Ascetic: Enhancing Cross-Iterations Data Efficiency in Out-of-Memory Graph Processing on GPUs

Ruiqi Tang
tangruiqi@mail.nankai.edu.cn
Nankai University
Tianjin, China

Ziyi Zhao
troppingz@gmail.com
Nankai University
Tianjin, China

Kailun Wang
wangkailun@mail.nankai.edu.cn
Nankai University
Tianjin, China

Xiaoli Gong*
gongxiaoli@nankai.edu.cn
Nankai University
Tianjin, China

Jin Zhang
nkzhangjin@nankai.edu.cn
Nankai University
Tianjin, China

Wenwen Wang
wenwen@cs.uga.edu
University of Georgia
GA, USA

Pen-Chung Yew
yew@umn.edu
University of Minnesota
MN, USA

ABSTRACT

Graph analytics are widely used in real-world applications, and GPUs are major accelerators for such applications. However, as graph sizes become significantly larger than the capacity of GPU memory, the performance can degrade significantly due to the heavy overhead required in moving a large amount of graph data between CPU main memory and GPU memory.

Some existing approaches have tried to exploit data locality and addressed the issues of memory oversubscription on GPUs. However, these approaches have yet to take advantage of the data reuse cross iterations because of the data sizes in most large-graph analytics. In our studies, we have found that in most graph applications the graph traversals exhibit a roughly sequential scan over the graph data with an extremely large memory footprint. Based on the observation, we propose a novel framework, called *Ascetic*, to exploit temporal locality with very long reuse distances.

In *Ascetic*, the GPU memory is divided into a *Static Region* and an *On-demand Region*. The static region can exploit data reuse across iterations. The on-demand region is designed to load the data requested in the iteration of the graph traversal while not found in the static region.

We have implemented a prototype of the *Ascetic* framework and conducted a series of experiments on performance evaluation. The experimental results show that *Ascetic* can significantly reduce the data transfer overhead, and allow more overlapped execution between GPU and CPU, which leads to an average of 2.0x speedup over a state-of-the-art approach.

CCS CONCEPTS

• **Theory of computation → Shared memory algorithms**; Graph algorithms analysis; *Caching and paging algorithms.*

KEYWORDS

Graph Computing, GPU memory oversubscription, Partition-based method, Data Reuse

ACM Reference Format:
Ruiqi Tang, Ziyi Zhao, Kailun Wang, Xiaoli Gong, Jin Zhang, Wenwen Wang, and Pen-Chung Yew. 2021. Ascetic: Enhancing Cross-Iterations Data Efficiency in Out-of-Memory Graph Processing on GPUs . In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3472457

*Corresponding author.

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08…$15.00
https://doi.org/10.1145/3472456.3472457

1 INTRODUCTION

Graph analytics has become one of the major workloads in data centers. As the size of graph data grows dramatically,

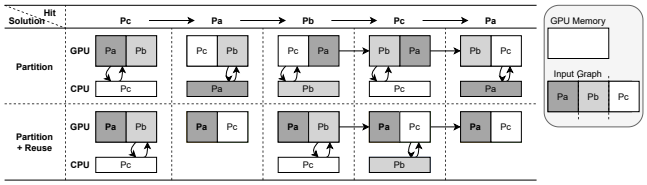


Figure 1: Data swap between CPU and CPU in a partition-based out-of-GPU-memory graph processing scheme

there are more applications that leverage GPUs for such compute- and memory-intensive workloads. However, the limited PCIe bandwidth between CPU and GPUs and the limited internal memory size in GPUs can incur a large data transfer overhead.

Most recent approaches use graph partitioning to fit subsets of the graph data in the internal memory. As shown in Figure 1, the example graph shown at the bottom has been divided into three partitions before being processed: *Pa*, *Pb*, and *Pc*. In most graph analytic applications, graph data are often traversed many times [4], and data partitions are brought into GPUs in some fixed order in each iteration, such as shown at the top: $Pa \rightarrow Pb \rightarrow Pc \rightarrow Pa \rightarrow Pb \rightarrow Pc$.

During the graph traversal, the *percentage of data accessed* and the *temporal locality* are usually quite small. Also, the data in different partitions could be cross-referenced. The accessed data needs to be swapped out in order to accommodate the new data to be brought in from CPU memory. Such data replacement could use the LRU policy[18], as shown in the *Partition* row. As graphs are traversed iteratively, data reuse can occur *across* iterations. *Pa* will be reused in the next iteration. The reuse distance (i.e. the time between the data reuse occurs) will be across 3 partitions. The reuse distance is often too large that the data in *Pa* will have been replaced long before they can be reused. Due to the size of the graph, *Pa*, *Pb*, *Pc* are always swapped out before their next reuse, resulting in data thrashing between CPU and GPU. We run PageRank [16] with *friendster-konect* [9] on a GPU with 11GB GPU memory. It runs for 43 iterations. From the measurements, the data transfer from CPU to GPU is about 1,306GB, which means an average of 30.4GB of data is transferred to GPU per iteration - almost twice the original size of the graph data. This reveals the shortfalls of the current data management in partition-based approaches, i.e. the temporal locality is not properly used.

In this paper, we propose a different data management strategy that aims to exploit the temporal locality across iterations. To avoid data thrashing, we reserve a portion of the GPU memory as *static region* to store graph data intended to be reused across iterations. The *Partition + Reuse* row in Figure 1 shows a simplified example on how it works. We select *Pa* to be kept in the static region and prevent it from being swapped out. Using this simple strategy, we find the data transfer can be reduced to 966GB – a 26% reduction.

