorithms on the `orkut` dataset. For the other cases where Tigr-V+ wins, the speedup ranges from 1.04X to 2.93X.

Despite improvements for most datasets and algorithms, Tigr-V+ performs worse than some existing methods in a few cases, especially with the PR algorithm. This is mainly because Tigr-V+ implements a push-based programming strategy. Different from the other evaluated algorithms, PR requires to processes every node in each iteration. For such kind of computation pattern, a pull-based graph processing (like CuSha) often performs more efficiently, by taking the advantages of parallel scan-style parallelism.

6.3 Performance Breakdown of Tigr

Figure 13 reports the speedups of different versions of *Tigr* over the baseline – a lightweight GPU graph engine without any transformations, for SSSP algorithm. The speedups with other graph algorithms follow a similar trend.

Physical v.s. Virtual. First, the results indicate that both physical and virtual split transformations bring performance benefits to the original GPU graph framework, with 1.2X (Tigr-UDT) and 1.7X (Tigr-V) average speedups, respectively. The reason Tigr-UDT shows less speedup is that physically transforming graphs with splitting increases the number of hops among nodes, which cause more iterations to converge for graph algorithms (see Section 6.5), while the number of iterations remains the same on a virtually transformed graph, as the values are directly propagated at the physical layer – with no extra hoping (see Section 4.1).

Memory-coalescing Optimization. Besides, Figure 13 also shows that the proposed edge-array coalescing optimization boosts the performance of virtual split transformations from 1.7X to 2.1X. The benefits come from the enhanced memory

Table 5. Space Cost of Physical Transformation (UDT).

	K=100	K=1000	K=10000
pokec	100.13%	100.00%	100.00%
LiveJournal	100.41%	100.00%	100.00%
hollywood	101.37%	100.05%	100.00%
orkut	100.99%	100.01%	100.00%
twitter	101.29%	100.07%	100.00%
sinaweibo	100.96%	100.06%	100.00%

Table 6. Space Cost of Virtual Transformation.

	K=4	K=8	K=16	K=32	K=100
pokec	147.32%	124.42%	113.24%	108.00%	105.32%
LiveJournal	146.69%	124.28%	113.46%	108.47%	105.86%
hollywood	149.28%	124.66%	112.38%	106.29%	102.35%
orkut	149.05%	124.55%	112.31%	106.23%	102.28%
twitter	148.05%	125.07%	113.88%	108.52%	105.15%
sinaweibo	145.99%	126.61%	117.60%	113.51%	111.05%

Table 7. Transformation Time Cost (*ms*).

Dataset	pokec	LiveJournal	hollywood	orkut	twitter	sinaweibo
Physical	403	1,088	994	2,164	10,161	16,444
Virtual	20.7	38.6	50.4	98.3	211.5	289.7

locality with more intelligent edge assignments to the virtual nodes (see Section 4.4 for more details).

6.4 Transformation Costs of Tigr

As mentioned earlier (Section 6.2), despite the increases in graph size, *Tigr*-based graph processing still require less memory than other frameworks like CuSha and Gunrock. Here, we further examine the time and space costs of *Tigr*.

Space Cost. Tables 5 and 6 report the space increases for physical and virtual transformations, respectively, in terms of the graph size in CSR format. For physical transformation, in order to keep sufficient value propagation rate, K is often set to a relatively large value (Table 3). As a result, the size of the graph only increases marginally, by up to 1.37% (K=100). As the degree bound increases, the sizes of transformed graphs decrease since less number of nodes will be transformed.

In comparison, as shown in Table 6, the space cost of virtual graph transformation is much higher due to the use of relatively smaller K. Because virtual split transformation only introduces a virtual node and the edge array dominates the sizes of power-law graphs, the overall space cost of virtual transformation remains around 25% even for $K = 8$. Note that despite the increased graph sizes, the memory footprint of Tigr-V is still much smaller than some other general graph frameworks, like CuSha and Gunrock (see Section 6.2).

