UNIVERSITY OF CALIFORNIA
RIVERSIDE

Efficient Processing of Large Irregular Graphs on GPUs and Multicores

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Amir Hossein Nodehi Sabet

December 2021

Dissertation Committee:

Dr. Zhijia Zhao, Chairperson
Dr. Rajiv Gupta
Dr. Nael Abu-Ghazaleh
Dr. Daniel Wong

The Dissertation of Amir Hossein Nodehi Sabet is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I want to express my deep gratitude to my advisor Dr Zhijia Zhao. Thank you so much for the research opportunity you provided for me and for the motivation throughout my graduate career. I have enjoyed working under your supervision, I learned a lot from you.

I want to thank my dissertation committee, Dr Rajiv Gupta, Dr Nael Abu-Ghazaleh and Dr Daniel Wong. Thank you so much for your support and helpful guidance throughout my graduate research.

The text of chapter two is a reprint of the material as it appears in "Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing" in Asplos, 2018.

The text of chapter three is a reprint of the material as it appears in "Subway: Minimizing Data Transfer during Out-of-GPU-Memory Graph Processing" in EuroSys, 2020.

To Salma and Liana.

ABSTRACT OF THE DISSERTATION

Efficient Processing of Large Irregular Graphs on GPUs and Multicores

by

Amir Hossein Nodehi Sabet

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2021
Dr. Zhijia Zhao, Chairperson

Graph analytics is fundamental in unlocking key insights by mining large volumes of highly connected data. However, comparing to many other data analytics, it is difficult to perform graph analytics efficiently on modern computers due to three reasons. First, *the structures of graphs are often highly irregular*. For example, a small portion of vertices may own a large number of neighbors while most vertices have very few neighbors. The structure irregularity leads to computation irregularity, resulting in workload imbalance among worker threads. Second, *real-world graphs tend to be large*. It is oftentimes impractical to find individual machines with memory capacity that can accommodate such large graphs entirely, not to mention accelerators (like GPUs) which have more limited memory capacity. Third, it remains open questions *how graph processing systems should be designed for emerging graph analytics*, which often involves the tradeoff among multiple key factors, such as data locality, amount of parallelism, and computation redundancy.

This thesis aims to address the above three challenges of graph processing under some specific contexts. First, it focuses on improving the performance of graph analytics

on modern highly-parallel processors – GPUs. Note that GPUs are conventionally designed for regular computations. To address the irregularity of graph computations, this thesis proposes a graph transformation technique that can convert irregular graphs into regular ones while preserving the correctness of the graph analytics. Second, this thesis further examines the performance bottleneck in processing oversized graphs that cannot fit into the memory of GPUs. It finds that the data movement between CPU and GPU is very costly. To address the issue, this thesis proposes a subgraph extraction technique which can dynamically extract the active parts of the graph in each processing iteration – they are usually small enough to fit into the GPU memory. Finally, this thesis looks into an underexplored yet important graph application – large-scale program analysis, and proposes to systematically exploit the design space of a graph system for this new application in order to realize its full potential.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Major Challenges

Graph analytics is fundamental in unlocking key insights by mining large volumes of highly connected data, such as identifying influencers in social networks, spotting frauds in bank transactions, optimizing supply chain distribution, and developing recommendations and more effective medical treatments. However, comparing to many other data analytics, it is actually very difficult to perform graph analytics efficiently on modern computers due to the following three major challenges.

First, *the structures of graphs are often highly irregular*. For example, a small portion of vertices may own a large number of neighbors while most vertices have very few neighbors (i.e., the degree distribution follows the power law). The structure irregularity leads to computation irregularity. Under the popular "vertex-centric" programming model for graph analytics, each vertex is assigned to a thread to process based on states of its neighbors. As a result, threads assigned with high-degree vertices (vertices with a large

number of neighbors) take much longer to complete than those running on vertices of low degrees, leading to workload imbalance.

Second, *real-world graphs tend to be large*. For example, the Twitter follow graph (as of 2012) has over 175 million vertices and approximately 20 billion edges. It is oftentimes impractical to find individual machines with memory capacity that can accommodate such large graphs entirely, not to mention accelerators (such as GPUs) which have more limited memory capacity. As a result, such "oversized" graphs have to be partitioned, and each partition has to be loaded into the main memory (or device memory) every processing iteration, also known as out-of-core (or out-of-GPU-memory) processing. A basic design of the out-of-core processing strategy may lead to a huge amount data movements for large graphs, seriously impacting the performance of graph processing.

Third, *it remains open questions how graph processing systems should be designed for emerging graph analytics*. Given a new graph analytics, there are often different ways to model the underlying graph computations. It is non-trivial to estimate the performance under each model due to the influences of multiple factors, such as data locality, amount of parallelism, and computation redundancy.

1.2 Contributions of This Thesis

This thesis aims to address the above three challenges of graph processing under some specific contexts, as illustrated in Figure 1.1. First, **Chapter 2** focuses on improving the performance of graph analytics on modern highly-parallel processors – GPUs. Note that GPUs are conventionally designed for accelerating computations on regular data with

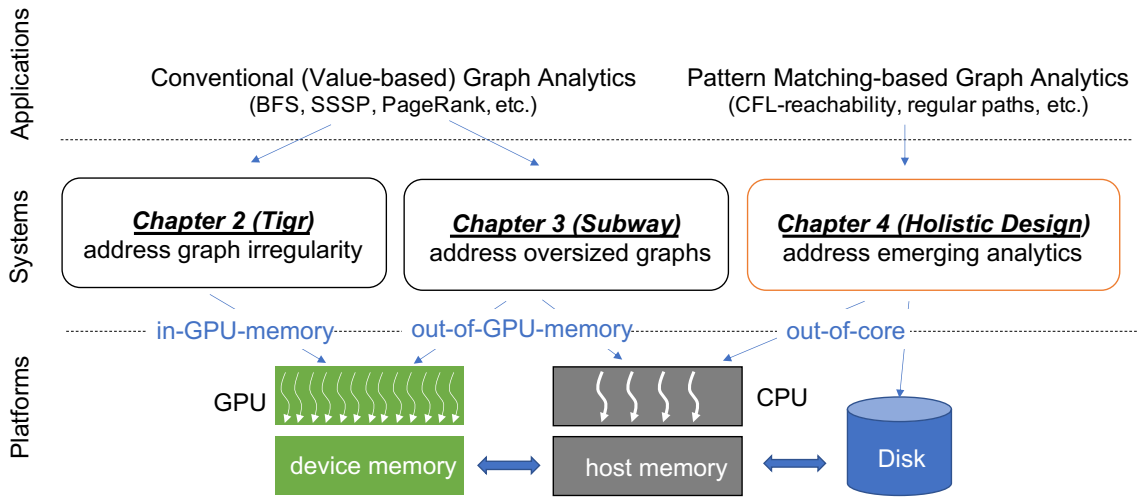


Figure 1.1: Contributions of This Thesis.

SIMD executions, but graph processing tends to be highly irregular. This mismatch can seriously compromise the computing power of GPUs. 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, Chapter 2 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, Chapter 2 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.

To address the second challenge, **Chapter 3** focuses on processing oversized graphs on the memory-limited GPUs (i.e., GPU memory oversubscription). When the graph size exceeds the GPU memory capacity, a large graph has to be partitioned and loaded into the GPU memory from the CPU memory partition by partition in each iteration of the graph processing. Due to the sheer amount of data transfer between CPU and GPU, the performance of graph processing often degrades dramatically. To reduce the volume of data transfer, existing approaches track the activeness of graph partitions and only load the ones that need to be processed. In fact, the recent advances of unified memory implements this optimization implicitly by loading memory pages on demand. However, either way, the benefits are limited by the coarse-granularity activeness tracking – each loaded partition or memory page may still carry a large ratio of inactive edges (i.e., edges needing not to be processed in the current iteration).

Different from prior efforts, Chapter 3 presents the first solution (to our best knowledge) that only loads the active edges of the graph to the GPU memory. To achieve this, it proposes a fast subgraph generation algorithm with a simple yet efficient subgraph representation and a GPU-accelerated implementation. They allow the subgraph generation to be applied in almost every iteration of the vertex-centric graph processing. Furthermore, the solution brings asynchrony to the subgraph processing, delaying the synchronization

between a subgraph in the GPU memory and the rest of the graph in the CPU memory. This can safely reduce the needs of generating and loading subgraphs for many common graph algorithms. A prototyped system, *Subway* (subgraph processing with asynchrony) is developed and evaluated. The results show over 4X speedup on average comparing with existing out-of-GPU-memory solutions and the unified memory-based approach, based on an evaluation with six common graph algorithms.

Finally, regarding the third challenge mentioned above, **Chapter 4** addresses the graph system design for an emerging graph application – large-scale program analysis. In general, it is challenging to perform interprocedural program analysis at large scale. To address this, recent research proposes to build dedicated graph systems for solving program analyses that can be formalized as context free language (CFL)-reachability problems, where new edges are iteratively inserted into an edge-labeled graph based on a set of context-free grammar rules. Despite some promising results, several major design questions remain to be systematically addressed to tape into the full potential of such a graph system.

Unlike the previous method, Chapter 4 presents a systematic exploration of the design space of the graph system used for large-scale program analysis by revealing and exploiting multiple key design aspects that critically affect the performance. First, inspired by the graph system design for classic graph problems, it introduces two basic computation models for CFL-reachability analysis, namely *vertex-centric* and *edge-centric* models, where each model exhibits its own pros and cons. On top of them, it proposes a *grammar-driven processing* scheme which enables the use of indexing to avoid unnecessary graph traversals and grammar rule matching. Finally, to ensure termination, the graph system needs to check

and remove duplicated edges when new edges are inserted to the graph, for which Chapter 4 shows that the *hashing-based mechanism* is more efficient than the existing sorting-based solution. Based on the above exploration, multiple representative graph systems for CFL-reachability analysis are implemented and evaluated on program analysis graphs extracted from real-world large system software. The results show that all the proposed graph systems can significantly outperform the state-of-the-art solution. Among them, the best performed one achieves significant speedups over the state-of-the-art under in-memory and out-of-core processing scenarios, respectively.

Note that Chapter 4 does not cover GPUs, instead, it focuses on CPU-based graph processing and out-of-core processing. However, the ideas proposed in Chapter 4 might be expanded to include GPUs as part of the underlying platform, which would allow to explore the synergy among the ideas from all three chapters (Chapters 2-4). This could be an interesting yet challenging future research work.

1.3 Dissertation Organization

This dissertation contains published works. Specifically, Chapter 2 is based on a paper accepted for publication at the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18) [92]. Chapter 3 is based on a paper published in the Proceedings of European Conference on Computer Systems (EuroSys'20) [110]. Chapter 4 is based on a to-be-submitted work which explores the design space of graph system for large-scale program analysis.

Chapter 2

Transforming Irregular Graphs for GPU-Friendly Graph Processing

2.1 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 [86], spotting frauds in bank transactions [111], optimizing supply chain distribution [133], and developing recommendations [29] and more effective medical treatments [20]. 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 [61] has shown orders

of magnitude efficiency improvement over traditional CPU-based graph processing, such as GraphLab [70]. 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 [73].

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 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 [61].

State of The Art. To address the above barrier, research so far either modifies GPU thread execution models [46, 40, 58] or changes the graph programming paradigm [35, 61, 60, 137]. For instance, warp segmentation [58] and maximum warp [46] 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

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

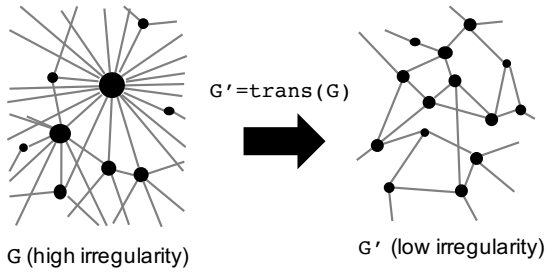


Figure 2.1: Illustration of Graph Irregularity Reduction.

to adapt as GPU architectures quickly evolve. By contrast, CuSha [61] and Gunrock [137] 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 2.1. Note that this is radically different from changing graph representations [61, 69] (e.g., CSR to CSR5 format) or partitioning graphs [70, 36]. *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.

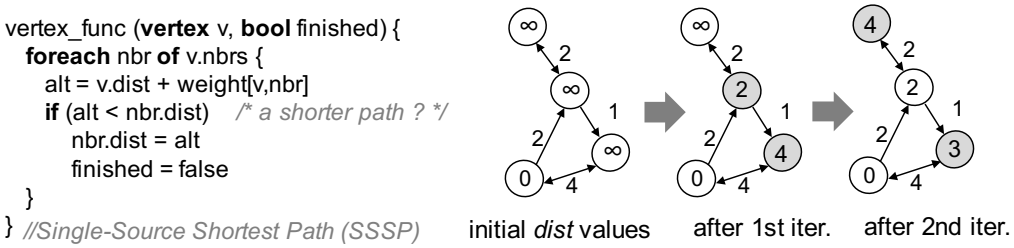


Figure 2.2: Example of (Push-based) Vertex-Centric Programming.

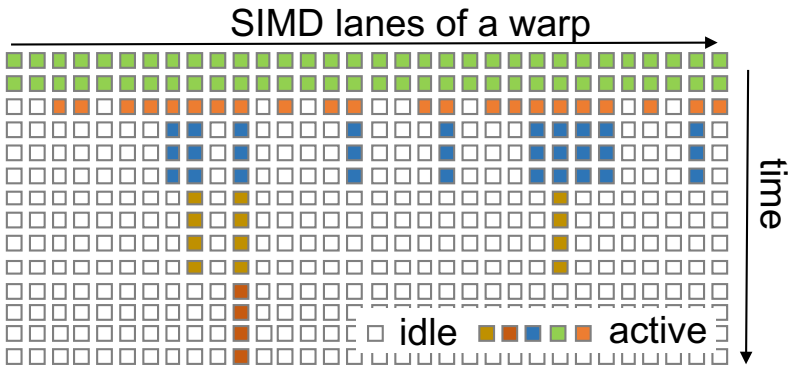


Figure 2.3: SIMD Execution.

- 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. (Repository: https://github.com/amirnodehi/Tigr)

2.2 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.

2.2.1 Vertex-Centric Programming

Graph analytics is notoriously difficult to parallelize due to its inherent dependencies [73]. 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 [76], this model has been widely adopted by many parallel graph engines, including Apache Giraph [10], GraphLab [70], PowerGraph [36], MaxWarp [46], CuSha [61], 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 [131], 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 2.2 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.

2.2.2 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 2.3.

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

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

2.2.3 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` [67], and `Hollywood` [17]) 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 2.2.1). 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.

2.3 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*.

2.3.1 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 2.2.1), 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 2.4. 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*

$$\mathcal{T} : (v,\ N_v) \mapsto (I \cup B,\ N_I \cup N_B) \tag{2.1}$$

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

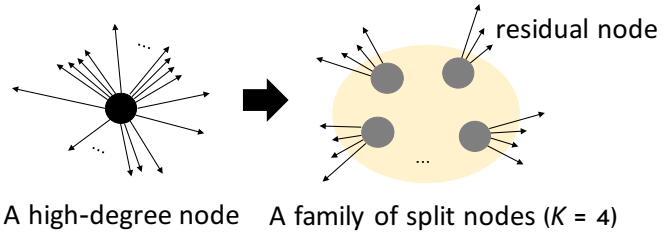


Figure 2.4: Illustration of Split Transformation.

where (i) I is the ***internal split node set*** *i.e., $N_I \cap N_v = \varnothing$; (ii) B is the* ***boundary split node set*** *i.e., $B = \{v' | N_{v'} \cap N_v \neq \varnothing\}$; (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 2.5). 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.

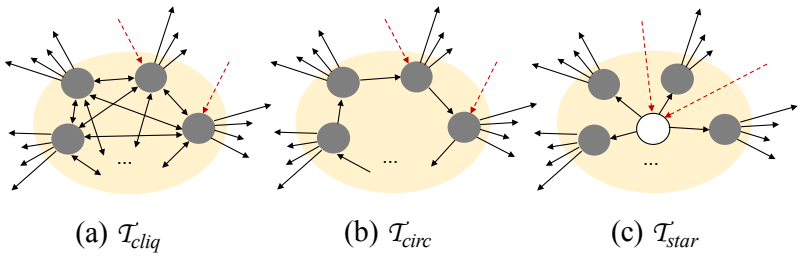


Figure 2.5: Three Example Connections.

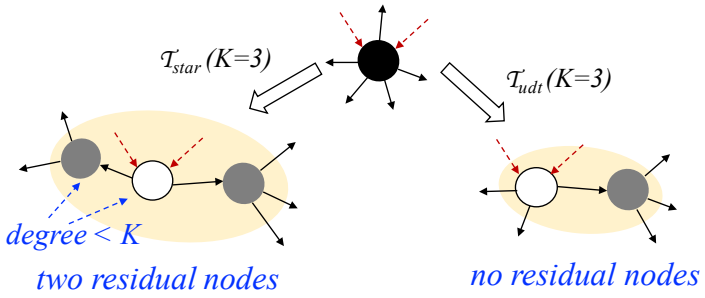


Figure 2.6: Comparison between $\mathcal{T}_{\text{star}}$ and $\mathcal{T}_{\text{udt}}$.

Table 2.1 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, $\mathcal{T}_{\text{cliq}}$ 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 2.1, $\mathcal{T}_{\text{circ}}$ performs the worst as it needs up to $\lceil d/K \rceil - 1$ hops to propagate a value from one split node (with

Table 2.1: 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

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.