Obviously, the reserved static region will reduce the available GPU memory for other computation needed in each iteration, called *on-demand region*, and exacerbate the memory pressure on the GPU internal memory. With a smaller *on-demand region*, we may need to divide the remaining input graph into even smaller partitions which will require more frequent data transfer and may incur more data transfer time. Therefore, to make the proposed scheme work efficiently, we need to address several challenges:

(1) We need to determine the best ratio between the size of the *static region* and that of the *on-demand region*.
(2) We need to reduce GPU wait time when we have to transfer data to the on-demand region from the CPU.
(3) We need to select appropriate data to be kept in the static region and exploit temporal locality as much as possible.
(4) We need to reduce the amount of data transferred into the on-demand region to better utilize its memory space.

Therefore we propose an efficient graph framework, called *Ascetic*. It quantitatively determines the ratio with empirical data. It also tries to overlap as much as possible the CPU data transfer time and the GPU computation time to reduce the GPU wait time. Moreover, it uses an adaptive updating mechanism to enhance the reuse of the static region.

In summary, we have made the following contributions.

(1) We explore the potential of data reuse in the large graph computing workloads that usually oversubscribe the GPU internal memory. We provide a comprehensive analysis on the access patterns of graph analytic applications and addressed the limit of conventional locality-based memory management approaches.
(2) We propose *Ascetic*, a novel graph computing framework. Ascteic exploits temporal locality across iterations by partitioning the GPU memory, and improves the GPU efficiency by maximizing CPU-GPU concurrency, which significantly improves the overall system performance.
(3) We have implemented a prototype of *Ascetic* with CUDA. Comprehensive experiments have been conducted and the results show that Ascetic can achieve average 2.0x speedup over a state-of-the-art graph processing approach.

The rest of this paper is organized as follows. Section 2 provides some background in graph processing and to motivate our proposed approach. Section 3 describes some design and implementation details of our approach. Section 4 presents the experimental setup and evaluation analysis. Section 5

discusses the design issues of Ascetic and Section 6 discusses the related work. Section 7 concludes the paper.

2 BACKGROUND AND MOTIVATION

It has been known to be very challenging to exploit data locality in large graph analytic workloads. In general, the access patterns of graph data shows significant irregularity at the granularity of edges or vertexes. In an experiment, we keep all vertices in GPU memory and edges in UVM, and acquire the edge-access traces using nvprof [15]. Figure 2 shows the access patterns of some common graph processing algorithms such as pagerank (PR), single-source shortest path (SSSP) and connected component (CC) on Friendster-konect dataset. Each data chunk has 4 million edges. The access patterns are acquired by monitoring memory accesses in UVM. It can be seen that all access patterns exhibit a roughly sequential scan of data chunks in each iteration. The access frequency to each data chunk is roughly the same, and only a small fraction of edges are accessed in each iteration, i.e. the access patterns are very sparse in each iteration, and there is no noticeable hot spot. That is because the loaded memory pages (mostly 64KB-2MB) of UVM may contain a large number of inactive edges. Due to the large working set in each iteration, there is an extremely long distance before the data reuse. It means the UVM-based LRU strategy does not work well on graph analytic applications.

There is an opportunity to take advantage of cross-iteration data reuse in graph algorithms and to avoid the repetitive data transfer between CPU and GPU.

2.1 Partitioning-Based Schemes

Graph partitioning is an obvious choice for out-of-memory graph processing. The graph is divided into partitions, each can fit into GPU internal memory. There are many schemes, such as GraphReduce [18] and Graphie [3] that use this approach to allow efficient out-of-memory graph processing on GPUs.

However, due to the sparse accesses in each iteration, lots of data movement is redundant. Table 1 shows the percentage of active edges processed in each iteration using four graph traversal algorithms, bread-first search (BFS), single source shortest path (SSSP), connected component (CC) and pagerank (PR) on the friendster-konect [9] and uk-2007-04 [1] datasets. It shows that the active edges in each iteration only accounts for a very small fraction of the total edges, even below 30% in some cases.

Table 1: Average percentages of active edges per iteration

Dataset	BFS	SSSP	CC	PR
Friendster-konect	4.5%	3.1%	14.1%	28.7%
UK-2007-04	0.8%	3.1%	3.0%	25.1%

Unified Virtual Memory (UVM) was introduced by NVIDIA in 2016 [14]. It provides an easier programming interface to deal with memory oversubscription. A unified memory access framework is provided to include both GPU internal memory and CPU main memory. It allows pages to be migrated from CPU main memory to GPU internal memory transparently with a demand paging mechanism.

Actually, the UVM-based approach can be considered as a fine-grained partitioning scheme of graph data, i.e. each partition is a page. Instead of using explicit swap operations, data to be accessed is migrated to GPU memory on demand.

However, the handling of page faults can cause non-negligible overheads [8].

Also, the page alignment can amplify the memory access irregularity and sparsity, which often makes the situation worse. More detailed discussions are provided in Section 4.4.

2.2 Fine-grained Memory Management

Recently, Gupta et.al [17] proposed a novel scheme to reduce the amount of data transferred between CPU an GPU by more accurately selecting the required data to be transferred.

