Out of GPU Memory Graph Processing

Ankit Kurani

MTech Project Report

Abstract

Processing large graph data with limited GPU memory is a significant challenge in real-world applications, including social networking websites, Google Maps, and the World Wide Web. To tackle this challenge, several techniques have been proposed, such as the Unified Virtual Memory (UVM) approach, partitioning-based approach, and zero-copy approach. However, comparing these techniques fairly has been hindered by the absence of normalized CUDA kernel functions and inconsistent data structures in previous studies. In this work, we present a comprehensive comparative analysis of these techniques by normalizing the CUDA kernels, integrating a partitioning-based approach into an existing UVM-based framework, and ensuring consistent data structures. We evaluate the performance and behavior of two widely used graph applications, Breath First Search (BFS) and Pagerank, on various graph datasets using both the UVM and partitioning-based approaches. Our experimental results shed light on the strengths and limitations of each technique for GPU-based graph processing

1 INTRODUCTION

In many graph applications, graphs naturally tend to grow as time progresses. To better understand what growth means in the above point, let's take an example of social networking websites. Over time, more people can join social networking websites and connect with others, and this process continues while increasing the number of nodes and edges as time progresses. When the input graph size becomes more than the GPU memory size, it is known as memory oversubscription. Many graph processing applications fail to process the graph dataset, which causes memory oversubscription or there is a dramatic slowdown in processing. Billion node graphs that exceed the memory capacity of standard machines are not well supported by popular Big Data tools like MapReduce, which are notoriously poor performing for the graph algorithms.

1.1 State of the Art

There are three general ways to process out-of-memory graphs in GPUs. The first approach is to use NVIDIA's Unified Virtual Memory (UVM) [5]. UVM brings CPU and GPU memory into a shared address space. The UVM driver migrates pages from CPU memory to GPU's memory on page faults whenever DRAM resident data is accessed from the GPU. UVM allows GPU to access the host memory transparently, with memory pages migrated on demand. By adopting unified memory, graph systems do not have to track the activeness of graph partitions. Instead, the memory pages containing active edges/vertices will be automatically loaded to the GPU memory triggered by the page faults. UVM helps in ease of programming, as the user does not have to explicitly partition the graph or keep track of it due to on-demand page faults. [1]Though on-demand page faulting is not free, there are significant overheads with page fault handling (such as TLB invalidations and page table updates).

The second approach involves partitioning the original input graph dataset into partitions. Partitions are made in such a way that they can completely fit into GPU memory. This technique is referred to as a partitioning-based approach. The oversized graph is first partitioned, then explicitly loaded to the GPU memory in each iteration of the graph processing. The graph is processed partition by partition, one at a time. A major challenge for this approach is the low computation-to-data transfer ratio due to the nature of iterative graph processing. Processing of the next partition of the graph can start once it is loaded onto the GPU's memory after the GPU finishes processing on the previous partition. Some prior work also focuses on overlapping data transfer operations with the GPU kernel execution, but this approach also has limitations of the high data movements between CPU and GPU. [1]Subway is one of the prior works which uses a partitioning-based approach for out-of-memory graph processing.

The third approach, the zero-copy approach, keeps the entire graph dataset on the CPU DRAM, and the GPU accesses the DRAM resident graph at the cache-line granularity. This approach allows GPUs to access the host memory in cache-line granularity directly. With zero-copy, no complicated data migration is needed, and GPUs can fetch data as small as 32 bytes from the host memory. This way of graph processing was first introduced by a prior work named [2]EMOGI. This approach is free of costly GPU page faults, but each access needs to cross PCIe interconnect and can slow the process.

1.2 Contributions

This work focuses on following points:

1. Why is normalizing cuda kernel functions important?

2. What challenges we faced while normalizing the cuda kernels?

3. Performance improvement over original vanilla subway[1] framework.

4. Behaviour of BFS on techniques like UVM and partition-based approaches on different graphs.

5. Analyzing Performance of UVM on BFS

6. Behaviour of Pagerank on techniques like UVM and partition-based approaches on different graphs.

2 Background

This section describes the basics needed for graph analytics, their programming model, the format in which graphs are stored, a few techniques of graph processing, and also a discussion of major issues in GPU-based graph processing.

2.1 Vertex Centric Programming

[11]Considering recent graph applications developments, vertex-centric programming has been widely adopted. It is a new graph programming paradigm that helps in simplicity, high scalability, and strong expressiveness. It was recently incorporated into distributed processing frameworks to address challenges in large graph processing. Billion node graphs that exceed the memory capacity of standard machines are not well supported by popular Big Data tools like MapReduce, which are notoriously poor performing for the graph algorithms like PageRank.

We think like a vertex rather than a graph(Traditionally, we have a global view of the complete graph, but here we have a single vertex point of view). During the processing, the vertex function is evaluated on all the active vertices iteratively until all the vertex values stop changing or iterations have reached the limit. It helps to improve the locality. Graph algorithms can be parallelized because all vertices can execute the vertex program in parallel(in the same iteration) with better fault tolerance, etc.

A vertex-centric algorithm consists of a so-called vertex program. This program is executed at each vertex. In each iteration, the vertex program is executed by the vertices, and the messages are exchanged between vertices. The algorithm terminates when no messages are sent from any vertex, indicating a halt.

2.2 Compressed Sparse Row

[1]Compressed Sparse Row(CSR) format is a well know standard format to store the graph in a vertex-centric programming model. It is a commonly used graph representation that captures the graph topology with two arrays: vertex array and edge array. As shown in Figure 1, the vertex array is made up of indexes to the edge array for locating the edges of the corresponding vertices

2.3 SubCSR format

SubCSR[1] is a format designed to represent a subset of vertices along with their corresponding edges in a Compressed Sparse Row (CSR)-represented graph. In Figure 2-(b), we illustrate the SubCSR representation for the active vertices and edges of the CSR graph shown in Figure 2-(a). At a high level, the SubCSR format closely resembles the CSR representation with a notable distinction in the structure of the vertex array.