2.3.2 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 2.6-(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 2.6-(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

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())
```

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 $O(D \cdot log_K(|E|/d))$.

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

2.3.3 Enforcing the Correctness for $\mathcal{T}_{\mathbf{udt}}$

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 3 *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,*

$$P'(v_1, v_2) = P(v_1, v_2) \cup E_{new} \tag{2.2}$$

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 2.7. 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

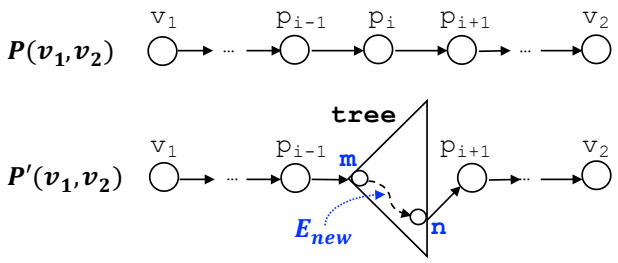


Figure 2.7: A Path Before and After UDT.

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

Based on Theorem 3, we have three corollaries.

Corollary 4 *UDT preserves graph connectivity.*

Proof. By the definition of connectivity and Theorem 3. ■

Corollary 4 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 5 and Corollary 5).

Corollary 5 *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 2.2 in Theorem 3. 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 5, 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 2.8 shows the UDT with dumb weights for SSSP. The distance between A and B remain six after transformation.

Corollary 6 *UDT preserves the minimal edge weight in a path by assigning weight infinity to all UDT-introduced edges.*

Proof. See Equation 2.2 in Theorem 3. 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 6 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 7 *For push-based and pull-based vertex-centric programming, UDT preserves the indegrees and outdegrees, respectively, for all nodes in the original graph.*

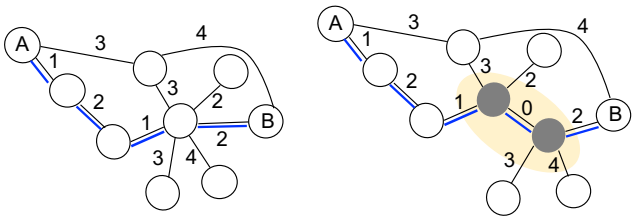


Figure 2.8: Example of UDT with dumb weights for SSSP.

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 7 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 slow-

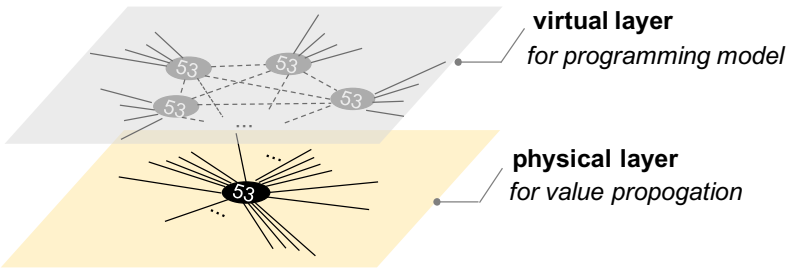


Figure 2.9: Illustration of Virtual Split Transformation.

down of value propagation (caused by splitting). To address these issues, we propose to *virtually* apply split transformations, without physically changing the graphs.

2.4 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.

2.4.1 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 2.9. 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* 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 2.3.1). 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

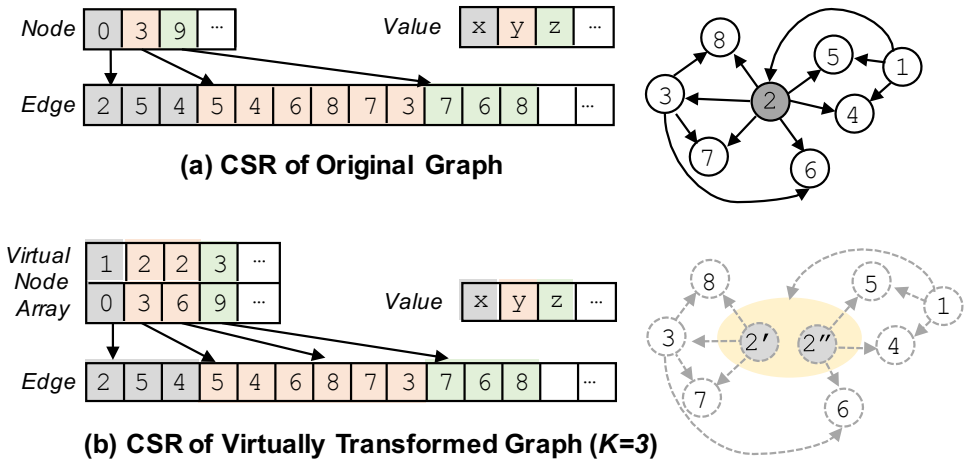


Figure 2.10: Integrating Virtual Node Array into CSR Format.

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 2.10, 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 3.4).

- *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 2.10. 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 [4] as shown in Figure 2.10. 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 2.10, 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 2.10. 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] Similar ideas are also applicable to *on-the-fly mapping reasoning*.

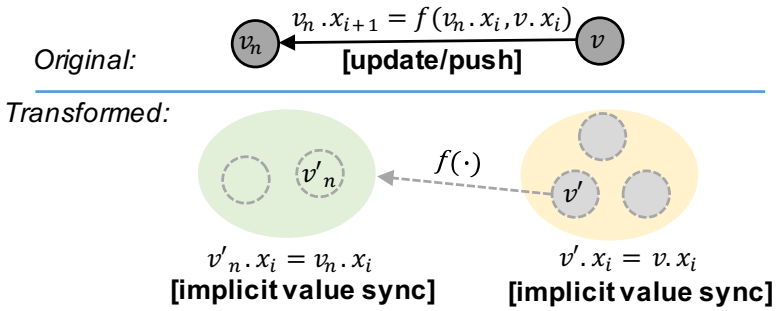


Figure 2.11: Correctness for Push-based Scheme.

2.4.2 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.

Theorem 8 *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 2.11. 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 9 *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 $v.w = f(v.w, v_1.w, v_2.w, \cdots, v_n.w)$[5], $v_i \in v.nbrs$. 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 [61], such as SUM, MIN, and MAX. These include popular ones like SSSP, BC, SSWP, BFS, and PR[6]. 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 8 and 9 guarantee correctness for vertex-centric graph analyses in a broad sense.

[5]We assume vertex function includes the node value itself as a parameter.
[6]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
```

2.4.3 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`.

2.4.4 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.

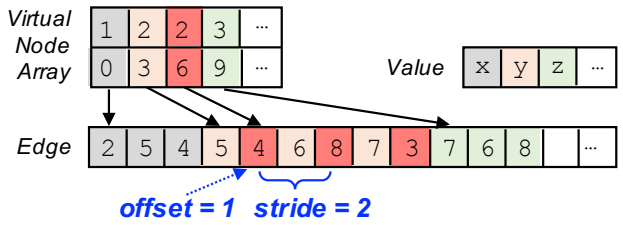


Figure 2.12: Edge-array Coalescing.

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 2.10. 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 2.10, 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 2.12). 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 way, when the virtual nodes of the same family are scheduled to the same warp (as they are consecutive in the

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
```

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.

2.5 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 2.3.1); For virtual transformation, *Tigr* uses *virtual node array* (Section 2.4.1) for lower runtime cost. In addition to *edge-array coalescing* (Section 2.4.4), 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.

2.6 Evaluation

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

2.6.1 Methodology

We compare *Tigr*-based GPU graph processing with three state-of-the-art general GPU graph processing frameworks: maximum warp [46], CuSha [61], and Gunrock [137]. Both implementations of CuSha and Gunrock are obtained from their public repositories. For maximum warp, we use an implementation from the CuSha repository. Table 2.2 lists the methods used in our comparison.

Besides, we compared with low-level implementations of some specific graph primitives, such as ECL-CC [51], Elsen and Vaidyanathan's PR [30], Davidson and others'

Table 2.2: Methods in Evaluation.

Abbr.	Framework
MW	Maximum warp w/ warp size range: 2˜32 [46]
CuSha	CuSha w/ G-Shards or Concatenated Windows [61]
Gunrock	Gunrock graph processing library [137]
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

Table 2.3: 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 [67]	1.6M	31M	8.8K	11	500	10
LiveJournal [67]	4.0M	69M	15K	13	1K	10
Hollywood [17]	1.1M	114M	11K	8	1K	10
Orkut [67]	3.1M	234M	33K	7	1K	10
Sinaweibo [108]	59M	523M	278K	5	10K	10
Twitter2010 [108]	21M	530M	698K	15	10K	10

SSSP [28], as well as the BFS by Merrill and others [82]. In fact, Gunrock [137] 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 [7].

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 2.3 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*

[7] `https://github.com/amirnodehi/Tigr`.

Table 2.4: 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**

search (BFS), *connected components* (CC), *single-source shortest path* (SSSP), *single-source widest path* (SSWP), *between centrality* (BC), and *PageRank* (PR).

2.6.2 Comparison With Existing Methods

Tables 2.4, 2.5 and 2.6 report the performance results of tested methods (we will discuss Tigr-UDT and Tigr-V in Section 2.6.3). For MW with varying virtual warp sizes, the best performance is chosen. Similarly, for CuSha, we report results of the better one

Table 2.5: Performance Comparison for SSWP algorithm.
execution time: ms; the best performance is bolded

Alg.	Dataset	MW	CuSha	Tigr-V+
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**

Table 2.6: Performance Comparison for CC algorithm.
execution time: ms; the best performance is bolded

Alg.	Dataset	Gunrock	Tigr-V+
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**

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. Our performance results indicate 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 2.4). 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 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.

2.6.3 Performance Breakdown of Tigr

Figure 2.13 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

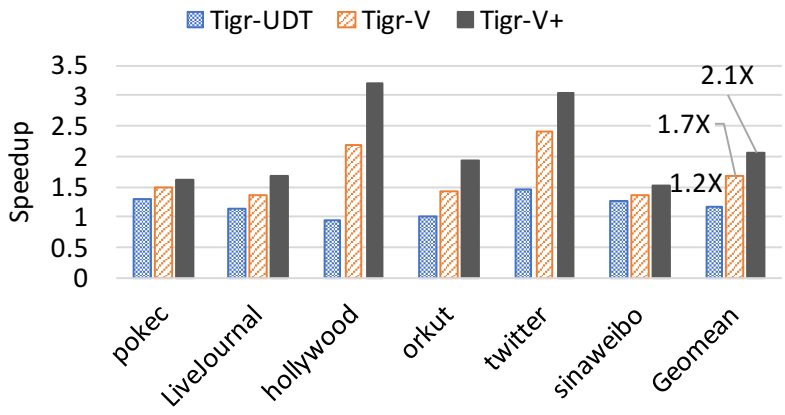


Figure 2.13: Speedups of *Tigr* over baseline (SSSP).

(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 2.6.5), 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 2.4.1).

Memory-coalescing Optimization. Besides, Figure 2.13 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 locality with more intelligent edge assignments to the virtual nodes (see Section 2.4.4 for more details).

Table 2.7: 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 2.8: 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%

2.6.4 Transformation Costs of Tigr

As mentioned earlier (Section 2.6.2), 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 2.7 and 2.8 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 2.3). 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 2.8, the space cost of virtual graph transformation is much higher due to the use of relatively smaller K. Because virtual split transfor-

Table 2.9: 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

Table 2.10: 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%

mation 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 2.6.2).

Time Cost. Table 2.9 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.

2.6.5 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 2.10 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.

2.7 Related Work

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

2.7.1 General Parallel Graph Processing

There is a rich body of work on designing distributed graph programming systems. Boost Graph Library [120], as an early effort, offers a high-level abstraction for programming graphs. To enable parallel execution, Gregor and Lumsdaine implement parallel BGL [38]. Inspired by the Bulk Synchronous Parallel model [131], Google designs the first vertex-centric graph programming framework Pregal [76]. Since then, vertex-centric graph programming has been adopted by many parallel graph engines, such as Apache Giraph [10], GraphLab [71], and PowerGraph [36]. Targeting distributed platforms, the above systems require to partition graphs and store the partitions across machines, based on edges [57, 98], vertices [36], or value vectors [148]. PowerLyra [24] has shown improved performance by differentiating the partitioning between high-degree and low-degree vertices.

Though vertex partitioning [36, 24] *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 [119] and Galois [99] support programming over a subset of vertices. Featuring amorphous data parallelism, Galois offers a new perspective on irregular graphs processing [90, 101]. Charm++ [54] and STAPL [31, 138] 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 [65], GraphQ [136], and Graspan [135].

In addition, some work focuses on specific parallel graph algorithms, such as connected components [39], BFS [13], SSSP[25, 84], and betweenness centrality [19] or the design choice between pull and push-based processing schemes [16]. Some recent work parallelizes automata executions which are essentially input-guided graph traversals [155, 154, 102].

2.7.2 Graph Processing on GPUs

By mapping nodes of a graph to GPU threads, Harish and others [43] implement a GPU graph processing framework based on vertex-centric programming. To minimize path divergence and load imbalance in GPU graph processing, Maximum warp [46] decomposes GPU warps into smaller sub-warps. By contrast, CuSha [61] 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 [137] 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 [74, 82], GPU-optimized con-

nected components [121, 39], SSSP based on Δ-stepping or hybrid approaches [93, 28] and betweenness centrality based on Brandes formulation [52, 113, 79].

Besides, GPU graph processing for specific applications, such as program analysis [81], and general applications, like compiler-level optimizations [95] have also been proposed. There are also a series of work on multi-GPU graph processing, including TOTEM [33], Medusa [158], METIS [56], as well as hybrid CPU-GPU methods [35, 47]. Graphie [41] and GraphReduce [116] 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.

2.7.3 GPU Efficiency Optimizations

Minimizing non-coalesced memory accesses is shown as a NP-complete problem [139]. To optimize memory access efficiency on GPUs, Dymaxion [23] and G-streamline [147] use methods like data reordering, memory remapping, and job swapping. Similar to the load balancing with local worklists [89], a queue-based approach handles irregularities in task loads [130]. By contrast, other work [145, 66] 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 [14] for SpMV operation on CSR matrices. Sartori and Kumar [114] 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 [88], and a study on identifying

the bottlenecks of implementing GPU applications, such as data transfers, kernel invocations and memory latencies [142].

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.

2.8 Summary

This chapter 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 chapter 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 chapter 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.

Chapter 3

Reducing Data Transfer during Out-of-GPU-Memory Graph Processing

3.1 Introduction

In many graph-based applications, graphs naturally grow over time. Considering web analytics [18, 94], the sizes of web graphs quickly increase as more webpages are crawled. In social networks [85, 115] and online stores [100], graphs related to user interactions (e.g., following and liking) and product purchases (e.g., who bought what item) also grow consistently. These growing graphs pose special challenges to GPU-based graph processing systems. When the size of an input graph exceeds the GPU memory capacity, known as

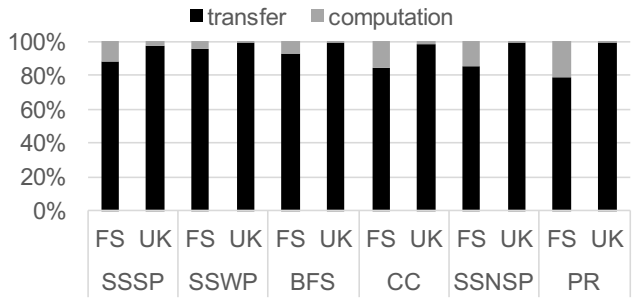


Figure 3.1: Time Breakdown of Partitioning-based Approach
(six analytics on two real-world graphs from Section 3.4).

memory oversubscription, existing GPU-based graph systems either fail to process (such as CuSha [61], Gunrock [137], and Tigr [92]), or process it with dramatic slowdown [116, 42].

State of The Art. Existing efforts [116, 62, 42] in addressing the GPU memory oversubscription for graph applications mainly follow ideas in out-of-core graph processing, such as GraphChi [64], X-Stream [109], and GridGraph [161]. Basically, the oversized graph is first partitioned, then explicitly loaded to the GPU memory in each iteration of the graph processing. Hereinafter, we refer to this explicit memory management as the *partitioning-based approach*. A major challenge for this approach is the low computation to data transfer ratio due to the nature of iterative graph processing. As a result, the data movement cost usually dominates the execution time, as shown in Figure 3.1. For this reason, one focus in optimizing this approach is *asynchronously* streaming the graph data to the GPU memory to overlap the data transfer with the GPU kernel executions [116, 62, 42]. Unfortunately, as Figure 3.1 indicates, for many graph applications, the computation cost is substantially smaller than the data transfer cost. Moreover, it varies significantly over iterations. Both factors limit the benefits of this overlapping. A more promising direction is to directly

reduce the data movements between CPU and GPU. To achieve this, GraphReduce [116] and Graphie [42] track the partitions with to-be-processed (active) vertices/edges and only load those to the GPU memory, which have shown promises in reducing data movements. However, the benefits of this activeness checking are limited to the partition level. For example, a partition with few active edges would still be loaded entirely. Moreover, for real-world power-law graphs, most partitions may stay active due to their connections to some high-degree vertices, further limiting the benefits.

As a more general solution, *unified memory* recently has become available with the release of CUDA 8.0 and the Pascal architecture (2016) [1]. It allows GPU applications to access the host memory transparently with memory pages migrated on demand. By adopting unified memory, graph systems do not have to track the activeness of graph partitions, instead, the memory pages containing active edges/vertices will be automatically loaded to the GPU memory triggered by the page faults. Despite the ease of programming and on-demand "partition" loading, unified memory-based graph systems suffer from two limitations: First, on-demand page faulting is not free. As shown later, there are significant overheads with page fault handling (such as TLB invalidations and page table updates); Second, similar to the explicit graph partition activeness tracking, the loaded memory pages may also contain a large ratio of inactive edges/vertices, wasting the CPU-GPU data transfer bandwidth.

Unlike prior efforts and unified memory-based approach, this work aims to load only the active edges (and vertices), which are essentially a subgraph, to the GPU memory.

[1] CUDA 6.0 supports automatic data migration between CPU and GPU memories, but does not support GPU memory oversubscription.

The challenges lie in the costs. Note that the subgraph changes in every iteration of the graph processing. The conventional wisdom is that repetitively generating subgraphs at runtime is often too expensive to be beneficial [132]. Contrary to the wisdom, we show that, with the introduction of (i) a concise yet efficient subgraph representation, called SubCSR; (ii) a highly parallel subgraph generation algorithm; and (iii) a GPU-accelerated implementation, the cost of the subgraph generation can be low enough that it would be beneficial to apply in (almost) every iteration of the graph processing.

On top of subgraph generation, we bring asynchrony to the in-GPU-memory subgraph processing scheme. The basic idea is to delay the synchronization between a subgraph (in GPU memory) and the rest of the graph (in host memory). After the subgraph is loaded to GPU memory, its vertex values will be propagated asynchronously (to the rest of the graph) until they have reached a local convergence before the next subgraph is generated and loaded. In general, this changes the behavior of value convergences and may not be applicable to every graph algorithm. However, as we will discuss later, it can be safely applied to a wide range of common graph algorithms. In practice, the asynchrony tends to reduce the number of iterations and consequently the number of times a subgraph must be generated and loaded.

Together, the proposed techniques can reduce both the number of times the input graph is loaded and the size of loaded graph each time. We prototyped these techniques into a runtime system called **Subway**. By design, Subway can be naturally integrated into vertex-centric graph processing systems with the standard CSR graph representation. Our evaluation with six commonly used graph applications on a set of real-world and synthesized

graphs shows that Subway can significantly improve the efficiency of GPU-based graph processing under memory oversubscription, yielding 5.6X and 4.3X speedups comparing to the unified memory-based approach and the existing techniques, respectively.

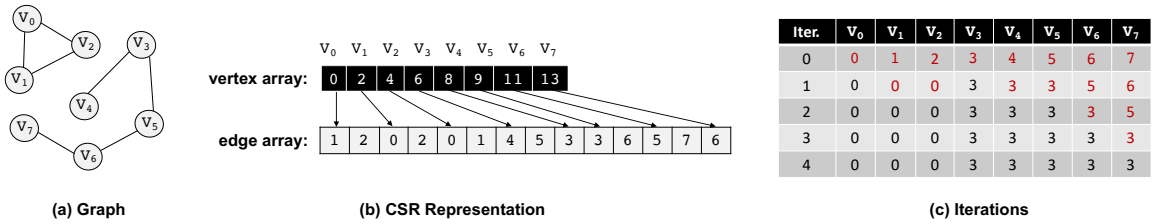
Contributions. In summary, this work makes three major contributions to GPU-based graph processing:

- First, this work presents a highly efficient subgraph generation technique, which can quickly and (often) significantly "shrink" the size of the loaded graph in each graph processing iteration.
- Second, it brings asynchrony to the in-GPU-memory subgraph processing, making it possible to reduce the times of subgraph generations and loading.
- Third, it compares the proposed techniques (Subway) with existing out-of-GPU-memory graph processing solutions, unified memory-based graph processing, as well as some of the state-of -the-art CPU-based graph systems. The source code of Subway is available at: https://github.com/AutomataLab/Subway.

Next, we first provide the background of this chapter.

3.2 Background

This section first introduces the basics of graph applications and their programming model, including a discussion of the major issues in GPU-based graph processing.



Iter.	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
0	0	1	2	3	4	5	6	7
1	0	0	0	3	3	3	5	6
2	0	0	0	3	3	3	3	5
3	0	0	0	3	3	3	3	3
4	0	0	0	3	3	3	3	3

Figure 3.2: Graph Representation and Vertex-Centric Graph Processing (Connected Components).

3.2.1 Graph Applications and Programming Model

As a basic yet versatile data structure, graphs are commonly used in a wide range of applications to capture their linked data and to reveal knowledge at a deeper level, ranging from influence analysis in social networks [87] and fraud detection in bank transactions [112] to supply chain distribution [134] and product recommendations [50]. As a sequence, there have been consistent interests in developing graph processing systems, covering shared-memory graph processing (e.g., Galois [91] and Ligra [119]), distributed graph processing (e.g., Pregel [76] and Distributed GraphLab [70]), out-of-core graph processing (such as GraphChi [64] and X-Stream [109]), and GPU-accelerated graph processing (such as CuSha [61] and Gunrock [137]). More details about these graph systems and others will be shown later in Section 3.5.

To support graph application developments, *vertex-centric programming* [76] has been widely adopted as a popular graph programming model, thanks to its simplicity, high scalability, and strong expressiveness. It defines a generic vertex function $f(\cdot)$ based on the values of neighbor vertices. During the processing, the vertex function is evaluated on all (active) vertices iteratively until all vertex values stop changing or the iterations have reached a limit. Depending on the value propagation direction, the vertex function can be

Algorithm 4 Vertex Function (CC) on CSR.