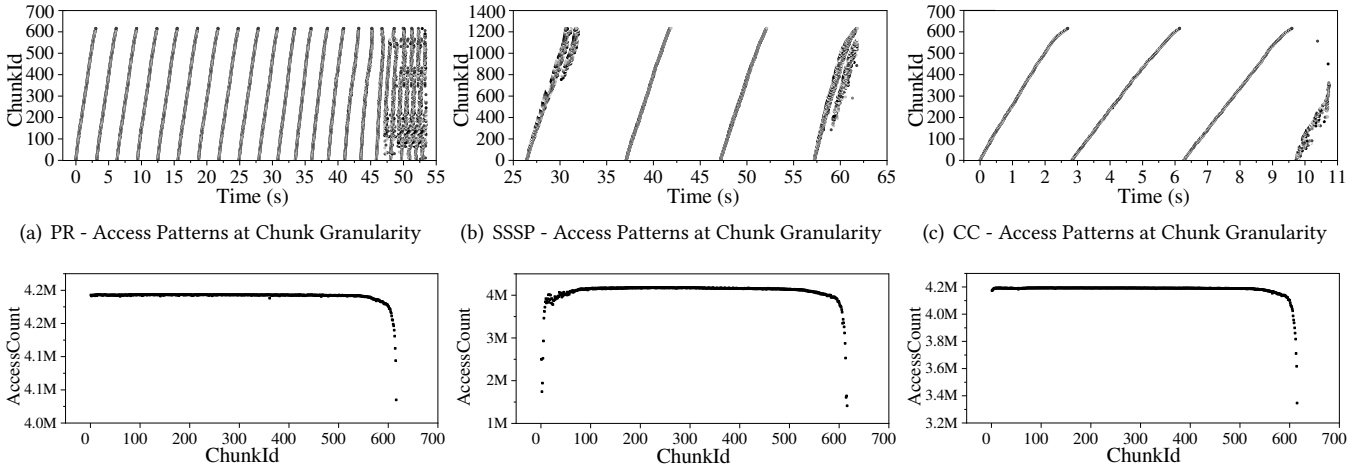
In the proposed scheme, the vertexes are owned by both CPU and GPU while the edges are located in main memory. There are 3 steps in each iteration: (a) GPUs are employed to locate a fine-grained sub-graph required for the current iteration, and predetermine the data structures of edges to be loaded; (b) CPU fills the data structure with the edge data in main memory using multiple threads, and then transmits the data to GPU through the PCIe bus; (c) GPU performs the graph processing on the transferred data for this iteration. Using the preprocessing to identify the required data for each iterations, the amount of data to be transferred can be significantly reduced. However, there are still several limitations. Firstly, as these steps are done sequentially, i.e. the CPU and GPU have to wait for each other to complete the previous step. Our study shows that **68%** of GPU time is idle in BFS algorithm on Friendster-konect dataset.

Second, the GPU memory is not fully utilized. As shown in Table 2, we measured the average GPU memory usage in each iteration. It can be seen that in most cases, for a GPU with a memory capacity of 8-16 GB, only a very small portion of GPU memory is utilized in each iteration, i.e. GPU memory is significantly underutilized.

Lastly, it does not exploit potential temporal locality, i.e. data reuse, and throw away the data after each iteration.

Table 2: Average memory usage per iteration

Dataset	BFS	SSSP	CC	PR
Friendster-konect	0.45GB	0.64GB	1.64GB	2.97GB
UK-2007-04	0.11GB	0.94GB	0.46GB	3.80GB



(a) PR - Access Patterns at Chunk Granularity (b) SSSP - Access Patterns at Chunk Granularity (c) CC - Access Patterns at Chunk Granularity

(d) PR - Access Count of Chunks in one Iteration (e) SSSP - Access Count of Chunks in one Iteration (f) CC - Access Count of Chunks in one Iteration

Figure 2: Access patterns of different graph processing algorithms at the data-chunk granularity

3 DESIGN OF *ASCETIC*

Based on the above observations, we propose a partitioning framework, called Ascetic. Ascetic tries to improve performance by better utilization of the GPU memory, exploiting cross-iteration data reuse, and to increase the overlap (i.e. parallelism) between CPU and GPU to reduce the wait time. In Ascetic, we partition the GPU memory into two regions and store the graph data based on their access behavior. More specifically, we try to fill one region with potentially reusable data across iterations and load the needed data that are not in that region on demand. In this section, we introduce the design and the implementation of Ascetic in detail.

3.1 Overview

As shown in Figure 3, there is one region reserved in GPU memory that is dedicated to enhancing data reuse, called **Static Memory Region**. To process data not covered in the Static Region, we dynamically transfer the additional data to the **On-demand Memory Region** from the CPU. The on-demand region consists of the active data not included in the static region but is needed in the current iteration. In addition, Ascetic has two controllers in the workflow. One is the **Manager** running on the GPU side to load and compute data, and issue data requests to the CPU on demand. The other is the **On-demand Engine** running on the CPU side to prepare and organize the data for the on-demand memory region.

There is a bitmap to facilitate the memory access of the static region. In each iteration, the bitmap is looked up to identify where the requested graph data by the GPU Manager are located (noted as pink in Figure 3). For the data located in

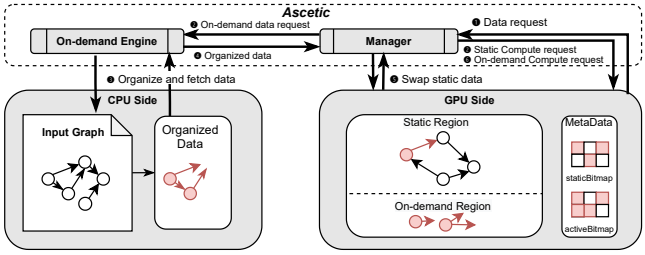


Figure 3: An overview of Ascetic that includes its two major components and detailed dataflow. On-demand Engine runs on CPU to organize and fetch data. On GPU side, the GPU memory is partitioned into Static Region and On-demand Region. Manager handles the data request and data transfer for both regions.

the GPU memory, the data processing can proceed without delay, while the missing data is labeled and sent to the on-demand engine to be organized for data requests to the CPU.

