Scaph: Scalable GPU-Accelerated Graph Processing with Value-Driven Differential Scheduling

Long Zheng, Xianliang Li, Yaohui Zheng, Yu Huang, Xiaofei Liao, and Hai Jin, *Huazhong University of Science and Technology;* Jingling Xue, *UNSW Sydney;* Zhiyuan Shao and Qiang-Sheng Hua, *Huazhong University of Science and Technology*

https://www.usenix.org/conference/atc20/presentation/zheng

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

Scaph: Scalable GPU-Accelerated Graph Processing with Value-Driven Differential Scheduling

Long Zheng[1] Xianliang Li[1] Yaohui Zheng[1] Yu Huang[1] Xiaofei Liao[1] Hai Jin[1]
Jingling Xue[2] Zhiyuan Shao[1] Qiang-Sheng Hua[1]
[1]*National Engineering Research Center for Big Data Technology and System/Service Computing Technology and System Lab/Cluster and Grid Computing Lab, Huazhong University of Science and Technology*
[2]*UNSW Sydney*
{longzh, xianliang, yaohui, yuh, xfliao, hjin, zyshao, qshua}@hust.edu.cn; j.xue@unsw.edu.au

Abstract

We introduce Scaph, a GPU-accelerated graph system that achieves scale-up graph processing on large-scale graphs that are initially partitioned into subgraphs at the host to enable iterative graph computations on the subgraphs on the GPU. For active subgraphs to be processed on GPU at an iteration, the prior work always streams each in its entirety to GPU, even though only the neighboring information for its active vertices will ever be used. In contrast, Scaph boosts performance significantly by reducing the amount of such redundant data transferred, thereby improving the effective utilization of the host-GPU bandwidth drastically. The key novelty of Scaph is to classify adaptively at each iteration whether a subgraph is a high-value subgraph (if it is likely to be traversed extensively in the current and future iterations) or a low-value subgraph (otherwise). Scaph then schedules a sub-graph for graph processing on GPU using two graph processing engines, one for high-value subgraphs, which will be streamed to GPU entirely and iterated over repeatedly, one for low-value subgraphs, for which only the neighboring information needed for its active vertices is transferred. Evaluation on real-world and synthesized large-scale graphs shows that Scaph outperforms the state-of-the-art, Totem (4.12×), Graphie (8.93×), and Garaph (3.71×), on average.

1 Introduction

Graph processing is used in a variety of real-world applications, including path navigation [23], social network analysis [9], and financial fraud detection [27]. Graph processing, typically memory-bound, often benefits substantially from memory optimizations [50]. Compared to CPU-based graph systems [15, 16, 30, 36, 40, 46, 51, 68], GPU-accelerated graph systems can have high internal bandwidth and massive parallelism, therefore offering superior speedup [19, 25, 39, 66], even for graph algorithms that involve substantial light-weight integer and comparison-based operations [28].

Unfortunately, many real-world graphs still cannot fit into GPU memory to enjoy high-performance in-memory graph processing. For example, NVIDIA's high-end Tesla V100 has 32GB global memory [42], while real-world graphs such as `Facebook`'s can easily reach the terabyte-scale [9]. This gap has spurred the development of many distributed graph systems, which partition a graph into sub-graphs and then assign these sub-graphs to different machines for distributed computing [13, 15, 16, 33, 36, 67]. However, these distributed graph systems suffer from prohibitive communication overheads [8, 15, 58] and also require an extensive range of domain knowledge to maintain [11, 16, 24, 38, 56, 59].

	GTX980	K40	P100
Arch.	Maxwell	Kepler	Pascal
#SMXs	16	15	56
#Cores	2048	2880	3584
Memory	4GB	12GB	16GB
BW	224 GB/s	288 GB/s	720 GB/s

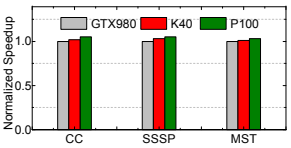


Figure 1: Performance of Graphie [17] for three representative graph algorithms on `fb-2009` (a graph with 139.1M vertices and 12.3B edges, taking 137.8GB (unweighted) and 275.6GB (weighted)) on three different generations of GPUs plugged (separately) in a 28-core host machine with 512GB memory

There is nowadays a viable alternative of turning a single machine plugged in with a GPU to support scale-up large-scale graph processing. Such a GPU-accelerated heterogeneous platform is easy to use and maintain [30, 62, 68]. In addition, we can take advantage of the large host memory (at the terabyte scale) to store large-scale graphs while still enjoying high-performance graph processing on GPU.

In this paper, we focus on building graph systems on GPU-accelerated heterogeneous platforms to achieve scale-up graph processing for large graphs that cannot fit into GPU memory. This would enable high-performance graph analytics on large-scale graphs everywhere by simply plugging a GPU into an off-the-shelf commodity PC. In this case, a large graph must be partitioned into subgraphs at the host. Any subgraphs to be processed on GPU must be streamed asynchronously to GPU when some previously transferred subgraphs are being concurrently processed on GPU (in an overlapping manner). We consider vertex-centric graph processing [36], where a

graph algorithm is performed in a sequence of iterations until convergence [15, 36]. In each iteration, a graph algorithm processes only the *active vertices* (vertices with ongoing updates) in each subgraph, updates their neighbors (along their outgoing edges) and activates the neighbors whose values have been updated. In this paper, we restrict ourselves to handle large-scale graphs that can entirely fit into the host memory. Meanwhile, all the vertex data, including vertex states (active or not), are assumed to be resident in the GPU memory. In contrast, the edge data of a graph are stored at the host and partitioned into subgraphs. During graph processing, *active subgraphs* (containing all out-going edges of an active vertex) must be transferred to GPU for iterative processing.

Achieving scale-up graph processing for large-scale graphs on GPU-accelerated heterogeneous platforms is challenging. The power-law graphs [15] can result in substantial load imbalance among threads and warps [39]. Irregular data accesses made in graph algorithms often lead to non-coalesced memory accesses for GPU graph processing. Fortunately, effective techniques for addressing these performance-limiting issues exist [14, 17, 34]. Currently, the performance bottleneck in a GPU-accelerated graph system has shifted to the limited host-GPU bandwidth, which was relatively sufficient in the past (e.g., ∼11.4GB/s for PCI-Express 3.0). However, existing graph processing engines [17, 26, 34, 47] focus still on overcoming the GPU memory capacity limitation to enable large-scale graph processing, without paying adequate attention to the effective utilization of the host-GPU bandwidth.

Simple heuristics are used to reduce the number of data transfers. Totem [14] partitions a graph into two subgraphs, one for the host and one for GPU, by keeping the amount of data transfers to a minimum at the expense of severe load imbalance. Garaph [34] concurrently processes all active subgraphs on both the host and GPU. Graphie [17] processes all subgraphs on GPU but re-processes only the recently processed subgraphs in the next iteration (before they are removed from GPU memory). However, these graph systems always transfer an active subgraph in its entirety to GPU (even though only the neighboring information for its active vertices will usually be used), resulting in poor utilization of the host-GPU bandwidth. To see this, Figure 1 compares the performance results of Graphie [17] for running three graph algorithms on a large graph on a PC with three generations of GPUs (one at a time). We see little performance gains when increasingly more powerful GPUs are used. For example, P100 has over 3× as many #SMX's and 4× as much memory as GTX980, but it offers small performance improvements.

Recently, hardware vendors have launched several advanced interconnect technologies to mitigate the impact of the "bandwidth wall". For example, compared to PCI-E 3.0, NVLINK 2.0 (50GB/s per link) and PCI-E 4.0 (32GB/s) are several times faster, but still cannot keep up with the growth in GPU computing capabilities. Specifically, these advanced technologies cannot yet provide ∼500GB/s required by graph analytics under existing computing platforms [1].

In this work, we argue that we can improve the performance of large-scale graph processing on a GPU-accelerated architecture significantly by improving the effective utilization of the host-GPU bandwidth. Our key observation is that the majority of the data in an active subgraph (once streamed to GPU) are never used in current and future iterations (§2.2). We introduce Scaph that achieves significantly improved performance than state of the art by adopting value-driven differential scheduling for active subgraphs. The key novelty is to classify an active subgraph adaptively into a *high-value subgraph* (if it will be extensively traversed in current and future iterations) and a *low-value subgraph* (otherwise). Thus, a high-value subgraph contains a significant amount of *useful data (UD)* to be used by active vertices in the current iteration and of *potentially useful data (PUD)* to be used by its future active vertices in future iterations. On the other hand, a low-value subgraph contains a lot of *never-used data (NUD)* in current and future iterations.

Unlike earlier graph systems [17, 26, 34, 47], which transfer an active subgraph to GPU in its entirety (but with only its UD used usually), Scaph uses the host to stream an active sub-graph to GPU by using two graph processing engines for handling high-value and low-value subgraphs, respectively. For the high-value subgraph, it will be transferred to GPU entirely. Inspired by the data movement reduction in out-of-core settings [2, 53, 69], we propose to compute each high-value subgraph multiple times to exploit its PUD ahead of schedule for accelerating convergence. Unlike these earlier efforts focusing on exploiting only the PUD in a subgraph, we present a GPU-context-friendly delayed scheduling to enable exploiting the PUD *across subgraphs on GPU* such that the value of the high-value subgraphs can be maximized. For the low-value subgraph, only the neighboring information for its active vertices is transferred and scheduled once.