```
1: /* Connected Components */
2: procedure CC
3:     tid = getThreadID()
4:     if isActive[tid] == 0 then
5:         return
6:     sourceValue = value[tid]
7:     for i = vertex[tid] : vertex[tid+1] do
8:         nbr = edge[i]
9:         if sourceValue < value[nbr] then
10:            atomicMin(value[nbr], sourceValue)
11:            isActiveNew[nbr] = 1     /* active in next iter. */
```

either *pull*-based (gathering values along in-coming edges) or *push*-based (scattering values along out-going edges) [91].

Algorithm 4 illustrates the push-based vertex function for connected components (CC), where the graph is in CSR (Compressed Sparse Row) format, a commonly used graph representation that captures the graph topology with two arrays: *vertex array* and *edge array*. As shown in Figure 3.2-(b), the vertex array is made up of indexes to the edge array for locating the edges of the corresponding vertices (Line 8-9 in Algorithm 4). In addition, the vertex function also accesses *activeness labeling array* to check vertex activeness (Line 4) and update its neighbors' (Line 12), as well as the *value array* for reading and updating the connected component IDs (Line 7 and 10-11). Its iterations on the example graph are shown in Figure 3.2-(c). Initially, all vertices are active with connected component IDs the same as their vertex IDs. After three iterations, the vertices fall into two connected components, labelled with the smallest vertex IDs.

3.2.2 GPU-based Graph Processing

With the abundant parallelism exposed by the vertex-centric programming, GPUs built with up to thousands of cores have great potential in accelerating graph applications [44, 46, 83, 61, 137, 15, 92, 153]. Despite this promise, two main challenges arise in GPU-based graph processing: the *highly irregular graph structures* and the *ever-increasing graph sizes*.

The graph irregularity causes non-coalesced accesses to the GPU memory and load imbalances among GPU threads. Existing research on GPU-based graph processing, including CuSha [61], Gunrock [137], and Tigr [92], mainly focus on addressing the graph irregularity problem and have brought significant efficiency improvements. For example, CuSha brings new data structures (i.e., G-Shards and Concatenated Windows) to enable fully coalesced memory accesses. In compaison, Gunrock designs a frontier-based abstraction which allows easy integrations of multiple optimization strategies. More directly, Tigr proposes to transform the irregular graphs into more regular ones.

However, big gap remains in efforts towards the second challenge - processing oversized graphs. Despite that GPU memory capacity has been increasing, it is still too limited to accommodate many real-world graphs [116, 42]. There are two basic strategies to handle such cases: (i) *partitioning-based approach* and (ii) *unified memory-based approach*.

Partitioning-based Approach. This approach follows the ideas in out-of-core graph processing [64, 109, 161] – it first partitions the oversized graph such that each partition can fit into the GPU memory, then loads the graph partitions into the GPU memory in a round-robin fashion during the iterative graph processing. Most existing solutions adopt this strategy, including GraphReduce [116], GTS [62], and Graphie [42]. To improve

the processing efficiency under GPU memory oversubscription, two main optimizations have been proposed: the first one tries to hide some data transfer cost by asynchronously streaming the graph partitions to the GPU memory while the kernels are executing [116, 62, 42]; The second optimization tracks the activeness of each graph partition and only loads the ones with active edges (i.e., active partitions) to the GPU memory [116, 42]. Unfortunately, as discussed in Section 3.1, the benefits of both optimizations are limited by the sparsity nature of iterative graph processing – in most iterations, only a small subset of vertices tend to be active (a.k.a the frontier). First, the sparsity results in low computation-to-transfer ratio, capping the benefits of overlapping optimization. Second, in each loaded active partition, there might still be a large portion of inactive edges (i.e., their vertices are inactive) due to the sparsity.

Unified Memory-based Approach. Rather than explicitly managing the data movements, a more general solution is adopting unified memory [2], a technique has been fully realized in recent Nvidia GPUs. The main idea of unified memory is defining a managed memory space in which both CPU and GPU can observe a single address space with a coherent memory image [2]. This allows GPU programs to access data in the CPU memory without explicit memory copying. A related concept is *zero-copy memory* [1], which maps pinned (i.e., non-pageable) host memory to the GPU address space, also allowing GPU programs to directly access the host memory. However, a key difference is that, in unified memory, the memory pages containing the requested data are automatically *migrated* to the memory of the requesting processor (either CPU or GPU), known as *on-demand data migration*, enabling faster accesses for the future requests.

Adopting the unified memory for graph applications is straightforward: when allocating memory for the input graph, use a new API `cudaMallocManaged()`, instead of the default `malloc()`. In this way, when the graph is accessed by GPU threads and the data is not in the GPU memory, a page fault is triggered and the memory page containing the data (active edges) is migrated to the GPU memory. On one hand, this implicitly avoids loading memory pages with only inactive edges, sharing similar benefits with the partition activeness-tracking optimization in the partitioning-based approach. On the other hand, it also suffers from a similar limitation – loaded memory pages may contain a large ratio of inactive edges. Moreover, as we will show later, the page fault handling introduces substantial overhead, compromising the benefits of on-demand data migration.

In addition, unified memory can be tuned via APIs such as `cudaMemAdvise()` and `cudaMemPrefetchAsync()` [2] for better data placements and more efficient data transfer (more details will be given in Section 3.4). In addition, a recent work, ETC [68], also proposes some general optimization strategies, such as proactive eviction, thread throttling, and capacity compression. However, as we will show in Section 3.4, such general optimizations either fail to improve the performance or bring limited benefits, due to their inability in eliminating the expensive data transfer of inactive edges in each iteration of the graph processing.

In summary, both the existing partitioning and unified memory-based methods load the graph based on coarse-grained activeness tracking, fundamentally limiting their performance benefits. Next, we present Subway, a low-cost CPU-GPU graph data transfer solution based on fine-grained activeness tracking.

Table 3.1: Ratio of Active Vertices and Edges
V-avg (max): average ratio of active vertices (maximum ratio); E-avg (max): average ratio of active edges (maximum ratio).

	friendster-snap [67]		uk-2007 [4]	
Algo.	V-avg (max)	E-avg (max)	V-avg (max)	E-avg (max)
SSSP	4.4% (43.3%)	9.1% (85.1%)	4.6% (60.4%)	5.1% (67.7%)
SSWP	2.1% (38.4%)	5.2% (78.3%)	0.6% (12.6%)	0.6% (12.4%)
BFS	2.1% (32.3%)	4.1% (75.8%)	0.6% (12.6%)	0.6% (12.4%)
CC	8.1% (100%)	9.8% (100%)	3.2% (100%)	3.2% (100%)
SSNSP	2.1% (32.3%)	4.1% (75.8%)	0.6% (12.6%)	0.6% (12.4%)
PR	6.6% (100%)	24.1% (100%)	1.1% (100%)	1.7% (100%)

3.3 Subway

The core to Subway is a *fast subgraph generation* technique which can quickly extract a subgraph from a standard CSR formatted graph based on the specified vertices. When fed with the active vertices (i.e., the frontier), this technique can "shrink" the original graph such that only a subgraph needs to be loaded to the GPU memory. In addition, Subway offers an *in-GPU-memory asynchronous subgraph processing* scheme, which can reduce the needs of subgraph generation and loading. Together, these techniques can significantly bring down the cost of CPU-GPU data transfer, enabling efficient out-of-memory graph processing on GPUs. Next, we present these techniques in detail.

3.3.1 Fast Subgraph Generation

For many common graph algorithms, it is often that a subset of vertices are active (need to be processed) in an iteration of the graph processing. In most iterations, the ratio of active vertices is often very low, so as to the ratio of active edges. Table 3.1 reports the average and maximum ratios of active vertices and edges across all iterations, collected from six graph algorithms on two real-world graphs. In these tested cases, the

average ratios of active vertices and active edges are always below 10%. Motivated by this fact, we explore the possibility of separating the active parts of the graph from the rest and only load those to the GPU memory. This can greatly improve the state of the art [42, 116] which is only able to separate the inactive partitions. Despite the promise, the key challenge with this fine-grained separation is the cost. Recent work [132] shows that dynamically restructuring a graph could be very expensive in the out-of-core graph processing. Fortunately, in GPU-based graph processing, we can always leverage the power of GPU to accelerate this subgraph generation. Along with a concise design of the subgraph representation and a highly parallel generation algorithm, we find that the cost of subgraph generation can be quite affordable in comparison to the benefits that it brings. Next, we first introduce the subgraph representation, then present its generation algorithm.

Subgraph Representation. Subway assumes that the input graph is in CSR format [2], the commonly used graph format in GPU-based graph processing (see Section 3.2). Also, following a prior assumption [42], Subway allocates the vertex array (and its values) in the GPU memory and the edge array (usually dominating the graph size) in the host memory, as shown in Figure 3.3-(a). This design completely avoids writing data back to host memory at the expense of less GPU memory for storing the graph edges. Recall that the task is to separate the active vertices and edges (i.e., a subgraph) from the rest of the graph, while satisfying two objectives:

- First, the representation of the separated subgraph should remain concise, just like the CSR format;

[2]Note that using edge list, another common graph format, does not simplify the subgraph generation, but increases the memory consumption.

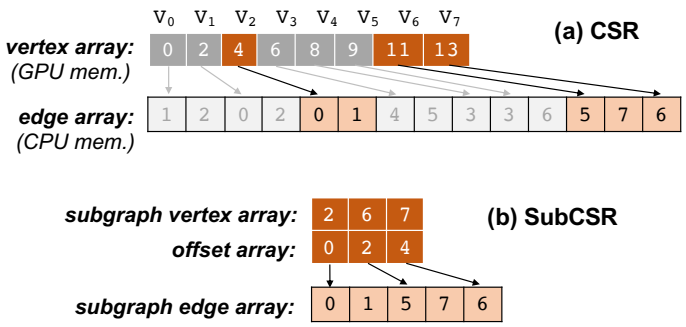


Figure 3.3: SubCSR Representation.

- Second, the vertices and edges of the subgraph can be efficiently accessed during the graph processing.

To achieve these goals, we introduce SubCSR – a format for representing a subset of vertices along with their edges in a CSR-represented graph. Figure 3.3-(b) shows the SubCSR for the active vertices and edges of the CSR in Figure 3.3-(a). At the high level, it looks close to the CSR representation. The main difference is that the vertex array in the CSR is replaced with two arrays: *subgraph vertex array* and *offset array*. The former indicates the positions of active vertices in the original vertex array, while the latter points to the starting positions of their edges in the *subgraph edge array*. In the example, the active vertices are $\mathtt{V}_2$, $\mathtt{V}_6$, and $\mathtt{V}_7$, hence their indices (2, 6, and 7) are placed in the subgraph vertex array. The subgraph edge array in the SubCSR consists only the edges of selected vertices. Obviously, the size of SubCSR is linear to the number of selected vertices plus their edges.

To demonstrate the access efficiency of SubCSR, we next illustrate how SubCSR is used in the (sub)graph processing. Algorithm 5 shows how a GPU thread processes the subgraph after it is loaded to the GPU memory. Comparing it with Algorithm 4, except that `vertex[]` and `edge[]` are replaced with `offset[]` and `subEdge[]`, the only difference is at

Algorithm 5 Vertex Function (CC) on SubCSR.

```
1: /* Connected Components */
2: procedure CC
3:     tid = getThreadID()
4:     vid = subVertex[tid] /* difference: an extra array access */
5:     if isActive[vid] == 0 then
6:         return
7:     sourceValue = value[vid]
8:     for i = offset[tid] : offset[tid+1] do
9:         nbr = subEdge[i]
10:        if sourceValue < value[nbr] then
11:            atomicMin(value[nbr], sourceValue)
12:            isActiveNew[nbr] = 1
```

Line 4, an extra access to the subgraph vertex array `subVertex[]`. This extra array access may slightly increase the cost of vertex evaluation. However, as shown in the evaluation, this minimum increase of computation can be easily outweighed by the significant benefits of subgraph generation.

Next, we describe how to generate the SubCSR efficiently for a given set of active vertices with the help of GPU.

Generation Algorithm. Algorithm 6 and Figure 3.4 illustrate the basic ideas of GPU-accelerated SubCSR generation. The inputs to the algorithm include the CSR (i.e., `vertex[]` and `edge[]`), the vertex activeness labeling array `isActive[]` [3], and the degree array `degree[]` (can also be generated from CSR). The output of the algorithm is the SubCSR for active vertices (i.e., `subVertex[]`, `offset[]`, and `subEdge[]`). At the high level, the generation follows six steps.

Step-1: Find the indices of active vertices `subIndex[]` using an exclusive prefix sum over the vertex activeness labeling array `isActive[]` (Line 2). Assuming the active vertices

[3]Note that using a labeling array of the same length as `edge[]` may simplify the subgraph generation, but this large labeling array may not fit into the GPU memory, thus making its maintenance very expensive.

Algorithm 6 SubCSR Generation.

```
1: procedure GENSUBCSR(vertex[], edge[], isActive[], degree[])
2:     subIndex[] = gpuExclusivePrefixSum(isActive[])
3:     subVertex[] = gpuSC1(isActive[], subIndex[])
4:     subDegree[] = gpuResetInactive(isActive[], degree[])
5:     subOffset[] = gpuExclusivePrefixSum(subDegree[])
6:     offset[] = gpuSC2(isActive[], subIndex[], subOffset[])
7:     subEdge = cpuSC(vertex[], edge[], subVertex[], offset[])
8:     return subVertex[], offset[], subEdge[]
9:
10: procedure GPUSC1(isActive[], subIndex[])          ▷ /* GPU stream compact vertex indices */
11:     tid = getThreadID()
12:     if isActive[tid] == 1 then
13:         subVertex[subIndex[tid]] = tid
14:     return subVertex[]
15: procedure GPURESETINACTIVE(isActive[], degree[])   ▷ /* GPU reset degrees of inactive vertices */
16:     tid = getThreadID()
17:     if isActive[tid] == 0 then
18:         subDegree[tid] = 0
19:     else
20:         subDegree[tid] = degree[tid]
21:     return subDegree[]
22: procedure GPUSC2(isActive[], subIndex[], subOffset[])   ▷ /* GPU stream compact offset array */
23:     tid = getThreadID()
24:     if isActive[tid] == 1 then
25:         offset[subIndex[tid]] = subOffset[tid]
26:     return offset[]
27: procedure CPUSC(vertex[], edge[], subVertex[], offset[])   ▷ /* CPU stream compact edge array */
28:     parallel for i = 0 to numActiveVertices do
29:         v = subVertex[i]
30:         subEdge[offset[i]:offset[i+1]]
31:         = edge[vertex[v]:vertex[v+1]]
32:     end parallel for
33:     return subEdge[]
```

are V_2, V_6, and V_7, this step puts the IDs of active vertices into corresponding positions of `subIndex[]` (see Figure 3.4).

Step-2: Create the subgraph vertex array `subVertex[]` with a stream compaction based on `subIndex[]`, `isActive[]`, and the array index `tid` (Line 3, 11-18). If a vertex is active, put its ID (such as 2, 6, or 7 in the example) into `subVertex[]`.

Step-3: Based on the degree array of CSR `degree[]` (assume it is available or has been generated), create a degree array `subDegree[]` for the active vertices, where the degrees of inactive vertices are reset to zeros (Line 4, 20-29).

Step-4: Compute the offsets of active vertices `subOffset[]` with an exclusive prefix sum over `subDegree[]` (Line 5). In the example, the offset of the first active vertex (V_2) is always zero, and since V_2 has two edges (see `degree[]`), the second active vertex (V_6) has offset two. Similarly, V_7 has an offset of four. Note that this step depends on the reset in Step-3.

Step-5: Compact `subOffset[]` into `offset[]` by removing the elements of inactive vertices (Line 6, 31-38). Note that this step needs not only `isActive[]`, but also `subIndex[]`.

Step-6: Finally, compact the edge array `edge[]` to `subEdge[]` by removing all inactive edges (Line 7, 40-48). This requires to access the CSR as well as `subVertex[]` and `offset[]`. Basically, edges of active vertices are copied from `edge[]` to `subEdge[]`. Note that `offset[]` is critical here in deciding the target positions of the copying.

Though there are six steps, each step only involves a few simple operations. More importantly, all the six steps are highly parallel, making it straightforward to take advantage of the massive parallelism of GPU. More specifically, as the comments in Algorithm 6 indicate, the first five steps are offloaded to the GPU, where the activeness labeling array `isActive[]` is maintained. After the fifth step, two arrays (`subVertex[]` and `offset[]`) are transferred back to the host memory, where the sixth step is performed. At the end of the generation, the SubCSR for the active vertices are formed in the host memory and are ready to be loaded to the GPU memory.

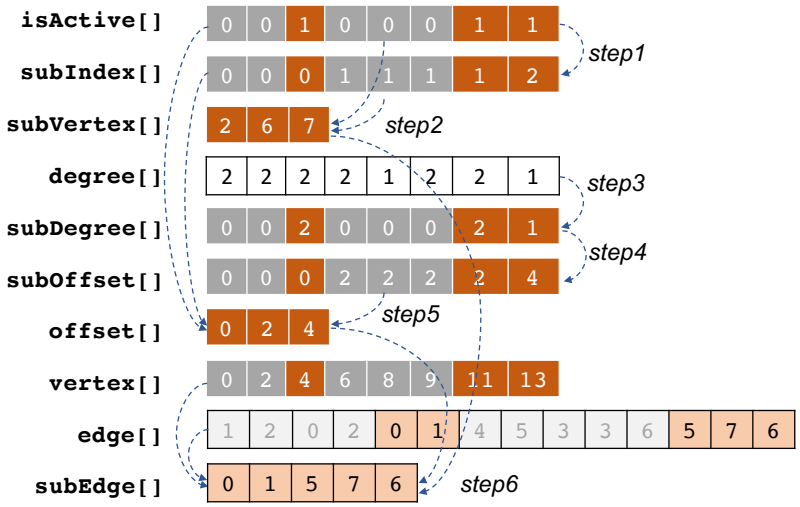


Figure 3.4: SubCSR Generation for Example in Figure 3.3.

Cost-Benefit Analysis. In Algorithm 6, the first five steps have a time complexity of $O(|V|)$, where V is the vertex set; while the last step takes $O(|E_{active}|)$, where E_{active} is the set of active edges. Hence, the overall time complexity of SubCSR generation is $O(|V| + |E_{active}|)$. Since set V is fixed, the cost of SubCSR generation varies depending on the amount of active edges. On the other hand, the benefit of SubCSR generation comes from the reduced data transfer: instead of loading the whole edge array (with a size of $|E|$, where E is the edge set) to the GPU memory, only the SubCSR of active vertices is loaded, with a size of $|E_{active}| + 2 * |V_{active}|$, where V_{active} is the set of active vertices. Note that, there is also data transfer during the SubCSR generation (between Step-5 and Step-6), with a size of $2 * |V_{active}|$. Therefore, the total saving of data transfer is $|E| - |E_{active}| - 4 * |V_{active}|$. Assuming the CPU-GPU data transfer rate is r_{trans}, then the time saving $S_{trans} = r_{trans} * (|E| - |E_{active}| - 4 * |V_{active}|)$.

Assuming the concrete cost of SubCSR generation is C_{gen}, that is, $C_{gen} = O(|V| + |E_{active}|)$, then, theoretically speaking, if $C_{gen} < S_{trans}$, applying SubCSR would bring net

benefit to the out-of-memory graph processing on GPUs. In practice, we can set a threshold for enabling SubCSR generation based on a simpler metric – the *ratio of active edges*, denoted as P_{active}. As P_{active} increases, the cost of SubCSR generation increases, but the benefit decreases. Hence, there is a sweet spot beyond which the SubCSR generation will not bring any net benefit (i.e., the threshold). Based on our evaluation (Section 3.4), we found $P_{active} = 80\%$ is a threshold that works well in general for tested applications and graphs. In fact, for most tested cases, we found P_{active} is below (often well below) 80% for almost all iterations of the graph processing, making the SubCSR generation applicable across (almost) all iterations. When P_{active} is beyond 80%, SubCSR generation would be disabled and the conventional partitioning-based approach would be employed as a substitution.

Oversized SubCSR Handling. Though P_{active} is usually low enough that the generated SubCSR can easily fit into the GPU memory, there are situations where the SubCSR remains oversized. To handle such cases, the conventional partitioning-based approach can be adopted. Basically, an oversized SubCSR can be partitioned such that each partition can fit into the GPU memory, then the SubCSR partitions are loaded into GPU memory and processed one after another. The partitioning of SubCSR is similar to the partitioning of the original graph (CSR): logically partition the subgraph vertex array `subVertex[]` such that each vertex array chunk, along with its offset array `offset[]` and subgraph edge array `subEdge[]`, are close to but smaller than the available GPU memory size. Since logical partitioning is free and the total cost of SubCSR loading remains the same, handling oversized SubCSR keeps the benefits of SubCSR generation.

3.3.2 Asynchronous Subgraph Processing

Traditionally, there are two basic approaches to evaluate the vertex function: the *synchronous approach* [76] and the *asynchronous approach* [70, 64]. The former only allows the vertices to synchronize at the end of each iteration, that is, all (active) vertices read values computed from the last iteration. This design strictly follows the bulk-synchronous parallel (BSP) model [131]. By contrast, the asynchronous approach allows the vertex function to use the *latest values*, which may be generated in the current iteration (intra-iteration asynchrony). In both schemes, a vertex is only *evaluated once per iteration*. This simplifies the development of graph algorithms, but results in a low *computation to data transfer ratio*, making the data transfer a serious bottleneck under GPU memory oversubscription. To overcome this obstacle, Subway offers a more flexible vertex function evaluation strategy – *asynchronous subgraph processing*.

Asynchronous Model. Under this model, after a subgraph (or a subgraph partition) is loaded into the GPU memory, it will be processed asynchronously with respect to the rest of the graph in the host memory. Algorithm 7 illustrates its basic idea. Each time when a subgraph partition, say P_i, is loaded to the GPU memory (Line 6), the vertices in P_i are iteratively evaluated until there is no active ones in P_i (Line 7-11). This adds a second level of iteration inside the whole-graph level iteration (Line 1-13). The outer level iteration ensures that the algorithm will finally reach a global convergence and terminate, while the inner level iteration maximizes the value propagations in each loaded subgraph partition. Since the inner iteration makes the vertex values "more stable" – closer to their eventual values, the outer level iterations tend to converge faster. Thus, there will be less need for

generating and loading subgraphs (Line 3 and 6). Next, we illustrate this idea with the example in Figure 3.5.

Example. Initially, all the vertices are active (also refer to Figure 3.2-(c)). Assume that their edges cannot fit into the GPU memory, hence the (sub)graph is partitioned into two parts: $P1$ and $P2$, as shown in Figure 3.5-(a). First, $P1$ is loaded to the GPU memory and processed iteratively until their values are (locally) converged. Note that, at this moment, the values of V_4 and V_5 have been updated (under a push-based scheme). After that, $P2$ is loaded to the GPU memory and processed in the same way. Since vertices in $P2$ observe the latest values (3 for V_4 and V_5), they can converge to "more stable" values, in this case, their final values. The whole processing takes only one outer iteration to finish, comparing to three iterations in the conventional processing (see Figure 3.2-(c)).

Note that choosing the above asynchronous model does not necessarily reduce the total amount (GPU) computations, which include both the inner and outer iterations. According to our experiments, the change to the total computations fluctuates slightly case by case (see Section 3.4). On the other hand, the saving from reduced subgraph generation and loading is significant and more consistent. As shown in the above example, it only needs to load the (sub)graph into the GPU memory once, rather than three times.

Related Ideas. One closely related processing scheme is the TLAG model ("*think like a graph*") proposed in distributed graph processing [129]. In this case, a large input graph is partitioned and stored on different computers connected by the network. During the iterative processing, different computers send messages to each other to synchronize their

Algorithm 7 Asynchronous Subgraph Processing

