2023

EXPLORING HIGH PERFORMANCE AND ENERGY EFFICIENT GRAPH PROCESSING ON GPU

Robert P. Watling
Michigan Technological University, rwatling@mtu.edu

EXPLORING HIGH PERFORMANCE AND ENERGY EFFICIENT GRAPH PROCESSING ON GPU

By

Robert P. Watling

A THESIS

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2023

This thesis has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Computer Science.

Department of Computer Science

Thesis Co-advisor:	*Dr. Zhenlin Wang*
Thesis Co-advisor:	*Dr. Junqiao Qiu*
Committee Member:	*Dr. Soner Onder*
Department Chair:	*Dr. Andy Duan*

Dedication

To Dr. Junqiao Qiu,

for his flexibility despite difficult circumstances.

To my mother,

who exemplifies perseverance and curiosity–which guides me in all of my pursuits.

Contents

List of Figures

List of Tables

List of Algorithms

Definitions

graph	Structure that contains a set of vertices and a set of edges
vertex	A point on a graph representing an endpoint of information
edge	A pair of vertices representing a connection
node	Synonymous with vertex
thread	Smallest possible parallel entity for computations
block	Collection of warps on GPU
warp	Collection of 32 threads on GPU
frontier	Set of nodes available for updates in graph processing
TLB	Mechanism that translates addresses

List of Abbreviations

ASYNC	Asynchronous Graph Processing
BFS	Breadth-first search algorithm
CC	Connected components algorithm
CUDA	Compute Unified Device Architecture
DD	Data driven
EEGRAPH	Energy Efficiency Graph
FPGA	Field-Programmable Gate Array
GPU	Graphical Processing Unit
KONECT	The KONECT Project
NVIDIA	Nvidia Corporation
NVML	Nvidia Management Library
PR	PageRank Algorithm
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SNAP	Stanford Network Analysis Project
SSSP	Single Source Shortest Path Algorithm
SSWP	Single Source Widest Path Algorithm
SYNC	Synchronous Graph Processing

TD	Topology Driven
TLB	Translation lookaside buffer
UM	Unified Memory

Abstract

Parallel graph processing is central to analytical computer science applications, and GPUs have proven to be an ideal platform for parallel graph processing. Existing GPU graph processing frameworks present performance improvements but often neglect two issues: the unpredictability of a given input graph and the energy consumption of the graph processing. Our prototype software, EEGraph (Energy Efficiency of Graph processing), is a flexible system consisting of several graph processing algorithms with configurable parameters for vertex update synchronization, vertex activation, and memory management along with a lightweight software-based GPU energy measurement scheme. We observe relationships between different configurations of our software, performance, and GPU energy for processing in-memory and out-of-memory graphs. The ideal parameters are discovered for specific input graphs by analyzing the observed relationships. We also present the utility of subgraph generation to predict the performance and energy consumption of complete graph configurations. EEGraph improves upon state-of-the-art GPU-based graph processing software by 2.08 times for performance and 1.60 times for GPU energy for processing in-memory graph datasets. Additionally, EEGraph improves upon the state-of-the-art by 3.30 times for performance and 1.63 times for GPU energy for processing large out-of-memory graph datasets.

Chapter 1

Introduction

Graphs and graph processing algorithms are fundamental to computer science and are commonly used in related fields to deliver deep contextual knowledge in analysis applications. These applications include compiler design, computer security, artificial intelligence, bioinformatics, navigation systems, and more. Thus parallel graph processing softwares have been subsequently developed for CPU and GPU platforms to provide more performance-sensitive solutions for these applications. Recently, GPU-based graph processing implementations have proven advantageous due to the GPU's massive parallelism, high memory bandwidth, and energy efficiency. Energy efficiency has been increasingly focused on in computer systems research to reduce computer systems' economic and environmental impacts. However, despite their popularity and effectiveness, there is limited research on the energy efficiency of parallel graph

processing algorithms for GPU-based systems. Therefore we present our prototype software EEGraph (Energy Efficiency of Graph processing), an extension of state-of-the-art GPU graph processing frameworks with configurable parameters for performance and energy efficiency.

EEGraph can perform five graph processing algorithms with combinations of two critical processing parameters and three memory options. EEGraph has GPU-based parallel implementations of Breadth First Search (BFS), Connected Components (CC), PageRank (PR), Single Source Shortest Path (SSSP), and Single Source Widest Path (SSWP). The fundamentals of graphs, the graph processing algorithms, and graph processing algorithms on GPUs are presented in Chapter 2.