In summary, this paper makes the following contributions:

- *Subgraph Value Characterization.* We quantify the value of a subgraph adaptively (dynamically) in terms of its UD and PUD used in current and future iterations.
- *Value-Driven Differential Scheduling.* We propose a scheduler that adaptively distinguishes high- and low-value subgraphs in each iteration and dispatches a subgraph to an appropriate graph processing engine for acceleration.
- *Value-Driven Graph Processing Engines.* We introduce two graph processing engines to squeeze the most value out of high- and low-value subgraphs to maximize the effective utilization of the host-GPU bandwidth in each case.
- *Evaluation.* We evaluate Scaph on both real-world and synthesized large graphs. Scaph outperforms state-of-the-art heterogeneous graph systems, Totem (4.12×) [14], Graphie (8.93×) [17], and Garaph (3.71×) [34], on average.

The rest of this paper is organized as follows. §2 describes the background and motivation. §3 gives an overview of

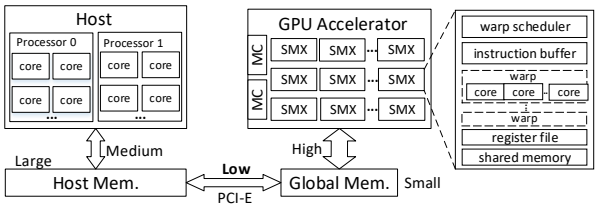


Figure 2: A GPU-accelerated heterogeneous architecture

Scaph. §4 describes value-driven differential scheduling while §5 discusses how to accomplish this effectively. §6 presents results. §7 discusses the related work. Finally, §8 concludes.

2 Background and Motivation

We first review the background. We then present some case studies to reveal why the poor host-GPU bandwidth utilization has limited the performance achieved by existing heterogeneous graph systems, finally motivating Scaph.

2.1 Host-GPU Heterogeneous Architectures

Figure 2 shows a representative GPU-accelerated heterogeneous architecture that integrates the hardware advantages of the host (with a larger host memory) and the GPU (with a stronger computing ability). A GPU consists of multiple *streaming multiprocessors* (SMXs), each of which includes hundreds of cores. Compared to the high-speed internal bandwidth (e.g., ∼720GB/s for NVIDIA Tesla P100 [41]) of GPU cores accessing global memory, a GPU is generally connected to the host with a relatively slow interface. For example, the host-GPU bandwidth via PCI Express 3.0 can be limited to be as low as ∼11.4GB/s in practice [5]. This significant performance gap often severely limits the performance potential achieved on a GPU-accelerated heterogeneous architecture if the host-GPU data transfers are frequent [17, 26]. This work makes use of a PCI Express interconnect since it is commonly used in the current commodity market.

2.2 A Motivating Study

Existing heterogeneous graph systems [17, 26, 47], with Graphie [17] as a representative compared against in our evaluation, generally use host memory to store large-scale graphs (partitioned into subgraphs) and rely on GPUs exclusively to accelerate graph analytics on these subgraphs. Figure 3 depicts their generic graph processing engine used, with the function calls in blue executed on GPU. Due to the limited GPU memory, a graph G is first divided into subgraphs, $\tilde{G}_1, \cdots, \tilde{G}_n$ (line 2). During the entire iterative graph processing, the vertex data of G always reside in GPU memory, but the edge data of G, which are spread across these subgraphs, will be streamed to GPU on-demand [17, 26, 34, 47].

At each iteration (lines 5 – 12), $\tilde{G}_{active}$ represents the set of active subgraphs, i.e., the ones containing some out-going edges of an active vertex. In each iteration, all active vertices

```
1  Procedure SimpleSubgraphEngine(Graph G)
2      Load G̃'s subgraphs in {G̃_1, ··· , G̃_n} into the host
3      VertexInitialization(G)
4      G̃_active ← FindActiveSubgraph(G)
5      while G̃_active ≠ ∅ do
6          foreach G̃_i ∈ G̃_active do
7              stream ← DispatchStream(G̃_i)
8              if G̃_i is not resident in GPU memory then
9                  GBuf ← AllocateDeviceMemory( )
10                 TransferData(stream, GBuf, G̃_i, CPU2GPU)
11             Kernel (stream, G̃_i)
12         G̃_active ← FindActiveSubgraph(G)
   /* Graph Processing Kernel on the GPU */
13 Procedure Kernel(Subgraph G̃)
14     foreach v ∈ G̃.SetOfVertices do
15         if v is active then
16             foreach e ∈ v.outedges do
17                 if Update(v, e) = SUCCESS then
18                     Activate(e.destination_vertex)
```

Figure 3: Existing graph processing engine on a GPU-accelerated heterogeneous architecture (with the function calls in blue executed on GPU and all the rest on the host)

Table 1: The amount of used/unused data in the subgraphs transferred to GPU

	Algo.	Used	Unused
TW	CC	12.15GB	21.44GB
	SSSP	22.74GB	77.42GB
	MST	25.78GB	106.47GB
UK	CC	43.41GB	688.43GB
	SSSP	81.64GB	1302.85GB
	MST	134.97GB	2099.25GB

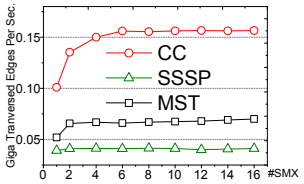


Figure 4: Performance of Graphie for TW with different number of SMXs

are processed. If their out-going edges are not in the GPU, their containing (active) subgraphs are transferred to GPU in their entirety. Afterward, these active vertices will be processed on the GPU (lines 13 – 18) to activate more destination vertices possibly. Note that Graphie [17] may schedule first the subgraphs processed at the end of the previous iteration as they are still in GPU memory (line 8).

This simple graph processing engine does not effectively utilize the limited, scarce host-GPU bandwidth since many vertices in an active subgraph are not active. Simply transferring an entire subgraph to GPU (line 10) but consuming only a fraction of its data (lines 14 – 15) will waste a considerable amount of the host-GPU bandwidth. As a result, all the required data cannot arrive at the GPU promptly, limiting the performance that can be potentially achieved on GPU.

Let us examine the ratios of the unused over used data in the subgraphs transferred to GPU for three graph algorithms operating on two graphs, twitter (TW) [29] and uk-2007 (UK) [6], by Graphie [17] using the graph processing engine given in Figure 3. Table 1 gives the results obtained through an offline trace analysis, showing that these ratios range from 6.29 to 36.17. This indicates that the host-GPU bandwidth under Graphie is utilized rather ineffectively. Consequently, as shown further in Figure 4, the performance of Graphie for

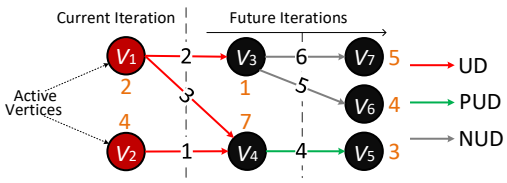
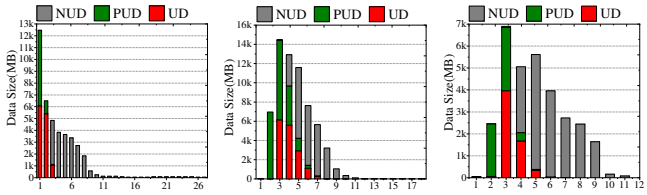


Figure 5: UD, PUD, and NUD in a subgraph, which may change across the iterations, illustrated for SSSP. The weight of an edge denotes its distance. The shortest distance found so far by SSSP at a vertex is depicted next to it in orange.



(a) CC (b) SSSP (c) MST

Figure 6: The amount of UD, PUD, and NUD across the iterations for three graph algorithms on twitter (TW) [29]

each graph algorithm (operating on TW) has plateaued as soon as #SMXs = 4. However, mainstream GPU accelerators usually have far more than 4 SMXs. For example, NVIDIA's Tesla K80 has 26 SMXs, while P100 has been integrated with 56 SMXs. Thus, a significant gap remains between the poor provision of data and high-speed computation of GPU.

2.3 Value-Driven Subgraph Scheduling

For a subgraph, its active vertices vary across the iterations. However, from the perspective of an active vertex, it always contains three types of edge data, as illustrated in Figure 5:

- *Useful Data (UD).* These are the edge data associated with all the active vertices in a subgraph, i.e., $V_1 \xrightarrow{2} V_3$, $V_1 \xrightarrow{3} V_4$, and $V_2 \xrightarrow{1} V_4$ in Figure 5. UD will definitely be used in the current iteration (lines 15 – 16 in Figure 3) and must be transferred to GPU [17, 26, 47].
- *Potentially Useful Data (PUD).* These are the edge data associated with all the future active vertices in future iterations in a subgraph. In Figure 5, PUD will be just $V_4 \xrightarrow{4} V_5$, since V_4 will be the only one activated by both V_1 and V_2 in current and future iterations. Unlike UD, PUD is not actually used in the current iteration, but may be transferred repeatedly to GPU if not handled carefully (as in the case of Figure 3 where PUD is usually discarded).
- *Never Used Data (NUD).* These are the edge data that will never be used again in a subgraph, associated with its vertices that have converged and will thus never be active. In Figure 5, NUD are $V_3 \xrightarrow{5} V_6$ and $V_3 \xrightarrow{6} V_7$.