```
1: do /* whole-graph-level iteration */
2:     V_active = getActiveVertices(G)
3:     G_sub = genSubCSR(G, V_active)
4:     /* the subgraph may be oversized, thus partitioned */
5:     for P_i in partitions of subgraph G_sub do
6:         load P_i to GPU memory
7:         do /* partition-of-subgraph-level iteration */
8:             for v_i in vertices of P_i do
9:                 f(v_i)  /* evaluate vertex function */
10:        while anyActiveVertices(P_i) == 1
11: while anyActiveVertices(G) == 1
```

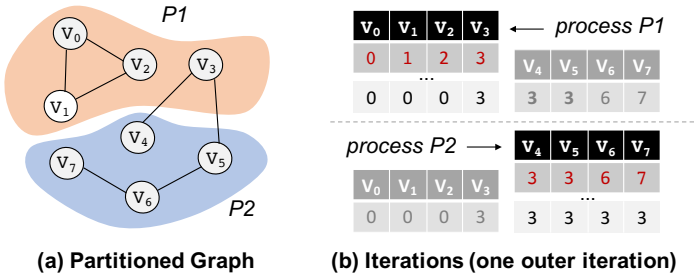


Figure 3.5: Example under Asynchronous Model.

values. To reduce the amount of messages generated in the distributed system, TLAG lifts the programming abstraction from vertex level to partition level. In the new abstraction, graph programming becomes partition-aware, permitting vertex values to be freely propagated within each partition. As the vertex values may become "more stable" when they are propagated to the other partitions, the communications across partitions tend to be reduced, so as to the messages generated in the distributed systems.

Both TLAG and asynchronous subgraph processing extend the conventional intra-iteration asynchrony to some kind of inter-partition asynchrony. However, the key difference is that, in TLAG, all partitions are processed simultaneously, while in asynchronous subgraph processing, the partitions are processed in serial, one partition at a time. The dif-

ference makes the latter "more asynchronous" than the former: in a partition-by-partition processing, a later loaded partition can directly observe the latest values computed from earlier partitions, while in TLAG, the latest values from the other partitions are not exposed until all partitions have reached their local convergences. Due to this reason, asynchronous subgraph processing may converge even faster than TLAG. For example, it takes one more outer iteration for the example in Figure 3.5 to converge when implemented in TLAG.

Correctness. The use of asynchronous subgraph processing may alter the value propagation priority (i.e., preferring to evaluate vertices in the "current" partition), similar to TLAG. Therefore, it may not be suitable for every graph algorithm, especially those that are sensitive to the value propagation order. For example, prior work [129] has shown that TLAG is applicable to three major categories of graph algorithms: *graph traversal*, *random walk*, and *graph aggregation*. But, for some other algorithms (e.g., PageRank), it requires some changes to the algorithm design. As a general sufficient (but not necessary) condition, as long as the final vertex values do not depend on the vertex evaluation order, it is ensured that the asynchronous subgraph processing will preserve the convergence and the converged values. It is not hard to manually verify that the commonly used graph traversal algorithms, such as SSSP (single-source shortest path), BFS (breath-first search), SSWP (single-source widest path), and CC (connected components), all satisfy the above condition. In addition, after adopting the accumulative update-based algorithm [152], PageRank can also satisfy this condition, thus runs safely under asynchronous subgraph processing. The correctness of these graph algorithms has also been verified by our extensive experiments (see Section 3.4).

Note that, automatically reasoning about the necessary correctness conditions for a specific graph algorithm under asynchronous subgraph processing is a challenging research problem not covered by this work. For this reason, Subway provides asynchronous subgraph processing as an optional feature, enabled only when the correctness has been verified.

3.3.3 Implementation

We prototyped Subway as a runtime system with both the fast subgraph generation technique (Section 3.3.1) and the asynchronous subgraph processing (Section 3.3.2). In order to demonstrate the effectiveness of Subway, we integrated it into an existing open-source GPU-based graph processing framework, called Tigr [4] [92]. In fact, Subway uses Tigr for in-memory graph processing when the input graph fits into GPU memory. Another reason we choose Tigr is for its use of the standard vertex-centric programming and the CSR graph representation, which make it immediately ready to adopt Subway. For simplicity, we refer to this integrated graph system as Subway when the reference context is clear. By default, the fast subgraph generation is enabled when the ratio of active vertices in the current iteration is beyond the threshold 80%. By contrast, the asynchronous subgraph processing scheme is disabled by default for the correctness reason mentioned earlier. In the implementation of fast subgraph generation, we used the `Thrust` library [6] for the exclusive prefix sum in the first and fourth steps of the SubCSR generation, and the `Pthread` library for the parallel edge array compaction in the sixth step (see Section 3.3.1).

In-Memory Processing. Subway automatically detects the size of the input graph, when the graph fits into the GPU memory, it switches to the in-memory processing mode (i.e.,

[4] `https://github.com/AutomataLab/Tigr`

Tigr). The performance of Subway would be the same as Tigr, which has been compared to other well-known GPU-based frameworks such as Gunrock [137] and CuSha [61], showing promising results [92]. On the other hand, when the graph cannot fit into the GPU memory, optimizations from Tigr would be disabled and the system mainly relies on Subway runtime (subgraph generation and asynchronous processing if applicable) for performance optimizations.

3.4 Evaluation

In this section, we evaluate the prototyped system Subway under the scenarios of GPU memory oversubscription, with an emphasis on the cost of CPU-GPU data movements and the overall graph processing efficiency.

3.4.1 Methodology

Our evaluation includes the following approaches:

- *Basic Partitioning-based Approach* (`PT`): This one follows the basic ideas of partitioning-based memory management without further optimizations. It logically partitions the graph based on the vertex array, then loads the partitions into the GPU memory one by one during each iteration of the graph processing. We include this approach to make the benefits reasoning of other approaches easier.

- *Optimized Partitioning-based Approach* (`PT-Opt`): On top of `PT`, we incorporated several optimizations from existing solutions [116, 42], including asynchronous data streaming (with 32 streams), active partition tracking (see Section 3.1), and reusing loaded active

partitions [42]. This approach roughly approximates the state-of-the-art in GPU-based graph processing under memory oversubscription.

- *Optimized Unified Memory-based Approach* (`UM-Opt`): To adopt unified memory, we allocated the graph (i.e., edge array) with `cudaMallocManaged()` [5], so active edges can be automatically migrated to the GPU memory as needed. After that, we tried to optimize this approach based on CUDA programming guidance [2] and ideas from a recent work on unified memory optimizations, ETC [68]. First, we provided a *data usage hint* via `cudaMemAdvise()` API to make the edge array `cudaMemAdviseSetReadMostly`. As the edge array does not change, this hint will avoid unnecessary writes back to the host memory. Based on our experiments, this optimization reduces the data transfer by 47% and total processing time by 23% on average. Second, we applied *thread throttling* [68] by reducing the number of active threads in a warp. However, we did not observe any performance improvements. A further examination revealed two reasons: (i) the maximum number of threads executing concurrently on a GPU is much smaller than the number of vertices (i.e., 2048×30 on the tested GPU versus tens of millions of vertices), so the actual working set is much smaller than the graph (assuming that each thread processes one vertex), easily fitting into the GPU memory; (ii) the accesses to the edge array exhibit good spatial locality thanks to the CSR representation. For these reasons, thread throttling turned out to be not effective. Besides the above two optimizations, another optimization we considered but found not applicable is *prefetching* [2]. The challenge is that the active vertices in next processing iteration are unpredictable, so to the edges

[5]The version of CUDA driver is v9.0.

needed to load. Finally, some lower-level optimizations such as memory page compression [68] and page size tuning [6] might be applicable, but they are not the focus of this work; we leave systematic low-level optimizations to the future work.

To the best of our knowledge, this is the first time that unified memory is systematically evaluated for GPU-based graph processing, since it is fully realized recently.

- *Synchronous Subgraph Processing* (`Subway-sync`): In this approach, the asynchronous processing scheme in Subway is always disabled. Therefore, its measurements will solely reflect the benefits of subgraph generation technique.

- *Asynchronous Subgraph Processing* (`Subway-async`): In the last approach, the asynchronous processing scheme in Subway is enabled, but may be disabled temporarily as needed, depending on the ratio of active edges.

Datasets and Algorithms. Table 3.2 lists the graphs used in our evaluation, including five real-world graphs and one synthesized graph. Among them, `friendster-konect` and `twitter-mpi` are from the Koblenz Network Collection [3], `friendster-snap` is from the Stanford Network Analysis Project [67], `uk-2007` and `sk-2005` are two web graphs from the Laboratory for Web Algorithmics [4], and `RMAT` is a widely used graph generator [21]. In addition, Table 3.2 reports the numbers of vertices and edges, the range of estimated diameters, and the in-memory graph sizes with and without the edge weights, respectively.

There are six widely used graph analytics evaluated. They include breath-first search (BFS), connected components (CC), single-source shortest path (SSSP), single-source

[6] The default page size, 4KB, is used in our evaluation.

Table 3.2: Graph Datasets
$|V|$: number of vertices; $|E|$: number of edges; Est. Dia.: estimated diameter range; Size_w: in-memory graph size (CSR) with edge weights; Size_{nw}: in-memory graph size (CSR) without edge weights.

Abbr.	Dataset	$\|V\|$	$\|E\|$	Est. Dia.	Size_w	Size_{nw}
SK	sk-2005 [4]	51M	1.95B	22-44	16GB	8GB
TW	twitter-mpi [3]	53M	1.96B	14-28	16GB	8GB
FK	friendster-konect [3]	68M	2.59B	21-42	21GB	11GB
FS	friendster-snap [67]	125M	3.61B	24-48	29GB	15GB
UK	uk-2007 [4]	110M	3.94B	133-266	32GB	16GB
RM	RMAT [21]	100M	10.0B	5-10	81GB	41GB

widest path (SSWP), single-source number of shortest path (SSNSP), and PageRank (PR). Note that SSSP and SSWP work on weighted graphs, thus, the sizes of their input graphs are almost doubled comparing to other algorithms. To support asynchronous subgraph processing, PageRank and SSNSP are implemented using accumulative updates [152].

Evaluation Platform. We evaluated Subway mainly on a server that consists of an NVIDIA Titan XP GPU of 12 GB memory and a 64-core Intel Xeon Phi 7210 processor with 128 GB of RAM. The server runs Linux 3.10.0 with CUDA 9.0 installed. All GPU programs are compiled with `nvcc` using the highest optimization level.

Out-of-GPU-memory Cases. With edge weights (required by SSSP and SSWP), none of the six graphs in Table 3.2 fit into the GPU memory. In fact, besides the first two graphs (`SK` and `TW`), the sizes of the other four graphs, as shown in the second last column of Table 3.2, are well beyond the GPU memory capacity (12GB). Without weights, three graphs (`SK`, `TW`, and `FK`) fit into the GPU memory. In the following, we only report results of the out-of-GPU-memory cases.

Next, we first compare the overall performance of different approaches, then focus on evaluating `UM-Opt` and the two versions of Subway: `Subway-sync` and `Subway-async`.

3.4.2 Overall Performance

Table 3.3 reports the overall performance results, where the `PT` column reports the raw execution time, while the following ones show the speedups of other methods over `PT`.

First, `PT-Opt` shows consistent speedup over `PT`, 2.0X on average, which confirms the effectiveness of optimizations from existing work [42, 116]. By contrast, `UM-Opt` does not always outperform `PT`, depending on the algorithms and input graphs. The numbers in italics correspond to the cases `UM-Opt` runs slower than `PT`. We will analyze the benefits and costs of `UM-Opt` in detail shortly in Section 3.4.3. On average, `UM-Opt` still brings in 1.5X speedup over `PT`.

Next, `Subway-sync` shows consistent improvements over not only `PT` (6X on average), but also existing optimizations `PT-Opt` (3X on average) and unified memory-based approach `UM-Opt` (4X on average). The significant speedups confirm the overall benefits of the proposed subgraph generation technique, despite its runtime costs. Later, in Section 3.4.4, we will breakdown its costs and benefits.

Finally, `Subway-async` shows the best performance among all, except for six algorithm-graph cases, where `Subway-sync` performs slightly better. More specifically, it yields up to 41.6X speedup over `PT` (8.5X on average), 15.4X speedup over `PT-Opt` (4.3X on average), and 12.2X speedup over `UM-Opt` (5.7X on average). In addition, it outperforms synchronous Subway (`Subway-sync`) by 1.4X on average. These results indicate that, when asynchronous subgraph processing is applicable (ensured by developers), it is worthwhile to adopt it under the out-of-GPU-memory scenarios. We will evaluate `Subway-async` in depth in Section 3.4.5.

Table 3.3: Performance Results
Numbers (speedups) in bold text are the highest among the five methods; Numbers (speedups) in italics are actually slowdown comparing to PT.

		PT	PT-Opt	UM-Opt	Subway-sync	Subway-async
SSSP	SK	118.3s	1.5X	3.5X	5.8X	**9.5X**
	TW	20.4s	1.7X	*0.8X*	2.9X	**6.0X**
	FK	53.0s	1.7X	*0.6X*	4.2X	**8.0X**
	FS	68.5s	1.6X	*0.7X*	4.2X	**6.7X**
	UK	492.7s	2.9X	1.9X	6.5X	**15.6X**
	RM	66.6s	1.3X	*0.6X*	2.0X	**3.1X**
SSWP	SK	174.7s	1.8X	5.2X	13.2X	**23.1X**
	TW	19.7s	2.2X	1.2X	4.4X	**7.0X**
	FK	50.3s	2.1X	1.2X	7.4X	**13.1X**
	FS	71.3s	1.8X	1.1X	8.0X	**12.5X**
	UK	350.8s	3.7X	4.9X	**38.8X**	36.3X
	RM	58.3s	1.1X	*0.5X*	2.2X	**3.7X**
BFS	FS	30.9s	1.9X	*0.9X*	6.7X	**9.6X**
	UK	176.3s	3.2X	10.3X	**28.8X**	21.8X
	RM	32.7s	1.5X	*0.7X*	3.2X	**3.7X**
CC	FS	22.9s	2.1X	1.1X	4.2X	**5.7X**
	UK	388.1s	5.7X	4.0X	10.9X	**26.0X**
	RM	25.5s	1.3X	*0.5X*	1.9X	**2.3X**
SSNSP	FS	59.1s	1.5X	*0.9X*	4.4X	**5.6X**
	UK	349.4s	4.0X	8.9X	**25.6X**	25.2X
	RM	61.7s	1.1X	*0.8X*	**4.6X**	3.9X
PR	FS	278.4s	2.5X	1.9X	**2.8X**	2.2X
	UK	577.9s	2.7X	3.4X	16.5X	**41.6X**
	RM	319.5s	1.2X	*0.9X*	**3.2X**	3.0X
GEOMEAN			2.0X	1.5X	6.0X	**8.5X**

3.4.3 Unified Memory: Benefits and Costs

To better understand the inconsistent benefits from unified memory, we further analyze its benefits and costs with more detailed measurements. First, as mentioned earlier, recent releases of unified memory (CUDA 8.0 and onwards) come with on-demand data migration, which essentially resembles the partition activeness-tracking optimization [116, 42]. With this new feature, CUDA runtime only loads the memory pages containing the

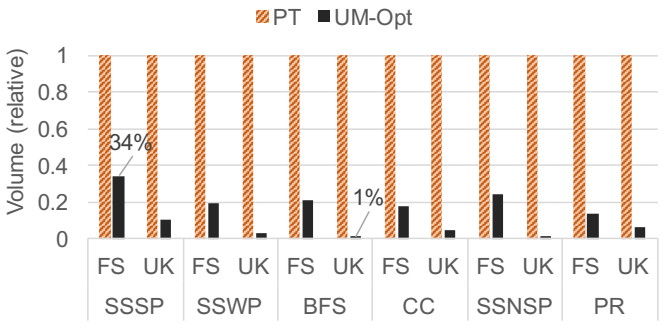


Figure 3.6: CPU-GPU Data Transfer (by volume).

data that GPU threads need, skipping the others. In the context of vertex-centric graph processing, on-demand data migration avoids loading the memory pages consisting only inactive edges. To confirm this benefit, we measured the volume of data transfer between CPU and GPU in `UM-Opt` and compared it with that in `PT`. The data was collected with the help of Nvidia Visual Profiler 9.0 [5]. As reported in Figure 3.6, comparing to `PT`, the data transfer in `UM-Opt` is greatly reduced, up to 99% (occurred to `BFS-UK`). Moreover, unified memory also simplifies the programming by implicitly handling out-of-GPU-memory cases.