Time Cost. Table 7 reports the transformation time. Note that the current implementation of transformations is serial and can be parallelized. In general, the transformation time is proportional to the size of the graph for both physical and virtual graph transformations. For the same K, virtual transformation is more lightweight than physical transformation as it only needs to build a virtual node array, rather than creating new nodes and edges. Since physical transformation can be performed offline, its cost can be amortized across different runs. For virtual transformation, it can be easily integrated into the graph loading phase, in which case the transformation time cost could be negligible.

Table 8. Performance Details (SSSP, LiveJournal, $K = 8$).

	Without Worklist				With Worklist		
	#iter	time / iter.	#instr.	warp effi.	#iter	#instr.	warp effi.
Original	14	29.92	3.3×10^9	25.98%	18	9×10^8	60.53%
Physical	29	24.68	8.9×10^9	91.15%	45	4.6×10^9	70.11%
Virtual	14	17.64	7.6×10^9	92.81%	18	2.2×10^9	85.51%

6.5 Case Study: SSSP

To obtain deeper insights on how irregular graph processing benefits from physical and virtual split transformations, we use SSSP as an example and break down the performance into lower-level contributing factors, such as the number of iterations, runtime of each iteration, GPU warp efficiency, and the total amount of instructions executed.

Table 8 lists the detailed profiles of SSSP running on the original `LiveJournal` graph, the physically transformed graph and the virtually transformed graph, respectively. When the worklist optimization is not used, all the nodes in the graph are processed in each iteration. In this case, the physically transformed graph needs over 2X iterations to converge, due to the increased node distances caused by physical splitting. By contrast, the virtually transformed graph needs no extra iterations at all, thanks to its implicit value synchronization. As to processing time per iteration, both physical and virtual transformations are able to reduce it substantially, due to the irregularity reduction. Meanwhile, both of them lead to more instructions to execute because of the extra computations on new nodes and edges. As shown by the warp efficiency columns, both transformations boost the efficiency with more balanced computations.

The results for the cases with worklist optimization, in general, follow similar patterns. However, the processing time per iteration can vary a lot depending on the set of active nodes, hence is not listed. Note that the total number of instructions is dramatically reduced in all three cases, as only active nodes are involved in the computations.

7 Related Work

This section discusses related work in three aspects: general graph processing frameworks, GPU-based graph processing, and techniques for GPU efficiency optimizations.

7.1 General Parallel Graph Processing

There is a rich body of work on designing distributed graph programming systems. Boost Graph Library [62], as an early effort, offers a high-level abstraction for programming graphs. To enable parallel execution, Gregor and Lumsdaine implement parallel BGL [18]. Inspired by the Bulk Synchronous Parallel model [65], Google designs the first vertex-centric graph programming framework Pregal [41]. Since then, vertex-centric graph programming has been adopted by many parallel graph engines, such as Apache Giraph [1], GraphLab [38], and PowerGraph [17]. Targeting distributed platforms, the above systems require to partition graphs and store the partitions across machines, based on edges [29, 52], vertices [17], or value vectors [75]. PowerLyra [9] has shown improved performance by differentiating the partitioning between high-degree and low-degree vertices.

Though vertex partitioning [9, 17] *shares similarities with split transformation, the two approaches differ in a few key aspects*. First, split transformation allows to change the graph topology meanwhile does not result in any graph partitions; Second, targeting distributed platforms, vertex partitioning requires to synchronize the partitioned vertices explicitly; More critically, vertex partitioning often has to replicate both high-degree and low-degree vertices (called *mirroring*).

On shared-memory platforms, Ligra [61] and Galois [53] support programming over a subset of vertices. Featuring amorphous data parallelism, Galois offers a new perspective on irregular graphs processing [49, 54]. Charm++ [27] and STAPL [14, 70] are general parallel programming systems. The former supports intensively for irregular computations, while the latter features a parallel container data structure for graph processing. For easier adoption, graph processing based on single PCs also receives significant attentions, such as GraphChi [33], GraphQ [68], and Graspan [67].

In addition, some work focuses on specific parallel graph algorithms, such as connected components [19], BFS [2], SSSP[10, 45], and betweenness centrality [6] or the design choice between pull and push-based processing schemes [4]. Some recent work parallelizes automata executions which are essentially input-guided graph traversals [55, 76, 77].