Note that the same vertex may be activated many times in different iterations. Given a subgraph, its UD, PUD, and NUD computed at different iterations can vary dynamically.

Figure 6 shows the amount of UD, PUD, and NUD for the active subgraphs across all the iterations for three graph algorithms operating on twitter (TW) [29], partitioned sequentially into subgraphs of 32MB each. Graphie [17], a representative of existing heterogeneous graph systems [17, 26, 47], wastes the host-GPU bandwidth in two ways (Figure 3). First, PUD, usually discarded by Graphie but needed in future iterations, is substantial in earlier iterations. Second, NUD, which is becoming increasingly more dominant as the iteration progresses, is streamed to GPU redundantly.

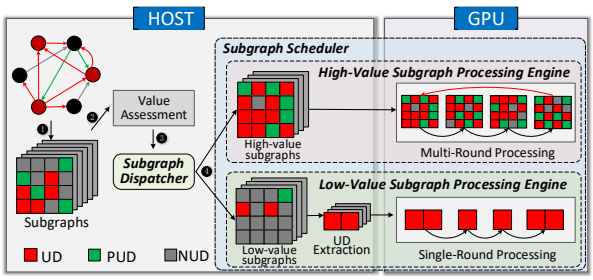


Figure 7: The workflow of Scaph

For a subgraph, it will be cost-ineffective to stream just its UD, since its PUD cannot be exploited simultaneously. Instead, our key insight for improving the effective utilization of the host-GPU bandwidth is to look beyond the current iteration, by considering not only its UD in the current iteration but also its PUD in future iterations. Based on a cost-benefit analysis, we aim to leverage rather than discard its PUD (once streamed to GPU) in iterative graph processing. Thus, the value of a subgraph at an iteration should be measured in terms of not only its UD but also its PUD.

Now, how do we extract the UD and PUD from a subgraph at a given iteration so that both can be transferred to GPU? Extracting the UD from a subgraph is easy as its active vertices in the current iteration are known (lines 4 and 12 in Figure 3). However, extracting precisely the PUD (without NUD) from a subgraph is difficult, as its future active vertices are not known yet during the current iteration.

For a given subgraph, we propose to predict its PUD size at an iteration from the UD sizes in the current and past iterations. This enables to adopt a value-driven differential scheduler that computes the value of a subgraph adaptively and schedules it depending on if it has a high value (when its UD and PUD are dominant) or a low value (otherwise).

3 Scaph Overview

Figure 7 shows the workflow of Scaph, in which all the subgraphs of a graph are computed on the GPU while the host is responsible for their preparation. At each iteration, its dispatcher classifies a subgraph into either a high-value or low-value subgraph and sends it to its corresponding engine to facilitate value-driven differential scheduling. Both engines schedule their subgraphs for acceleration on GPU independently but concurrently.

Value-Driven Subgraph Dispatcher. Conceptually, the value of a subgraph at a given iteration is proportional to the

amount of its UD and PUD. The key insight here is that, for a given subgraph, although accurately computing its PUD is difficult, its PUD size can be approximated based on the UD sizes in the current and past iterations. For a subgraph at a given iteration, Scaph's subgraph dispatcher (§4), classifies it adaptively as a high-value subgraph if it contains a sufficient amount of UD and PUD to justify its transfer in its entirety to GPU and a low-value subgraph to request only its UD to be transferred to GPU otherwise. This is done adaptively as the value of a subgraph changes as the iteration progresses.

Value-Driven Subgraph Scheduler. Scaph has two separate graph processing engines, described in §5, to process differentially high- and low-value subgraphs. For a high-value subgraph, we use a queue-assisted multi-round processing engine, which streams it entirely from the host to GPU (if it is not in GPU memory) and exploits both its UD and PUD adequately to enable faster convergence. For a low-value subgraph, Scaph relies on the graph processing engine given in Figure 3 but transfers only its UD to GPU, with the UD extracted in a NUMA-aware manner on the host.

Scaph is essentially a hybrid graph system that allows out-of-order computation of high-value subgraphs in each synchronous iteration. The use of asynchronous execution allows fast convergence but also changes the vertex scheduling priority of subgraphs. Therefore, a graph algorithm can use Scaph safely for preserving the convergence and the converged values, if it satisfies the correctness condition that the final vertex results are insensitive to the value propagation order.

4 Value-Driven Subgraph Dispatching

In Section 4.1, we quantify the value of a subgraph. In Section 4.2, we discuss how to estimate the value of a subgraph to support value-driven differential scheduling.

4.1 Quantifying the Value of a Subgraph

Graph computations proceed iteratively until convergence. Conceptually, the value of a subgraph $\tilde{G}$ can be measured in terms of its UD used in the current iteration and its PUD used in future iterations. Therefore, the *value* of $\tilde{G}$, denoted $Val(\tilde{G})$, from the current iteration Cur to the MAX-th iteration (beyond which $\tilde{G}$ is no longer active), is defined as:

$$Val(\tilde{G}) = \sum_{i=Cur}^{MAX} \sum_{v \in \tilde{G}.SetOfVertices} D(v) * A^i(v) \qquad (1)$$

where $D(v)$ represents the number of out-going edges of vertex v restricted to $\tilde{G}$ and $A^i(v) \in \{0,1\}$ indicates that v is active (inactive) in the i-th iteration when $A^i(v) = 1$ ($A^i(v) = 0$). $Val(\tilde{G})$ represents the amount of computations arising from $\tilde{G}$ from the current iteration until convergence. According to Equation (1), the PUD of a subgraph is quantized by the number of its edges that will be used in future iterations.

The value of a subgraph depends upon its active vertices and their degrees. In the case of uniform degree distributions, the activation status of vertices can still differentiate the amount of UD, PUD, and NUD for a subgraph.

4.2 Value-Driven Differential Scheduling

Scaph emphasizes value-driven data transfers, which should directly reflect how the bandwidth is *effectively* utilized in order to enable faster convergence.

The intuition behind $Val(\tilde{G})$ is clear. If $Val(\tilde{G})$ is high, $\tilde{G}$ should be a high-value subgraph. Then we should transfer $\tilde{G}$ as a whole to GPU and also exploit its UD and PUD adequately by iterating over $\tilde{G}$ multiple times before it is removed from GPU memory. Otherwise, $\tilde{G}$ should be treated as a low-value subgraph. In this case, we will opt to transfer only its UD to GPU and just iterate over the resulting $\tilde{G}$ once.

If $\tilde{G}$ is a high-value subgraph, then the throughput of processing $\tilde{G}$ on GPU can be measured as follows:

$$T_{HV}(\tilde{G}) \quad = \quad \frac{|UD| + \lambda |PUD|}{|\tilde{G}|/BW + t_{barrier}} \qquad (2)$$

The denominator $|\tilde{G}|/BW + t_{barrier}$, which represents the data transfer time for $\tilde{G}$, is used to approximate the time elapsed on processing $\tilde{G}$ by assuming a complete overlap between data transfers and computations on GPU. As $\tilde{G}$ is transferred in its entirety to GPU, $|\tilde{G}|$ denotes the amount of data thus transferred, BW represents the host-GPU bandwidth, and $T_{barrier}$ is the synchronization overhead for $\tilde{G}$ (amortized by the number of active subgraphs processed). The numerator $|UD| + \lambda * |PUD|$ represents the amount of UD and PUD accessed when $\tilde{G}$ is iterated over on GPU. We use a balancing factor λ to decay $|PUD|$, where $0 \leqslant \lambda \leqslant 1$, to signify the actual amount of PUD accessed.

If $\tilde{G}$ is a low-value subgraph, then we have:

$$T_{LV}(\tilde{G}) \quad = \quad \frac{|UD|}{|UD|/BW + t_{barrier}} \qquad (3)$$

This time, only the UD of $\tilde{G}$ is streamed to GPU.

Now, $\tilde{G}$ is a high-value subgraph if $T_{HV}(\tilde{G}) \geqslant T_{LV}(\tilde{G})$ and a low-value subgraph otherwise. Thus, we need to analyze:

$$|UD| + \lambda |PUD| (1 + \frac{t_{barrier}}{|UD|/BW}) > |\tilde{G}| \qquad (4)$$

To verify $T_{HV}(\tilde{G}) \geqslant T_{LV}(\tilde{G})$, the key lies in determining $|PUD|$, which is difficult to obtain directly. In fact, for a subgraph, its PUD is technically activated from its UD, motivating us to estimate the PUD of a subgraph heuristically based on the UD of the same subgraph. In this work, we consider a subgraph to have a high value if either of the following two conditions (which we found to work well across all of our applications, as confirmed in §6) holds to simplify Equation (4):

```
1  Procedure VDDSEngine(Graph G)
2      Distribute G's subgraphs {G̃1,···,G̃n} to NUMA
         nodes
3      VertexInitialization(G)
4      G̃active ← FindActiveSubgraph(G)
5      Transfer VertexStates from GPU to CPU
6      while G̃active ≠ ∅ do
7          foreach G̃ ∈ G̃active do
8              if Predictor( G̃) = "HIGH-VALUE" then
9                  Push(HVworklist, G̃)
10             else
11                 Push(LVworklist, G̃)
12         HVSPEngine(HVworklist)
13         LVSPEngine(LVworklist, VertexStates)
14         G̃active ← FindActiveSubgraph(G)
15         Transfer VertexStates from GPU to CPU
```

Figure 8: Value-driven differential scheduling for high- and low-value subgraphs, with the calls in blue executed on GPU

- $|UD|/|\tilde{G})| > \alpha$. This indicates that UD is dominant among $\tilde{G}$. Intuitively, $\tilde{G}$ is a high-value subgraph.
- $|UD_{current}| - |UD_{last}| > 0$ and $|UD|/|\tilde{G}| > \beta$. UD remains a medium level and is also growing increasingly over iteration, indicating the potentially growing PUD. $\tilde{G}$ can be thus treated as a high-value subgraph.