Figure 4 shows the workflow of Ascetic. We choose the push-based vertex-centric programming model because it is widely used in recommender systems, search engines and others. We use a vertex-centric model in the framework and keep all vertices in the GPU memory. The graph is presented in the CSR format. In each iteration, the vertexes to be accessed are marked as *active*. The edges connected to active vertexes are loaded into the GPU memory for processing. In each iteration, we mark the active vertexes on a bitmap *ActiveBitmap*. In Ascetic, there is another bitmap *StaticBitmap* for the static region. It shows if the associated edges are available in Static Region. By comparing the bitmap of *StaticBitmap* and *ActiveBitmap*, we can get the vertex bitmap

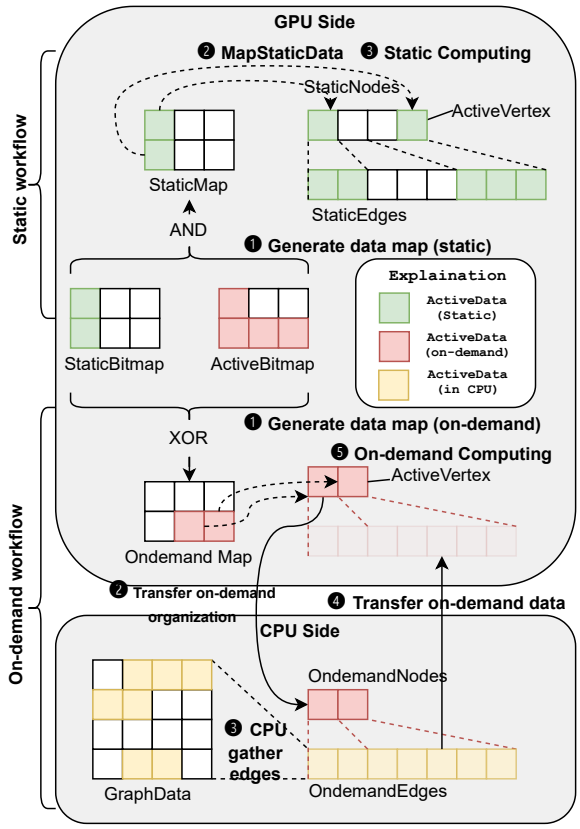


Figure 4: Detailed workflow of Ascetic.

StaticMap located in GPU, and the vertex bitmap *On-demand Map*, which marks the graph data to be transferred from CPU in the current iteration. The array of *StaticNodes* is generated from *StaticMap*, and is sent to Manager for data processing such as graph traversal or weight counting.

At the same time, the *On-demandNodes* is generated based on *On-demand Map*, and processed based on the vertexes information such as *Degree* to generate data requests. Such requests are sent to On-demand Engine, which is similar to the scheme used in Subway [17]. The On-demand Engine gets the requested edges *OndemandEdges* from the CPU main memory *Edgelist*, and send it to On-demand Region. At the end of the current iteration, the active vertexes for the next iteration will be generated based on the connected edges.

3.2 Overlapping Computation and Data Transfer

As presented in Section 2.2, GPU or CPU has to wait for each other to complete the calculation and thus can incur extra overhead. The partitioned memory in Ascetic provides us with an opportunity to overlap the data processing in the

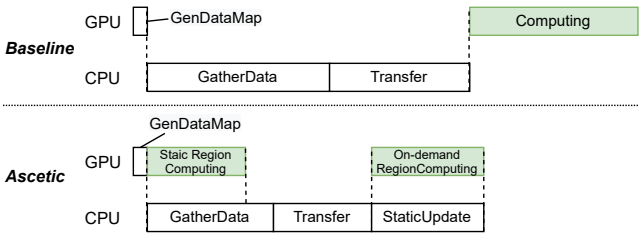


Figure 5: Computation overlap in Ascetic.

Static Region and the On-demand Region. As shown in Figure 5, after generating the data map for the active vertexes (marked as *GenDataMap* in the figure), the **Manager** can start the data processing with the data available in the *Static Region*, while simultaneously the **On-demand Engine** can gather the requested data from CPU and transfer it to the *On-demand Region*. It will significantly reduce the idle time on GPU because the data preparation time can be partially overlapped with the GPU compute time. Also, there is another opportunity in that the *Static Region* can be updated by CPU with data prefetching when the GPU is processing the *On-demand Region*. In this way, Ascetic can further improve the performance by improving the utilization of both hardware resources.

3.3 GPU Memory Partition

Since the static region is designed for data reuse across iterations and to overlap with the data preparation, we may be tempted to make the static memory region as large as possible. However, if the On-demand Region is too small, we have to divided the on-demand data into many smaller fragments in order to fit into the On-demand Region, and then transfer and process them in turn similar to the patterns shown in Figure 1, which may cause a significant performance penalty. Therefore, it is critical to set an appropriate partition ratio between the two regions for better performance.

In Ascetic, we determine the partition ratio empirically, and the ratio can be adjusted before each iteration. Firstly, assume the percentage of the edges processed in an iteration is K, and these edges are distributed more or less evenly across the data sets. Assume M is the size of the GPU memory, and M_{static} is reserved for the *Static Region*. If the size of the dataset is D, the data to be loaded into the *On-demand Region* should be $(D - M_{static}) \times K$ on average in each iteration. To avoid further data partitioning to fit the *On-demand Region*, we should have:

$$(D - M_{static}) \times K + M_{static} \leq M \quad (1)$$

Assume R is the portion allocated to the *Static Region*, i.e. $R = M_{static}/M$. To maximize the *Static Region*, we should have:

$$R = (1 - K \times D/M)/(1 - K). \quad (2)$$

As shown in Table 1, the percentage of active edges in the data set in each iteration is mostly around 10%, except PR. In our implementation, we choose 10% as the default value of K.