In the CSR format, the vertex array directly indicates the positions of all vertices in the original graph. However, in SubCSR, the vertex array is replaced with two arrays: the subgraph vertex array and the offset array. The subgraph vertex array denotes the positions of active vertices from the original vertex array, representing the subset of vertices being considered. On the other hand, the offset array identifies the starting positions of the edges associated with these active vertices in the subgraph edge array.
Overall, the SubCSR format presents a refined approach for representing a subset of vertices along with their edges within a CSR-represented graph. Through its adaptation of the vertex array into the subgraph vertex array and offset array, SubCSR offers a more efficient and concise representation, enhancing the performance of graph algorithms that operate on specific subsets of vertices and their associated edges.

2.4 Unified Virtual Memory

The real-world graph can be of large order magnitude. The easiest way to enable GPU-based graph traversal on such graphs can be to use [5]UVM. It provides a single memory address space accessible by both CPU and GPU through a page faulting mechanism. It reduces the burden from the shoulders of the programmer as they do not have to manage where the data resides explicitly, so they do not have to explicitly manage the data movements between CPU and GPU. The memory pages containing the requested data are automatically migrated to the memory of the requesting processor (either CPU or GPU), known as on-demand data migration.
When allocating memory for the input graph, use a new API, cudaMallocManaged(), instead of the default malloc(). In this way, when the graph is accessed by GPU threads and the data is not in the GPU memory, a page fault is triggered, and the memory page containing the data (active edges) is migrated to the GPU memory. But page fault handling adds substantial overheads, which reduces the benefits of on-demand page faulting. Also, it suffers from where loaded memory pages may contain a large ratio of inactive edges.

2.5 Paritioning Based apporach

[1]In this approach, the oversized graph is partitioned first such that each partition fits into the GPU memory, then loads the graph partitions into the GPU memory in a round-robin fashion during the processing. This technique also has some problems, like low computation to data transfer ratio. Most of the time is spent transferring data rather than computing operations. Also, programmers have to keep track of which partition to move to GPU each time and explicitly handle it.

2.6 Zero-Copy

[2]Another approach to processing graphs is the zero-copy approach. It maps the pinned memory to the GPU address space, allowing GPU programs to access to the host memory directly. Basically, it allows GPUs to access the host memory in cache line granularity. With zero-copy, no complicated data migration is needed, and GPU can fetch data as small as 32 bytes from the host memory. To allow GPU threads access to the external memory in smaller granularity than UVM[5], GPUs support marking memory address ranges as zero-copy memory. GPU threads can access the zero-copy memory as if it was GPU global memory, and the GPU transforms memory requests from the threads to memory requests over an external interconnect like PCIe. In this approach entire graph resides on the CPU DRAM, and the GPU accesses the DRAM-resident graph at cache granularity over the PCIe interconnect.

3 Current Contributions

3.1 Normalizing Kernel

Normalizing the kernel refers to the process of integrating Subway's partition-based approach into the existing Rapids codebase. It is crucial to normalize kernels in order to perform fair comparisons between different techniques for graph applications. In many research work they directly compare the execution time without normalizing kernel and checking for the correctness of the results. This lack of normalization can lead to biased results and make it challenging to evaluate the true performance differences between techniques. By keeping the Cuda kernel consistent, the focus can be solely on comparing the impact of different techniques on graph processing.

Another aspect that needs to be taken into account is the usage of different data structures in different research works. Various graph processing frameworks may employ different data structures for representing graphs, which can affect performance. To achieve fairness, it is important to normalize the data structures used across different techniques and ensure that

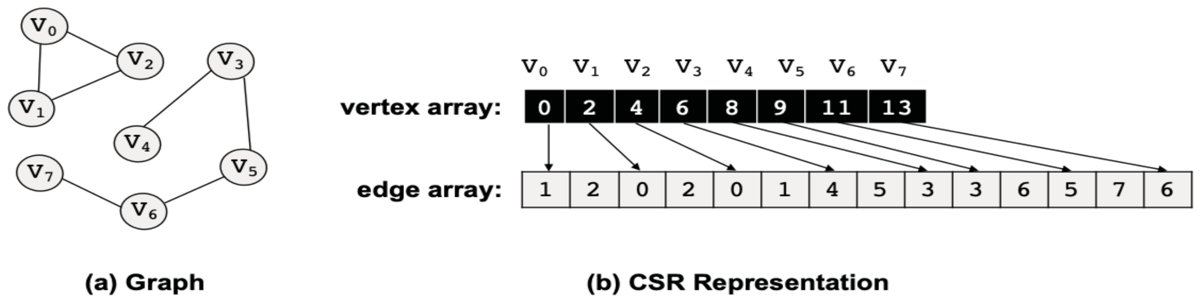

Figure 1: [1]CSR Format

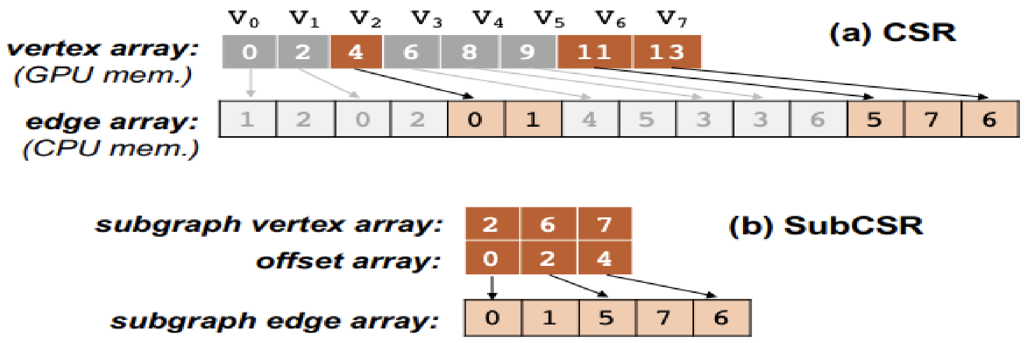

Figure 2: [1]SubCSR Format

the comparison is based solely on the impact of the techniques themselves.

We have two code repos with us; the first is the Rapids[4] framework with UVM implementation, and the other is the Subway[1] partitioning-based framework.

To address the challenges mentioned above and promote fair comparisons, the Subway[1] partitioning-based approach was integrated into the existing Rapids[4] codebase, which originally focused on UVM-based graph processing. This integration allowed for a direct comparison between the UVM approach and the partitioning-based approach, using a normalized CUDA kernel and consistent data structures.