Despite the promises, the benefits of using unified memory for graph processing are limited by two major factors. First, the on-demand data migration is not free. When a requested memory page is not in the GPU memory, a page fault is triggered. The handling of page faults involves not only data transfer, but also TLB invalidations and page table updates, which could take tens of microseconds [2]. To estimate the significance of this extra overhead, we first collected the data transfer cost and the cost of graph computations, then subtracted them from the total runtime of `UM-Opt` [7]. The remaining time is used as

[7]We also tried to measure the page fault cost with Nvidia Visual Profilers 9.0 and 10.0, which unfortunately do not produce reasonable results.

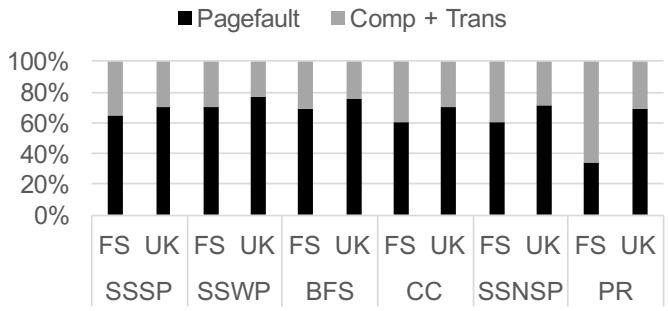


Figure 3.7: Page Fault Overhead in Unified Memory.

the estimation of page fault-related overhead. Figure 3.7 reports the breakdown, where overhead related page fault takes 23% to 69% of the total time. These substantial overhead may outweigh the benefits of reduced data transfer. As shown in Table 3.3 (`UM-Opt` column), in 5 out of 18 tested cases, `UM-Opt` runs slower than `PT`.

The second major factor limiting the benefits of unified memory-based graph processing lies in the *granularity* of data migration – memory pages: a migrated page may still carry inactive edges due to the sparse distributions of active vertices. We will report the volume of unnecessary data migration in the next section. While reducing memory page size may mitigate this issue, it increases the costs of page fault handling as more pages would need to be migrated.

3.4.4 Subgraph Generation: Benefits and Costs

Similar to the unified memory-based approach, using our subgraph generation brings both benefits and costs. The benefits come from reduced data transfer – only active edges are transferred to the GPU memory. On the other hand, the costs of subgraph generation occur on the critical path of the iterative graph processing – the next iteration

waits until its subgraph (SubCSR) is generated and loaded. In addition, there is a minor cost in the subgraph processing due to the use of SubCSR, as opposed to CSR (see Section 3.3.1). Before comparing its costs and benefits, we report the frequencies of SubCSR generation and partitioning first.

Table 3.4 reports how frequently the SubCSR is generated and how many times the SubCSR/CSR is partitioned across iterations. Here, we focus on `Subway-sync`, we will discuss `Subway-async` shortly in Section 3.4.5. Note that the criterion for enabling SubCSR generation is that the ratio of active edges is over 80%. Among 24 algorithm-graph combinations, in 11 cases, this ratio is always under 80%; in 10 cases, the ratio exceeds 80% only in one iteration; and in the remaining three cases (`CC-RM`, `PR-FS`, and `PR-RM`), the ratio exceeds 80% more often: in 2 iterations, 11 iterations, and 7 iterations, respectively. This is because `RM` is the largest graph among the tested ones and algorithms `PR` and `CC` often activate more edges due to their nature of computation – all vertices (and edges) are active (100%) initially according to the algorithms.

When SubCSR is not generated (i.e., the activeness ratio is below 80%), the graph (CSR) has to be partitioned; Otherwise, the graph (SubCSR) only needs to be partitioned if it remains oversized for the GPU. Both partitioning cases in general happen infrequently, except for algorithms `PR` and `CC` and graph `RM`, due to the same reasons just mentioned earlier.

Next, we report the benefits of reduced data transfer. As Figure 3.8 shows, the volume of data transfer in `Subway-sync` is dramatically reduced comparing to `PT`, with a reduction ranging from 89.1% to 99%. It is worthwhile to note that the reduction in

Table 3.4: SubCSR Generation and Partitioning Statistics
Itr: total number of iterations; $Itr_{>80\%}$: number of iterations with more than 80% active edges (iteration IDs); $Itr_{>GPU}$: number of iterations with SubCSR greater than GPU memory capactity (iteration IDs).

		Subway-sync			Subway-async		
		Itr	$Itr_{>80\%}$	$Itr_{>GPU}$	Itr	$Itr_{>80\%}$	$Itr_{>GPU}$
SSSP	SK	90	0	0	86	1(1)	1(1)
	TW	15	1(5)	1(5)	10	1(1)	1(1)
	FK	30	1(7)	3(6-8)	22	1(1)	1(1)
	FS	27	1(7)	3(6-8)	21	1(1)	1(1)
	UK	187	0	16(17-32)	179	1(1)	1(1)
	RM	8	1(3)	2(3-4)	8	1(1)	3(1-3)
SSWP	SK	134	0	0	115	1(1)	1(1)
	TW	15	0	0	6	1(1)	1(1)
	FK	30	1(7)	2(6-7)	18	1(1)	1(1)
	FS	30	0	2(6-7)	25	1(1)	1(1)
	UK	134	0	0	122	1(1)	1(1)
	RM	10	1(3)	2(3-4)	9	1(1)	1(1)
BFS	FS	24	0	1(6)	15	1(1)	1(1)
	UK	134	0	0	122	1(1)	1(1)
	RM	6	1(3)	1(3)	4	1(1)	1(1)
CC	FS	15	1(1)	2(1-2)	8	1(1)	1(1)
	UK	291	1(1)	6(1-6)	122	1(1)	1(1)
	RM	4	2(1-2)	2(1-2)	4	1(1)	1(1)
SSNSP	FS	24	0	1(6)	18	1(1)	1(1)
	UK	134	0	0	127	1(1)	1(1)
	RM	6	0	1(3)	5	1(1)	2(1-2)
PR	FS	75	11(1-11)	17(1-17)	44	1(1)	12(1-12)
	UK	359	1(1)	3(1-3)	310	1(1)	1(1)
	RM	45	7(1-7)	21(1-21)	33	5(1-5)	14(1-14)

`Subway-sync` is more significant than that in `UM-Opt` (Figure 3.8 vs. Figure 3.6). On average, the data transfer volume in `UM-Opt` is 3.3X of that in `Subway-sync`. The extra 2.3X data transfer is due to the saving of loading inactive edges carried by the migrated memory pages.

As to the cost of subgraph generation, instead of reporting its cost ratios, we combine the costs of subgraph generation and the subgraph transfer together (a breakdown

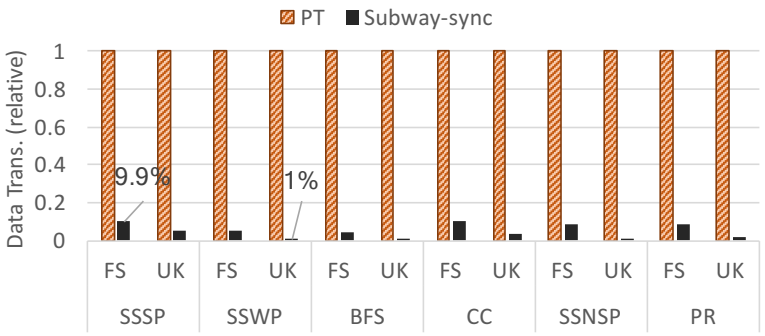


Figure 3.8: CPU-GPU Data Transfer (by volume).

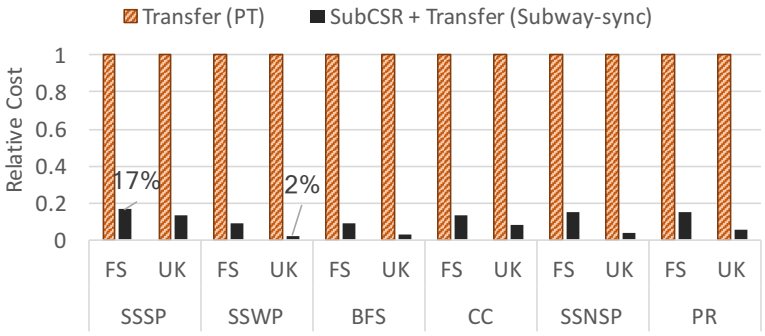


Figure 3.9: Time Costs of SubCSR Generation + Data Transfer.

between the two can be found in Figure 3.11), then compare the total cost to the transfer cost without subgraph generation. As reported in Figure 3.9, even adding the two costs together, the total remains significantly less than the data transfer cost without subgraph generation, with a reduction of time ranging from 83% to 98%. These results confirm the subgraph generation as a cost-effective way to reduce the data transfer. For this reason, `Subway-sync` exhibits significantly higher speedup than `UM-Opt` on average (see Table 3.3).

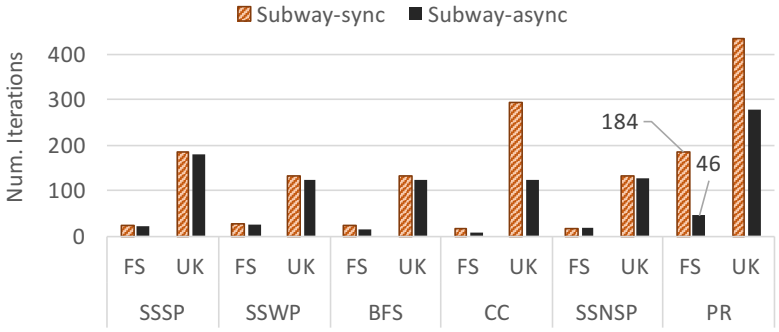


Figure 3.10: Impacts on Numbers of (Outer) Iterations.

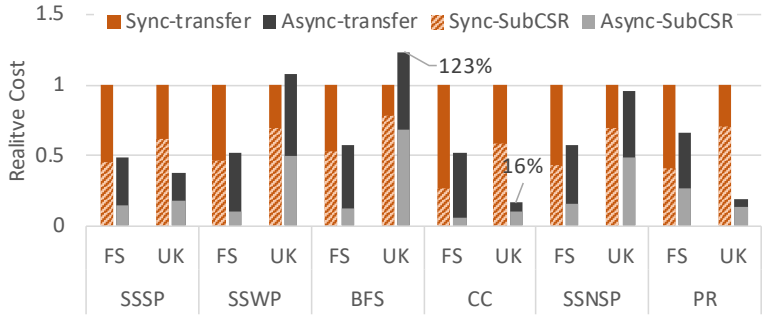


Figure 3.11: Impacts on SubCSR Generation + Transfer.

3.4.5 Asynchronous Processing: Benefits and Costs

Next, we examine the benefits and costs of asynchronous subgraph processing. As discussed in Section 3.3.2, adopting the asynchronous model tends to reduce the (outer) iterations of graph processing, thus saving the needs for subgraph generations and loading. To confirm this benefit, we profiled the total number of iterations in the outer loop of graph processing (also refer to Algorithm 7). Figure 3.10 compares the numbers of iterations with and without the asynchronous model. In general, the numbers of iterations are consistently reduced across all tested cases, except for `SSNSP-FS`, where the number of iterations remains unchanged. On average, the number of iterations is reduced by 31%. Correspondingly, the number of times for generating a subgraph and loading it to GPU is also reduced, as shown

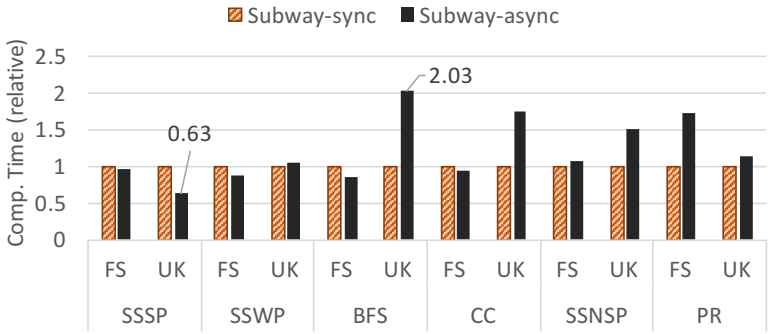


Figure 3.12: Impacts on Graph Computation Time.

on the right of Table 3.4. Note that in asynchronous processing mode, to maximize the value propagation within a subgraph (i.e., more inner iterations), `Subway-async` always partitions and loads the entire graph in the first outer iteration.

However, it is interesting to note that the costs of subgraph generation and data transfer may not be reduced in the same ratios as the number of iterations, as indicated by Figure 3.11. For example, in the case of `PR-UK`, the number of iterations is reduced from 434 to 278 (about 36% reduction). However, its costs of subgraph generation and data transfer are reduced more significantly, by 71%. The opposite situations may also happen (e.g., in the case of `PR-FS`). The reason is that the asynchronous model affects not only the number of (outer) iterations, but also the amounts of active vertices and edges in the next (outer) iteration, thus altering the cost of subgraph generation and the volume of data transfer. In general, the overall impacts on the costs of subgraph generation and data transfer are positive, leading to a 52% reduction on average. Among the 12 examined cases, only 2 cases (`SSWP-UK` and `BFS-UK`) show cost increases, by up to 23%.

At last, by altering the way that values are propagated, the asynchronous model may also change the overall amount of graph computations, as reported in Figure 3.12. In

general, the changes to the graph computation time vary across tested cases, ranging from 0.63X to 2.03X.

Adding the above impacts together (Figures 3.11 and 3.12), adopting the asynchronous model remains beneficial overall, yielding speedups in 18 out of 24 tested cases (see Table 3.3). For the others, the performance loss is within 10% on average.

3.4.6 Out-of-GPU-Memory vs. CPU-based Processing

Instead of providing out-of-GPU-memory graph processing support, another option is switching to the CPU-based graph processing, though this will put more pressure on the CPU, which may not be preferred if CPU is already overloaded. Nonetheless, we compare the performance of the two options for reference purposes. Note that the performance comparison depends on the models of CPU and GPU. In our setting, the GPU is Nvidia Titan XP with 3840 cores and 12GB memory while the CPU is Intel Xeon Phi 7210 processor with 64 cores. Both processors are hosted on the same machine with a RAM of 128GB. Note that the cost of the CPU is 4 to 5X more expensive than the GPU (according to our purchasing price). The CPU-based graph processing system we chose for comparison is a state-of-the-art system, Galois [91]. We also tried Ligra [119], another popular shared-memory graph system, however, due to its high memory demand, we found none of our tested graphs can be successfully processed on our machine. Table 3.5 reports the running time of both graph systems for the graph algorithms that both natively support. Note that Galois provides alternative implementations for each graph analytics. We used the best implementation in our setting: `default` for BFS and CC, and `topo` for SSSP. Overall, we found the performance of Subway is comparable to Galois in our experimental setup. In

Table 3.5: Subway (Out-of-GPU-memory) vs. Galois (CPU)
GPU: Titan XP (3840 cores, 12GB);
CPU: Xeon Phi 7210 (64 cores); RAM: 128GB

		Subway-sync	Subway-async	Galois
SSSP	SK	20.28s	12.51s	**5.42s**
	TW	6.94s	**3.41s**	7.94s
	FK	12.54s	**6.65s**	22.129s
	FS	16.17s	**10.26s**	29.28s
	UK	75.47s	31.54s	**13.44s**
	RM	33.1s	**21.49s**	29.63s
BFS	FS	4.65s	**3.21s**	8.35s
	UK	6.12s	8.1s	**5.07s**
	RM	10.2s	**8.72s**	17.32s
CC	FS	5.49s	**4.03s**	5.23s
	UK	35.76s	14.93s	**4.88s**
	RM	13.7s	11.11s	**10.47s**

fact, Subway (async version) outperforms Galois in 7 out of 12 tested algorithm-graph combinations. Also note that, as an out-of-GPU-memory solution, the time of Subway includes not only all the data transfer time from CPU to GPU, but also the SubCSR generation time.

3.5 Related Work

Graph Processing Systems. There have been great interests in designing graph processing systems. Early works include Boost Graph Library [120] and parallel BGL [38]. Since the introduction of Pregal [76], vertex-centric programming has been adopted by many graph engines, such as Giraph [10], GraphLab [70], and PowerGraph [36]. More details about vertex-centric graph processing systems can be found in a survey [77]. A number of graph processing systems are built on distributed platforms to achieve high scalability [76, 70, 36, 37, 24, 160, 26, 122, 27]. They partition graphs and store the partitions

across machines, based on edges [57, 98], vertices [36], or value vectors [148]. Some of the distributed graph processing systems adopt the idea of asynchronous processing among machines to improve the convergence rate and/or reduce the communication costs [70, 129, 27].

On shared-memory platforms, Ligra [119] and Galois [91] are widely recognized graph processing systems for their high efficiencies. Charm++ [55] and STAPL [127] are two more general parallel programming systems, with intensive supports for irregular computations, like graph processing.

A more relevant body of research is the out-of-core graph processing systems. Some representative systems include GraphChi [64], X-Stream [109], GraphQ [136], GridGraph [161], CLIP [8], Wanderland [149], Graspan [135], and among others. Some of their ideas have been adopted for handling GPU memory oversubscription, as discussed earlier. In addition, ideas in some prior work [132, 8] are also closely related to the techniques proposed in this work, but different in both contexts and technical details. In prior work [132], a dynamic graph partitioning method is proposed for disk-based graph processing, which operates on shards [64], a data structure optimized for disk-based graph processing. In comparison, our subgraph generation is based on CSR, a more popular representation in GPU-based graph processing. Furthermore, our technique features a new subgraph representation and a GPU-accelerated generation. In another prior work, CLIP [8], local iterations are applied to a graph partition loaded from the disk, which resembles our asynchronous model, but in a different context. Moreover, they only applied it to two graph analytics. In comparison, we have discussed the correctness of the asynchronous model and successfully applied it to a much broader range of graph analytics.

GPU-based Graph Processing. On GPU-based platforms, research has been growing to leverage the computation power of GPUs to accelerate graph processing. Early works in this area include the one from Harish and others [43], Maximum warp [46], CuSha [61], Medusa [159], and many algorithm-specific GPU implementations [83, 121, 39, 93, 52, 32]. More recently, Gunrock [137] introduced a frontier-based model for GPU graph processing, IrGL [96] presents a set of optimizations for throughput, and Tigr [92] proposed a graph transformation to reduce the graph irregularity.

Most of the above systems assume that the input graph can fit into the GPU memory, thus they are unable to handle GPU memory oversubscription scenarios. To address this limitation, GraphReduce [116] proposed a partitioning-based approach to explicitly manage the oversized graphs, with the capability to detect and skip inactive partitions. More recently, Graphie [42] further improved the design of the partitioning-based approach, with an adoption of X-Stream style graph processing and a pair of renaming techniques to reduce the cost of explicit GPU memory management.

In general, Subway is along the same direction as the above systems, with two critical advancements. First, it introduces a GPU-accelerated subgraph generation technique, which pushes the state-of-the-art partition-level activeness tracking down to the vertex level. Second, it brings asynchrony to in-GPU-memory subgraph processing to reduce the needs for subgraph generations and reloading. As demonstrated in Section 3.4, both techniques can significantly boost the graph processing efficiency under memory oversubscription.

Besides single-GPU graph processing scenarios, prior work also built graph systems on multi-GPU platforms [59, 97, 15, 159, 53] and proposed hybrid CPU-GPU processing

scheme [34, 75, 45]. In these cases, the idea of asynchronous processing can be adopted among different graph partitions to reduce inter-GPU and CPU-GPU communications, similar to that in distributed graph processing (such as TLAG [129]).

3.6 Summary