Considering the irregular data access in graph computing, we may want to adjust the value of R after we generate the data map as shown in Figure 4. We define V_{static} as the amount of data to be accessed in *Static Region*; $V_{ondemand}$ as the amount of data to be loaded into *On-demand Region*; V as all of the data to be accessed in one iteration. If $V_{ondemand}$ exceeds the capacity of *On-demand Region* and $V_{static}/M_{static} < 0.5 \times V/D$, which means the percentage of the accessed data has significantly exceeded K, while the *Static Region* is under utilized, we should expand the *On-demand Region* and reduce the *Static Region* by

$$M_{static} \times V/D \tag{3}$$

After such re-partitioning, we need to update the corresponding bitmap and re-generate the data map accordingly.

3.4 Data Replacement in Static Region

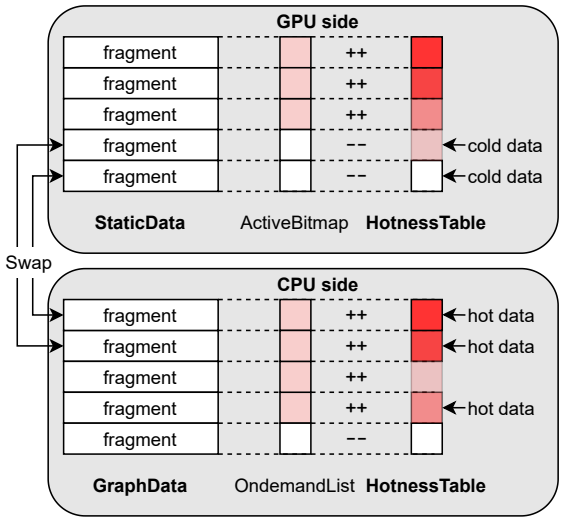


Figure 6: Data replacement mechanism in Ascetic

Despite the fact that the data in the static region can be reused cross iterations, its set of reusable data can change over time [8] and become "stale" after some iterations. To keep the reusable data "fresh" in the static region, we gradually replace the "stale" data in the static region when GPU is processing the data in the *On-demand Region*.

According to the access patterns, we divide the graph dataset into 16KB chunks, which are also amenable to the PCI-e burst transfer mechanism. As shown in Figure 6, for each chunk, a counter is assigned to record the number of accesses in the earlier iterations. If the counter exceeds a threshold, it means the chunk is stale. Different replacement policies can be used based on different graph applications. For example, the counter in BFS can record the number of accesses in all of the past iterations to determine if the chunk is stale. While the counter in PageRank determines the status of chunk by the number of accesses in the last iteration. A server thread in the On-demand Engine handles the data replacement by evicting the stale data chunks in the Static Region, and replace them with new data chunks from the CPU memory. The related meta-data will be updated accordingly after the new data chunks are swapped in.

4 EXPERIMENT AND EVALUATION

4.1 Experimental Setup and Methodologies

Our test platform is equipped with Intel Xeon Silver 4210 10-core CPU, 128GB DRAM, and a NVIDIA Tesla P100 GPU with 56 SMX clusters, 3584 cores and 16GB GDDR5 memory(to calculate more real-world datasets, we limit the GPU memory as 10GB in most environments). The Pascal GPU is enabled with CUDA 11.0 runtime and the 450.29 driver, while the host side is running Ubuntu Linux 18.04 with Linux kernel version 4.15. The source codes are compiled with -O3 optimization. We use $K = 10\%$ to determine the size of the *Static Region*, and no partition adjustment is monitored. Four widely-used graph algorithms are evaluated that include breadth-first search (BFS), connected components (CC), single-source shortest path (SSSP), and page rank (PR).

Table 3: The datasets used in experiments

Abbr.	Name	Vertices	Edges
GS	gsh-2015-host(d) [12]	68.66 M	1.80 B
FK	friendster-konect(u) [9]	68.35 M	2.59 B
FS	friendster-snap(u) [19]	124.83 M	3.61 B
UK	uk-2007-04(d) [12]	106.86 M	3.79 B
RMAT	RMAT-rand(u) [19]	40-100 M	2.5-12 B

As shown in Table 3, we choose four real-world large-scale graphs and one synthesized graph as the datasets in our evaluation. Among them, Friendster-konect and friendster-snap are undirected social network data sets. UK-2007 and gsh-2010 are directed web graph datasets. The size of a dataset is determined by the amount of memory used by all vertices and edges in the graph according to the data type used, as well as the required temporary data buffers and records of vertexes/edges during calculation. The synthesized graphs are generated by RMAT (a widely-used graph generator). Due to the limited size of the CPU memory, we set the maximum number of edges to be 12 billions. Note that the size of the edge data is doubled for SSSP because there is an additional data field for the weight.

For comparison, we also implemented a partition-based graph processing system (PT) [18], a Unified Virtual Memory

(UVM) based system [5], and Subway [17], a state-of-the-art graph processing system. In the partition-based system, the large graphs are partitioned and the active partitions are transferred to GPU memory, which is a common technique used to handle out-of-GPU-memory graph processing. Unified Virtual Memory(UVM) is a hardware- and system-supported scheme to handle GPU memory oversubscription problem. We use CUDA API such as cudaMemAdvise() to optimize the active data migration and exploit the potential data reuse. Subway [17] is a recent effort to address the challenge of out-of-GPU-memory graph processing. We take its open-source release and implement the Pagerank algorithm faithfully. Each program is run 10 times and their arithmetic mean is used in our evaluation.

Table 4: Performance results

The values of Subway and Ascetic are normalized to the data of PT. The highest number among the three methods is in bold.