By normalizing the Cuda kernel, integrating the Subway[1] partition code, and ensuring consistency in data structures, we aim to provide a fair comparison of the performance and behavior of different techniques, such as UVM[5] and partitioning, for graph applications like BFS. This approach will enable a more accurate assessment of the strengths and limitations of each technique and facilitate informed decision-making for GPU-based graph processing.

Note: This required significant amount of engineering effort

3.2 Challenges Faced

We faced many challenges while integrating a partition-based approach in Rapids code base. Following are the challenges one may encounter and must solve while writing a partition-based code.

1. **Multiple Paths possible to a node; considering multiple partitions**

   According to Rapid's implementation, they store a level array to keep track of at what level a node lies as compared to the source node. In Rapids[4], each node's level distance can be updated at most once because it does not write code considering partitions greater than one

   In the partition-based approach, a challenge arose in the BFS implementation within Rapids[4]. Due to the nature of partitioning, certain nodes may not be processed until a later partition is loaded into the GPU memory. This delay in processing can result in incorrect level distances for some nodes.

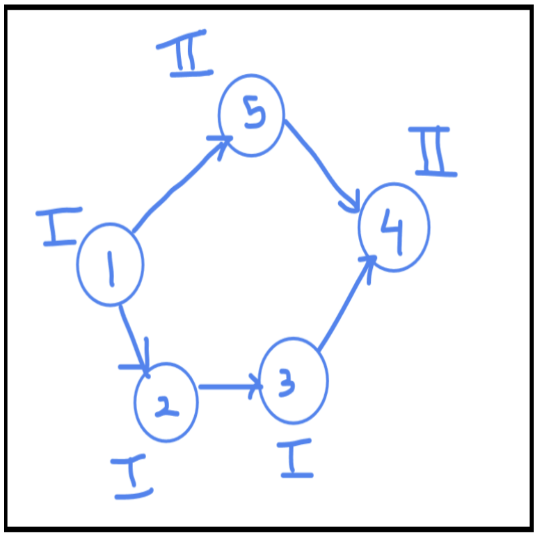

Figure 3: Partition 1 : Node 1, 2, 3 ;
Partition 2: Node 4 and 5

For example in Figure 3, consider node 1 as source node and we want to calculate level of node 4 with respect to source node in the graph. In the first path, the traversal follows the nodes 1, 2, 3, and finally reaches node 4, updating its distance to 3. However, in the second path, the traversal takes a different route through nodes 1, 5, and then reaches node 4. At this point, the correct minimum distance for node 4 should be 2, as it is reachable through a shorter path. However, since the second partition was not loaded at the time of processing node 4, its distance was incorrectly set to 3.

To address this challenge, the implementation in the BFS algorithm within Rapids[4] needed to be modified. Specifically, in the update frontier function, changes were made to allow multiple updates to the same node's value array. This modification enabled the comparison of the visited node's distance with its previous value and ensured that the minimum distance was correctly written to the node's value array.

By making this adjustment, the BFS implementation in Rapids could handle scenarios where the traversal of nodes may occur across multiple partitions. This improvement ensured that each node would receive the correct minimum distance during the graph traversal, even if certain nodes were processed in different partitions.

2. **Multiple Predecessor to update same node**

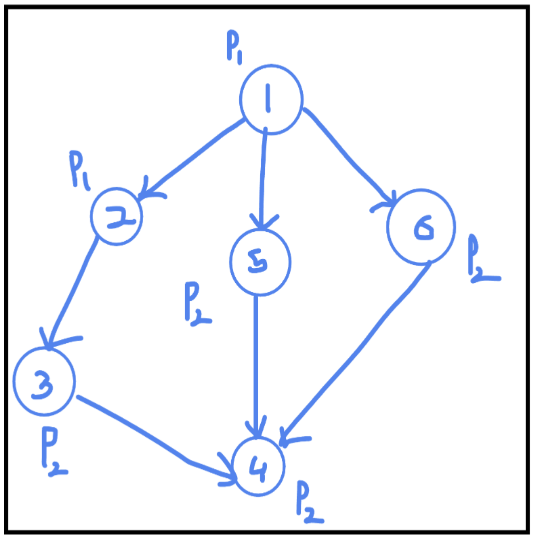

Figure 4: Partition 1 : Node 1, 2 ;
Partition 2: Node 3, 4, 5 and 6

In the graph given shown at figure 4, during the processing of Partition 2, node 4 is reached through three different predecessors: nodes 3, 5, and 6. In the original implementation, only the numerical value of the predecessors was considered, resulting in the selection of the smallest predecessor value (in this case, node 3) to update the distance of node 4. However, this approach led to an incorrect distance for node 4 since its correct distance should be based on the shortest path from the source node.

To address the issue in the Rapids[4] implementation related to handling multiple predecessors in the partition-based approach, a modification was made to consider not only the numerical value of the predecessors but also their respective distance values.

We proposed a solution to store not only the frontiers and their predecessors but also the current distances of the predecessors. For example, the data structure could be:

(a) Frontier 4, with predecessor 5 and distance[5] = 1 .. (**We selected this path**)

(b) Frontier 4, with predecessor 6 and distance[6] = 1

(c) Frontier 4, with predecessor 3 and distance[3] = 2 (**Rapids was selecting this path in case of multiple predecessor to a node**)

By considering the distances of the predecessors, it becomes possible to determine the shortest distance among them. In this case, the last two items in the list can be dropped because the distance values of nodes 5 and 6 are higher than the distance of node 3. This modification ensures that only the frontier with the shortest distance predecessor is selected and used to update the distance of a node.

By incorporating this change, the BFS implementation in Rapids[4] can handle cases where multiple predecessors exist for a node in the partition-based approach. It ensures that the correct shortest distance is considered for each node, resulting in accurate traversal and distance updates during graph processing.

3. **Integrating SubCSR**

   In the partition-based approach of Subway[1], a concept called "SubCSR" format is utilized to store a subgraph. This format was not originally present in the Rapids[4] code base. To make the flow of the partition-based approach compatible with Rapids, the subcsr format was introduced into the Rapids code.