The specific implementation of EEGraph is presented in Chapter 3. We present the software system used to develop EEGraph. In Chapter 3, we also discuss the software that inspires EEGraph. We also motivate our intuition for out-of-memory graph processing using unified memory and our choice of vertex-centric graph processing for full graphs and subgraphs. Lastly, we conceptualize EEGraph's specific measurement collections for execution time and GPU energy.

In Chapter 4, we evaluate both in-memory and out-of-memory graph datasets. We discuss the relationships between the various options for EEGraph and compare them

to the state-of-the-art graph processing software Subway [29]. The averages of several vertex-induced subgraphs are then evaluated for each dataset, suggesting vertex-induced subgraphs are a viable option for profiling graph algorithms and other tasks.

We then discuss similar works and outline EEGraph's contribution in Chapters 5 and 6, respectively. EEGraph is derived from many existing parallel graph processing approaches in computer systems. Chapter 5 discusses those existing approaches, including works on graph processing performance, heterogeneous graph processing performance, and energy efficiency on GPUs. Finally, we conclude that our software, EEGraph, provides significant performance and GPU Energy efficiency benefits for graph processing on GPUs.

EEGraph improves on the state-of-the-art graph processing software for both in-memory graph processing and out-of-memory graph processing. Our software provides configurations for vertex update synchronization, vertex activation, and memory resource selection. To our knowledge, EEGraph has one of the first lightweight software-based GPU energy measurement schemes. Additionally, we observe the viability of vertex-induced subgraphs for profiling graph processing software. Ultimately, EEGraph significantly improves upon the state-of-the-art graph processing software, Subway [29]. On average, EEGraph explicit data transfer implementation has a mean speedup of 2.08 for in-memory graphs and an energy improvement of 1.60 over unified memory and Subway implementations for in-memory graph datasets. EEGraph

improves upon Subway using a unified memory implementation for processing out-of-memory graph datasets with a speedup of 3.30 and an energy improvement of 1.63 on our system. Therefore, EEGraph is a flexible software for graph processing both in-memory and out-of-memory graphs on GPUs.

Chapter 2

Background

2.1 Graph Basics

A graph $G = (V, E)$ is a set of V vertices and a set E of pairs of vertices called edges [13]. The vertices represent points of information, and the edges are connections between those points. We will use the term node and vertex interchangeably. An edge can be directed (one-way) or undirected (two-way). Then for two vertices x and y, we say x is adjacent to y if an edge exists between them. This edge is *incident* to both x and y. The degree of a vertex is the number of edges incident to it. We explore both directed and undirected graphs of various sizes and average degrees in this work.

We also discuss more intermediate graph theory concepts: the diameter of graphs and induced subgraphs of a graph. To define the diameter, we first need to define a walk. A walk is a sequence of visited nodes along edges in a graph. A path is a walk of distinct vertices of a graph, and the distance between two distinct vertices is the shortest path between them. Ultimately, the diameter of a graph is the greatest distance between any pair of vertices in the graph [13]. The diameter of an input graph in our scenario helps describe the graph's connectivity.

A subgraph is a set $G' = (V', E') \subseteq G = (V, E)$ for some graph G such that $V' \subseteq V$ and $E' \subseteq E$. An edge-induced subgraph is a selected subset of edges and all nodes of each of those edges. A vertex-induced subgraph is a selected subset of vertices and every edge between a pair of selected vertices. Vertex-centric graph processing is popular in research, so it follows that we explore vertex-induced subgraphs in this work. Therefore we generate a vertex-induced subgraph of an input graph in our work. Ideally, a well-generated vertex-induced subgraph will exhibit the same characteristics as the whole input graph but require less processing. A quality vertex-induced subgraph would help select the best configuration of critical parameters while still being performance sensitive in an application setting.

2.2 Graph Processing

2.2.1 Graph Processing Algorithms

Several graph algorithms condense, search, or describe information represented by graphs. We can classify graph processing algorithms into eight general categories: graph traversal algorithms, graph analysis algorithms, components, communities, centrality, pattern matching, graph anonymization, and other operations [8]. We employ graph traversal algorithms, graph component calculation, and centrality measures. The graph traversal algorithms that we discuss are breadth-first search (BFS), single-source shortest path (SSSP), and single-source widest path (SSWP). The component calculation algorithm is a traditional connected components algorithm (CC) [9]. The centrality algorithm we analyze is an iterative algorithm called PageRank [28]. These algorithms are extensively used in computer science and uniquely operate on graphs.