For GPU-based graph processing, managing GPU memory oversubscription is a fundamental yet challenging issue to address. This chapter provides a highly cost-effective solution to extracting a subgraph that only consists of the edges of active vertices. As a consequence, the volume of data transfer between CPU and GPU is dramatically reduced. The benefits from data transfer reduction outweigh the costs of subgraph generation in (almost) all iterations of graph processing, bringing in substantial overall performance improvements. In addition, this chapter introduces an asynchronous model for in-GPU-memory subgraph processing, which can be safely applied to a wide range of graph algorithms to further boost the processing efficiency. In the evaluation, the proposed system (Subway) is compared with not only the existing techniques, but also an optimized unified memory-based implementation. The results confirm both the effectiveness and efficiency of the proposed techniques.

Chapter 4

Towards a Holistic Graph System Design for CFL-Reachability Analysis

4.1 Introduction

As software applications become more sophisticated and more deeply integrated into daily life, concerns regarding their correctness, efficiency, and security also increase. To address these concerns, researchers and engineers have been increasingly leveraging program analysis to find bugs [11] and reveal software vulnerabilities [9], not to mention its original uses in performance optimizations [7]. In general, there are strong interests in increasing both the analysis precision, as well as the ability to scale to large system software.

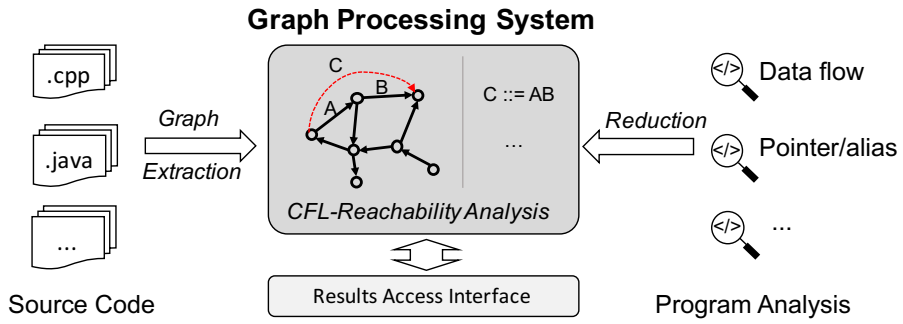


Figure 4.1: Graph System for Program Analysis [135].

Recently, Wang and others [135] proposed a novel dedicated graph system, called Graspan, to support large-scale program analysis with high precision. As illustrated by Figure 4.1, the basic idea is to leverage a classic formalization [106] that can reduce an interprocedural program analysis to an equivalent CFL (Context-Free Language)-reachability problem, where an edge-labeled graph is extracted from the source code and a corresponding context-free grammar is derived based on the analysis. By iteratively concatenating symbols along graph paths and matching them against the grammar rules, new edges with labels are generated and inserted to the graph, until no new edges can be found (i.e., reaching a fixed point). This formalization enables the use of graph systems to solve program analysis problems. Thanks to its "big data" perspective, the graph system makes it possible to analyze large system software, such as Linux kernel and PostgreSQL database, within hours or even minutes [135]. As a consequence of the analyses, more and deeper-level software issues have been discovered and reported.

Despite its promising results, this work finds that the full potential of a graph system for program analysis has not yet been realized. There are several major design questions that need to be addressed in depth:

- First, from a graph system design's perspective, what are the alternative ways to model the computations involved in CFL-reachability analysis? What are the pros and cons of each computation model, in terms of redundancy, data locality, and the degree of parallelism?

- Second, comparing to the conventional value-based graph systems (like Ligra [119] and Galois [91]), are there any new opportunities for optimizing a graph system dedicated for grammar rule matching?

- Third, to prevent edge duplications, the existing graph system leverages sorting to identify and remove duplicated edges. However, is it possible to avoid inserting duplicated edges in the first place? Can it be more cost-efficient?

- Finally, during out-of-core processing, is it possible for the graph system to avoid loading graph partitions in pairs, which in theory requires quadratic times of loading with respect to the number of partitions?

The above questions may critically affect the performance of a graph system for large-scale program analysis. The main goal of this work is to *systematically explore the design space of graph processing system for CFL-reachability analysis by addressing the aforementioned questions.*

First, inspired by the conventional graph systems [78, 109] designed for solving classic value-based graph problems, like single source shortest path (SSSP) and PageRank, this work categorizes the computations involved in CFL-reachability analysis into two basic models, which are referred to as the *vertex-centric* model and *edge-centric* model.

- The *vertex-centric* model arranges computations from the perspective of a vertex—it traverses the neighbors of *a given vertex* to find possible new edges of the vertex;
- The *edge-centric* model, instead, organizes computations from the perspective of an edge—it examines the adjacent edges of *a given edge* to discover new edges.

Interestingly, as this work reveals, the two "perspectives" can lead to distinct computation patterns with substantially different data locality, redundancy (unnecessary grammar rule matching), and the degree of parallelism. As shown later in evaluation, depending on their interactions with the input graph and the grammar, the performance gaps between the two computation models can vary dramatically.

Second, most existing graph systems are designed mainly for value-based analysis (e.g., SSSP and PageRank). Unlike these scenarios, CFL-reachability analysis is centered around grammar rule matching, which is based on symbols, rather than numeric values. To take advantage of this property, this work proposes a *grammar-driven processing* scheme, which arranges the data structure and computations based on the symbols appearing in the grammar rules. This enables the use of indexing to bypass certain graph traversals and avoid unnecessary grammar rule matching.

Third, to ensure termination, it is critical that the graph system can detect edge duplications and remove/avoid them. The existing graph system Graspan [135] proposes an efficient merge-sorting-based solution to detect edge duplications in each iteration of the graph processing. However, instead of detecting and removing the duplicated edges, this work finds that it could be more efficient to prevent inserting duplicated edges in the first

place. In specific, it shows that a hashing-based duplication edge checking can be more cost-efficient than the sorting-based edge duplication removal.

Finally, to make the system scalable to large graphs that may not fit into the memory (before or after edge insertions), it is important that the graph system provides an efficient out-of-core processing mode. To achieve this, Graspan loads partitions pair by pair to find possible new edges within and between partitions. In the worst case, however, this design requires to load partitions $P(P-1)/2$ times, where P is the number of partitions. To avoid intensive partition loading in unfavorable scenarios, this work presents a new out-of-core processing strategy, which only loads one partition per time. On top of that, it can process each partition asynchronously to accelerate the convergence.

To demonstrate their efficiency, we implemented multiple representative designs of the graph system based on the proposed techniques, then evaluated them using a group of program analysis graphs extracted from real-world system software. The results are aligned with the discussion of the two basic computation models, confirm the effectiveness of the proposed optimizations, showing significant speedups over the state-of-the-art graph system Graspan for in-memory graph processing and out-of-core graph processing.

In summary, this works makes a three-fold contribution to the system development for large-scale program analysis.

- First, it categorizes the computations in CFL-reachability analysis into two basic models and discusses their pros and cons in terms of redundancy, locality, and parallelism;
- Second, it introduces multiple critical optimizations to the graph system that substantially boost the performance in both in-memory and out-of-core processing scenarios;

- Finally, it evaluates multiple designs of the proposed graph system and demonstrates their performance benefits over the state-of-the-art graph system, making it possible to analyze large software within minutes or even seconds.

Next, we first present the background of this work.

4.2 Background

In this section, we first introduce the classic formalization that maps a large class of interprocedural program analysis problems to graph reachability problems, then we present a state-of-the-art graph processing system for solving such graph reachability problems.

4.2.1 Program Analysis and CFL-Reachability

In their seminal work [106], Reps and others introduced the ideas of transforming a class of interprocedural data-flow analysis problems, including but not limited to, the classical separable problems (like reaching definitions, live variables and available expressions) and many non-separable problems (like constant propagation and uninitialized variables), into graph reachability problems. Following this formalization, many program analyses have been proposed [12, 63, 103, 128, 140, 143, 150, 151] based on graph reachability, including variations of points-to analysis [156, 124]. In the following, we use pointer analysis as an example to illustrate its basic idea.

The goal of pointer analysis or points-to analysis is to statically reason about a set of heap objects (allocation sites) that a pointer may point to—the *points-to set*. Based on the points-to information, one can also derive aliasing relations among pointers—whether

two pointers may point to the same heap object or not. For simplicity, we present a *flow and field-insensitive* pointer analysis [135] which considers all the pointer relevant operations (as listed below) in a program but ignores the control flows and field accesses.

`x = y`	value assignment	`x = *y`	indirect load
`*x = y`	store to address	`x = &y`	address of

To formulate the pointer analysis as a graph reachability problem, a graph is constructed where nodes represent the variables and abstract locations and edges indicate the value flows among the variables. Each edge in the graph can be of one of the following three types:

- *Assignment edge (A)*: There is an A edge from `y` to `x` if and only if there is an assignment `x = y` in the program;
- *Dereference edge (D)*: There is a D edge from `x` to `*x` and a D edge from `&x` to `x` for each `*x` and `&x` in the program;
- *Alloc edge (M)*: For each allocation `x = malloc()`, there is an M edge from the allocation site to its reference `x`.

Figure 4.2-(b) illustrates an example (incomplete) graph for pointer analysis for the code segment given in Figure 4.2-(a), based on the context free grammar given in Figure 4.2-(c), where the blue arrows are generated based on the code, while the orange arrows are transitive edges added based on the grammar rules. The first rule in the grammar defines a value flow – how a value flows among variable expressions. Note that each value can flow from the variable carrying it to the variable itself – a self-loop in the graph (not

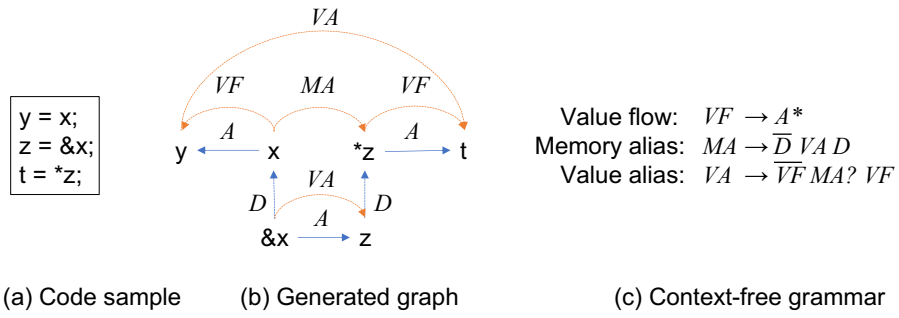


Figure 4.2: Pointer Analysis as CFL-Reachability Analysis [135].

shown here for conciseness). The second and third rules define the memory and value aliases, recursively based on each other and the referencing/dereferencing relations, as well as the value flows (a symbol with a bar on top means a direction-reversed edge with the symbol). For example, consider the nodes `&x` and `z` in the graph, they are clearly value aliases (according to the first and third rules), thus a *VA* edge is added between them. After that, there is a path from `x` to `*z` satisfying the second rule, so an *MA* edge is added between `x` and `*z`, indicating their memory aliasing relation. Finally, a path from `y` to `t` becomes available to match the third rule, thus a *VA* edge is added between the two nodes. A more comprehensive example involving memory allocations can be found in [135].

Next we present the basic idea of a state-of-the-art graph system solution to the above CFL-reachability problem.

4.2.2 Existing Graph System for CFL-Reachability Analysis

Motivated by its importance, Wang and others recently developed a dedicated graph system for solving CFL-reachability analysis in the context of large-scale program analysis, namely, Graspan [135]. Unlike traditional solutions that do not add new edges

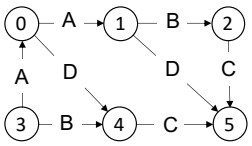
Value flow:	$VF \rightarrow A\ VF \mid \varepsilon$
Memory alias:	$MA \rightarrow \overline{D}\ T1$
Temp:	$T1 \rightarrow VA\ D$
Value alias:	$VA \rightarrow \overline{VF}\ T2$
Temp:	$T2 \rightarrow MA\ VF \mid VF$

Figure 4.3: CFG in Chomsky Normal Form.

to the graph physically (with the concern of memory blowup), Graspan explicitly add transitive edges to the graph. As new edges are inserted, the graph may grow quickly, representing an evolving dataset. Furthermore, the computations involved in this analysis are relatively very simple – just adding new edges based on the grammar rule matching. These two characteristics make it a great opportunity to develop a "big data" solution – a graph system that supports frequent edges insertions, efficient graph traversals and symbol matching, as well as handling scenarios where the graph cannot fit into the memory.

First, Graspan generates the initial graph from the given program using a modified compiler front end. To support the high-precision interprocedural analysis, the front end inlines functions based on a bottom-up traversal of the call graph. For recursive functions, it collapses the functions in each strongly connected component (SCC) into a single function and treats it context insensitively. With the generated graph, it is the users' responsibility to specify a context-free grammar that guides the insertion of transitive edges to the graph. Following an *edge-pari-centri model*, it requires the grammar to be normalized into the Chomsky normal form, where the right-hand side of each grammar rule contains at most two symbols. Figure 4.3 lists the grammar rules for the prior example after applying normalization. Note that two extra grammar symbols ($T1$ and $T2$) are created to facilitate the normalization. Graspan takes the initial graph and the normalized CFG as inputs.

Before the actual processing, Graspan first partitions the graph by dividing vertices into logical intervals, such that edges whose source vertices fall into the same interval belong to the same partition. Moreover, edges are sorted first based on the source vertex ID, then based on the target vertex ID. Figure 4.4 shows the partitioning of an example graph.



Partition 0

src	dst	lab
0	1	A
	4	D
1	2	B
	5	D

Partition 1

src	dst	lab
2	5	C
3	0	A
	4	B
4	1	C

Figure 4.4: Example Partitions in Graspan.

During the actual processing, Graspan loads two partitions each time and adds new transitive edges whose source vertices belong to the two partitions. This process is repeated until no new edges can be added to any pair of partitions – a fixed point. Assume the number of partitions is P, then it takes at least $P(P-1)/2$ rounds of partition pair loading to complete. To avoid unnecessary loading, Graspan maintains some meta-information – the *destination distribution map* (DDM) which capture the percentage of edges in one partition p with destinations in another partition q. With DDM, Graspan prioritizes the loading of partition pairs that are better "matched". However, in the worst case, it still needs $P(P-1)/2$ rounds of loading.

Algorithm 8 shows the detailed processing of a pair of partitions in Graspan. First, after the partitions are loaded, they are immediately merged into one graph (V, E). The in-memory representation of the initial graph is given in Figure 4.5. To avoid unnecessary

Algorithm 8 Graspan's Algorithm for CFL-Reachability