When α is relatively large, which implies that the UD in a subgraph tends to be dominant, we can determine if it is a high-value subgraph by considering only its UD. β is needed to identify the high-value subgraphs where the amount of UD is relatively low and that of PUD is potentially high. Thus, β is often smaller than α. As shown in Table 1, considering both together is often more effective than considering either alone. In this work, α and β are set empirically as 50% and 30% to represent a nice point for yielding good results.

Figure 8 gives our value-driven differential scheduler, `VDDSEngine()`, for scheduling a graph G. Initially, G is partitioned into subgraphs, $\tilde{G}_1, \cdots, \tilde{G}_n$, at the host and distributed across its NUMA nodes (to facilitate their scheduling). Scaph uses two graph processing engines, as described in §5 below, `HVSPEngine()` for scheduling high-value subgraphs, and `LVSPEngine()` for scheduling low-value subgraphs. In line 8, Scaph uses the above heuristic predictor to estimate the value of an active subgraph. Note that both engines work independently but concurrently. `LVSPEngine()` needs *VertexStates* in order to perform UD extraction for the active vertices in each subgraph. The UD extraction can be overlapped effectively with the data transfers in `HVSPEngine()`. At the end of each iteration (line 15), Scaph will transfer back the updated vertices from the GPU to the CPU. Edges, which are not modified, are thus not transferred.

5 Value-Driven Subgraph Processing

Scaph has two graph processing engines. We describe the one for handling high-value subgraphs in §5.1 and the one for handling low-value subgraphs in §5.2.

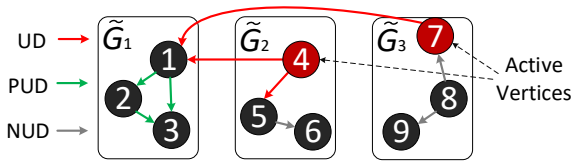


Figure 9: An example illustrating value propagation across the subgraphs, with ❶ activatable by ❹ and ❼. The PUD in $\tilde{G}_1$ can be exploited only if $\tilde{G}_2$ and/or $\tilde{G}_3$ are processed first.

5.1 High-Value Subgraph Processing

The key to extracting the most value out of high-value subgraphs lies in how to fully exploit their PUD. A useful idea of running each loaded subgraph multiple times is leveraged in the out-of-core settings [2, 53, 69] to exploit the *intrinsic value in a subgraph* for reducing the number of I/Os between memory and disk. However, under a GPU-accelerated heterogeneous architecture, subgraphs must often be small enough (in several tens of millions of bytes [17, 26]) against the ones in out-of-core settings, to enable fine-grained GPU scheduling. In this case, simply iterating over such a small-sized subgraph multiple times is often ineffective, since it can exploit only the PUD of its active vertices activated by its other active vertices but not active vertices from other subgraphs.

In Scaph, we improve the PUD exploitation significantly by enabling exploiting the *external value across the subgraphs*. Our key observation is that: *given a subgraph already available in GPU memory, scheduling it again* ***after a period of delay*** *can expose its PUD more fully than processing it repeatedly*. Figure 9 illustrates this with three subgraphs, exhibiting some complex inter-subgraph data dependencies (as is often the case in practice). We see that ❶ in $\tilde{G}_1$ can be activated by ❹ in $\tilde{G}_2$ and ❼ in $\tilde{G}_3$. Once ❶ in $\tilde{G}_1$ is activated, ❷ in $\tilde{G}_1$ may get activated (as shown). In this case, the edge data for ❶→❷, ❶→❸, and ❷→❸ are part of the PUD of $\tilde{G}_1$. By processing $\tilde{G}_1$ after $\tilde{G}_2$ or $\tilde{G}_3$ or both (even better), we can exploit such PUD to enable faster convergence. That is, repeatedly processing $\tilde{G}_1$ would not help.

Queue-Assisted Multi-Round Processing. The scheduling of high-value subgraphs at a given iteration is shown in Figure 10. We use a k-level priority queue $(PQ_1, \ldots, PQ_k)$ to enable re-scheduling a GPU-resident subgraph after some delay, where k indicates the maximum number of times some subgraphs have been processed in the current iteration. Thus, k varies from iteration to iteration. Figure 11 shows a case.

In each differential scheduling iteration orchestrated by `VDDSEngine` (Figure 8), `HVSPEngine(worklist)` is invoked, where *worklist* contains all the high-value subgraphs in this iteration. During the pre-processing (lines 2–6), each subgraph $\tilde{G}_i$ in *worklist* is examined in turn. $\tilde{G}_i$ will be enqueued into PQ_1 (if not already there) if $\tilde{G}_i$ remains to be GPU-resident (i.e., in one of $\{PQ_1, \ldots, PQ_k\}$) from the previous iteration and inserted into *TransSet* (waiting to be streamed to GPU) otherwise. Thus, there are two concurrently executed modules,

```
1  Procedure HVSPEngine(worklist)
2      foreach G̃_i ∈ worklist do
3          if G̃_i is resident in GPU memory then
4              Push(PQ_1, i)
5          else
6              Push(TransSet, i)
       /* Subgraph Transferring Module */
7      while TransSet ≠ ∅ do
8          if copystream is available then
9              i ← Pop(TransSet)
10             if GPU has available memory for one subgraph then
11                 Gbuf ← AllocateDeviceMemory( )
12             else
13                 j ← Pop(PQ_k)
14                 Gbuf ← GetGbuf(G̃_j)
15             TransferData(copystream, Gbuf, G̃_i, CPU2GPU)
16             Push(PQ_1, i)
       /* Subgraph Scheduling Module */
17     while worklist ≠ ∅ do
18         if at least one stream in execstreams is available then
19             stream ← Available(execstreams)
               /* Exploit the UD of a subgraph in PQ_1 */
20             if PQ_1 ≠ ∅ then
21                 i ← Pop(PQ_1)
22                 Kernel(stream,G̃_i)
23                 Erase(worklist, G̃_i)
               /* Exploit the PUD of a subgraph in PQ_i, where i≠1
               */
24             else
25                 for p ← 2 to k do
26                     if PQ_p ≠ ∅ then
27                         i ← Pop(PQ_p)
28                         Kernel(stream,G̃_i)
29                         break
30             priority ← GetPriority(G̃_i)
31             Push(PQ_priority+1, G̃_i)
```

Figure 10: High-value subgraph processing in each iteration (called from Figure 8). The Kernel function is from Figure 3. The two colored code regions are executed in parallel.

Subgraph Transferring and Subgraph Scheduling.

The Subgraph Transferring module (lines 7 – 16) is responsible for streaming asynchronously the subgraphs in *TransSet* to GPU. This is done by using some free GPU memory whenever possible (line 11) or making some free by dequeuing a subgraph from the multi-level queue (lines 13 – 14). Due to lines 4 and 16, all subgraphs in *worklist* are initially enqueued into PQ_1, and thus assigned with the highest priority.

The Subgraph Scheduling module (lines 17 – 31) is responsible for scheduling the subgraphs in $PQ_1, \cdots, PQ_k$. The subgraphs in PQ_1 are processed first (for the first time in the current iteration) to exploit their UD (lines 21 – 23). If $PQ_1 = \emptyset$ (implying that some subgraphs are still being transferred to GPU asynchronously), the scheduler will dequeue a subgraph from a non-empty PQ_i, where i is the smallest, to exploit its PUD (lines 25 – 29), as this will be the i-th time that the subgraph is processed (in the i-th round) of the current iteration. Simultaneously, the data transfers for PQ_1 and the computations for $PQ_2, \cdots, PQ_k$ are maximally overlapped, too. In either case, the priority of a subgraph, once processed, drops by one (lines 30 – 31). This delayed re-scheduling attempts to maximize the PUD exploitation, by, e.g., increasing the chances for $\tilde{G}_1$ to be processed after $\tilde{G}_2$ and/or $\tilde{G}_3$ in Figure 9 (as motivated earlier). Consider $\tilde{G}_3$, which resides in PQ_1, in Figure 11. Once we have exploited its UD, we will move it to PQ_2 so that we can exploit its PUD after $\tilde{G}_7$, $\tilde{G}_4$, $\tilde{G}_8$, and $\tilde{G}_6$ have been processed.

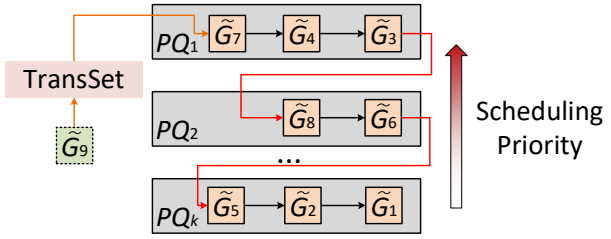


Figure 11: Subgraph processing with a k-level priority queue. PQ_i represents a queue PQ with the i-th priority. The smaller i is, the higher the priority is. All the subgraphs streamed from the host to GPU enter into PQ_1 initially.