7.2 Graph Processing on GPUs

By mapping nodes of a graph to GPU threads, Harish and others [22] implement a GPU graph processing framework based on vertex-centric programming. To minimize path divergence and load imbalance in GPU graph processing, Maximum warp [23] decomposes GPU warps into smaller sub-warps. By contrast, CuSha [32] addresses the efficiency issues with two new graph representations, namely G-shard and concatenated windows, to achieve coalesced accesses. Based on the concept of frontiers, Gunrock [69] proposes a new programming abstraction for GPU graph processing.

Some other work targets efficient GPU implementations of specific graph algorithms, including hierarchical queue or prefix-sum based BFS [40, 44], GPU-optimized connected components [19, 63], SSSP based on Δ-stepping or hybrid approaches [11, 50] and betweenness centrality based on Brandes formulation [26, 42, 58].

Besides, GPU graph processing for specific applications, such as program analysis [43], and general applications, like compiler-level optimizations [51] have also been proposed. There are also a series of work on multi-GPU graph processing, including TOTEM [15], Medusa [78], METIS [28], as well as hybrid CPU-GPU methods [16, 24]. Graphie [21] and GraphReduce [60] target the GPU memory constraints for processing large graphs – another important problem in GPU-based graph processing. In general, our proposed methods are orthogonal to these existing techniques.

7.3 GPU Efficiency Optimizations

Minimizing non-coalesced memory accesses is shown as a NP-complete problem [71]. To optimize memory access efficiency on GPUs, Dymaxion [8] and G-streamline [74] use methods like data reordering, memory remapping, and job swapping. Similar to the load balancing with local worklists [48], a queue-based approach handles irregularities in task loads [64]. By contrast, other work [34, 73] proposes static decomposition to overcome load imbalance in nested patterns. The idea of dividing the warp into virtual warps is also used in CUSP library [3] for SpMV operation on CSR matrices. Sartori and Kumar [59] explore the tradeoff between path divergence and the accuracy of the results, by forcing all the warp lanes to follow the majority.

In addition to path divergence elimination, there are also methods trying to avoid the uses of atomic instructions for processing irregular graphs [47], and a study on identifying the bottlenecks of implementing GPU applications, such as data transfers, kernel invocations and memory latencies [72].

In fact, the graph irregularity issue exhibits as a special case of the path divergence problem in GPU processing. However, we are not aware of any systematic studies that address the divergence issue at the input graph level by directly transforming the structures of graph data.

8 Conclusion

This work addresses the critical irregularity issue in GPU graph processing by transforming irregular input graphs. Comparing to existing solutions, graph transformation does not require significant changes to the graph programming system or the GPU thread execution model.

Specifically, to reduce the graph irregularity, this work presents a class of *split transformations*, which split nodes with high degrees into sets of nodes with lower degrees. It further identifies a type of split transformation – *UDT*, with desirable properties, including correctness guarantees for a variety of graph algorithms. To reduce the transformation costs, this work introduces a virtual transformation scheme, which allows a separation between the programming model and the graph data. Based on implicit value synchronization, the correctness of virtual split transformation is guaranteed for vertex-centric graph analyses in a broader sense. Finally, the evaluation confirms the effective and efficiency of the split transformations on real-world power-law graphs.

Acknowledgments

We thank all anonymous reviewers for their constructive comments and suggestions. We also thank Dr. Rajiv Gupta for his positive feedback during the early stage of this project. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1565928.

References

[1] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on Hadoop. *Proceedings of the Hadoop Summit. Santa Clara* 11 (2011).

[2] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 12.

[3] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*. ACM, 18.

[4] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 93–104.

[5] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web*. ACM, 587–596.

[6] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001), 163–177.

[7] Ed Bullmore and Olaf Sporns. 2009. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience* 10, 3 (2009), 186–198.

[8] Shuai Che, Jeremy W Sheaffer, and Kevin Skadron. 2011. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*. ACM, 13.

[9] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, 1:1–1:15. http://doi.acm.org/10.1145/2741948.2741970

[10] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. 1998. A parallelization of Dijkstra's shortest path algorithm. *Mathematical Foundations of Computer Science 1998* (1998), 722–731.