```
1: Input: Partitions P_1 and P_2
2: Combine P_1 and P_2 into one graph (V, E)
3: for edges of each vertex v ∈ E do
4:     O_v = ∅ /* old edges of vertex v */
5:     D_v = edges of v /* delta: (newly added) edges of vertex v */
6: for each vertex v whose D_v ≠ ∅ do
7:     mergeResult = ∅
8:     /* merge old edges of v with new edges of others */
9:     V_1 = (target vertices of O_v) ∩ V /* exclude those that are not in V */
10:    listsToMerge = O_v
11:    for v' ∈ V_1 do
12:        listsToMerge.add(D_v')
13:    mergeResult = MatchAndMergeSortedArrays(listsToMerge)
14:
15:    /* merge new edges of v with all edges of others */
16:    V_2 = (target vertices of D_v) ∩ V /* exclude those that are not in V */
17:    listsToMerge = {D_v, mergeResult}
18:    for v' ∈ V_2 do
19:        listsToMerge.add(O_v', D_v')
20:    mergeResult = MatchAndMergeSortedArrays(listsToMerge)
21:
22:    /* update O_v and D_v */
23:    listsToMerge = {O_v, D_v}
24:    O_v = MergeSortedArrays(listsToMerge)
25:    D_v = mergeResult − Q_v
```

checking of previously examined cases, Graspan separates the edges of a vertex v into the old edge list O_v and the new (delta) edge list D_v. When merging the edges of two adjacent vertices v_1 and v_2, it is safe to avoid those from the two old edge lists (i.e., O_{v_1} and O_{v_2}). To achieve this, the algorithm first initializes O_v with $\emptyset$ and D_v with the initial edges of v, respectively (see Line 4-5). Then, for each vertex with a non-empty D_v (i.e., new edges added in the last iteration), the algorithm considers two scenarios: (i) merging the old edges of v with new edges of its neighbors (Line 8-13) and (ii) merging the new edges of v with all edges of its neighbors (Line 15-20). In both scenarios, the key operation is to merge the sorted input lists into a new sorted list with new edges inserted (Lines 13 and 20). For high efficiency, Graspan uses a min-heap style mechanism which repetitively finds the

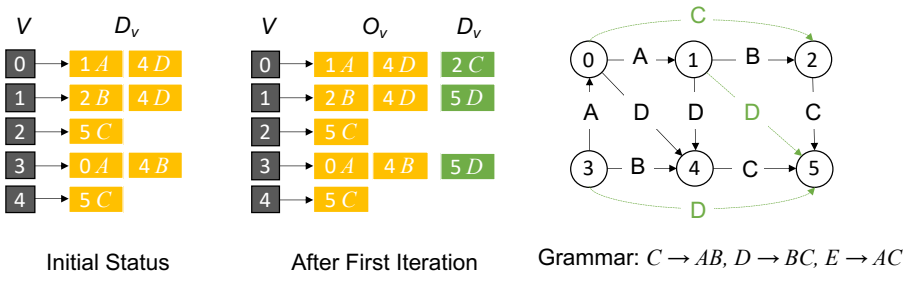


Figure 4.5: In-Memory Graph Representation in Graspan.

minimum in a slice of the lists and make a copy of it to an output list. Label matching is also performed before an edge is copied to the output.

Note that the sorting can automatically remove redundant edges, which is critical to the termination of the iterations – if redundant edges are allowed, the algorithm may continuously add duplicated edges without finishing. Moreover, with the partitioning-based design, Graspan naturally supports out-of-core processing. When a pair of loaded partitions generate too many new edges, it gets re-partitioned.

So far, we have presented the basic ideas of the existing graph system for CFL-reachability analysis. In the following, we will discuss various design aspects of the system and explain some insights in better designing such a graph system.

4.3 Overview: A Holistic Approach to Graph System Design

Despite that Graspan has demonstrated promising performance results over some other existing solutions to the CFL-reachability problem, we believe that the full potential of a graph system has not yet been realized. In this section, we provide a high-level discussion on the design space of the graph system for CFL-reachability analysis. The discussion will

be arranged along three major dimensions: (i) the modeling of underlying computations; (ii) the data structures; (iii) and the processing mode, as illustrated in Figure 4.6.

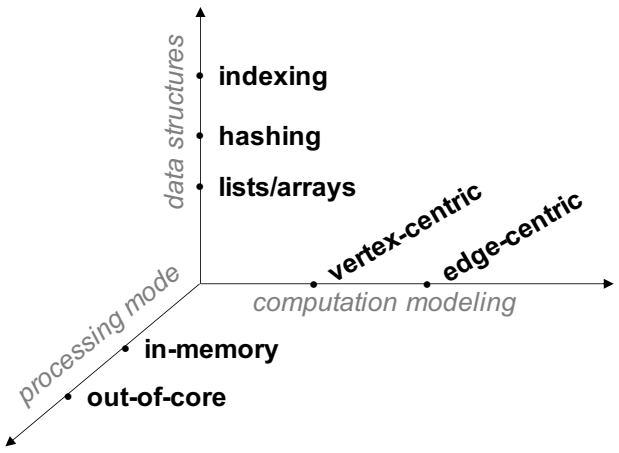


Figure 4.6: Design Dimensions of A Graph System for CFL-Reachability Analysis.

First, for conventional value-based graph systems, it is well-known that the graph computations can be modeled differently, such as *push* and *pull* models for in-memory graph processing and the *GAS* (gather-apply-scatter) model for distributed platforms. Though different algorithmic solutions have been discussed for CFL-reachability analysis, there are no systematic discussions on its alternative computation models from the perspective of a graph system. In this work, we present two basic computation models for CFL-reachability analysis, namely the *vertex-centric model* and the *edge-centric model*. More importantly, we will discuss their pros and cons in terms of the degree of parallelism, the amount of computation redundancy, and the data locality.

Second, from the data structures point of view, the existing solution (Graspan) mainly leverage lists and sorting for better efficiency. In this work, we will also consider the uses of hashing and indexing to further boost the performance. More specifically, we will

consider a hashing-based edge duplication checking mechanism, as opposed to the sorting-based strategy, and to further accelerate the grammar rule matching, we will consider to integrate symbol indexing into the graph representation. We find that both techniques can substantially improve the performance.

Finally, regarding the processing mode, a well-designed graph system should be able handle the graph efficiently during both in-memory and out-of-core processing. Note that supporting out-of-core processing is a highly demanded feature given that the graph's growing nature (new edges are inserted during the processing). Though the existing solution (Graspan) supports out-of-core processing, its design may suffer from degraded performance when new edges are scattered across partitions. As it loads partitions in pair, in the worst case, it may take $P(P-1)/2$ rounds of loading (for just one iteration). In this work, we will discuss an alternative design that only loads one partition to the memory each time.

In the following sections, we first present the basic computation models, discuss their pros and cons, then we will focus on the design of an optimized model.

4.4 Computation Modeling

4.4.1 Two Basic Models

Inspired by prior work on value-based graph systems [78, 109], we propose to model the computations in CFL-reachability analysis from the "perspective of a vertex or an edge". The resulted models are referred to as the *vertex-centric model* and the *edge-centric model*, respectively. Next, we present each of them with more details.

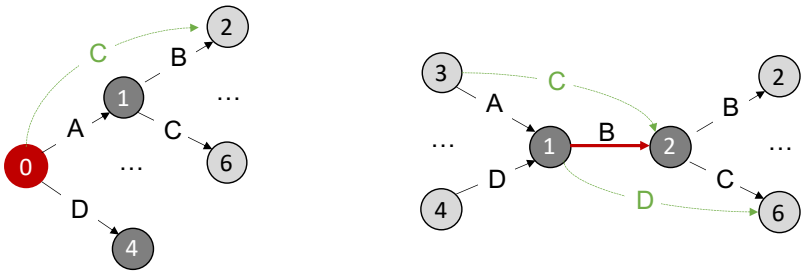


Figure 4.7: Vertex-Centric Model v.s. Edge-Centric Model.

- **Vertex-centric model**. As illustrated by Figure 4.7-(a), for each vertex $v \in V$, the model tries to find new transitive edges by concatenating its outgoing edges with its out-neighbors' outgoing edges. This requires the model to traverse v's out-neighbors and their out-neighbors. Here, a out-neighbor adjacency list is sufficient.

- **Edge-centric model**. As illustrated by Figure 4.7-(b), for each edge $e \in E$, the model tries to find new transitive edges by concatenating edges adjacent to e, including the incoming edges of e's source and the outgoing edges of e's destination. Thus, this model needs both the out-neighbor and in-neighbor adjacency lists.

Note that, based on its design explained in the prior section, Graspan clearly falls into the vertex-centric model (see Line 6 in Algorithm 8). In the following, we compare the two models in aspects that critically affect the overall performance of the processing.

4.4.2 Comparison: Pros and Cons

Before the discussion, we first make one basic assumption: to ease the analysis and avoid data contention in parallel processing, we assume the edges newly added in the

current iteration are not subsequently considered for generating other new edges within the same iteration, referred to as the *synchronized processing*. To support this, the new edges found in the current iteration need to be saved in a separated list. Later, we will include the discussion of *asynchronous processing* in the context of parallel processing.

In the following, we first discuss the basic implementations of both models in different aspects, along with some optimization ideas.

Time Complexity. First, assume the processing is *synchronized*, then both models would have the same number of iterations to complete, denoted as I. For the vertex-centric model, assume the average number of out-neighbors is K (i.e., $|E|/|V|$), then the time complexity of vertex-centric model is $O(|V| \times K \times K \times I) = O(|E|^2/|V| \times I)$[1]. For the edge-centric model, the time complexity is $O(|E| \times (K + K) \times I) = O(2|E|^2/|V| \times I)$. Clearly, the edge-centric model doubles the total amount of work comparing to the vertex-centric model, because each pair of adjacent edges in the edge-centric model are considered twice in each iteration. Consider the example in Figure 4.7-(b), the edge pair $3 \rightarrow 1 \rightarrow 2$ is considered when edge $1 \rightarrow 2$ is processed and when edge $3 \rightarrow 1$ is processed.

Memory Cost. As explained earlier in the prior section, the vertex-centric model needs the out-neighbor adjacency list, while the edge-centric model needs both the out-neighbor and in-neighbor adjacency lists. As a result, the graph memory consumption of edge-centric model is doubled comparing to that of the vertex-centric model.

Parallelism. In terms of the *parallelism* exposed by the two computation models, both can be easily parallelized with abundant amount of parallelism, as different vertices (or edges) can be processed independently from each other under the *synchronized processing* strategy.

[1]Assume the grammar rule match cost for each pair of edges is a constant.

Note that if the processing is *asynchronous* – new edges found in the current iteration are used for finding other new edges, then there are risks of data races as the new edge list may be updated and read at the same time by different threads.

Data Locality. Assume adjacency list(s) are used for both models. For the vertex-centric model, consider vertex v, like vertex 0 in Figure 4.7-(a), it is needed to access the outgoing edge lists of out-neighbors of v, which can be located in different memory regions, thus incurring irregular accesses. However, the traversal of each edge list of a out-neighbor (e.g., edge list of vertex 1) is sequential – good spatial locality. In comparison, the edge-centric model only needs to access the in/out-neighbors of the source and the destination of an edge, both of which are sequential. In this sense, the data locality of edge-centric model appears to be better that that of the vertex-centric model. However, as the edge-centric model needs to access two adjacency lists, which puts higher pressure on the cache.

Explicit Redundancy. With a basic implementation, both models may produce a large ratio of redundant computations. Recall that Graspan separates the newly added edges from the old edges to avoid unnecessary concatenations of some of the edge pairs, referred to as *new-old edge separation*. As Graspan follows the vertex-centric model, thus the latter also suffers from the same redundancy. For edge-centric model, the situation of redundancy is slightly different. Consider the processing of edge $1 \rightarrow 2$ in Figure 4.7-(b) in iteration i, a basic implementation would concatenate the edge with every outgoing edge of the destination (vertex 2) and every incoming edge of the source (vertex 1), regardless when they are generated. It is possible that some of the edges have been concatenated with this edge before. The *new-old edge separation* can be adopted here as well to avoid such

unnecessary concatenations. So far, we have discussed the "explicit redundancy" regarding unnecessary edge concatenations and grammar rule matching, which can be addressed with the *old-new edges separation* optimization (from Graspan).

Implicit Redundancy. In fact, even with the above old-new edges separation, the models may still exhibit some "implicit redundancy" regarding unnecessary graph traversals. For the vertex-centric model, every vertex $v \in V$ may still need to be processed and its out-neighbors need to be traversed. However, not every vertex may need to be processed in each iteration. If a vertex along with its out-neighbors do not have any new edges added in the last iteration, then its processing can be safely skipped in the current iteration. However, to find this out, we need to track, for each vertex, not only the "new edge status", but also the "new edge status of neighbors", which could be expensive ($O(|V| + |E|)$). Similarly, for the edge-centric model, not every edge may need to be processed in each iteration. If an edge was not added in the prior iteration, and neither its source nor its destination has new edges added in the prior iteration, then its processing can be safely skipped in the current iteration. More precisely, we can prove that *only edges added in the prior iteration need to be processed in the current iteration* (thanks to its consideration of both incoming and outgoing edges). With this optimization, the edge-centric model turns out to be the same as a classic *worklist-based algorithm* for solving CFL-reachability [80].

So far, we have discussed the pros and cons of the basic implementations of the two models, meanwhile, we also introduced optimizations for addressing some of the "cons". In summary, both models can adopt the *old-new edges separation* optimization. For the

edge-centric model, it can further adopt the *worklist* optimization. Next, we assume these optimizations are enabled, then revisit some aspect discussed earlier.

Parallelism - Revisited. First, under the *synchronous processing* mode, the *old-new edges separation* optimization does not affect the degree of the parallelism of each model, as it simply separates the edges added in prior iterations into two sub-lists, while the new edges found in the current iteration are store separately and write-only. As to the *worklist* optimization for the edge-centric model, the situation is more complex. If the worklist is implemented as a global data structure shared among threads, then updates to the worklist need to be serialized (i.e., atomic operations are needed). As there could be many new edges found in one iteration, the cost of this worklist maintenance could become the bottleneck. Later, we will introduce some alternative designs to better support the parallelism.

In summary, both basic models have their pros and cons. Overall, we consider the edge-centric model with the worklist optimization is more promising, especially consider the extra optimizations that will be discussed in the following. For this reason, we next focus on this model, especially its parallel implementation.

4.5 Parallel Edge-Centric Model

In this section, we present a detailed design of the parallel edge-centric model. Note that the aforementioned worklist optimization to the edge-centric model is like a "double-edged sword". On one hand, it reduces the redundancies; on the other hand, it makes it more challenging to achieve parallel efficiency due to the use of a global edge list shared by different threads. Based on our experiments, processing the worklist using multiple threads

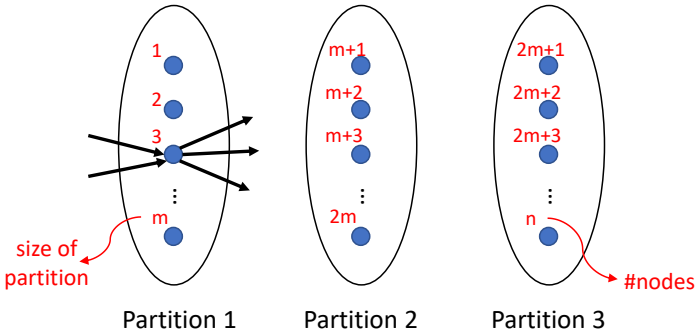


Figure 4.8: Partitioning of n vertices with m as the size of each part.

with locks is not efficient; actually, the performance is no better than the sequential version. In the following, we design a parallel lock-free edge-centric algorithm.

4.5.1 Partitioning

First, we partition the graph into subgraphs in a way that each thread can process one subgraph at a time. There are two challenges. First, the graph grows during the processing. Second, to process an edge, all the incoming edges to the source and outgoing edges from the destination are required. To make the partitioning feasible, we do not restrict our algorithm to process an edge using one thread and in one partition – new incoming and outgoing edges can be generated using two different threads in different partitions and there are no constraints.

In specific, the partitioning is based on the vertices. Vertices will be partitioned into disjoint sets of vertices. Each partition contains all the incoming and outgoing edges to the vertices of that partition (Figure 4.8).

Each edge may be in one partition or in two partitions. If both the source and the destination of an edge are in one partition, the edge belongs to that partition only; otherwise

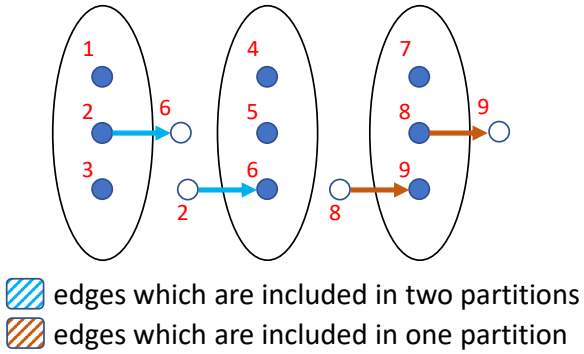


Figure 4.9: Handling of Edges during Partitioning.

it belongs to two partitions. If an edge belongs to two partitions, the computations on it are split and will be done by two threads. New incoming and outgoing edges will be generated by two different threads, respectively, and the new edges corresponding to rules with one symbol on the right-hand side will be generated by both threads. If an edge only belongs to one partition, the new edges will be generated using one thread. Figure 4.9 illustrates both cases mentioned above.

4.5.2 Processing

The computations have two phases. Phase 1 is to generate new edges and add the new edges to the current partition if they belong to it. Phase 2 is to add the newly generated edges that belong other partitions to the corresponding partitions. Even for an edge with both source and destination belonging to one partition, it is possible to generate new edges that belong to other partitions. There is a synchronization after each phase. Phase 1 and 2 repeat until there are no new edges generated.

If we assign one simple worklist to each partition, the process of generating new edges for other partitions and gathering edges from other partitions creates conflicts among

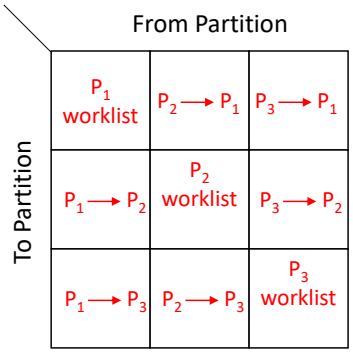


Figure 4.10: 3D-Worklist

threads. To address this, we introduce 3D-worklist, a lock-free data structure to prevent conflicts and speedup the processing.

As illustrated in Figure 4.10, in a 3D-worklist, we assign one worklist for each partition in each partition. When a partition is being processed using one thread and new edges are generated, each edge will be added to the worklist it belongs to. Assume there are 4 partitions and partition 1 is being processed. There would be 4 worklists in partition 1 for each partition. When a new edge is generated, based on its source and destination, it would be added to one partition if both vertices belong to one partition or it would be added to two partitions if the vertices belong to different partitions. As a result, the new edges will be pushed into the worklists of their partitions.

To prevent adding redundant edges, it still uses a hashset. Each vertex has one hashset for incoming edges and one hashset for outgoing edges. But to prevent conflicts, partitions do not have access to the hashsets of each other. In this way, if a partition generates a new edge which that belongs to other partitions (none of the source and destination are in the partition), it is not possible to check if it is already in the graph, but it will be checked later by its own partitions. If the source or the destination of a new edge belongs

to the current partition, the redundancy check can happen immediately. Thus, there are two types of new edges: (i) the ones that we know are not redundant and have to be added to the graph and (ii) the ones that may be redundant and have to be ignored. Therefore, we use two 3D-worklists to keep track of the new edges.

Another important consideration in the parallel edge-centric model is the choice of *synchronous versus asynchronous processing*. As mentioned earlier, synchronous processing does not allow the new edges found in the current iteration to be used for finding new edges. This simplified our discussion in the earlier section, but may not be necessary in practice. However, in the context of parallel processing, asynchronous processing may require to lock the new edge list as different threads may read/write it. In this case, we consider a hybrid design: the local processing within one partition (using one thread) is asynchronous while globally, different partitions are processed synchronously, which aligns well with the above two-phase design.

The following algorithm summarizes the major ideas of parallel lock-free edge-centric processing.

- Preprocessing
    - Break the node IDs into P ranges for each partition
    - Make each partition by putting together all the incoming and outgoing edges of each partition ID range
    - Define two 3D-worklists: $nextEdges$ and $possibleNextEdges$
    - For each partition p: Push all the edges of p into $nextEdges[p][p]$
- Phase 1 - Do in parallel for each partition p:

- For each edge $e = (i, j, A)$ in $nextEdges[p][p]$
  - $newEdges = \{\}$
  - $PartI =$ the partition of i
  - $PartJ =$ the partition of j
  - Add new edges to $newEdge$ based on rules of type $B ::= A$
  - If $partI == p$: Add new edges to $newEdges$ based on rules of type $C ::= BA$
  - If $partJ == p$: Add new edges to $newEdges$ based on rules of type $C ::= AB$
  - For each edge $e_n = (u, v, X)$ in $newEdges$
    - $PartU =$ the partition of u
    - $PartV =$ the partition of V
    - If ($PartU \neq p$ and $PartV \neq p$):
      - If $PartU == PartV$:
        - Add e_n to $possibleNextEdges[PartU][p]$
      - Else:
        - Add e_n to $possibleNextEdges[PartU][p]$
        - Add e_n to $possibleNextEdges[PartV][p]$
    - Else If $PartU == PartV$:
      - Add e_n to $nextEdges[PartU][p]$ if it is not redundant
    - Else:
      - Add e_n to $nextEdges[PartU][p]$ if it is not redundant
      - Add e_n to $nextEdges[PartV][p]$ if it is not redundant

- Synchronization
- Phase 2 - Do in parallel for each partition p:
  - For each partition i in $[1 : P]$ except p:
    - Add $nextEdges[p][i]$ to $nextEdges[p][p]$
    - Add $possibleNextEdges[p][i]$ to $nextEdges[p][p]$ if they are not redundant
- If Union of $nextEdges[i][i]$ for i in $[1 : P]$ is not empty, go to Phase 1

So far, we have introduced the parallel design of the edge-centric model. Next, we will present a new optimization specific to symbol-matching-based graph problems.

4.5.3 Grammar-Driven Processing Scheme

The number of possible edge labels in CFL graph reachability problem is limited and usually a very low number. This gives us an opportunity for an indexing optimization.

As it is mentioned before, the graph representation used in this system is adjacency list. To apply the indexing optimization, it needs to be modified. For each vertex, instead of keeping all the neighbors in a single list, there is a separate list for each label. This is referred to as *labeled adjacency list*. This indexing optimization can be used in both vertex-centric and edge-centric models. Instead of looking into the grammar rules to check if an edge or a pair of edges can generate a new edge, for each rule, we look for edges or pair of edges which can generate new edges. In this way, the possible new edges can be found in constant time using the indexing optimization. We refer to the above processing scheme as the *grammar-driven processing scheme*.

With this new scheme, the time complexity for searching in the neighbor edges of a vertex to find edges with a specific label decreases from the order of the vertex degree to constant time. Later, our evaluation will demonstrate the effectiveness of this optimization.

4.5.4 Supporting Out-of-Core

In graph reachability problem, the graph grows fast. New edges are added to the graph and the number of new edges are not predictable. It can easily grow up and become larger than the main memory, which raises the need for out-of-core processing.

In fact, the parallel data driven algorithm we proposed in the earlier section fits well into the out-of-core scenario thanks to its partition-based two-phase design. In Phase 1, processing one partition is independent of processing other partitions and in Phase 2, each partition can gather its new edges independent from other partitions.

First, the graph is partitioned and all the partitions are stored in the external memory. Then, in each iteration, partitions are loaded and processed one at a time. The 3-D worklist is kept in the main memory, but if the number of new edges for a partition in the worklist exceeds a threshold, to avoid running out of memory, the queue corresponding to that partition will be stored in the external memory and will be loaded back into the main memory when that partition is being processed.

Note With the above design,

4.6 Evaluation

4.6.1 Implementation

We implemented both the vertex-centric and edge-centric models, along with the indexing optimizations, for both in-memory and out-of-core processing in C++ language. The parallelization is achieved with the Cilk library.

More specifically, we used the `unordered_set<ull>` for implementing the hashmap for edge duplication checking. For vertex-centric model, the graph is stored as a 3-level vector: `vector<vector<vector<OutEdge>>>`, where the first level is for vertices, the second level is for separating the new edges (generated from the last iteration) from the old ones, and the third level is for the outgoing edges (i.e., target vertices). For the edge-centric model, the graph is stored in two 2D vectors: `vector<uint> **outEdgeVecs` and `vector<uint> **inEdgeVecs`, where the first dimension is for the grammar labels (i.e., label indexing), and the second dimension is for the outgoing/incoming edges of each vertex, respectively. To represent the grammar, we first separate the grammar rules into three sets based on the number of symbols on the right-hand side (RHS), then define each set of rules as a vector of vectors: `vector<vector<uint>>`, where the first level is the rule ID and the second level is the actual rule in the format of `(LHS, RHS-1, RHS-2)`. For each rule set, we build indices for faster access. The index is established as a direct mapping from RHS to the LHS. For example, given the RHS BC, we can directly access all the possible LHS of rules like $X ::= BC$. The index is also stored in a vector of vectors: `vector<vector<uint>>`, where the first level is an encoded value of RHS (e.g., BC is encoded as $B \times N_{sym} + C$), and the second level is for the corresponding list of possible LHS symbols. Note that, beside

indexing the RHS, we can also index other combinations of symbols in the rule, such as AB or AC in a rule like $A ::= BC$, which can be used based on the configurations.

4.6.2 Methodology

In this evaluation, we compare four representative configurations of our solution with the state-of-the-art CFL reachability analysis system—Graspan[135]. These include:

- *Vertex-Centric Model (*VCM*).* This configuration implements the vertex-centric model and uses the hashmap for edge duplication checking;
- *Edge-Centric Model (*ECM*).* This configuration implements the edge-centric model and uses the hashmap for edge duplication checking;
- *Vertex-Centric Model with Grammar Label Indexing (*VCM-IDX*).* This configuration implements the vertex-centric model along with the indexing optimization based on grammar labels, and uses the hashmap for edge duplication checking.
- *Edge-Centric Model with Grammar Label Indexing (*ECM-IDX*).* This configuration implements the edge-centric model along with the indexing optimization based on grammar labels, and uses the hashmap for edge duplication checking.

For in-memory processing, all the above four configurations are evaluated; for out-of-core processing, we mainly focus on the ECM-IDX configuration, for its best overall performance.

The input grammars and graphs are mainly collected from Graspan [135]. One additional grammar and graph pair is added from another recent work [162]. Table 4.1

Table 4.1: Graphs in Evaluation

		V	E_{before}	E_{after}
Data-flow analysis (DFA)	Linux	42.4M	44.0M	99.2M
	PSQL	29.8M	34.7M	56.0M
	httpd	5.7M	10.0M	19.2M