Our scheduler with a multi-level priority queue guarantees that subgraphs are scheduled fairly, preventing them from bearing too many *useless computations* in the sense that the data of a vertex is computed but not updated.

Time and Space Complexity Analysis. k is expected to be bounded by $\frac{BW'}{BW}$ where BW' is the internal bandwidth of GPU and BW is the host-GPU bandwidth. In our computing platform, $BW' = 224$GB/s and $BW = 11.4$GB/s. Thus, $k \leqslant 20$ is typically expected.

As for the space complexity, a k-level priority queue is used to keep track of only the indices of the active subgraphs processed in an iteration. Thus, the worst complexity is $O(\frac{Mem_{GPU} \times \text{sizeof(SubgraphIndex)}}{|\tilde{G}|})$, where Mem_{GPU} is the global memory size and $|\tilde{G}|$ is the size of a subgraph $\tilde{G}$. In our computing platform, we have used $\frac{4\text{GB} \times 4\text{B}}{32\text{MB}} = 0.5$KB.

5.2 Low-Value Subgraph Processing

The key to exploiting the most value of low-value subgraphs is to extract their UD efficiently. We use multiple CPU cores at the host to parallelize the UD extraction. Due to *non-uniform memory access* (NUMA), however, scanning naively all the vertices in a subgraph to extract its UD can still be costly. In addition, different subgraphs exhibit different amounts of UD. Such scanning tasks are also prone to load imbalance.

Figure 12 gives our scheduler for low-value subgraphs. In each value-driven scheduling iteration orchestrated by `VDDSEngine()` in Figure 8, `LVSPEngine(worklist, VertexStates)` is invoked, where *worklist* contains all the low-value subgraphs that are active in this iteration, with their active vertices indicated in *VertexStates*. There are three modules, UD Extraction, Subgraph Transferring, and Subgraph Scheduling, which all execute concurrently. The major contribution here is a NUMA-aware parallel UD extraction.

UD Extraction. Initially, all the subgraphs partitioned from a graph are evenly distributed to different NUMA nodes,

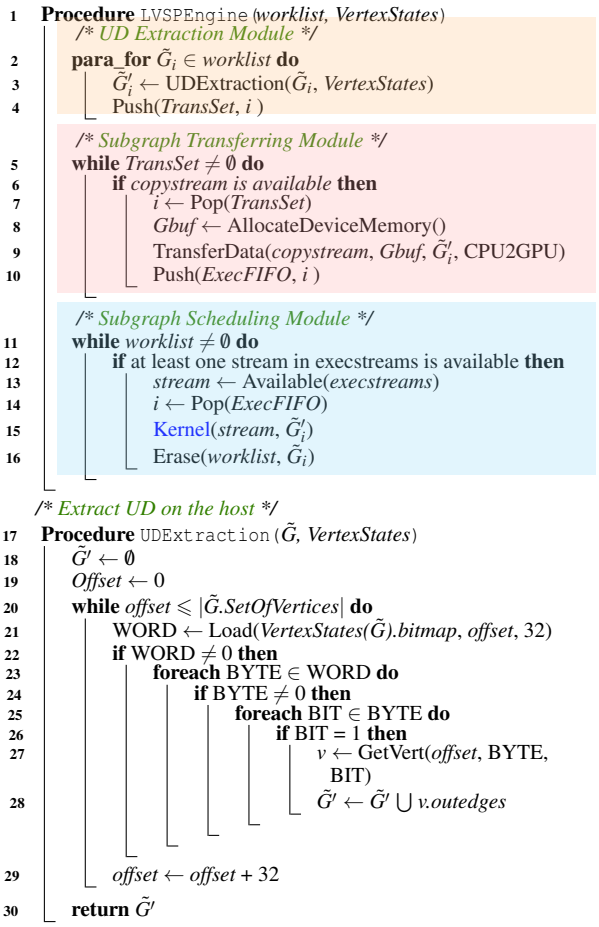


Figure 12: Low-value subgraph processing in each iteration (called from Figure 8). The Kernel function is from Figure 3. The three shaded code regions are executed in parallel.

with a NUMA node consisting of a CPU socket and its own memory banks (line 2 in Figure 8). The UD extraction module is given in terms of lines 2 – 4 and lines 17 – 30. To boost performance and improve intra-node load balancing, the UD extraction for each subgraph is done in its own thread, which is bound to the NUMA node storing the subgraph (line 3). To improve inter-node load balancing (as a minor optimization), we also duplicate in a NUMA node an equal number of randomly selected subgraphs from the other nodes (if there is still some memory space available). We adopt a simple bitmap-based approach to extract the UD from a subgraph $\tilde{G}$ efficiently (lines 17–30). All its vertices are stored in a bitmap, $VertexStates(\tilde{G}).bitmap$, with 1 (0) indicating that the corresponding vertex in $\tilde{G}$ is active (inactive). To accelerate its construction, the total of active vertices is computed on GPU.

Unlike high-value subgraphs, which can each be stored in the same-sized chunk in GPU memory (§5.1), low-value subgraphs may give rise to UD-induced subgraphs of varying sizes. To reduce fragmentation, Scaph further divides each chunk for storing a subgraph into smaller tiles (totaling 32 in our implementation). To store a UD-induced subgraph in GPU memory, Scaph will try to find consecutive tiles first in a partially filled chunk and then in a vacant chunk.

Subgraph Transferring. As in the case of high-value subgraph streaming in Figure 10, this module proceeds similarly except that a multi-level queue is no longer used.

Subgraph Scheduling. As in the case of scheduling high-value subgraphs to GPU in Figure 10, this module schedules UD-induced subgraphs (without using a multi-level queue).

6 Evaluation

We evaluate the efficiency and scalability of Scaph by answering the following four research questions (RQs):

- ***RQ1:*** How much more efficient is Scaph over state-of-the-art heterogeneous graph systems?
- ***RQ2:*** How effective is Scaph's value-driven differential scheduling in helping it achieve the overall performance?
- ***RQ3:*** How well does Scaph scale?
- ***RQ4:*** How much runtime overhead does Scaph introduce?

6.1 Experimental Setup

We compare Scaph with the following three state-of-the-art CPU-GPU heterogeneous graph systems:

- Totem [14]. A graph is divided into two subgraphs, which are processed by CPU and GPU, respectively. At the end of each iteration, the states of the active vertices that are activated reciprocally by the two subgraphs are exchanged.
- *Graphie* [17]. Like Scaph, a graph is initially partitioned at the host CPU and the subgraphs are then streamed to GPU for graph processing. Unlike Scaph, however, all active subgraphs are transferred to GPU in their entirety.
- *Garaph* [34]. At an iteration, all the subgraphs that are partitioned from a graph are processed concurrently by both the host and GPU if the number of outgoing edges of all active vertices in the entire graph exceeds 50% of the total number of edges and on the host only otherwise.

Subgraph Size. For Totem, Graphie, and Garaph, the sizes of subgraphs are selected from their papers. In Scaph, a graph is partitioned into subgraphs of 32MB each for several reasons. First, the host-GPU bandwidth tends to be under utilized with smaller sizes. Second, subgraphs will be streamed to GPU more frequently with larger sizes, as they tend to contain active vertices for more iterations. Finally, the kernel launching overheads appear to be well hidden with 32MB.

Graph Applications. We consider the first three typical graph algorithms (from different categories) and the latter two actual graph workloads (with different complexities): (1) *Single-Source Shortest Path* (SSSP) [60]–Sequential traversal, (2) *Connected Components* (CC) [20]–Parallel traversal, (3) *Minimum Spanning Tree* (MST) [37]–Graph mutation, (4) *Neural Network Digit Recognition* (NNDR) [4], and (5) *Graph-based Circuit Simulation* (GCS) [25]. All these algorithms fit the correctness criteria discussed in §3, though NNDR and GCS are already typically executed in an asyn-

Table 2: Graph datasets. The graph size is evaluated in the weighted edgelist representation.

Dataset	#Vertices	#Edges	Avg. Degree	Size
twitter (TW)	41.7M	1.47B	39.5	32.8GB
comfriend (FR)	124.8M	1.81B	14.5	40.4GB
sk-2005 (SK)	50.6 M	1.95B	38.5	43.6GB
uk-2007 (UK)	105.9M	3.74B	35.3	83.6GB
altavista-2002 (AV)	1.41G	6.64B	4.695	148.3GB
fb-2009 (FB)	139.1M	12.3B	88.7	275.6GB
RMAT-k (25<k<31)	2^k	2^{k+4}	16	-

Table 3: Execution times of Scaph, Totem, Graphie, and Garaph. Here, 'N/A' indicates that a graph algorithm has abnormally terminated due to some runtime error.