   Introducing the SubCSR format instead of the CSR format in the Rapids code base presented a challenge in determining where the extra indexing should be performed to identify the exact node being processed within the current partition. The Rapids code base is quite extensive, which made this task more complex.

   By carefully studying the code and making the necessary modifications, the integration of the SubCSR format into Rapids was successfully achieved. This enabled accurate indexing and identification of nodes within the current partition during graph processing, ensuring the correct handling of subgraphs and their associated data.

   Overall, while the task of identifying the appropriate locations for extra indexing within the large Rapids code base was challenging, a thorough analysis and understanding of the code allowed for the successful integration of the SUBCSR format and the necessary modifications to support the partition-based approach.

4. **Managing Out of Bound accesses on frontier data structure**

   In the Rapids[4] implementation, a problem arose when traversing nodes from the "current frontier" and creating a "new vertex frontier buffer" to store their neighbors. Since Subway[1] supports graph partitions where some neighbors may be non-resident in the current partition, an issue occurred when Rapids tried to update the distances and predecessors of nodes in the "new vertex frontier buffer" and then updated the "current frontier" to be the same as the "new vertex frontier".

   The problem emerged because when Rapids tried to access the index of the "subVertex" and "edge" arrays for a node that was not in the current partition, an "Array out of Index Error" occurred. This error happened because both the "subVertex" and "edge" arrays were created based on the nodes in the current partition, so accessing them for non-resident nodes led to out-of-bounds access.

   To address this issue, a solution was implemented in which Rapids was allowed to update the predecessors and distances of the non-resident nodes. However, these nodes were then removed from the "new vertex frontier" before updating the "current frontier".

   By implementing this solution, the "Array out of Index Error" in Rapids was resolved. It ensured that only nodes within the current partition were considered for array accesses and updates, preventing any out-of-bounds access errors.

5. **Adding CSR format creation on CPU side**
   In the original implementation of Rapids[4], we observed that the code did not terminate successfully when higher levels of oversubscription (85% or more) were applied. This issue was primarily attributed to thrashing during the creation of the Compressed Sparse Row (CSR) format. The Rapids codebase utilized a combination of CPU and GPU resources for the creation of the CSR format. To address this issue, we made modifications to the code by performing CSR format creation on the CPU side instead of the GPU side. This change allowed us to obtain output even for higher levels of oversubscription. However, it resulted in a tradeoff as the time taken to create the CSR format from the graph dataset on the CPU side was slower compared to the original implementation for lower level

of oversubscriptions. Nonetheless, this modification enabled us to successfully handle larger levels of oversubscription and obtain the desired output.

4 Experiments

Experiments were performed on a machine with NVIDIA GeForce RTX 3090 GPU having PCIe 4.0, CPU Main memory is 126GB. Available GPU memory is 24GB. We also take care that we run graph applications on those datasets that have a memory footprint less than CPU main memory, i.e., completely fit in CPU main memory.

We have made BFS compatible with the UVM[5] and partition-based approach[1].

We have used following graph datasets

1. Twitter Datset (1.47 billion edges, 41 million nodes):
2. New York Taxi Driver Dataset (1.19 billion edges, 266 nodes)
3. SK-UNION 2005 (1.9 billion edges, 51 million nodes)
4. Wikipedia links (en)[13] (437 million edges, 13 million nodes)
5. Wikipedia links (oc)[13] (24 million edges, 96k nodes)
6. Orkut[12] (117 million edges, 30 million nodes)
7. New Synthetic Dataset (1 billion edges, 100k nodes)

4.1 Definition of Oversubscription

Let x = Peak GPU memory needed when there is no oversubscription

Let y = Available GPU memory size

Oversubscription(%)= (1 - y/x) * 100

4.2 Comparison with Vanila Subway

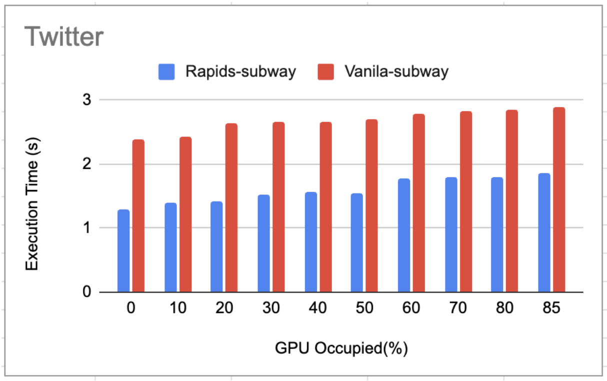

Figure 5

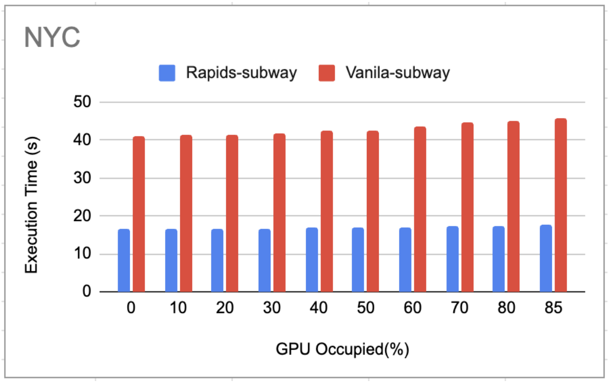

Figure 6

Above plots in figure 5 and 6 shows that integrating the partition-based approach into the Rapids[4] code base yielded promising results in performance improvement. By conducting experiments on real-world datasets such as Twitter and New York taxi driver data, we observed significant speed-ups of 1.7x and 2.5x on these respective datasets when using the partition-based approach compared to the vanilla(original) subway[1] framework. This outcome validates the effectiveness of normalizing the Cuda kernel and establishing a unified framework.

Furthermore, the improved performance of the partition-based approach enables a fair comparison with the Rapids[4] UVM approach. This demonstrates the validity of our decision to normalize the kernel and integrate the partition-based approach, highlighting its potential and advantages within the Rapids framework.