Graph traversal algorithms are essential for computing information flow in applications. In a sequential implementation, each algorithm we employ travels from vertex to vertex and performs a comparison. BFS traverses each graph level to assign the level to a value array or find a vertex by traversing levels. Parallel BFS compares each vertices level to its neighbors. The vertex value converges to the minimum neighbors'

level plus one in the parallel implementation. BFS applications include recommendation systems, puzzle solving, network broadcasting, and graph element searching. SSSP is similar to BFS. SSSP calculates the shortest distance to each vertex from a given source. Instead of minimizing the level of a vertex, the distance from a source node is minimized at each vertex. The previously mentioned comparison ultimately converges to the shortest possible distance from the source to a vertex. We adopt a common parallel implementation from the traditional Bellman-Ford and Dijkstra algorithms. An example of SSSP includes navigation systems and network routing. SSWP is the converse of SSSP. The traversal is the same, except the comparison results in the maximum distance from a source vertex to any other vertex. SSWP applications include computer network design, wireless connections, and credit maximization in financial settings. Each traversal algorithm involves moving through the graph and comparing the visited vertices. The key differences between these algorithms result from the comparison. This comparison typically raises a flag upon exhausted updates within a given loop that dictates the execution of the traversal.

The connected components algorithm identifies core subgraphs [8] [9]. Sequentially this is done by a traversal and a comparison as well. In parallel, this is done by generating "super vertices" as described by Hirschberg et al. [9]. Components are formed by assigning each node its sequential ID and updating clusters similar to our traversal algorithms. The vertex clusters share characteristics that are useful in dividing problem spaces. Image processing, compiler optimization, data mining, and

circuit analysis often divide a problem space using a components algorithm.

PageRank is an essential algorithm in the age of widespread internet access. Page et al. developed PageRank as one of the core algorithms for the original Google search engine. PageRank orders vertices (web pages) by importance by analyzing edges (hyperlinks) within the graph (the world wide web). A simplified version of the PageRank algorithm is best described by Page et al. as follows:

> Let u be a web page. Then let F_u be the set of pages u points to and B_u be the set of pages that point to u. Let $N_u = |F_u|$ be the number of links from u and let c be a factor used for normalization (so that the total rank of all web pages is constant).
>
> We begin by defining a simple ranking, R which is a slightly simplified version of PageRank:
>
> $$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N(v)} \text{ [28]}$$

The intuition behind PageRank is to propagate a rank of each vertex evenly to each of its neighbors. We implement PageRank in parallel by adopting constants for the initial ranks and subsequently calculating the change in ranking based on each node's degree. This is the common implementation of PageRank in parallel. Ultimately the

PageRank algorithm reaches a steady state. This state is reached after achieving a rank value within a given accuracy threshold. PageRank is significant in our analyses as it is a hugely popular iterative algorithm with high utility.

Graph algorithms are universal, efficient, and versatile structures essential to software applications. We highlight the key differences between the algorithms in Table 2.1 with a general outline of the sequential algorithms for BFS, CC, SSSP, and SSWP in Algorithm 1 and PR in Algorithm 2. In subsequent sections, we apply configurable options that include node activation, synchronization, and memory resource selection. These algorithms ultimately help generalize the relationships between energy consumption, performance, and input graph characteristics in a high-performance computing environment.

Algorithm 1 Algorithm for BFS, CC, SSSP, and SSWP

Require: $G = (vertices, edges)$
Require: $intialValue, update()$
  $distances \leftarrow$ unsigned int[vertices.size()]
  for $d \in distances$ **do**
    $d \leftarrow initialValue$
  end for
  for $v \in vertices$ **do**
    update(v, distance)
  end for

Table 2.1
Update and Initialization for Graph Algorithms

Algorithm	Initial Value	Source Initial Value	Update
BFS	∞	0	Minimum level
CC	node ID	node ID	Minimum component
PR	Constant	Constant	Constant * (Rank / Out-degree)
SSSP	∞	0	Minimum distance
SSWP	0	∞	Maximum distance