Field-insensitive pointer-analysis (PA)	Linux	11.2M	18.9M	186.5M
	PSQL	5.2M	9.3M	862.1M
	httpd	1.7M	3.0M	904.3M
Field-sensitive pointer analysis (FPA)	hdfs	5.3M	10.1M	1.8B
	Hadoop/MapReduce	21.8M	41.8M	99.1M

lists the graphs used for three program analysis grammars. The graphs are pre-generated from real-world large system software, including Linux 4.4.0-rc5, PostgreSQL 8.3.9, Apache httpd 2.2.18, and Hadoop-MR 2.7.5.

4.6.3 In-Memory Processing Performance

Table 4.2 reports the execution time of different methods on the benchmark graphs. First, we found all the four versions of our proposed solution run faster than Graspan across all the benchmarks evaluated. Among the four versions, VCM is the version closest to the design of Graspan, as Graspan is also vertex-centric. The major difference between the two is that Graspan uses sorting-based mechanism for edge duplication checking, while VCM uses hashmap for the same purpose. The results confirm that optimizing edge duplication checking itself can bring significant speedups, ranging from 4× to 25×.

Comparing the performance between VCM and ECM, we can find that each has its own winning cases. For example, VCM performs worse on the three DFA benchmarks, but better on the four PA/FPA benchmarks. As discussed earlier, there are some trade-offs between the two computation models. VCM needs more accesses to the graph structure,

Table 4.2: Execution Time Comparison

		Graspan	VCM	ECM	VCM-IDX	ECM-IDX
DFA	Linux	4h, 28m	6m, 30s	53s	9m, 25s	46.53s
	PSQL	3h, 12m	4m, 28s	22s	6m, 11s	20.1s
	httpd	9m, 34s	17s	8s	20s	6.79s
PA	Linux	13m, 41s	1m, 51s	2m, 27s	4m, 12s	1m, 29s
	PSQL	2h, 54m	29m, 38s	5h, 39m, 37s	5m, 20s	4m, 55s
	httpd	3h, 38m	50m, 51s	9h, 31m, 20s	4m, 6s	4m, 48s
FPA	Hadoop	29m, 8s	4m, 10s	1h, 6m, 33s	1m, 37s	39.75s

Table 4.3: Execution Time of Parallel Edge-Centric Model

		1t	2t	4t	8t	16t
DFA	Linux	1m, 26s	1m	48.5s	21.05s	20.4s
	PSQL	27.7s	26.1s	19.9s	10.5s	10.8s
	httpd	10.7s	7.8s	6.6s	3.7s	3.1s
PA	Linux	2m, 40s	2m, 21s	1m, 34s	48.4s	37.5s
	PSQL	13m, 40s	4m, 4s	7m, 6s	4m, 6s	3m, 42s
	httpd	15m, 38s	10m, 35s	7m, 30s	4m, 10s	3m, 24s

meanwhile, it exposes more synchronization-free parallelism. In addition, VCM only needs to maintain one copy of the graph (based on outgoing edges), while ECM needs to maintain two copies of the graph (based on both incoming and outgoing edges). As the grammars for PA/FPA are much more complex than that of DFA, the benefits brought by the extra parallelism of VCM exceed the overhead of extra graph accesses. Therefore, VCM performs better in these cases.

Finally, comparing the two models with and without the grammar-based indexing optimization, we found the mixing results. For DFA benchmarks, the indexing brings slowdown to vertex-centric model (i.e., VCM-IDX), and marginal benefits to edge-centric model (i.e., ECM-IDX). By contrast, for PA/FPA benchmarks, the benefits of indexing optimization are significant, ranging from 1.8× to 119×.

Table 4.4: Performance Comparison

		Graspan	ECM-IDX
PA	PSQL	6h, 1m	53m, 26s
	httpd	6h, 7m	49m, 58s
FPA	hdfs	> 24h	1h, 42m

Table 4.3 reports the execution time of the parallel edge-centric model using different number of threads, which can scale well up to around 8 threads (the machine has 16 physical cores). The sub-linear speedups are mainly due to the memory-intensive nature of the computations – a large amount of new edges are generated which produces frequent writes to the memory. Also note that the single-thread version of parallel edge-centric model is slower than the sequential edge-centric model, which indicates the overhead of parallelization (such as the synchronization among threads).

4.6.4 Out-of-Core Processing Performance

Table 4.4 reports the execution for out-of-core processing on a machine with much less memory capacity (8GB). As ECM-IDX performs the best for in-memory processing, we extended it for supporting out-of-core scenarios. The results show that ECM-IDX runs over 6× faster than Graspan. The gap between the two methods are mainly due to two reasons. First, Graspan requires to load every pair of partitions to the memory until no new edges are generated. In comparison, our solution only needs to every partition once in each round. Second, after the partitions are brought into the memory, our solution runs more efficiently than Graspan, as demonstrated in the in-memory processing evaluation.

4.7 Related Work

Context Free Language (CFL)-reachability problem was introduced by Yannakakis [146] and then has been used for program analysis in several works [105, 48, 49, 107, 104]. A cubic solution for CFL reachability is explained here [80] and for a long time it was accepted that there is an "O(n^3) bottleneck", but a subcubic algorithm has been found later [22].

CFL-reachability has been used extensively for static analyses in sequential settings [118, 123, 125, 144, 157, 141, 72, 117] . Su et al. [126] proposed the first parallel implementation of CFL-reachability-based pointer analysis on multi-core CPUs by using a data sharing scheme for concurrent query-processing and a query scheduling scheme to eliminate redundancies. Graspan [135] uses edge-pair centric computation model to solve the CFL reachability problem and supports both in-memory and out-of-core computation and implements pointer and dataflow analyses. BigSpa [162] proposes a distributed interprocedural static analysis engine running on the cloud.

4.8 Summary

Program analysis is fundamental to many important domains of applications. Modeling program analysis as CFL-reachability problems and solving them using a graph system can make program analysis much more scalable. However, the existing design of such a graph system often performs sub-optimal due to its ad-hoc design. In this work, we explore the design space of such a program analysis graph system, and examines alternative options in several major design dimensions, including computation models, traversal orders, indexing strategies, and the mechanisms for edge duplication checking. By analytically and

experimentally comparing different configurations, we reveal the several findings in the system design. Finally, our evaluation shows that our proposed graph system design based on such systematic exploration yields significant better efficiency than the state-of-the-art graph system for program analysis.

Chapter 5

Conclusions

In conclusion, this dissertation explores multiple critical design aspects to improve the efficiency of graph processing systems on modern computer architectures, including both multi-core CPUs and GPUs.

For GPU-based platforms, it first proposes a graph transformation technique to address the negative impacts of graph irregularity on the performance of GPU-based graph processing. The results of this study lead to the development of an in-GPU-memory graph processing system – *Tigr*. Comparing to prior GPU-based graph processing systems, *Tigr* demonstrates significant speedups on multiple graph benchmarks and real-world graphs. Then, this dissertation further studies the expensive data movements between CPU and GPU during the out-of-GPU-memory graph processing, and introduces a fast subgraph extraction technique to minimize the data movements. As a result, an out-of-GPU-memory graph system – *Subway*, is developed. A comprehensive evaluation shows that *Subway* can

substantially reduce the amount of the data movements, thanks to its ability to only load the active fraction of the graph into the GPU memory in each processing iteration.

For CPU-based platforms, this dissertation focuses on an emerging graph analytics – CFL-reachability analysis, which is fundamental to large-scale inter-procedural program analysis, but has not yet been systematically studied from the perspective of graph system design. In one of its major chapters, this dissertation examines multiple key design factors of a dedicated graph processing system for CFL-reachability analysis, including the degree of parallelism, data locality, computation redundancy, as well as the efficiency of the out-of-core processing scenario. The examination, along with a systematic evaluation, reveals several new opportunities in optimizing the performance of such a graph system, demonstrating substantial performance improvements in both in-memory and out-of-core processing scenarios.

Bibliography

[1] CUDA C best practices guide: Chapter 9.1.3. zero copy. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide. Accessed: 2019-06-30.

[2] CUDA C programming guide. https://docs.nvidia.com/cuda/archive/8.0/cuda-c-programming-guide/index.html. Accessed: 2019-06-30.

[3] The koblenz network collection. http://konect.uni-koblenz.de/. Accessed: 2019-06-30.

[4] Laboratory for web algorithmics. http://law.di.unimi.it/. Accessed: 2019-06-30.

[5] NVIDIA CUDA toolkit v8.0. http://developer.download.nvidia.com/compute/cuda/8_0/rel/docs/CUDA_Toolkit_Release_Notes.pdf. Accessed: 2019-06-30.

[6] Thrust - parallel algorithms library. https://thrust.github.io/. Accessed: 2019-06-30.

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, August 2006.

[8] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk I/O. In *2017 USENIX Annual Technical Conference (USENIX ATC)*, pages 125–137, 2017.

[9] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

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

[11] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. Using static analysis to find bugs. *IEEE software*, 25(5):22–29, 2008.

[12] Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 553–566, 2015.

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

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

[15] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. Groute: An asynchronous multi-GPU programming model for irregular computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 235–248, 2017.

[16] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 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*, pages 93–104. ACM, 2017.

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

[18] Anthony Bonato. A survey of models of the web graph. In *Workshop on Combinatorial and Algorithmic Aspects of Networking*, pages 159–172. Springer, 2004.

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

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

[21] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[22] Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 159–169, New York, NY, USA, 2008. Association for Computing Machinery.

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

[24] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 1:1–1:15, New York, NY, USA, 2015. ACM.

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

[26] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 752–768, 2018.

[27] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Vishwesh Jatala, Keshav Pingali, V Krishna Nandivada, Hoang-Vu Dang, and Marc Snir. Gluon-async: A bulk-asynchronous system for distributed and heterogeneous graph analytics. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 15–28. IEEE, 2019.

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

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

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

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

[32] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. Xbfs: exploring runtime optimizations for breadth-first search on GPUs. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 121–131, 2019.

[33] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 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*, pages 345–354. ACM, 2012.

[34] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 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*, pages 345–354, 2012.

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

[36] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[37] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 599–613, 2014.

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

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

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

[41] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. Graphie: Large-scale asynchronous graph traversals on just a GPU. In *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*, pages 233–245. IEEE, 2017.

[42] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. Graphie: Large-scale asynchronous graph traversals on just a GPU. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 233–245. IEEE, 2017.

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

[44] Pawan Harish and PJ Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *International conference on high-performance computing*, pages 197–208. Springer, 2007.

[45] Stijn Heldens, Ana Lucia Varbanescu, and Alexandru Iosup. Hygraph: Fast graph processing on hybrid CPU-GPU platforms by dynamic load-balancing. 2016.

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

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

[48] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[49] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. *ACM SIGSOFT Software Engineering Notes*, 20(4):104–115, 1995.

[50] Zan Huang, Wingyan Chung, Thian-Huat Ong, and Hsinchun Chen. A graph-based recommender system for digital library. In *Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, pages 65–73. ACM, 2002.

[51] Jayadharini Jaiganesh and Martin Burtscher. ECL-CC v1.0. `http://cs.txstate.edu/~burtscher/research/ECL-CC/`, 2018.

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

[53] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. A distributed multi-GPU system for fast graph processing. *Proceedings of the VLDB Endowment*, 11(3):297–310, 2017.

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

[55] Laxmikant V Kale and Sanjeev Krishnan. CHARM++: a portable concurrent object oriented system based on c++. In *OOPSLA*, volume 93, pages 91–108. Citeseer, 1993.

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

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

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

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

[60] Farzad Khorasani, Bryan Rowe, Rajiv Gupta, and Laxmi N Bhuyan. Eliminating intra-warp load imbalance in irregular nested patterns via collaborative task engagement. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 524–533. IEEE, 2016.

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

[62] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *Proceedings of the 2016 International Conference on Management of Data*, pages 447–461. ACM, 2016.

[63] John Kodumal and Alex Aiken. The set constraint/cfl reachability connection in practice. *ACM Sigplan Notices*, 39(6):207–218, 2004.

[64] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, 2012.

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

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

[67] Jure Leskovec and Andrej Krevl. SNAP datasets:stanford large network dataset collection. 2015.

[68] Chen Li, Rachata Ausavarungnirun, Christopher J Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 49–63. ACM, 2019.

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

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

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

[72] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An incremental points-to analysis with cfl-reachability. In *Proceedings of the 22nd International Conference on Compiler Construction*, CC'13, page 61–81, Berlin, Heidelberg, 2013. Springer-Verlag.

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

[74] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th design automation conference*, pages 52–55. ACM, 2010.

[75] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 195–207, 2017.

[76] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[77] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):1–39, 2015.

[78] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):1–39, 2015.

[79] Adam McLaughlin and David A Bader. Scalable and high performance betweenness centrality on the GPU. In *Proceedings of the International Conference for High performance computing, networking, storage and analysis*, pages 572–583. IEEE Press, 2014.

[80] David Melski and Thomas Reps. Interconvertbility of set constraints and context-free language reachability. *SIGPLAN Not.*, 32(12):74–89, December 1997.

[81] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU implementation of inclusion-based points-to analysis. *ACM SIGPLAN Notices*, 47(8):107–116, 2012.

[82] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

[83] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 117–128, 2012.

[84] Ulrich Meyer and Peter Sanders. δ-stepping: A parallel single source shortest path algorithm. In *European Symposium on Algorithms*, pages 393–404. Elsevier, 1998.

[85] Alan Mislove, Hema Swetha Koppula, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Growth of the flickr social network. In *Proceedings of the first workshop on Online social networks*, pages 25–30. ACM, 2008.

[86] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 29–42. ACM, 2007.

[87] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 29–42. ACM, 2007.

[88] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Atomic-free irregular computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 96–107. ACM, 2013.

[89] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph algorithms on GPUs. In *ACM SIGPLAN Notices*, volume 48, pages 147–156. ACM, 2013.

[90] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[91] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471. ACM, 2013.

[92] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for GPU-friendly graph processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 622–636, 2018.

[93] Hector Ortega-Arranz, Yuri Torres, Diego R Llanos, and Arturo Gonzalez-Escribano. A new GPU-based approach to the shortest path problem. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 505–511. IEEE, 2013.

[94] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[95] Sreepathi Pai and Keshav Pingali. 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*, pages 1–19. ACM, 2016.

[96] Sreepathi Pai and Keshav Pingali. 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*, pages 1–19, 2016.

[97] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D Owens. Multi-GPU graph analytics. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 479–490. IEEE, 2017.

[98] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, pages 493–498. Springer, 1996.

[99] 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. The Tao of parallelism in algorithms. In *ACM Sigplan Notices*, volume 46, pages 12–25. ACM, 2011.

[100] Alain Pirotte, Jean-Michel Renders, Marco Saerens, et al. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *IEEE Transactions on Knowledge & Data Engineering*, (3):355–369, 2007.

[101] Dimitrios Prountzos and Keshav Pingali. Betweenness centrality: algorithms and implementations. In *Acm Sigplan Notices*, volume 48, pages 35–46. ACM, 2013.

[102] Junqiao Qiu, Zhijia Zhao, and Bin Ren. Microspec: Speculation-centric fine-grained parallelization for fsm computations. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, pages 221–233. IEEE, 2016.

[103] Jakob Rehof and Manuel Fähndrich. Type-base flow analysis: from polymorphic subtyping to cfl-reachability. *ACM SIGPLAN Notices*, 36(3):54–66, 2001.

[104] Thomas Reps. Shape analysis as a generalized path problem. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '95, page 1–11, New York, NY, USA, 1995. Association for Computing Machinery.

[105] Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11-12):701–726, 1998.

[106] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[107] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. SIGSOFT '94, page 11–20, New York, NY, USA, 1994. Association for Computing Machinery.

[108] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[109] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[110] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. Subway: Minimizing data transfer during out-of-GPU-memory graph processing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[111] Gorka Sadowski and Philip Rathle. Fraud detection: Discovering connections with graph databases. *White Paper-Neo Technology-Graphs are Everywhere*, 2014.

[112] Gorka Sadowski and Philip Rathle. Fraud detection: Discovering connections with graph databases. *White Paper-Neo Technology-Graphs are Everywhere*, 2014.

[113] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V Çatalyürek. Betweenness centrality on GPUs and heterogeneous architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 76–85. ACM, 2013.

[114] John Sartori and Rakesh Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. *IEEE Transactions on Multimedia*, 15(2):279–290, 2013.

[115] John Scott. Social network analysis. *Sociology*, 22(1):109–127, 1988.

[116] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. Graphreduce: processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 28. ACM, 2015.

[117] Lei Shang, Yi Lu, and Jingling Xue. Fast and precise points-to analysis with incremental cfl-reachability summarisation: Preliminary experience. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, page 270–273, New York, NY, USA, 2012. Association for Computing Machinery.

[118] Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, page 264–274, New York, NY, USA, 2012. Association for Computing Machinery.

[119] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.

[120] Jeremy G Siek, Lie-Quan Lee, and Andrew Lumsdaine. The boost graph library: User guide and reference manual, portable documents, 2001.

[121] Jyothish Soman, Kothapalli Kishore, and PJ Narayanan. A fast GPU algorithm for graph connectivity. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[122] Shuang Song, Xu Liu, Qinzhe Wu, Andreas Gerstlauer, Tao Li, and Lizy K John. Start late or finish early: A distributed graph processing system with redundancy reduction. *Proceedings of the VLDB Endowment*, 12(2):154–168, 2018.

[123] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. *SIGPLAN Not.*, 41(6):387–400, jun 2006.

[124] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. *ACM SIGPLAN Notices*, 40(10):59–76, 2005.

[125] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 59–76, New York, NY, USA, 2005. Association for Computing Machinery.

[126] Yu Su, Ding Ye, and Jingling Xue. Parallel pointer analysis with cfl-reachability. In *2014 43rd International Conference on Parallel Processing*, pages 451–460, 2014.

[127] Gabriel Tanase, Antal Buss, Adam Fidel, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Xiabing Xu, Nedal Mourad, Jeremy Vu, et al. *The STAPL parallel container framework*, volume 46. ACM, 2011.

[128] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–95, 2015.

[129] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[130] Stanley Tzeng, Anjul Patney, and John D Owens. Task management for irregular-parallel workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*, pages 29–37. Eurographics Association, 2010.

[131] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[132] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *2016 USENIX Annual Technical Conference (USENIX ATC)*, pages 507–522, 2016.

[133] Stephan M Wagner and Nikrouz Neshat. Assessing the vulnerability of supply chains using graph theory. *International Journal of Production Economics*, 126(1):121–129, 2010.

[134] Stephan M Wagner and Nikrouz Neshat. Assessing the vulnerability of supply chains using graph theory. *International Journal of Production Economics*, 126(1):121–129, 2010.

[135] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 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*, pages 389–404. ACM, 2017.

[136] Kai Wang, Guoqing (Harry) Xu, Zhendong Su, and Yu David Liu. Graphq: Graph query processing with abstraction refinement-scalable and programmable analytics over very large graphs on a single pc. In *USENIX Annual Technical Conference*, pages 387–401, 2015.

[137] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 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*, page 11. ACM, 2016.

[138] Brandon West, Adam Fidel, Nancy M Amato, Lawrence Rauchwerger, et al. A hybrid approach to processing big data graphs on memory-restricted systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 799–808. IEEE, 2015.

[139] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. In *ACM SIGPLAN Notices*, volume 48, pages 57–68. ACM, 2013.

[140] Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *European Conference on Object-Oriented Programming*, pages 98–122. Springer, 2009.

[141] Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, page 98–122, Berlin, Heidelberg, 2009. Springer-Verlag.

[142] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. Graph processing on gpus: Where are the bottlenecks? In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 140–149. IEEE, 2014.

[143] Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 155–165, 2011.

[144] Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, page 155–165, New York, NY, USA, 2011. Association for Computing Machinery.

[145] Yi Yang and Huiyang Zhou. Cuda-np: realizing nested thread-level parallelism in GPGPU applications. In *ACM SIGPLAN Notices*, volume 49, pages 93–106. ACM, 2014.

[146] Mihalis Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, page 230–242, New York, NY, USA, 1990. Association for Computing Machinery.

[147] Eddy Z Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 369–380. ACM, 2011.

[148] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. Exploring the hidden dimension in graph processing. In *OSDI*, pages 285–300, 2016.

[149] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. Wonderland: A novel abstraction-based out-of-core graph processing system. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 608–621, 2018.

[150] Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 435–446, 2013.

[151] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. Efficient subcubic alias analysis for c. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 829–845, 2014.

[152] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Accelerate large-scale iterative computation through asynchronous accumulative updates. In *Proceedings of the 3rd workshop on Scientific Cloud Computing*, pages 13–22. ACM, 2012.

[153] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. DiGraph: An efficient path-based iterative directed graph processing system on multiple GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 601–614. ACM, 2019.

[154] Zhijia Zhao and Xipeng Shen. 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*, pages 619–630, 2015.

[155] Zhijia Zhao, Bo Wu, and Xipeng Shen. 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, 2014.

[156] Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 197–208, 2008.

[157] Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 197–208, New York, NY, USA, 2008. Association for Computing Machinery.

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

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

[160] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 301–316, 2016.

[161] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC)*, pages 375–386, 2015.

[162] Zhiqiang Zuo, Rong Gu, Xi Jiang, Zhaokang Wang, Yihua Huang, Linzhang Wang, and Xuandong Li. Bigspa: An efficient interprocedural static analysis engine in the cloud. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 771–780. IEEE, 2019.