4.3 Behaviour of BFS

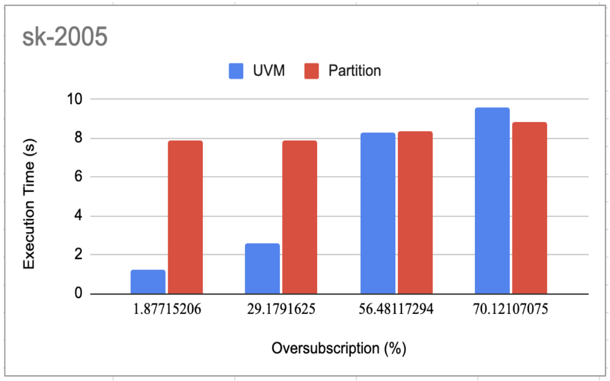

Figure 7: sk-union

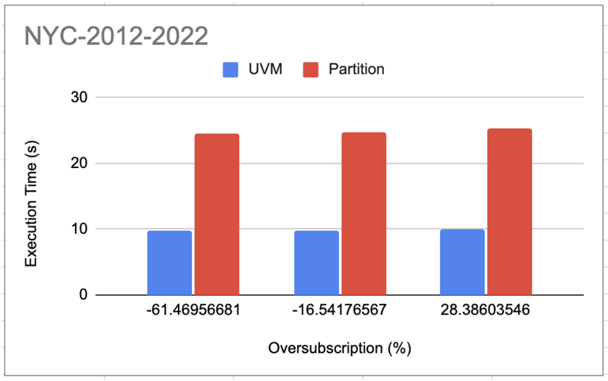

Figure 10: new york taxi driver, negative value of oversubscription indicates amount of GPU memory available is more than required

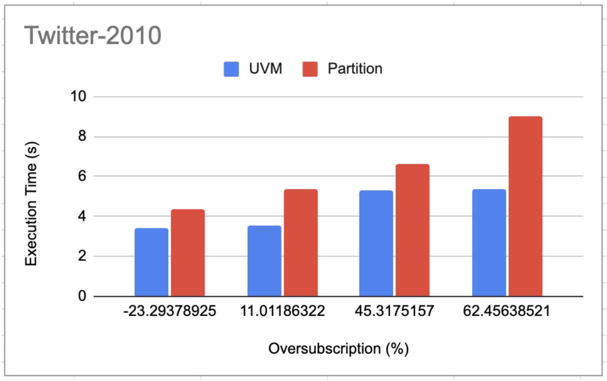

Figure 8: twitter, negative value of oversubscription indicates amount of GPU memory available is more than required

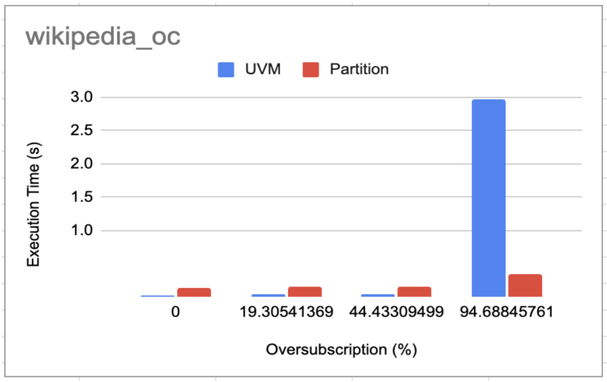

Figure 11: wiki-small

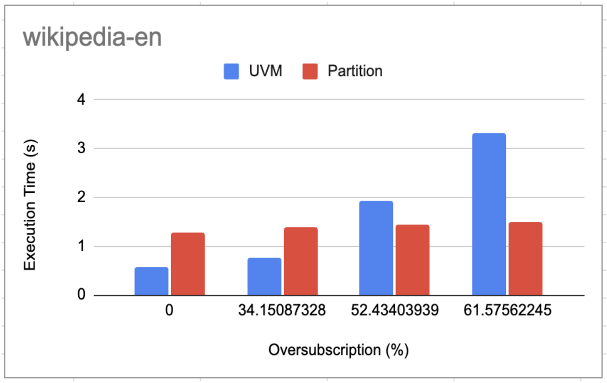

Figure 9: wikipedia-large

The behavior of the Breadth First Search (BFS) algorithm varies depending on the graph datasets used. From Figure 8 and 10, in the case of Twitter and New York Taxi Driver graphs, the Unified Virtual Memory (UVM) approach outperforms the partition-based approach. However, from figure 7, 9 and 11 it can be seen that for graphs such as sk-union, wikipedia-small, and wikipedia-large, the partition-based approach performs better than UVM[5], particularly for larger oversubscription ratios. These results indicate that the choice between UVM and the partition-based approach depends on the specific characteristics of the graph dataset being processed and the level of oversubscription.

