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.

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.
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-warsps. 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 Gurnrock [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)
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 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.

---

2Single-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. 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 \( N_S, N_S = \bigcup_{v \in S} N_v \). 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

\[
\mathcal{T} : (v, N_v) \mapsto (I \cup B, N_I \cup N_B)
\]

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 summarizes 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, \( \mathcal{T}_{\text{cliq}} \) 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 \( |d(K)| - 1 \) 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...
shown in Figure 6-(a). Applying $T_{star}$ 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 ($T_{udt}$), which ensures at most one residual node in the generated family.

Algorithm 1 illustrates 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 $T_{star}$.

**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.

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{split}$</td>
<td>$[d/K] - 1$</td>
<td>$((d/K) - 1) \cdot [d/K]$</td>
<td>$K + [d/K] - 1$</td>
<td>1</td>
<td>high</td>
<td>fast</td>
</tr>
<tr>
<td>$T_{merge}$</td>
<td>$[d/K] - 1$</td>
<td>$[d/K] - 1$</td>
<td>$K + 1$</td>
<td>low</td>
<td>high</td>
<td>slow</td>
</tr>
<tr>
<td>$T_{star}$</td>
<td>$[d/K]$</td>
<td>$[d/K]$</td>
<td>$\max{K + 1, [d/K]}$</td>
<td>1</td>
<td>low</td>
<td>varies</td>
</tr>
</tbody>
</table>

### Algorithm 1 UDT Transformation

1. \textbf{if} degree($v$) > $K$ \textbf{then} \quad \Rightarrow \text{for each high-degree node}
   2. \quad $q = \text{new\_queue}()$
   3. \quad \textbf{for} each $v_n$ from $v$'s neighbors \textbf{do}
   4. \quad \quad $q.add(v_n)$ \quad \Rightarrow \text{add all original neighbors}
   5. \quad \quad $v.remove\_neighbor(v_n)$
   6. \quad \textbf{while} $q.size() > K$ \textbf{do}
   7. \quad \quad $v_n = \text{new\_node}()$
   8. \quad \quad \textbf{for} $i = 1..K$ \textbf{do}
   9. \quad \quad \quad $v_n.add\_neighbor(q.pop())$
   10. \quad \quad $q.push(v_n)$ \quad \Rightarrow \text{add a new node}
   11. \quad \quad $S = q.size()$
   12. \quad \quad \textbf{for} $i = 1..S$ \textbf{do}
   13. \quad \quad \quad $v.add\_neighbor(q.pop())$ \quad \Rightarrow \text{connect to the root node}

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

### 3.3 Enforcing the Correctness for $T_{udt}$

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,

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

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

**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 $E_{new} = \emptyset$. 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
By the definition of connectivity and Theorem 1.

Proof. By the definition of connectivity and Theorem 1.

**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 \( E_{\text{new}} \), \( P'(v_1, v_2) \) 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 \( E_{\text{new}} \), \( P'(v_1, v_2) \) 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.

**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.
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 \( \text{map}_v \) and an edge mapping \( \text{map}_e \).

\[ v = \text{map}_v(v'), e = \text{map}_e(e') \]

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 \( (v'_2, v''_2) \).

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).

- **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 \( (6/3) \), 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.
For pull-based vertex-centric graph analyses, to Theorem 3. Virtual split transformations preserve the results of 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 \( f() \), the new value of the neighbor \( v_n \cdot x_{i+1} \) would also be the same as the one calculated on the original graph, that is, \( v'_n \cdot x_{i+1} = v_n \cdot x_{i+1} \). 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 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 \( f() \), the new value of the neighbor \( v_n \cdot x_{i+1} \) would also be the same as the one calculated on the original graph, that is, \( v'_n \cdot x_{i+1} = v_n \cdot x_{i+1} \). Since the equality holds from initialization (i.e., \( i = 0 \)), it will continue to hold for all the following iterations till convergence or termination. □