		PT	Subway	Ascetic
SSSP	GSH	279.9s	9.4X	**15.2X**
	FK	145.2s	7.3X	**10.9X**
	FS	177.9s	6.5X	**8.6X**
	UK	595.411s	16.5X	**23.7X**
PR	GSH	249.1s	1.9X	**2.5X**
	FK	97.9s	1.4X	**3.1X**
	FS	198.3s	2.1X	**2.8X**
	UK	393.6s	2.3X	**4.6X**
CC	GSH	40.5s	2.9X	**17.6X**
	FK	36.4s	1.8X	**6.0X**
	FS	59.4s	3.4X	**5.2X**
	UK	595.4s	16.5X	**23.7X**
BFS	GSH	49.2s	9.9X	**84.7X**
	FK	59.2s	10.6X	**28.0X**
	FS	84.7s	9.9X	**15.2X**
	UK	281.2s	35.3X	**50.2X**
GEOMEAN			5.6X	**11.4X**

4.2 Performance Analysis

Table 4 shows the overall performance results. For simplicity, we have normalized the execution time to the partition-based implementation (PT) [18]. It can be seen that compared to PT, Ascetic can achieve 50.2x speed up for the best case and 11.4x on average. Table 5 summarizes the actual amount of data transferred during the graph processing in each implementation. The results are normalized to the size of the original dataset. Note that they include data transferred during the initial data filling to the *Static Region*. It can be seen that Ascetic can reduce 95.6% of data transferred during graph processing compared to PT.

Figure 7 shows the speedups and the amount of data transferred of Ascetic over Subway. On average, Ascetic achieves 2.0X speedup compared to Subway. The data transfer is not

Table 5: Data transfer results

The values of PT, Subway and Ascetic are normalized to the total data required. The minimum number among the three methods is in bold.

		Size	PT	Subway	Ascetic
SSSP	GSH	13.7G	84.5X	4.2X	**2.3X**
	FK	19.5G	30.0X	2.1X	**1.3X**
	FS	27.4G	23.7X	1.8X	**1.5X**
	UK	28.6G	217.9X	12.1X	**9.8X**
PR	GSH	7.2G	90.0X	15.1X	**1.5X**
	FK	10.1G	45.0X	10.8X	**4.8X**
	FS	14.4G	42.8X	12.4X	**9.1X**
	UK	14.9G	87.3X	22.2X	**15.2X**
CC	GSH	7.0G	22.8X	4.0X	**0.04X**
	FK	9.9G	14.7X	3.0X	**1.0X**
	FS	13.9G	12.4X	2.0X	**1.3X**
	UK	14.5G	15.7X	5.2X	**3.3X**
BFS	GSH	7.0G	27.9X	1.0X	**0.02X**
	FK	9.9G	18.3X	1.0X	**0.3X**
	FS	13.9G	22.5X	1.0X	**0.7X**
	UK	14.5G	10.6X	0.9X	**0.6X**
GEOMEAN			32.5X	3.6X	**1.4X**

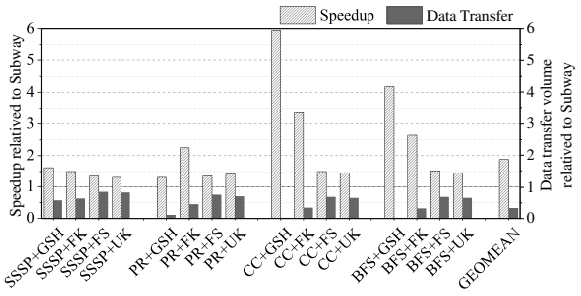


Figure 7: Performance and data transfer comparison with Subway

contain the static prestore data. As the figure shows, the average amount of data transferred in Ascetic is only about 39% of those in Subway. We can see that the reduced amount of data transferred clearly corresponds to the degree of improvement in the overall performance. A more detailed analysis is in Section 4.3.

4.3 Breakdown of the Optimizations

To further understand the details of the performance improvement, we use *Static savings* to represent the benefit from data reuse in the Static Region, and *Overlapping savings* to represent the benefit from overlapping the computation using the available data in the Static Region while filling the data in On-demand Region. We use the results from Subway as the baseline. We explicitly disable overlapping to measure the sole performance impact from *Static savings*, and activate overlapping to measure how much improvement can be

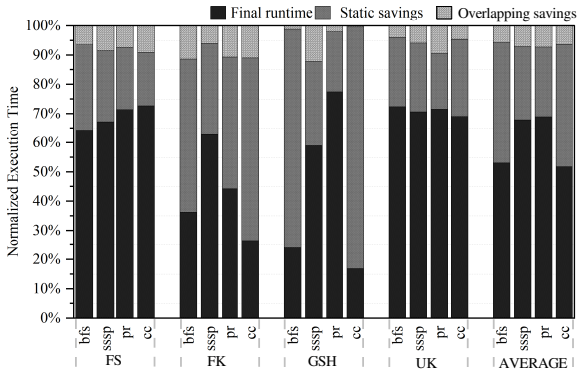


Figure 8: Breakdown of the optimization benefits

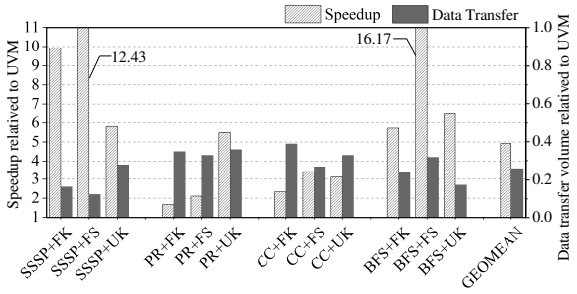


Figure 9: Performance and data transfer comparison with the UVM-based scheme

achieved from overlapping. Figure 8 shows the breakdowns of the performance gain from each component.