4.4 Analysis on Page Faults in UVM on BFS

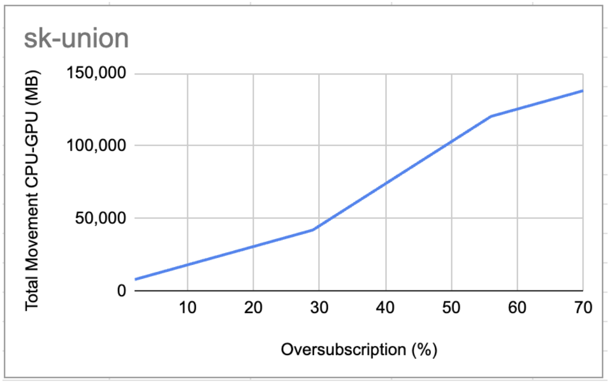

Figure 12: Graph: sk-2005

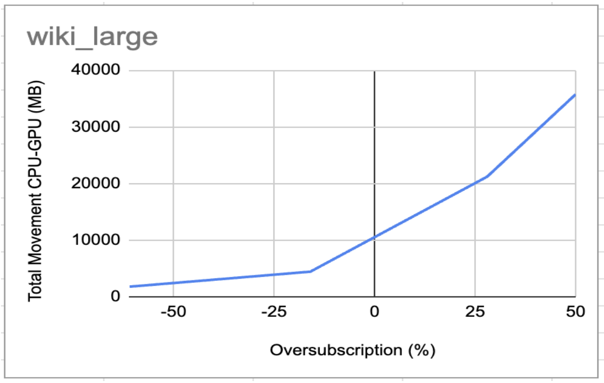

Figure 13: Graph: wikipedia_en

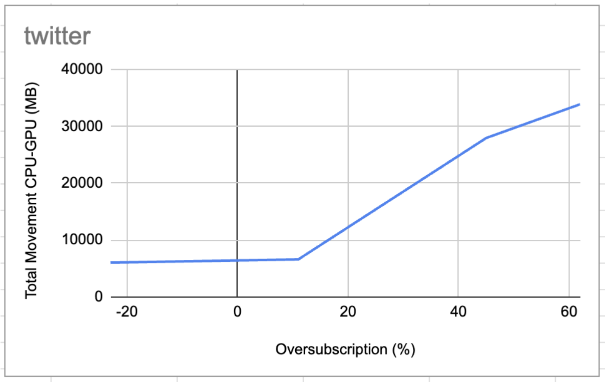

Figure 14: Graph: Twitter-2010

Observations

1. Figure 12, 13 and 14 highlights that the total memory movement between the CPU and GPU increases with higher levels of oversubscription in the UVM approach. The significant amount of data transferred between the CPU and GPU, in the range of gigabytes, indicates a higher occurrence of page faults.

2. Indeed, the increasing execution time with higher levels of oversubscription is closely related to the increased memory movement and page faults in the UVM approach.

4.5 Analysis of Partitions and Iterations in Partition-based on BFS

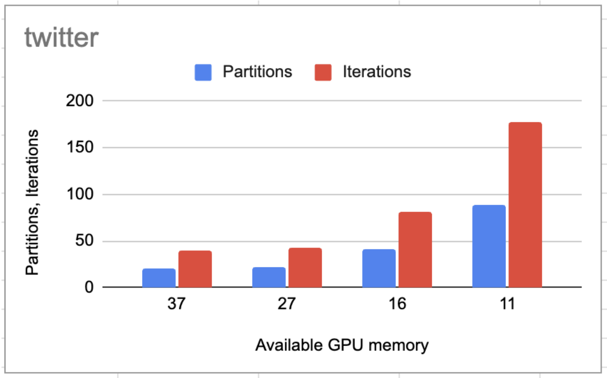

Figure 15: Available GPU memory in percentage on x axis

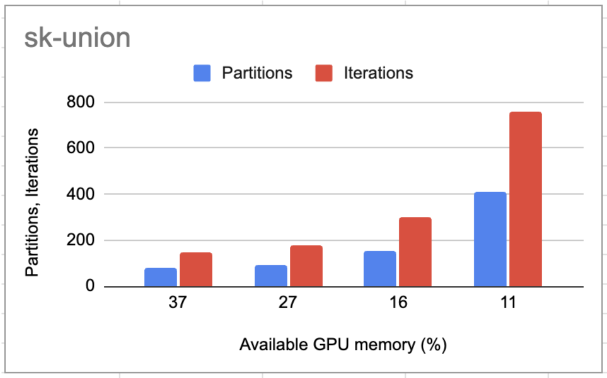

Figure 16

From the above plots of figure 15 and 16 following observations can be made

1. When GPU memory is scarce, the graph data needs to be divided into smaller partitions to fit within the available memory.

2. This necessitates multiple iterations of processing, where each partition handles a subset of the graph data and iterations in some way denote the recomputation of the same partition to reach a convergence point till all the nodes are not finished for processing.

3. As the number of partitions and iterations increases, the overall execution time also increases due to the additional overhead of managing and processing multiple partitions.

4.6 Performance of Unified Virtual Memory (UVM) on Synthetic Graphs

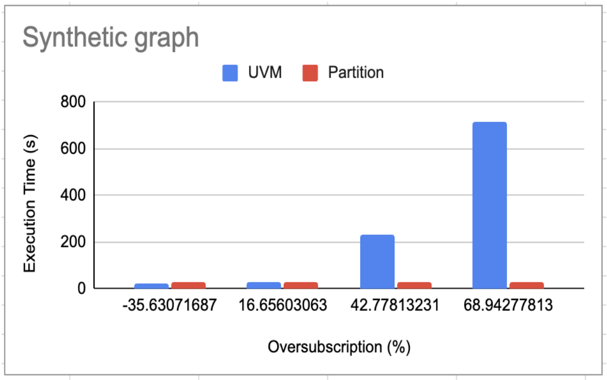

Figure 17: Graph Application: BFS

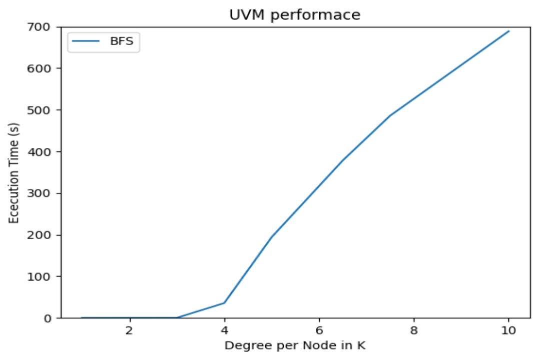

Figure 18: Graph Application: BFS

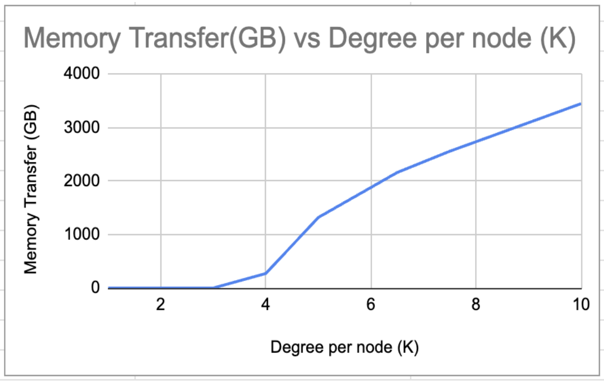

Figure 19

To study the performance of UVM[5] we tried to create a synthetic graphs with various graph properties. A synthetic graph with 100,000 nodes and 1 billion edges was generated, where each node had ten thousand randomly distributed neighbors. This ensured a diverse range of node degrees and access patterns, simulating real-world scenarios encountered in graph processing applications. Following are some observations of this experiment conducted.

The access pattern to neighboring nodes exhibited randomness, with children nodes being distant from their parent nodes in the Compressed Sparse Row (CSR) format. Upon evaluating the initial performance of UVM[5] and the partition-based approach on the synthetic graph, we observed that UVM exhibited poor performance, as it can be observed from figure 17. This performance disparity motivated us to investigate the underlying factors contributing to this discrepancy.