Algorithm	System	Execution Time (Secs)					
		TW	FR	SK	UK	AV	FB
CC	Totem	2.41	5.01	2.72	9.32	N/A	N/A
	Graphie	1.89	4.46	16.53	23.61	57.49	133.21
	Garaph	1.17	2.53	2.90	7.07	31.46	86.24
	Scaph	0.28	0.91	1.08	2.47	7.08	15.35
SSSP	Totem	5.94	5.78	7.07	19.21	N/A	N/A
	Graphie	5.32	9.24	52.01	89.44	218.51	413.07
	Garaph	3.71	4.45	6.83	16.52	114.68	204.35
	Scaph	0.92	1.67	3.17	6.64	29.06	38.87
MST	Totem	7.93	10.90	21.33	42.84	N/A	N/A
	Graphie	8.45	16.24	32.19	53.22	198.85	304.51
	Garaph	4.14	7.38	12.35	25.82	101.25	131.45
	Scaph	1.39	1.99	2.93	6.36	25.23	35.41
NNDR	Totem	6.47	6.63	12.17	29.43	N/A	N/A
	Graphie	5.38	7.32	28.19	49.81	234.04	457.13
	Garaph	3.41	4.76	9.28	28.74	116.34	175.34
	Scaph	1.77	2.08	2.99	5.13	20.19	33.55
GCS	Totem	19.77	23.04	59.51	98.11	N/A	N/A
	Graphie	24.08	38.84	50.34	93.29	454.41	834.59
	Garaph	10.53	15.56	20.438	39.45	185.58	299.76
	Scaph	3.33	4.08	10.46	16.13	39.52	54.94

chronous way, while the other algorithms are typically run in a synchronous, iterative manner.

Graph Datasets. We use (1) 6 real-world graphs [6, 31]) for performance evaluation, and (2) 5 large synthesized graphs (generated by the RMAT tool [7]) for scalability evaluation. Table 2 gives all the graphs used. For SSSP and MST that work on the weighted graphs, we randomly assign each edge of an unweighted graph with a weight ranging from 1 to 100.

Computing Platform. We evaluate Scaph on a machine where the host is equipped with two Intel 14-core Xeon CPUs, E5-2680v4@2.40GHz with 512GB memory (256GB on each of the two NUMA nodes). The GPU is NVIDIA P100 (with 56 SMXs, 3584 cores, and 16GB memory), connected to the host via the PCI Express 3.0 at 16x. The host-GPU bandwidth is around 11.4GB/s. We use NVCC V8.0.61 and g++ V5.4.0 to compile all the applications under "-O3". The operating system is Ubuntu 14.04 with Linux kernel 4.13.

6.2 RQ1: Efficiency

To answer RQ1, we compare Scaph against Totem [14], Graphie [17], and Garaph [34]. Table 3 depicts the results.

Scaph vs. Totem. The speedup of Scaph over Totem ranges from 2.23× (for SSSP on SK) to 7.64× (for CC on TW) with an average of 4.12×. Totem's critical performance bottleneck lies in its severe load imbalance, as it partitions each graph into only two subgraphs, one for the host (with 512GB memory) and one for GPU (with only 16GB memory). As a result, Totem cannot tap GPU's processing power to exploit adequately the UD and PUD in a graph. Its bottleneck is to ask the CPU to process most of the graph data, which would have been processed more efficiently by the GPU otherwise. A typical measurement for FB is for the CPU to handle 358.1GB and the GPU to handle only 16GB. In contrast, Scaph streams all subgraphs dynamically to GPU with value-driven differential scheduling, thereby exploiting more adequately GPU's processing power, and consequently, the UD and PUD in all the subgraphs. In the case of CC operating on FR, SK, and UK, their GPU portions under Totem are 39.6%, 36.7%, and 19.1%, respectively. As a result, Scaph outperforms Totem by 5.51x (FR), 2.52x (SK), and 3.77x (UK).

Scaph vs. Graphie. Scaph is faster than Graphie by 8.93× on average, with its speedup ranging from 3.03× (for NNDR on TW) to 16.41× (for SSSP on SK). Both Graphie and Scaph process all subgraphs on GPU only. So Graphie can be understood as a version of Scaph, where every subgraph is treated as a high-value subgraph except that only its UD is used but its PUD is exploited rather inadequately. Graphie is inferior to Scaph for several reasons. First, Graphie transfers an active subgraph entirely to GPU even though it contains only a few active vertices (i.e., a lot of NUD), wasting the host-GPU bandwidth. Second, Graphie exploits the UD only but PUD inadequately in an active subgraph.

Let us examine SSSP on SK, where the speedup of Scaph over Graphie is the highest (at 16.41×). Graphie converges in 75 iterations, by transferring 18,019 subgraphs totaling 374.4GB data to GPU. In contrast, Scaph converges in 16 iterations, by transferring 9,897 subgraphs totaling only 19.6GB data, comprising 13.2GB for 798 high-value subgraphs and 6.4GB for 9,099 low-value subgraphs. For Scaph, its significantly improved utilization for the host-GPU bandwidth has resulted in its significantly improved overall performance.

Scaph vs. Garaph. Scaph is faster than Garaph by 3.71×, with an overall rang from 1.93× (for NNDR on TW) to 5.62× (for CC on FB). Unlike Scaph, Garaph processes all the subgraphs on both the host and GPU if the active vertices in the entire graph have a lot of outgoing edges and on the host only otherwise (§6.1). Despite this, Garaph cannot distinguish high-value from low-value subgraphs as Scaph does. While being more effective than Graphie in reducing the amount of NUD transferred, Garaph is inferior to Scaph as it still transfers more NUD to GPU and exploits PUD less adequately.

Let us examine CC on FB, where the speedup of Scaph over Garaph is the highest (at 5.62×). Garaph processes all the subgraphs on the host only (as the outgoing edges of FB's active vertices over the total is under 6.9% at any iteration), by

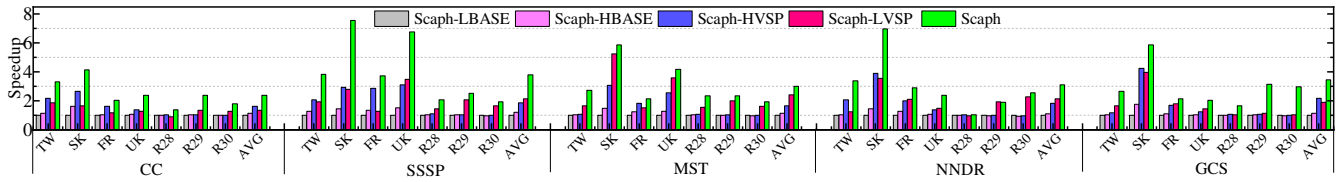


Figure 13: Speedup of Scaph, Scaph-HVSP, and Scaph-LVSP (normalized to Scaph-LBASE)

using a so-called notify-pull model. In contrast, Scaph uses a fine-grained value-driven differential scheduler to identify high-value and low-value subgraphs even though it has active vertices only in its local regions at any iteration, so that the GPU's processing power is adequately exploited.

6.3 RQ2: Effectiveness

To answer RQ2, we consider four variations of Scaph: (1) **Scaph-HVSP**, where all the low-value subgraphs can be understood as being misidentified as high-value subgraphs, (2) **Scaph-LVSP**, where all the high-value subgraphs can be understood as being misidentified as low-value subgraphs, (3) **Scaph-HBASE**, which applies the differential processing, but every subgraph transferred to the GPU has kept computation with a specific number of times (without using queue-based delayed scheduling), as used in CLIP [2], and (4) **Scaph-LBASE**, a variation of Scaph-LVSP except that every subgraph is streamed to GPU entirely (without UD extraction), as used in Graphie [17].

Figure 13 gives the results. We see that neither of Scaph-HVSP and Scaph-LVSP is always better than the other, and also Scaph is the best performer for all the algorithms on all the graphs. Thus, Scaph's value-driven differential scheduling with heuristic subgraph identification is highly effective.

Scaph-HVSP. Scaph-HVSP achieves better speedups for the graphs where algorithms take longer iterations to converge, as this allows it to exploit PUD more adequately and thus stream less redundant data to GPU. For example, each algorithm on SK has the longest number of iterations against on other graphs, thereby delivering considerable speedups. We also see that Scaph-HBASE is significantly inferior to Scaph-HVSP. This is because small subgraphs often contain very little PUD from themselves worthy of being exploited. Our queue-based scheduling allows the availability of PUD from other subgraphs via delayed scheduling. Thus, multi-time processing under Scaph-HVSP can expose significantly more PUD than that under Scaph-HBASE (i.e., by simply applying the idea from CLIP) for boosting performance.

Scaph-LVSP. Just like Scaph-HVSP, Scaph-LVSP can be quite effective in some cases. For example, the top two speedups achieved by Scaph-LVSP for MST are 5.26x and 3.58x on SK (14.8GB) and UK (27.61GB), respectively. The corresponding speedups from Scaph are 5.99x and 4.19x. However, Scaph-LVSP can be rather ineffective for the graphs that can nearly fit into the 16GB GPU memory, since Scaph-LBASE will then make GPU-resident for nearly all the subgraphs. For R28 with 16.78GB (unweighted) and 29.48GB (weighted), Scaph-LVSP offers little or even negative benefits for CC, NNDR, and GCS (on unweighted graphs) but positive ones for SSSP and MST (on weighted graphs).

Scaph. Scaph obtains the best of both worlds, Scaph-HVSP and Scaph-LVSP. For CC, SSSP, MST, NNDR, and GCS, the average speedups achieved by Scaph-HVSP (Scaph-LVSP) are 1.63x, 1.87x, 1.66x, 1.84, and 2.18x (1.33x, 2.12x, 2.41x, 2.15x, and 1.90x), respectively. As for Scaph, these average speedups are 2.38x, 3.79x, 3.01x, 3.12x, and 3.44x. Note that Scaph has the highest gain on SK, where Scaph-HVLP and Scaph-LVSP are also most effective.