Algorithm 2 Algorithm for PR

Require: $G = (vertices, edges)$
Require: $intialValue, update()$
  $value \leftarrow$ unsigned int[vertices.size()]
  $delta \leftarrow$ unsigned int[vertices.size()]
  for $val \in value$ **do**
    $val \leftarrow initialValue$
  end for
  for $d \in delta$ **do**
    $d \leftarrow 0$
  end for
  for $v \in vertices$ **do**
    update(v, distance, delta)
  end for

2.2.2 Graph Processing Algorithm Variants

We have implemented five parallel algorithms, namely Breadth-First Search (BFS), Connected Components (CC), PageRank, Single-Source Shortest Path (SSSP), and Single-Source Weighted Shortest Path (SSWP), for parallel processing. The parallel graph processing algorithms can be performed on GPU and the CPU. To analyze these algorithms, we have derived algorithm variants by combining two critical parameters: synchronization and node activation.

Synchronization refers to the process of updating vertex values. Each algorithm variant incurs a necessary barrier at the end of each GPU kernel which is considered synchronous in a traditional sense. However, the synchronization types are still commonly referred to as asynchronous and synchronous processing in other works [29] [31]. This is because synchronization in graph processing refers to possible updates.

In asynchronous processing, vertex values are updated in the current iteration, which can lead to faster convergence. However, this approach can also incur more irregular data communication, reducing performance. On the other hand, synchronous processing only allows vertices to synchronize at the end of each iteration of a given algorithm's GPU kernel, using values from the most recently executed GPU kernel. While synchronous processing may not converge as quickly as asynchronous processing, it has the advantage of regular data communication, which aligns with the GPU's preference for all threads performing similar tasks. Understanding the tradeoffs between convergence and regular communication is key to our analysis.

Node activation is the other critical parameter in our algorithm variants. If node activation is present, the computation is data-driven (DD). Data-driven computation reduces the number of unnecessary updates by tracking active and inactive vertices. This allows for more effective kernel execution regarding the total number of computations. Alternatively, topology-driven computation does not track node activation and may incur unnecessary computations on inactive vertices. The topology-driven computation would appear to underperform, but that is not necessarily the case. The topology-driven computation may offset the overhead of label management for node activation in data-driven variants [31]. Redundant computation by topology-driven variants may also be offset by many threads [31]. The effectiveness of DD or TD is likely related to the unique structure of each particular graph and how many nodes would potentially be activated. We combine this with the synchronization strategies

to implement popular designs in parallel graph processing.

This work also applies Tigr's transformation technique and these graph configurations to optimize graph processing on GPU platforms [24]. Tigr's technique involves splitting nodes with high outdegree into a virtual layer, which reduces graph irregularity and improves performance. This is illustrated in Figure 2.1. By using this technique and combining it with our configurations, we are also able to apply the high-performance strategy of vertex-centric graph processing effectively.

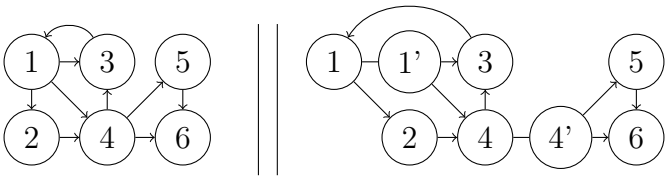

Figure 2.1: A Graph Before and After Tigr [24] Transformation (right)

Figure 2.2 illustrates the pairing of the synchronous and topology-driven parameters (Sync-TD). Sync-TD performs graph algorithms on fixed sets of vertices without restrictions on which nodes can receive updates. We relabel the fixed sets of vertices similar to SEP-Graph [31]. The updates performed are synchronous, implying that the update occurs after each kernel iteration. Figure 2.3 is quite similar to Figure 2.2 and illustrates the data-driven configuration counterpart for synchronous graph algorithms. The primary difference is the labeling of active and inactive nodes in addition to the synchronous updates.