**Algorithm 2** SSSP on Virtually Transformed Graph

```python
1: __global__ SSSP_Kernel(bool finished)
2: nodeid = virtualNodes[tid].physicalNodeId => main difference
3: d = distance[nodeid]
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 virtualNodes[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 virtualNodes[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

---

9We assume vertex function includes the node value itself as a parameter.

7PageRank requires modifying the logic of its vertex function.
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.

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>MW</td>
<td>Maximum warp w/ warp size range: 2-32 [23]</td>
</tr>
<tr>
<td>CuSha</td>
<td>CuSha w/ G-Shards or Concatenated Windows [32]</td>
</tr>
<tr>
<td>Gunrock</td>
<td>Gunrock graph processing library [69]</td>
</tr>
<tr>
<td>Tigr-UDT</td>
<td>UDT split transformation-based graph processing</td>
</tr>
<tr>
<td>Tigr-V</td>
<td>Virtual split transformation-based graph processing</td>
</tr>
<tr>
<td>Tigr-V+</td>
<td>Virtual split transformation w/ edge-array coalescing</td>
</tr>
</tbody>
</table>

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.

6.2 Datasets
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).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Nodes</th>
<th>#Edges</th>
<th>$d_{\text{max}}$</th>
<th>$d$</th>
<th>$K_{udt}$</th>
<th>$K_{v}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pokec social [35]</td>
<td>1.6M</td>
<td>31M</td>
<td>8.8K</td>
<td>11</td>
<td>500</td>
<td>10</td>
</tr>
<tr>
<td>LiveJournal [35]</td>
<td>4.0M</td>
<td>69M</td>
<td>15K</td>
<td>13</td>
<td>1K</td>
<td>10</td>
</tr>
<tr>
<td>Orkut [35]</td>
<td>3.1M</td>
<td>234M</td>
<td>33K</td>
<td>7</td>
<td>1K</td>
<td>10</td>
</tr>
<tr>
<td>Sinaweibo [56]</td>
<td>59M</td>
<td>523M</td>
<td>278K</td>
<td>5</td>
<td>10K</td>
<td>10</td>
</tr>
<tr>
<td>Twitter2010 [56]</td>
<td>21M</td>
<td>530M</td>
<td>698K</td>
<td>15</td>
<td>10K</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3 lists the graphs 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).

<https://github.com/amirnodehi/Tigr>
Table 4. Performance Comparison.

<table>
<thead>
<tr>
<th>Alg.</th>
<th>Dataset</th>
<th>MW</th>
<th>CuSha</th>
<th>Gunrock</th>
<th>Tigr-V+</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>pokec</td>
<td>60.32</td>
<td>21.73</td>
<td>28.23</td>
<td>14.64</td>
</tr>
<tr>
<td>BFS</td>
<td>Livejournal</td>
<td>149.6</td>
<td>57.62</td>
<td>51.47</td>
<td>27.76</td>
</tr>
<tr>
<td>BFS</td>
<td>hollywood</td>
<td>89.4</td>
<td>142.26</td>
<td>24.54</td>
<td>15.9</td>
</tr>
<tr>
<td>BFS</td>
<td>orkut</td>
<td>276.13</td>
<td>129.93</td>
<td>227.83</td>
<td>77.73</td>
</tr>
<tr>
<td>BFS</td>
<td>twitter</td>
<td>1514.44</td>
<td>1606.85</td>
<td>344.06</td>
<td>178.53</td>
</tr>
<tr>
<td>BFS</td>
<td>sinaweibo</td>
<td>1160.01</td>
<td>OOM</td>
<td>OOM</td>
<td>299.24</td>
</tr>
<tr>
<td>SSSP</td>
<td>pokec</td>
<td>94.37</td>
<td>44.49</td>
<td>73.34</td>
<td>40.77</td>
</tr>
<tr>
<td>SSSP</td>
<td>Livejournal</td>
<td>228.39</td>
<td>115</td>
<td>127.54</td>
<td>62.21</td>
</tr>
<tr>
<td>SSSP</td>
<td>hollywood</td>
<td>180.16</td>
<td>331.46</td>
<td>85.49</td>
<td>44.84</td>
</tr>
<tr>
<td>SSSP</td>
<td>orkut</td>
<td>538.99</td>
<td>279.33</td>
<td>452.89</td>
<td>159.85</td>
</tr>
<tr>
<td>SSSP</td>
<td>twitter</td>
<td>1670.21</td>
<td>OOM</td>
<td>533.47</td>
<td>269.75</td>
</tr>
<tr>
<td>SSSP</td>
<td>sinaweibo</td>
<td>1529.09</td>
<td>OOM</td>
<td>1297.46</td>
<td>699.35</td>
</tr>
<tr>
<td>PR</td>
<td>pokec</td>
<td>20.81</td>
<td>2.06</td>
<td>30.67</td>