The results show that, on average, 37% of the execution time improvement is from *Static savings*. The execution of CC on GS dataset achieves 82.7% improvement. The amount of data processed by the On-demand Engine in CC is also reduced, which leads to another optimization. More specifically, the data transfer in PR on FK dataset was reduced from 109.038GB to 48.070GB by Static savings, and the time of data transfer dropped from 70.709s to 31.447s. At the same time, the CPU spends on filling up the On-demand Region decreases as the on-demand data decreases. However, there is basically no data reuse in the Static Region in BFS. But as the Figure 8 shows, Static saving still gives 6.5% average execution time reduction for BFS. This is because the data located in the *Static region* can be processed directly without waiting for data transfer from the CPU. Note that we have included the initial data filling for the Static Region. In practice, the Static Region can be reused throughout the graph processing and benefits the reduction in data transfer

Furthermore, Overlapping savings contribute about 10% execution time reduction on average. The overhead of data preparation on the CPU is not negligible. For example, the data preparation time on CPU in CC using FK dataset was 3.417s after *Static savings*, which is 40.79% of the total execution time, while *Overlapping savings* can hide 3.115s after the overlapping mechanism is enabled.

4.4 Comparison with UVM-based scheme

Figure 9 shows the improvement of Ascetic over UVM. UVM is a straightforward way to deal with memory oversubscription and to exploit data locality. However, the UVM-based scheme is 6.2x slower than Ascetic.

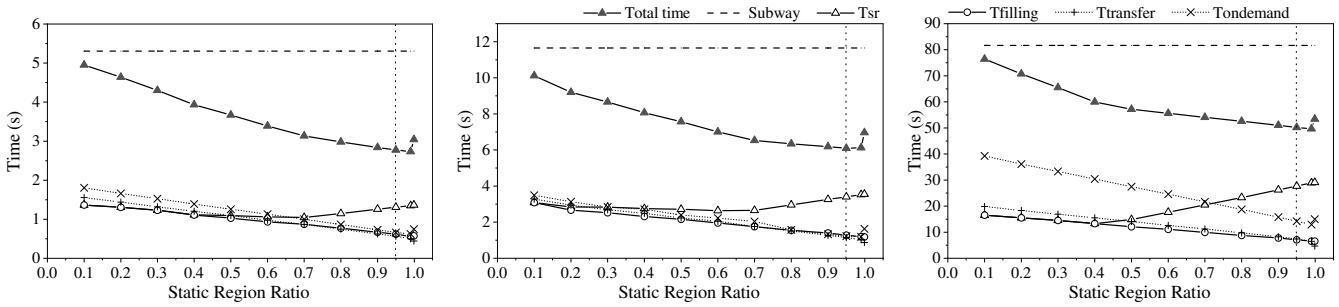
A more detailed analysis shows that there are several limitations when we apply UVM. First, the small page-level data migration increases the frequency of the data transfer between CPU and GPUs. The situation gets worse when we consider the irregularity and sparsity mentioned in Section 2 that can hurt the benefit of the spatial locality inherited in demand paging. Second, the long data reuse distance hurts the LRU-based replacement policy used in UVM. Based on the near-sequential access patterns of the graph processing, it is nontrivial to exploit the temporal locality with the automatic page migration in UVM. Third, page-fault handling can incur high overheads.[8].

5 DISCUSSION

We have conducted a serial of experiments on BFS/CC/PageRank on the FK dataset by changing the size of *Static Region* and show the results in Figure 10. As can be seen, these algorithms share a similar optimal value of around 95% of the total GPU memory. Our selected value for the size of the Static Region (marked as the dot-line in the figure) is quite close to the optimal.

From the observation in Section 2, the capacity of *Static Region* compared to the size of the dataset is critical to the data hit rate. We also show the execution time of different components in Ascetic in Figure 10. They include processing time in *Static Region* (marked as T_{sr}), data preparation and filling time for *On-demand Region* (marked as $T_{filling}$), *On-demand Region* data transfer time (marked as $T_{transfer}$), and graph processing time in *On-demand Region* (marked as $T_{ondemand}$). Just as shown in Figure 10, the larger the size of *Static Region*, the more time Ascetic spends in GPU computing in *Static Region*, which can be exploited for computation overlap. In addition, the time for On-demand transfer can be saved.

Figure 11 (on the left of dotted line) shows the evaluation of various *Static Region* sizes. We use the data set Friendster (15GB) for GPU memory with sizes ranging between 5GB and 13GB to evaluate the performance of Ascetic under different GPU memory sizes. The benefit from data reuse decreases as the available GPU memory is reduced. However, it can be seen that compared to Subway there is still a performance



(a) BFS (b) CC (c) PageRank

Figure 10: The impact of Static Region ratio on the execution time of different algorithms (BFS/CC/PageRank) using the FK dataset. The horizontal dashed line is the performance of Subway [17], and the vertical dot line is the ratio selected in Ascetic based on Equation (2)

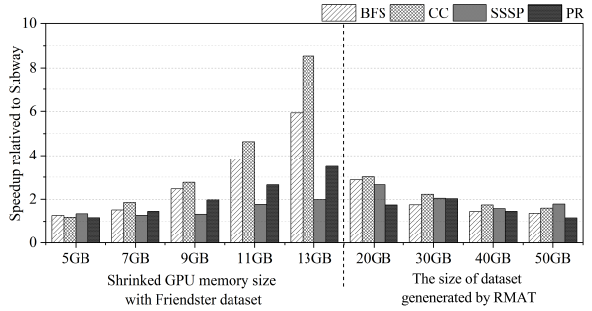


Figure 11: The performance comparison with Subway with different GPU memory sizes and datasets.

improvement of 24.6% even when the size of the available GPU memory is the only 35% of the size of the dataset.