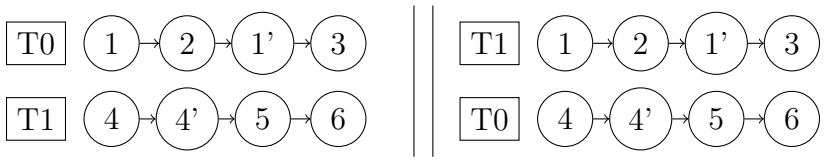

Figure 2.2: Synchronous TD Variant

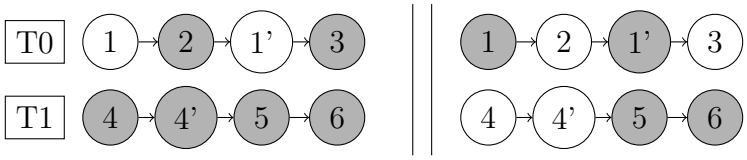

Figure 2.3: Synchronous DD Variant with Shaded Inactive Nodes

We also illustrate the asynchronous configurations of our parallel graph processing algorithms. In the asynchronous variant, updates can occur within the current processing iteration, resulting in irregularly exhausted computations at the end of each kernel. Figure 2.4 shows the asynchronous topology-driven (Async-TD) variant, which considers any available vertex until each thread has performed all possible work on its portion of vertices. On the other hand, the asynchronous data-driven (Async-DD) variant in Figure 2.5 only considers updates for active vertices. To contrast with the node activation from Sync-DD, it should be noted that all vertices are initially active in the Async-DD variant, whereas all vertices are inactive in the synchronous data-driven variant. This distinction is necessary to ensure correctness when considering the irregularity of the asynchronous variants' computations.

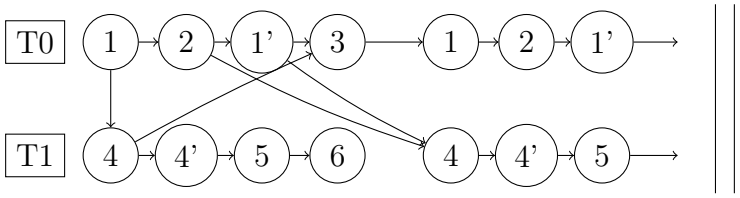

Figure 2.4: Asynchronous TD

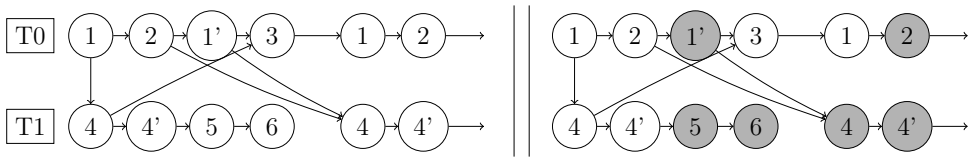

Figure 2.5: Asynchronous DD

2.2.3 Graph Processing on GPU Architectures

Graph processing is well suited for the GPU due to its unique organization. The organization of the GPU allows for massive parallelism, high memory bandwidth, and scalability. The GPU is advantageous due to its design focus on the flow of large amounts of information. The CPU design focus is on the execution of instructions with the additional consideration of manipulating memory. The CPU is expected to perform in more diverse settings that necessitate much of its design focus. Figure 2.6 depicts the differences between the CPU and GPU. The GPU is utilized in this work for several reasons.

Graph processing is a well-defined problem space that is naturally parallelizable on GPU architectures. The parallelization of some problems makes the GPU's conceptualization of threads, data, and instructions advantageous. The design of the GPU architecture is called single instruction multiple threads or SIMT [26]. SIMT is also called Single Instruction Multiple Data or SIMD in traditional Flynn categories of computer architectures [27]. The threads in SIMT architecture are data-centric and

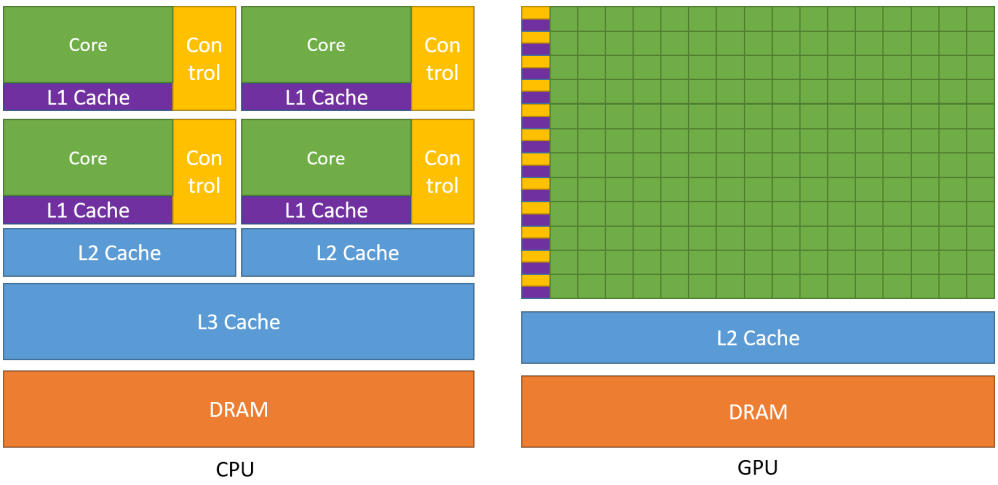

Figure 2.6: GPU Organization [26]