<td>22.1</td>
</tr>
<tr>
<td>PR</td>
<td>Livejournal</td>
<td>30.63</td>
<td>4.61</td>
<td>33.04</td>
<td>34.25</td>
</tr>
<tr>
<td>PR</td>
<td>hollywood</td>
<td>16.73</td>
<td>20.35</td>
<td>10.7</td>
<td>15.09</td>
</tr>
<tr>
<td>PR</td>
<td>orkut</td>
<td>135.65</td>
<td>16.29</td>
<td>171.7</td>
<td>156.32</td>
</tr>
<tr>
<td>PR</td>
<td>twitter</td>
<td>216.21</td>
<td>OOM</td>
<td>243.07</td>
<td>221.49</td>
</tr>
<tr>
<td>PR</td>
<td>sinaweibo</td>
<td>445.8</td>
<td>OOM</td>
<td>444.02</td>
<td>463.06</td>
</tr>
<tr>
<td>CC</td>
<td>pokec</td>
<td>54.94</td>
<td>17.94</td>
<td>37.44</td>
<td>42.32</td>
</tr>
<tr>
<td>CC</td>
<td>Livejournal</td>
<td>133.98</td>
<td>49.42</td>
<td>59.54</td>
<td>47.4</td>
</tr>
<tr>
<td>CC</td>
<td>hollywood</td>
<td>71.08</td>
<td>98.87</td>
<td>89.36</td>
<td>21.38</td>
</tr>
<tr>
<td>CC</td>
<td>orkut</td>
<td>221.67</td>
<td>132.37</td>
<td>170.51</td>
<td>207.93</td>
</tr>
<tr>
<td>CC</td>
<td>twitter</td>
<td>1427.73</td>
<td>979.03</td>
<td>683.89</td>
<td>573.53</td>
</tr>
<tr>
<td>CC</td>
<td>sinaweibo</td>
<td>928.45</td>
<td>OOM</td>
<td>772.52</td>
<td>579.13</td>
</tr>
<tr>
<td>SSWP</td>
<td>pokec</td>
<td>111.44</td>
<td>52.29</td>
<td>-</td>
<td>36.86</td>
</tr>
<tr>
<td>SSWP</td>
<td>Livejournal</td>
<td>353.02</td>
<td>163.58</td>
<td>-</td>
<td>65.67</td>
</tr>
<tr>
<td>SSWP</td>
<td>hollywood</td>
<td>141.2</td>
<td>239.13</td>
<td>-</td>
<td>22.63</td>
</tr>
<tr>
<td>SSWP</td>
<td>orkut</td>
<td>479.12</td>
<td>211.38</td>
<td>-</td>
<td>121.48</td>
</tr>
<tr>
<td>SSWP</td>
<td>twitter</td>
<td>1546.68</td>
<td>OOM</td>
<td>-</td>
<td>240.48</td>
</tr>
<tr>
<td>SSWP</td>
<td>sinaweibo</td>
<td>1527.14</td>
<td>OOM</td>
<td>-</td>
<td>635.23</td>
</tr>
<tr>
<td>BC</td>
<td>pokec</td>
<td>-</td>
<td>-</td>
<td>87.09</td>
<td>42.86</td>
</tr>
<tr>
<td>BC</td>
<td>Livejournal</td>
<td>-</td>
<td>-</td>
<td>109.56</td>
<td>73.61</td>
</tr>
<tr>
<td>BC</td>
<td>hollywood</td>
<td>-</td>
<td>-</td>
<td>55.77</td>
<td>39.21</td>
</tr>
<tr>
<td>BC</td>
<td>orkut</td>
<td>-</td>
<td>-</td>
<td>399.96</td>
<td>207.58</td>
</tr>
<tr>
<td>BC</td>
<td>twitter</td>
<td>-</td>
<td>-</td>
<td>732.28</td>
<td>475.23</td>
</tr>
<tr>
<td>BC</td>
<td>sinaweibo</td>
<td>-</td>
<td>-</td>
<td>1597.25</td>
<td>1033.97</td>
</tr>
</tbody>
</table>

**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.

**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 SSSP 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 vs. 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 (Tigr-UDT) and 1.7X (Tigr-V) average speedups, respectively. The benefits to the original GPU graph framework, with 1.2X over baseline (SSSP).

**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.
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.

### 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 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 processing 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 partition;
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-wars. 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 A-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 pro-
cessing, 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 ef-
ciciency 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 work-
lists [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 ma-
trices. 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.
[77] Zhijia Zhao, Bo Wu, and Xipeng Shen. 2014. Challenging the "Embar- 
rassingly Sequential": Parallelizing Finite State Machine-Based Compu-
tations 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 