We also use RMAT [6] to generate larger datasets and evaluate different algorithms taking the data bandwidth between CPU and GPU into consideration. As shown on the right of Figure 11, when the size of the dataset increases, the benefit from data reuse is reduced. However, even if we reserve only 10GB of GPU memory for the Static Region, which is roughly 20% of the input dataset size, Ascetic still achieves 1.5X speedup as compared to Subway. In general, Ascetic has a better performance when large datasets are used. Due to the increase in dataset size and the limited bandwidth between CPU and GPU, the data transfer time will be a larger portion of the total time.

Based on the observation that most graph analytic applications exhibit near sequential and evenly access of data chunks in the dataset, our conjecture is that the Static Region can be randomly filled with data chunks across the graph dataset. We have conducted a serial of experiments by filling up the Static Region with the front portion, the rear portion, and randomly selected data chunks in the dataset. In our measurement, the initial dataset in Static Region has negligible impact on the performance (less than 5%), which validates our conjecture.

The replacement of dataset in Static Region does not significantly improve the performance because the time left for **On-demand Engine** to update the *Static Region* is quite limited. Based on our measurements, only 28.40% of time is spent in the *On-demand Region*, and only about 2% of the total data transfer can be completed during that time.

6 RELATED WORKS

A number of graph processing frameworks have emerged in this era of big data, including Graspan [20] and Wonderland [22] for single-machine environment, and Pregel [13], GraphLab [12] for clustered environments. As the graph data exceeds the size of the main memory, out-of-core graph processing systems such as Graphchi [10] and Grid-Graph [23] have been proposed that partition the graph datasets into smaller sub-graphs and load them from large data storage devices. Similar to the CPU-based graph processing frameworks, there are many graph-processing systems based on GPUs, such as Cusha [7], Enterprise [11] and GunRock [21]. However, the GPU memory is also a limited resource for the large graph datasets. To address such GPU memory constraints, a lot of research has been conducted recently [2]. There are two major challenges that need to be addressed. The first challenge is that the partition-based approaches can have significant overheads due to redundant data transfer. The second challenge is that the data locality is difficult to exploit in graph analytic applications. Compare to other works, Ascetic exploits data reuse (i.e. temporal locality) across iterations using a *Static Region*, and load the data accurately in the *On-demand Region* to improve computation overlap.

Subway [17] is a state-of-the-art graph processing system proposed recently. It uses a fine-grained data transfer scheme by accurately organizing subgraph required in each iteration.

In this way, the data that needs to be transferred in each iteration can be significantly reduced. We also exploit such an approach to manage the *On-demand Region* in Ascetic. Compared to Subway, Ascetic further improves the utilization of GPU in two ways. Firstly, it improves GPU memory efficiency using *Static Region*. Secondly, it overlaps the *On-demand Region* data transfer time with the computation work in *Static Region* to hide the data transfer latency.

7 CONCLUSION

Processing large graph datasets that exceed capacity of GPU memory is a fundamental challenge in graph analytic applications. To improve data reuse and to avoid the overhead caused by redundant data transfer, we have analyzed the access patterns of different graph applications and find out that the access patterns exhibit a near sequential scan at the granularity of data chunks. Based on this observation, we propose Ascetic, a GPU-based framework to perform large-scale graph processing. We have implemented the Graph framework, and evaluate its performance with 4 conventional graphs applications running on different datasets. The results show that Ascetic can outperform other partition-based graph processing schemes, and archive an average of 2.0x speed up over a state-of-art graph processing system.

The source code of Ascetic is publicly available at `https://github.com/NKU-EmbeddedSystem/Ascetic`

ACKNOWLEDGMENTS

This work is partially supported by the National Key Research and Development Program of China (2018YFB1003405), the M. G. Michael Award of the Franklin College of Arts and Sciences at the University of Georgia, and a faculty startup funding of the University of Georgia.

REFERENCES

[1] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference*. ACM Press, Manhattan, USA, 595–601.

[2] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. 345–354.

[3] W. Han, D. Mawhirter, B. Wu, and M. Buland. 2017. Graphie: Large-Scale Asynchronous Graph Traversals on Just a GPU. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 233–245. https://doi.org/10.1109/PACT.2017.41

[4] Pawan Harish and P. J. Narayanan. 2007. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing* (Goa, India) *(HiPC'07)*. Springer-Verlag, Berlin, Heidelberg, 197–208.

[5] Mark Harris. 2021. *Unified Memory for CUDA Beginners*. Accessed: 2020-12-31.

[6] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Scalable SIMD-Efficient Graph Processing on GPUs. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT '15)*. 39–50.

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

[8] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. Batch-Aware Unified Memory Management in GPUs for Irregular Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1357–1370.

[9] Jérôme Kunegis. 2019. The koblenz network collection.

[10] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC.. In *OSDI*. USENIX Association, 31–46.

[11] Hang Liu and H Howie Huang. 2015. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[12] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A New Framework For Parallel Machine Learning.. In *UAI*. AUAI Press, 340–349.

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

[14] NVIDIA. 2006. *NVIDIA Tesla P100—The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100*. Accessed: 2020-12-31.

[15] NVIDIA. 2021. Nvprof : A CUDA profiling tool that traps memory access addresses. https://docs.nvidia.com/cuda/profiler-users-guide..

[16] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Stanford InfoLab. Previous number = SIDL-WP-1999-0120.

[17] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: minimizing data transfer during out-of-GPU-memory graph processing.. In *EuroSys*. ACM, 12:1–12:16.

[18] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: processing large-scale graphs on accelerator-based systems. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[19] SNAP. 2021. Stanford network analysis project. https://snap.stanford.edu/..

[20] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 389–404.

[21] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–12.

[22] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A novel abstraction-based out-of-core graph processing system. *ACM SIGPLAN Notices* 53, 2 (2018), 608–621.

[23] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning.. In *USENIX Annual Technical Conference*. USENIX Association, 375–386.