operate with thousands of threads. The threads found on the CPU are instruction-centric and number in tens of threads. The different perspective on threads enables the GPU to operate in parallel more broadly than the CPU. Thus the GPU has been characterized as a massively parallel architecture and is shown in Figure 2.7. Massively parallel architecture arranges the threads in synchronized groups with dedicated memory resources on chunks of data to maximize the throughput [26]. The arrangement of threads and data generates lanes of information that are optimal for data-intensive programs such as graph processing.

The memory hierarchy on the GPU enables much of the thread-level abstraction on the GPU. The memory hierarchy (Figure 2.6) of global memory, local memory, shared memory, and register memory. The threads can access the global memory in the same thread block with great abundance and slow access times [26]. Local memory operates similarly to global memory. There is an abundance of local memory, like

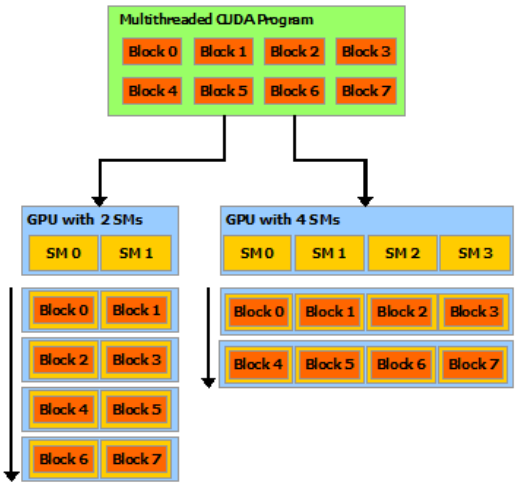

Figure 2.7: GPU Multihtreading [26]

global memory. This abundance means that the access times are essentially the same. Local memory differs from global memory by only allowing a single thread to access it. Therefore local memory allows for easy development but incurs a performance cost. Depending on the access pattern, global and local memory may be cached into L1 or L2 cache. The shared memory operates with block-level access, like global memory. However, it is significantly faster to access but is more scarce than global memory as it utilizes the GPU cache. The GPU also has a set of registers. Registers on the GPU operate similarly to CPU registers. They allow fast access times but are the most scarce memory type. This work utilizes global memory as the input graphs are large. Thus the capacity and transfer are more indicative of the performance than fast access times. Data transfer is the major bottleneck in many accelerated computing systems.

The data transfer is performed explicitly with software directives or implicitly using

unified memory. Unified memory is a single memory address space accessible from both the host (GPU) and the device (GPU) [26]. The single memory address space allows for implicit data transfer from the CPU host to the GPU device and easier programming. The main drawback of utilizing unified memory is the overhead of page faults. A page fault occurs when memory accessed by the GPU has not yet been transferred to the GPU memory [29]. The desired data is then paged into the GPU, which involves a transfer, translation look-aside buffer (TLB) updates, and page table updates [29]. This overhead can be a performance disadvantage but allows for easier programming and the ability to access memory beyond the GPU capacity. The increased accessible memory space is especially important for processing large, out-of-memory graphs, which are increasingly prevalent in real-world applications. Unified memory can also be used on smaller graphs and still require paging due to the page size of 4KB or 64KB depending on the system [6]. We explore the potential benefits of unified memory graph processing compared to explicit data transfer approaches.

Parallel graph processing on GPUs is advantageous but also incurs many challenges. These include data-driven computations, graph irregularity, and high data access to computation ratio [18] [8]. We perform data-driven computations, as previously mentioned. Our approach to data-driven graph processing is similar to existing works that utilize an active node list [2] [33] [31] [29]. Our extension of Tigr helps combat the irregularity problem on graphs [24]. The data access to computation ratio is reduced by choosing the GPU platform. The massively parallel architecture makes it

well suited for the multiple data accesses of graph algorithms. These solutions guide our implementation for graph processing and our choice of platform.