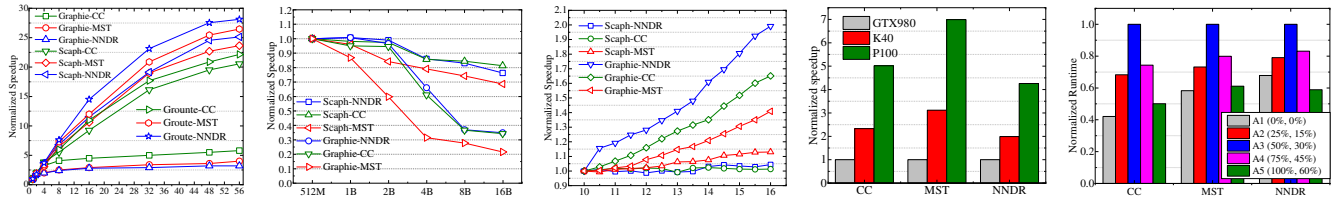
6.4 RQ3: Sensitivity Study

To answer RQ3, we investigate Scaph's scalability in terms of #SMXs, graph sizes, memory sizes, and GPU generations. We select Graphie as a reference on CC, MST, and NNDR.

#SMXs. Figure 14(a) compares Scaph and Graphie in terms of CC, MST, and NNDR on UK [6] for varying #SMXs by using all the 8GB GPU memory available. Scaph is significantly more scalable than Graphie for all the three graph algorithms, since Scaph can utilize the host-GPU bandwidth more effectively as already motivated earlier (Figures 1 and 4). For example, Graphie-MST reaches its plateau when #SMXs $= 2$, but Scaph-MST continues to offer a scalable performance improvement. CC and NNDR exhibit a similar trend.

However, Scaph's scalability degrades gradually as #SMXs increases, due to the integrated impacts of the intrinsic random accesses of graph processing on GPU [5, 25, 57] and the increasingly more SMXs competing for the memory bandwidth. As also shown in Figure 14(a), Groute [5], an in-memory graph system that can not handle over-subscription, on UK-2007@1M [6] (a sample graph with 1M vertices and 41M edges generated from UK), suffers from exactly the same scalability problem, which is beyond the scope of this work. We leave addressing this problem in future work.

Graph Sizes. Figure 14(b) compares Scaph and Graphie as the graph size increases. For CC and NNDR working on unweighted graphs, Scaph (Graphie) can store up to 4 billion (2 billion) edges in GPU memory. For MST working on the weighted graph, these edge counts drop to roughly 2 billion and 1 billion. Both Scaph and Graphie maintain their throughput well as the graph size increases but degrade visibly for the graphs that can no longer fit into GPU memory. However,



(a) #SMXs (b) Edge sizes (c) GPU memory (GB) (d) GPU generations (e) Varying α and β

Figure 14: Performance of Scaph and Graphie (including Groute for (a) only) in terms of varying (a) #SMXs: UK [6] and 8GB GPU memory, (b) graph sizes: R26–R30 [7] and 16GB GPU memory and #SMX=56, (c) GPU memory capacities: FB [31] and #SMX=56, (d) GPU genreations: FB [31], and (e) configurations of (α, β): FB [31], respectively. All results are normalized to the one obtained by itself with the smallest configuration.

Scaph has a slower performance reduction rate than Graphie, for two reasons. First, Scaph can better tap GPU's processing power due to its use of a multi-level priority queue for exploiting PUD more adequately and overlapping data transfers and GPU computation more effectively. Second, Scaph avoids transferring a large amount of NUD for low-value subgraphs.

GPU Memory Capacities. Figure 14(c) compares Scaph and Graphie for varying GPU memory sizes. Graphie is highly sensitive to the GPU memory capacity used, which determines directly how many subgraphs can be resident on GPU at an iteration and how many of these get re-processed in the ensuing iteration (before they are removed from GPU memory). In contrast, Scaph is nearly insensitive, since it exploits UD and PUD for high-value subgraphs and UD only for low-value subgraphs always. Note that Scaph is significantly faster than Graphie (Table 3). In Figure 14(c), Graphie improves over itself (normalized to 10GB) as the GPU memory size increases.

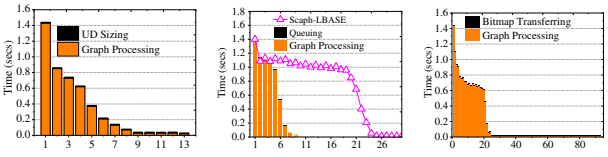
GPU Generations. Figure 14(d) characterizes the performance of Scaph on different GPU generations. Compared to Graphie that shows few speedups as shown in Figure 1, Scaph enables the significant speedups for K40 (1.99×∼3.12×) and P100 (4.26×∼5.02×) against that of GTX980.

Varying α and β. Figure 14(e) shows the sensitivity of the performance results of Scaph with respect to α and β. Here, A1 can be understood as Scaph-HVSP and A5 as Scaph-LVSP. Looking at A3, we see that increasing α and β causes more subgraphs to be mis-identified as low-value subgraphs (A4 and A5) and decreasing α and β causes more subgraphs to be mis-identified as high-value subgraphs (A1 and A2). Thus, A3 seems to represent a nice sweet spot for yielding good performance results. As for the problem of finding an optimal setting, we leave it as future work.

6.5 RQ4: Runtime Overhead

We discuss Scaph's overheads incurred in its *value-driven differential scheduling* (VDDS) given in Figure 8, *high-value subgraph processing* (HVSP) given in Figure 10, and *low-value subgraph processing* (LVSP) given in Figure 12.

VDDS. The cost of computing the subgraph value comes from computing the UD size for each iteration, on GPU, in line 8 of Figure 8. This is negligible, as shown in Figure 15(a).



(a) VDDS (b) HVSP (c) LVSP

Figure 15: Scaph's runtime overhead for running CC on UK [6] across the iterations

HVSP. The main overhead of HVSP lies in its queue management. In Figure 15(b), the cost incurred per iteration is small, representing an average of 0.79% of the total processing time. This small overhead is more than offset by the benefit reaped. In particular, the iteration count is reduced since most of the PUD can be computed ahead of schedule. The per-iteration time can be improved mainly because most of the NUD is discarded (rather than transferred expensively).

LVSP. The main overhead of LVSP lies in transferring a bitmap representation for all the active vertices in a subgraph from GPU to the host. As shown in Figure 15(c), the average cost incurred per iteration represents 4.3% of the total graph processing time. However, this cost increases relatively towards the last few iterations, reaching 57.4% at the end, where each subgraph has little UD to be acted upon.

6.6 Limitations

Graph Partition. Various partitions may show different value variations of subgraph at runtime. Scaph adopts a greedy vertex-cut partition [15] with the time taken depending on the number of partitions. It would be interesting future work to find a more reasonable partition method that can make most of UD and PUD exploited in the early stage of graph processing for faster convergence.

Disk-based Heterogeneous Graph Systems. The performance of Scaph is insensitive to the difference between CPU and GPU memory, given that the whole graph is assumed to fit into the CPU memory. To support even larger graphs on a single machine, using the disk (e.g., SSD) as secondary storage is promising. In this case, a new dimension of performance bottleneck will be the I/O inefficiency, which has been studied in prior work [2, 32, 35, 55]. We can combine Scaph

with these past disk-based solutions to cooperatively handle graphs that cannot fit into the host memory.

Performance Profitability. Scaph delivers performance benefits by processing all the subgraphs differentially. Scaph is currently not expected to be applied to graph algorithms where the set of active vertices does not shrink as computation goes on. For example, all vertices in PageRank are active in every iteration. Thus, all the data of a subgraph can be regarded as UD without any PUD. In fact, we can extend Scaph to distinguish these all-active subgraphs further for PageRank by considering not only the degrees and the activation but also the state variation rate for each vertex, which is a potential direction of future work.

7 Related Work

Heterogeneous Graph Systems. Such systems have been studied on a range of heterogeneous architectures equipped with varying hardware resources [21, 35, 44]. Compared to GPU-accelerated solutions [43, 47], FPGA-accelerated alternatives are advantageous in energy-efficiency [10, 61]. In developing Scaph, we focus on improving host-accelerator bandwidth utilization. The basic idea behind can also be applied to improve the scalability of FPGA-accelerated heterogeneous graph systems with a few hardware specializations.

Distributed Graph Systems. The rationale is to aggregate multiple machines to enable processing large-scale graphs. The main challenge lies in obtaining good graph partitions [3, 8, 16, 48, 52] so as to minimize the communication overheads across the machines. Some recent studies take advantage of emerging high-speed networks (e.g., RDMA) to reduce communication overheads [49, 58]. Aspire [54] designs a relaxed consistency model to exploit asynchronous parallelism for iterative algorithms. Gemini [67] includes a series of adaptive runtime optimizations to enable obtaining an attractive scale-out efficiency.

Disk-based Graph Systems. Many disk-based graph systems [12, 45, 64, 65] exist for supporting large-scale graph processing. GraphChi [30] relies on parallel sliding windows to optimize disk accesses. GridGraph [68] uses 2-level hierarchical partitioning to reduce the I/O overhead. TurboGraph [18] applies a pin-and-slide model to exploit the multicore and I/O parallelism. Due to the low disk-to-memory bandwidth, disk-based graph systems are often at least two orders-of-magnitude slower than heterogeneous solutions.