Analyzing the access pattern of neighboring nodes in the synthetic graph, we found that the children nodes were often distant from their parent nodes in the CSR format. This resulted in a higher frequency of page faults during memory transfers between the CPU and GPU. Since GPU page faults are expensive operations, the increased number of page faults negatively impacted the performance of the UVM approach.

To further explore the impact of node degrees on UVM performance, we conducted experiments varying the degrees of nodes in the synthetic graph. Figures 18 and 19 show that as the degree of a node increases, indicating a higher number of edges connected to that node, UVM performance degrades..

These findings emphasize the significance of considering degree and access pattern in graph processing algorithms and highlight the challenges faced by UVM[5] in scenarios with high degrees and random access patterns.

Observation: The average degree of the sk-union, wikipedia-en, and wikipedia-oc graphs was found to be higher compared to the twitter and new york taxi driver datasets. This disparity in average degree could be a contributing factor to the superior performance of the partition-based approach over UVM on large oversubscription ratios. Higher average degrees imply a larger number of edges per node, resulting in more intensive memory operations and potentially increased page faults in the UVM approach. The partition-based approach, which distributes the graph data across multiple partitions, can better handle the higher average degrees by reducing the memory movement and optimizing data locality, leading to improved performance in such scenarios.

4.7 Behaviour of Pagerank

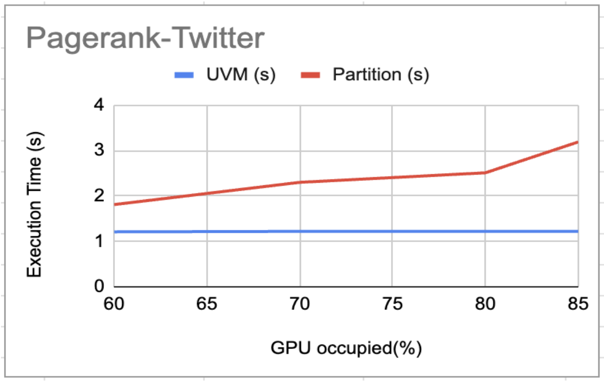

Figure 20

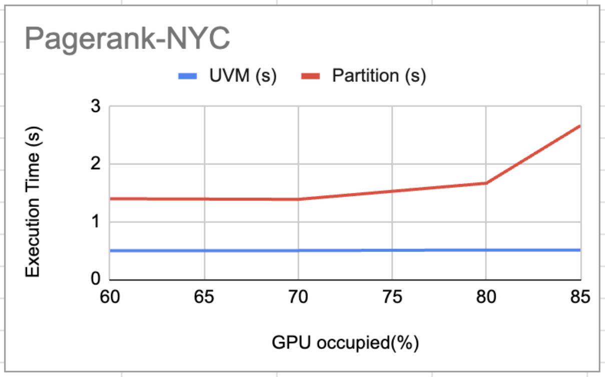

Figure 21

In our integration of the partition-based approach into the rapids[4] code base, we extended its application to the Pagerank graph algorithm. This involved a careful study of the Pagerank algorithm itself and a thorough examination of how rapids had implemented Pagerank using the Unified Virtual Memory (UVM) approach.

Unlike traversal algorithms such as Breadth First Search (BFS) and Single Source Shortest Path (SSSP), the Pagerank algorithm does not heavily rely on internal data structures that need to track the progress of each node at the end of each iteration. Instead, it involves distributing simple probabilities to neighboring nodes after each iteration.

The comparison between the UVM and partition-based approaches, as depicted in Figures 20 and 21, highlights the performance superiority of UVM in the specific context of the graph datasets used in our experiments. This can be attributed to the absence of oversubscription, as the GPU memory usage for these graphs was significantly low, with only 1GB out of the available 24GB being utilized. Consequently, the data transfer between the CPU and GPU was minimal, resulting in improved performance for the Pagerank algorithm.

Considering the primary objective of our work, which is focused on solving the problem of out-of-GPU memory graph processing, algorithms that require less data structures, such as Pagerank, are of lower priority. The rapids[4] code base is optimized to deliver outputs in the shortest possible time, and in this specific scenario, UVM outperforms the partition-based approach due to its efficient utilization of resources.

While our integration of the partition-based approach showcases its effectiveness in improving the performance of graph applications such as BFS and SSSP, it is important to acknowledge that its advantages may not be as pronounced when applied to algorithms like Pagerank, which have minimal GPU memory requirements and limited data transfer between the CPU and GPU. Our focus remains on addressing the challenges of large-scale graph processing with out-of-memory constraints, and for this purpose, the rapids code base, optimized for rapid output generation, currently favors the UVM approach for Pagerank computations.

5 Conclusions

In conclusion, our study focused on comparing the performance of UVM and partition-based approaches in graph processing applications. We began by investigating the impact of normalizing CUDA kernels on fair comparisons between different techniques. The results highlighted the importance of kernel normalization for accurate performance evaluations. We then integrated the Subway[1] partition-based approach into the existing Rapids[4] codebase, addressing various challenges, such as handling non-resident neighbors and implementing the SubCSR format. This integration enabled us to evaluate the behavior of graph applications, specifically Breath First Search (BFS), on different datasets under both UVM and partition-based approaches after normalization.

Our findings demonstrated that the performance of UVM and partition-based approaches varies depending on the graph dataset and the level of oversubscription. On datasets such as Twitter and the New York taxi driver graph, UVM outperformed the partition-based approach. However, on graphs with higher average degrees, such as sk-union, wikipedia-small, and wikipedia-large, the partition-based approach exhibited better performance, particularly at larger oversubscription ratios. This can be attributed to the increased memory movement and potential page faults observed in the UVM approach, indicating the impact of data access patterns influenced by the degree of the nodes in the graph.

Moreover, we conducted experiments on a synthetic graph with varying node degrees, revealing a degradation in UVM performance as the node degree increased. This observation further supported the notion that higher average degrees lead to more random memory access patterns, resulting in increased memory movement between the CPU and GPU and a potential increase in page faults. In contrast, the partition-based approach effectively mitigated these challenges by distributing the graph data across multiple partitions, optimizing data locality, and minimizing memory movement.