2.3 Motivation

There are many different implementations of GPU-based graph processing with different objectives. However, each implementation often focuses on a performance component in isolation. EEGraph combines several state-of-the-art frameworks, utilizing their collective high performance with additional analysis of GPU energy consumption. Our software also introduces a subgraph profiler to assist in the analysis of the graph processing parameters in a reduced way.

EEGraph explores graph processing algorithm variants as high-performance approaches to graph processing. A recent work, SEP-Graph, establishes this by performing active node tracking during the execution of graph processing algorithms to switch algorithm variants dynamically [31]. While the active node tracking during execution is a viable proxy for performance, we observe the performance after the graph processing. Our approach allows for more explicit performance measurement but, more importantly, allows for the evaluation of GPU energy. GPU energy is equivalent to instant power over time and can only be measured after execution. Memory

and thread usage profiling can often only be observed post-execution as well. Therefore, a post-execution model is better suited when considering the variety of profiling information possible for graph processing on GPUs.

EEGraph also includes the unified memory approach due to its popularity and utility in recent GPU applications. Unified memory has two primary benefits: programmability and out-of-memory computation. The more accessible programmability reduces the complexity of the software development process and allows developers to focus more on the problem space than the nuances of GPU programming. More importantly, unified memory's shared address space allows for out-of-memory computations. Subway explores using implicit unified memory graph processing and direct memory transfer approaches for out-of-memory graph processing[29]. They show that their direct approach has advantages over unified memory. However, more recent versions of unified memory have subsequently been released since the publication of Subway. Therefore we observe unified memory again for graph processing with a more fair performance measurement and the inclusion of GPU energy consumption. GPU energy consumption is of interest as the on-demand paging mechanisms of unified memory has the potential to offer energy savings resulting from operating only on relevant portions of the graph datasets for both small and large datasets.

Lastly, vertex-induced subgraphs are implemented in EEGraph to reduce the problem space's demands while replicating the complete graph's behavior. Subgraphs are

utilized primarily to perform the previously motivated post-execution profiling. It allows EEGraph to capture this valuable post-execution information within the bound of GPU memory while only slightly misrepresenting the whole graph in most cases.

Ultimately EEGraph builds on established graph processing frameworks from a post-execution profiling standpoint with the critical addition of GPU energy. We observe the results of algorithm variants and unified memory, which are popular in GPU graph processing. Additionally, vertex-induced subgraphs are generated to collect valuable profiling information with reduced space demands. EEGraph improves upon previous works and provides insightful analysis for GPU graph processing.

Chapter 3

EEGraph

3.1 Overview

At a high level, EEGraph contains four significant components, including algorithm variants, memory management, subgraph profiling, and measurement from bottom to top for five GPU-based graph processing algorithms, BFS, CC, PR, SSSP, and SSWP. The critical parameters of vertex synchronization and vertex activation define the algorithm variants. EEGraph also includes the ability to process both in-memory and out-of-memory graphs. Vertex-induced subgraphs are also generated for both types of memory management to infer the configurations' performance for algorithm variants and memory. The extension of unique graph processing software motivates

a precise measurement of execution time and GPU energy. These components guide our analysis of graph processing on GPUs.

3.2 Variant Processing Design

EEGraph compares algorithm variants for small and large graphs on GPU architectures. The critical parameters of vertex synchronization and vertex activation dictate the design of the variants. To achieve this, we have implemented each algorithm variant using popular parallel implementations using CUDA. Each algorithm variant has its own associated control loop, kernel, and activation lists inspired by other GPU-based implementations [24] Tigr, SEP-Graph [31], and Subway [29] which can be called as a library function after an input graph is read. We consider the four variants of Async-DD, Async-TD, Sync-DD, and Sync-TD using implicit (unified memory) and explicit data transfer (classic and subway) that are selected via command-line arguments. We assume our variant design, and proof of the variant designs, particularly Sync and Async, is beyond the scope of this paper. The algorithm variants are compared to Subway, which is an extension of Tigr for in-memory graphs and an out-of-memory graph processor for large graphs.

The objective of the algorithm variants is to observe the best combination of the critical parameters for graph processing. Each algorithm variant could offer the best

performance or GPU energy for a given graph, necessitating the profiling of all variants. This possibility is due to the unpredictability of the properties of a given input graph resulting from the extraordinary expressiveness of the structure.