Data Movement Reduction. Several previous studies leverage an analogous idea of running graph partitions multiple times for different purposes. CLIP [2] iterates over each loaded subgraph multiple times to squeeze out the value of each subgraph so that less amount of disk I/O is required. GraphQ [69] enables computing the local subgraphs multiple times in order to tolerate long latency across the compute nodes. Unlike these efforts, Scaph emphasizes on a GPU context that often requires small-size subgraphs to enable fine-grained scheduling. Thus, simply computing a subgraph multiple times is not sufficient to exploit its PUD fully. Scaph enables value exploitation not only within a subgraph but also *across the subgraphs* via a delayed scheduling mechanism.

In LUMOS [53], a subgraph in an iteration can be exploited asynchronously iff its updated values are independent of the subsequent iteration. This dependency-aware technique allows enjoying the efficiency of asynchronous execution while ensuring synchronous processing semantics. Applying this technique into Scaph can help identify the high-value subgraphs that contain across-iteration dependencies, so that Scaph can be extended to handle synchronous algorithms [22] safely by scheduling these high-value subgraphs once. However, the downside is that many dependency-free low-value subgraphs may also be allowed to be computed multiple times, wasting the GPU computational and storage resources.

Wonderland [63] uses graph abstraction as a bridge over on-disk subgraphs to speed up convergence. However, under the context of small-sized subgraphs, such a graph abstraction is often hard to keep concise, and extracting it from the whole graph is also non-trivial. PowerLayer [8] presents differentiated processing on high-degree and low-degree vertices to improve the trade-off between load balance and communication overheads in a distributed setting. However, applying the idea of PowerLayer cannot often identify the value of a subgraph accurately while Scaph does with a fine-grained solution. Mosaic [35] adopts a subgraph compression technique, which can be used to work together with Scaph to improve the bandwidth-efficiency of heterogeneous graph system further.

8 Conclusion

This paper tackles the challenge faced in achieving scale-up large-scale graph processing on a GPU-accelerated heterogeneous architecture. We introduce Scaph, a value-driven heterogeneous graph system that differentially schedules the subgraphs partitioned from a graph according to their values in order to improve the effective utilization of the host-GPU bandwidth. Scaph outperforms state of the art, as evaluated with representative graph algorithms operating on a range of graph datasets. In addition, these performance benefits scale up as more computing resources are available.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. In particular, we thank our shepherd, Xiaosong Ma, for her valuable suggestions. We would also like to thank Pengcheng Yao, Chuangyi Gui, Qinggang Wang, and Jieshao Zhao for their support. This work is supported by the National Key Research and Development Program of China under Grant No. 2018YFB1003502, National Natural Science Foundation of China under Grant No. 61702201, 61825202, 61832006 and 61929103, and Australian Research Council DP180104069. The correspondence of this paper should be addressed to Xiaofei Liao.

References

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117. IEEE, 2015.

[2] Zhiyuan Ai, Mingxing Zhang, and Yongwei Wu. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 125–137, 2017.

[3] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. In *Proceedings of the Hadoop Summit*, volume 11, pages 5–9, 2011.

[4] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Proceedingns of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, 2009.

[5] 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 (PPoPP)*, pages 235–248. ACM, 2017.

[6] Paolo Boldi and Sebastiano Vigna. The webgraph framework I: Compression techniques. In *Proceedings of the 13th International World Wide Web Conference (WWW)*, pages 595–601, Manhattan, USA, 2004. ACM.

[7] 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 (SDM)*, pages 442–446. SIAM, 2004.

[8] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the 10th European Conference on Computer Systems (Eurosys)*, pages 13–21. ACM, 2015.

[9] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[10] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 217–226. ACM, 2017.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. Large-scale graph processing on emerging storage devices. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (USENIX FAST)*, pages 309–316. USENIX, 2019.

[13] Wenfei Fan, Jingbo Xu, Yinghui Wu, Wenyuan Yu, Jiaxin Jiang, Zeyu Zheng, Bohan Zhang, Yang Cao, and Chao Tian. Parallelizing sequential graph computations. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pages 495–510. ACM, 2017.

[14] 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 (PACT)*, pages 345–354. ACM, 2012.

[15] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30. USENIX, 2012.

[16] 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 *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 599–613, 2014.

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

[18] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 77–85. ACM, 2013.

[19] Pawan Harish and Petter J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 2007 International Conference on high-performance computing (HiPC)*, pages 197–208. Springer, 2007.

[20] Lifeng He, Xiwei Ren, Qihang Gao, Xiao Zhao, Bin Yao, and Yuyan Chao. The connected-component labeling problem: A review of state-of-the-art algorithms. *Pattern Recognition*, 70(6):25–43, 2017.

[21] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 78–88. IEEE, 2011.

[22] Unit Kang, Duen Horng "Polo" Chau, and Christos Faloutsos. Inference of beliefs on billion-scale graphs. In *Proceedings of KDD Workshop on Large-scale Data Mining: Theory and Applications (LDMTA)*, pages 1–7, 2010.

[23] Gary J. Katz and Joseph T. Kider. All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 47–55. Eurographics Association, 2008.

[24] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems (Eurosys)*, pages 169–182. ACM, 2013.

[25] 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 (HPDC)*, pages 239–252, 2014.

[26] 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 (SIGMOD)*, pages 447–461. ACM, 2016.

[27] Efstathios Kirkos, Charalambos Spathis, and Yannis Manolopoulos. Data mining techniques for the detection of fraudulent financial statements. *Expert systems with applications*, 32(4):995–1003, 2007.

[28] Yusuke Kozawa, Toshiyuki Amagasa, and Hiroyuki Kitagawa. Gpu-accelerated graph clustering via parallel label propagation. In *Proceedings of ACM Conference on Information and Knowledge Management (CIKM)*, page 567–576, 2017.

[29] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, pages 591–600. ACM, 2010.

[30] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46. USENIX, 2012.

[31] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[32] Hang Liu and Howie H. Huang. Graphene: Fine-grained io management for graph computing. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC)*, pages 285–300. USENIX, 2017.

[33] 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.

[34] 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 *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 195–207, 2017.

[35] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 527–543. ACM, 2017.

[36] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. 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 (SIGMOD)*, pages 135–146. ACM, 2010.

[37] Abdullah A. Mamun and Sanguthevar Rajasekaran. An efficient minimum spanning tree algorithm. In *Proceedings of the IEEE Symposium on Computers and Communication (ISCC)*, pages 1047–1052, 2016.

[38] Christian Mayer, Muhammad Adnan Tariq, Chen Li, and Kurt Rothermel. Graph: Heterogeneity-aware graph

computation with adaptive partitioning. In *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 118–128. IEEE, 2016.

[39] 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 (PPoPP)*, volume 47, pages 117–128. ACM, 2012.

[40] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 456–471. ACM, 2013.

[41] Tesla NVIDIA. P100. *The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU, White paper, NVIDIA*, 2016.

[42] Tesla NVIDIA. V100. *NVIDIA Tesla V100 GPU Architecture Whitepaper, THE WORLD'S MOST ADVANCED DATA CENTER GPU, NVIDIA*, 2017.

[43] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 51, pages 1–19. ACM, 2016.

[44] Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. Graphphi: efficient parallel graph processing on emerging throughput-oriented architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–14. ACM, 2018.

[45] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 410–424. ACM, 2015.

[46] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 472–488. ACM, 2013.

[47] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. Graphreduce: processing large-scale graphs on accelerator-based systems. In *Proceedings of the 27th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 28:1–28:12. ACM, 2015.

[48] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 International Conference on Management of Data (SIGMOD)*, pages 505–516. ACM, 2013.

[49] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 317–332. USENIX, 2016.

[50] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. Graph processing on gpus: A survey. *ACM Computing Surveys*, 50(6), 2018.

[51] Julian Shun and Guy E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 48, pages 135–146. ACM, 2013.

[52] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 425–440. ACM, 2015.

[53] Keval Vora. Lumos: Dependency-driven disk-based graph processing. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, page 429–442, 2019.

[54] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. Aspire: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, page 861–878, 2014.

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

[56] Peng Wang, Kaiyuan Zhang, Rong Chen, Haibo Chen, and Haibing Guan. Replication-based fault-tolerance for large-scale graph processing. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 562–573. IEEE, 2014.

[57] 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 (PPoPP)*, pages 11–21. ACM, 2016.

[58] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. Gram: scaling graph computation to the trillions. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*, pages 408–421. ACM, 2015.

[59] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 15–28. USENIX, 2012.

[60] F. Benjamin Zhan. Three fastest shortest path algorithms on real road networks: Data structures and procedures. *Journal of Geographic Information and Decision Analysis*, 1(1):70–82, 1997.

[61] Jialiang Zhang, Soroosh Khoram, and Jing Li. Boosting the performance of fpga-based graph processor using hybrid memory cube: A case for breadth first search. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 207–216. USENIX, 2017.

[62] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 183–193. ACM, 2015.

[63] 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 (ASPLOS)*, page 608–621, 2018.

[64] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. Cgraph: a correlations-aware approach for efficient concurrent iterative graph processing. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 441–452. USENIX, 2018.

[65] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *Proceedings of 13th USENIX Conference on File and Storage Technologies (USENIX FAST)*, pages 45–58. USENIX, 2015.

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

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

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

[69] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. GraphQ: Scalable pim-based graph processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 712–725, 2019.