[11] Andrew Davidson, Sean Baxter, Michael Garland, and John D Owens. 2014. Work-efficient parallel GPU methods for single-source shortest paths. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 349–359.

[12] Pedro Domingos and Matt Richardson. 2001. Mining the network value of customers. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 57–66.

[13] E. Elsen and V. Vaidyanathan. 2013. A vertex-centric CUDA/C++ API for large graph analytics on GPUs using the gather-apply-scatter abstraction. https://github.com/RoyalCaliber/vertexAPI2. (2013).

[14] Adam Fidel, Nancy M Amato, and Lawrence Rauchwerger. 2012. The STAPL parallel graph library. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 46–60.

[15] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 345–354.

[16] Abdullah Gharaibeh, Tahsin Reza, Elizeu Santos-Neto, Lauro Beltrao Costa, Scott Sallinen, and Matei Ripeanu. 2013. Efficient large-scale graph processing on hybrid CPU and GPU systems. *arXiv preprint arXiv:1312.3018* (2013).

[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.. In *OSDI*, Vol. 12. 2.

[18] Douglas Gregor and Andrew Lumsdaine. 2005. The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)* 2 (2005), 1–18.

[19] John Greiner. 1994. A comparison of parallel algorithms for connected components. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*. ACM, 16–25.

[20] Tianyi David Han and Tarek S Abdelrahman. 2011. Reducing branch divergence in GPU programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 3.

[21] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-Scale Asynchronous Graph Traversals on Just a GPU. In *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*. IEEE, 233–245.

[22] Pawan Harish and PJ Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *International Conference on High-Performance Computing*. Springer, 197–208.

[23] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 267–276.

[24] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011. Efficient parallel graph exploration on multi-core CPU and GPU. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 78–88.

[25] Jayadharini Jaiganesh and Martin Burtscher. 2018. ECL-CC v1.0. http://cs.txstate.edu/~burtscher/research/ECL-CC/. (2018).

[26] Yuntao Jia, Victor Lu, Jared Hoberock, Michael Garland, and John C Hart. 2011. Edge v. node parallelism for graph centrality metrics. *GPU Computing Gems* 2 (2011), 15–30.

[27] Laxmikant V Kale and Abhinav Bhatele. 2016. *Parallel science and engineering applications: The Charm++ approach*. CRC Press.

[28] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[29] George Karypis and Vipin Kumar. 1998. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing* 48, 1 (1998), 96–129.

[30] Farzad Khorasani, Rajiv Gupta, and Laxmi N Bhuyan. 2015. Scalable SIMD-efficient graph processing on GPUs. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE, 39–50.

[31] Farzad Khorasani, Bryan Rowe, Rajiv Gupta, and Laxmi N Bhuyan. 2016. Eliminating Intra-warp Load Imbalance in Irregular Nested Patterns via Collaborative Task Engagement. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 524–533.

[32] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 239–252.

[33] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. USENIX.

[34] HyoukJoong Lee, Kevin J Brown, Arvind K Sujeeth, Tiark Rompf, and Kunle Olukotun. 2014. Locality-aware mapping of nested parallel patterns on gpus. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 63–74.

[35] Jure Leskovec and Andrej Krevl. 2015. SNAP Datasets:Stanford Large Network Dataset Collection. (2015).

[36] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 339–350.

[37] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.

[38] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. 2010. GraphLab: A new framework for parallel machine learning. *CoRR* abs/1006.4990 (2010). http://arxiv.org/abs/1006.4990

[39] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *Parallel Processing Letters* 17, 01 (2007), 5–20.