Overall, our study highlights the significance of considering graph characteristics, such as average degree, when choosing between UVM and partition-based approaches. While UVM may excel in certain scenarios, the partition-based approach offers better performance on graphs with higher average degrees, demonstrating its ability to manage memory movement efficiently. These insights contribute to the development of optimized graph processing techniques and facilitate informed decision-making for selecting the most suitable approach based on the specific characteristics of the graph dataset and the desired level of oversubscription.

6 Future Work

In future work, one potential direction is to develop a metric that dynamically determines whether the UVM or partition-based approach should be employed for graph processing based on the specific graph properties. This metric could consider factors such as average degree, graph size, and available GPU memory to make an informed decision at runtime. By dynamically selecting the most suitable approach, the overall performance and efficiency of graph processing can be further improved.

Additionally, extending the modified BFS code in the Rapids codebase to support other traversal algorithms, such as Single Source Shortest Path (SSSP), presents an interesting avenue for future exploration. Building upon the existing template, adapting it to handle SSSP would allow for efficient processing of this important graph algorithm and enable comprehensive comparisons between UVM, partition-based, and potentially other techniques.

Furthermore, integrating the Zero Copy technique into the existing approaches could be another area of future investigation. Zero Copy is a state-of-the-art technique that aims to minimize data movement between the CPU and GPU by directly accessing GPU memory from the CPU. By incorporating Zero Copy into UVM and partition-based approaches, its effectiveness and performance in graph processing can be evaluated and compared to existing techniques. This comparative analysis would provide valuable insights into the strengths and limitations of each approach and contribute to the ongoing advancements in efficient graph processing.

Overall, these future research directions aim to further enhance the performance, adaptability, and efficiency of graph processing techniques. By refining the decision-making process, extending the codebase to support additional algorithms, and exploring novel techniques, we can continue to optimize graph processing and enable more efficient analysis of large-scale graph datasets.

7 Related Work

7.1 Tigr

Tigr[10] addresses the challenge of the irregularity of graphs in modern-day graph analytics. As graphs are highly irregular structures, this work tries to show how irregularity can affect the performance of different graph applications. This work does not focus on making partitions of the graph but tries to transform the graph by performing some structural transformation. They do not focus on out-of-memory graph processing, but this work can be combined with the above three mentioned techniques for out-of-memory GPU graph processing to make them more efficient.

7.2 GRUS

Grus[14] is a system framework for GPU graph processing that tackles scalability and efficiency challenges. It addresses the limitations of GPU-dedicated memory by employing a Unified Memory (UM) trimming scheme and a lightweight frontier structure called Bitmap-Directed Frontier (BDF). Grus achieves up to 6.4 times speedup compared to existing frameworks and enables processing large graphs with billions of edges on a single GPU. By prioritizing space efficiency and reducing overhead through smart data access behaviors, Grus offers a competitive solution for efficient graph processing on GPUs.

7.3 HYTGraph

HyTGraph[15] is a GPU-accelerated graph processing framework designed to address the challenges of processing large graphs with limited GPU memory. The key bottleneck in such scenarios is the host-GPU data transfer, and existing frameworks employ different approaches to manage this transfer. However, these approaches often lead to suboptimal performance due to redundant data transfers, CPU compaction overhead, or low bandwidth utilization. In response, HyTGraph proposes a hybrid transfer management approach that combines the strengths of explicit and implicit transfer management at runtime. The framework incorporates effective task scheduling optimizations and achieves significant speedup compared to existing GPU-accelerated graph processing systems, such as Grus, Subway, and EMOGI, demonstrating its effectiveness in improving graph processing performance.

References

[1] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: Minimizing data transfer during out-of- GPU-memory graph processing. In Proceedings of the 15th EuroSys Conference (EUROSYS'20).

[2] S. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W. Hwu. EMOGI: efficient memory-access for out-of-memory graph-traversal in gpus. Proc. VLDB Endow

[3] Shao, Chuanming and Guo, Jinyang and Wang, Pengyu and Wang, Jing and Li, Chao and Guo, Minyi Oversubscribing GPU Unified Virtual Memory: Implications and Suggestions. In ICPE '22

[4] RAPIDS Development Team, RAPIDS: Libraries for End to End GPU Data Science.

[5] Mark Harris, NVIDIA UVM basics: Unified Memory for CUDA Beginners

[6]] CUDA C best practices guide: Chapter 9.1.3. zero copy. https:// docs.nvidia.com/cuda/cuda-c-best-practices-guide.

[7] Li, Chen and Ausavarungnirun, Rachata and Rossbach, Christopher J. and Zhang, Youtao and Mutlu, Onur and Guo, Yang and Yang, Jun. A Framework for Memory Oversubscription Management in Graphics Processing Units. ASPLOS '19

[8] Yangzihao Wang and Yuechao Pan and Andrew Davidson and Yuduo Wu and Carl Yang and Leyuan Wang and Muhammad Osama and Chenshan Yuan and Weitang Liu and Andy T. Riffel and John D. Owens. Gunrock: GPU Graph Analytics. ACM Transactions on Parallel Computing

[9] Xinjian Long, Xiangyang Gong, Huiyang Zhou. An Intelligent Framework for Oversubscription Management in CPU-GPU Unified Memory

[10] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming irregular graphs for GPU-friendly graph processing. In Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).

[11] Siyuan Liu. Vertex-centric graph processing: the what and why

[12] Stanford network analysis project. https://snap.stanford.edu/. Accessed: 2019-06-30.

[13] Jérôme Kunegis. KONECT – The Koblenz Network Collection. In Proc. Int. Conf. on World Wide Web Companion, pages 1343–1350

[14] Wang, Pengyu and Wang, Jing and Li, Chao and Wang, Jianzong and Zhu, Haojin and Guo, Minyi. Grus: Toward Unified-Memory-Efficient High-Performance Graph Processing on GPU. ACM Trans. Archit. Code Optim.

[15] Qiange Wang, Xin Ai, Yanfeng Zhang, Jing Chen, Ge Yu. HyTGraph: GPU-Accelerated Graph Processing with Hybrid Transfer Management.