[40] Lijuan Luo, Martin Wong, and Wen-mei Hwu. 2010. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th*

design automation conference. ACM, 52–55.
[41] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 135–146.
[42] Adam McLaughlin and David A Bader. 2014. Scalable and high performance betweenness centrality on the GPU. In *Proceedings of the International Conference for High performance computing, networking, storage and analysis*. IEEE Press, 572–583.
[43] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU implementation of inclusion-based points-to analysis. *ACM SIGPLAN Notices* 47, 8 (2012), 107–116.
[44] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 117–128.
[45] Ulrich Meyer and Peter Sanders. 1998. Δ-stepping: A parallel single source shortest path algorithm. In *European Symposium on Algorithms*. Elsevier, 393–404.
[46] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 29–42.
[47] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Atomic-free irregular computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM, 96–107.
[48] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Morph algorithms on GPUs. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 147–156.
[49] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 456–471.
[50] Hector Ortega-Arranz, Yuri Torres, Diego R Llanos, and Arturo Gonzalez-Escribano. 2013. A New GPU-based Approach to the Shortest Path Problem. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*. IEEE, 505–511.
[51] Sreepathi Pai and Keshav Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 1–19.
[52] François Pellegrini and Jean Roman. 1996. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*. Springer, 493–498.
[53] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. 2011. The Tao of parallelism in algorithms. In *ACM Sigplan Notices*, Vol. 46. ACM, 12–25.
[54] Dimitrios Prountzos and Keshav Pingali. 2013. Betweenness centrality: algorithms and implementations. In *Acm Sigplan Notices*, Vol. 48. ACM, 35–46.
[55] Junqiao Qiu, Zhijia Zhao, and Bin Ren. 2016. MicroSpec: Speculation-centric fine-grained parallelization for FSM computations. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*. IEEE, 221–233.
[56] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. http://networkrepository.com
[57] Gorka Sadowski and Philip Rathle. 2014. Fraud detection: Discovering connections with graph databases. *White Paper-Neo Technology-Graphs are Everywhere* (2014).
[58] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V Çatalyürek. 2013. Betweenness centrality on GPUs and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. ACM, 76–85.
[59] John Sartori and Rakesh Kumar. 2013. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. *IEEE Transactions on Multimedia* 15, 2 (2013), 279–290.
[60] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 28.
[61] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.
[62] Jeremy G Siek, Lie-Quan Lee, and Andrew Lumsdaine. 2001. The Boost Graph Library: User Guide and Reference Manual, Portable Documents. (2001).
[63] Jyothish Soman, Kothapalli Kishore, and PJ Narayanan. 2010. A fast GPU algorithm for graph connectivity. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*. IEEE, 1–8.
[64] Stanley Tzeng, Anjul Patney, and John D Owens. 2010. Task management for irregular-parallel workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*. Eurographics Association, 29–37.
[65] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.
[66] Stephan M Wagner and Nikrouz Neshat. 2010. Assessing the vulnerability of supply chains using graph theory. *International Journal of Production Economics* 126, 1 (2010), 121–129.
[67] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 389–404.
[68] Kai Wang, Guoqing (Harry) Xu, Zhendong Su, and Yu David Liu. 2015. GraphQ: Graph Query Processing with Abstraction Refinement-Scalable and Programmable Analytics over Very Large Graphs on a Single PC.. In *USENIX Annual Technical Conference*. 387–401.
[69] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 11.
[70] Brandon West, Adam Fidel, Nancy M Amato, Lawrence Rauchwerger, et al. 2015. A hybrid approach to processing big data graphs on memory-restricted systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 799–808.
[71] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. 2013. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 57–68.
[72] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. 2014. Graph processing on gpus: Where are the bottlenecks?. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE, 140–149.
[73] Yi Yang and Huiyang Zhou. 2014. CUDA-NP: realizing nested thread-level parallelism in GPGPU applications. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 93–106.
[74] Eddy Z Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011. On-the-fly elimination of dynamic irregularities for GPU computing. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 369–380.
[75] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. 2016. Exploring the Hidden Dimension in Graph Processing.. In *OSDI*. 285–300.
[76] Zhijia Zhao and Xipeng Shen. 2015. On-the-Fly Principled Speculation for FSM Parallelization. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*.

619–630. https://doi.org/10.1145/2694344.2694369

[77] Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. Challenging the "Embarrassingly Sequential": Parallelizing Finite State Machine-Based Computations through Principled Speculation. In *ASPLOS '14: Proceedings of 19th International Conference on Architecture Support for Programming Languages and Operating Systems*. ACM Press.

[78] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1543–1552.