3.3 Memory Management

Another component of EEGraph is memory management. Memory management refers to the different applications of unified memory for graph processing on GPUs. Unified memory's paging may be optimal for performance or GPU memory for processing both in-memory and out-of-memory graphs. We contrast the memory management of EEGraph to Subway, especially for the out-of-memory graph processing. Subway shows the benefits of subgraph computation for out-of-memory GPU memory management schemes for graph problems. Subway claims performance improvements compared to a CPU-based implementation Galois [22], but in 40% of their performance results, the CPU-based implementation has better performance [29]. Subway also primarily focuses on out-of-memory graph processing from a memory perspective, and their results indicate better memory utilization [29]. However, unified memory has had advancements since the publication of Subway. Subway uses CUDA Version 9 [29], and we use CUDA Version 11 on newer hardware. Subway also measures performance as just processing performance (i.e., kernel execution time) while separately evaluating the memory transfer. This method is practical but neglects that implicit

data transfer does not occur precisely at the GPU kernel boundary. Therefore, we consider the data transfer time in our performance measurement by segmenting the program at the first common assignment to GPU memory for the software presented in this work. We aim to enhance the knowledge pioneered by Subway by quantifying the performance and GPU energy consumption concerning implicit data transfers in a unified memory approach with Subway as the comparison.

To implement a unified memory graph processing software, we extend Tigr with unified memory (in addition to our algorithm variants) [24]. We enable unified memory by replacing `cudaMalloc()` with `cudaMallocManaged()` and removing explicit calls to data transfer functions in CUDA [26]. The usage hints `cudaMemAdviseSetReadMostly` for the `cudaMemAdvise()` function are applied to optimize the graph structure that is typically only read by the GPU in this software [26]. We observe memory management after reading an input graph and at the first common memory assignment for each memory management implementation.

This unified memory approach is compared to the explicit approach of our algorithm variants and Subway to understand the effects of modern GPU memory management for in-memory and out-of-memory graph processing. In our evaluation, we denote non-unified memory approaches as 'classic,' unified memory approaches as 'um,' and Subway as 'subway .' The analysis of the memory management for GPU-based graph processing is significant as graphs will inevitably increase in size, and newer devices

will continue to enhance new approaches to memory management. Our evaluation currently includes the memory implementations' performance and GPU energy consumption. The post-execution analysis and design of EEGraph also encourage other ways of quantifying memory management on GPUs for graph processing and other workloads.

3.4 Subgraph Profiling

The use of vertex-induced subgraphs in graph processing has several potential benefits that we explore in this work. First, we investigate whether generating a vertex-induced subgraph of an input graph can help infer the appropriate algorithm configuration for a given graph. This approach could be beneficial for graphs that necessitate an out-of-memory approach. In this situation, a subgraph could sufficiently select configurations reducing the memory footprint during the tuning stages of a project as opposed to running the whole graph. Subgraphs are also helpful for energy calculations. Since energy is power over time, we can only calculate energy after execution. Therefore we cannot switch runtime parameters like a previous work SEP-Graph [31]. The post-execution evaluation would also be the case for other profiling on GPUs, such as memory usage. By generating a subgraph and testing different configurations, we can identify relationships that suggest optimal configurations for the complete graph for performance and energy.

In our implementation, we generate the vertex-induced subgraph with a random selection of **five percent** of the vertices and associated edges for in-memory graphs and with **one percent** of the vertices for large out-of-memory graphs. We then randomly select 50 percent of the subgraph's vertices from the first 20 percent of vertices for the complete graph, as many real-world graphs have highly skewed connectivity patterns as shown in Table 4.1 and Table 4.2. Our software specifying the percentage of nodes in the subgraph should not exceed 40 percent, as this could result in an infinite loop in the current implementation. Additionally, knowledgeable users would see that many vertices are counter-intuitive to generating a subgraph in this context. Subgraph generation that retains the nature of the connectivity of the original graph may be a simple yet effective approach in graph processing on GPUs.

Once we generate the subgraph, we can evaluate it using the same algorithms and configurations as the complete graph. In our evaluation, we compare the best variants' speedup and relative energy consumption in the complete graph to the best variants' speedup and relative energy consumption in the subgraph and report the relative difference as a percentage. The results are similar if the difference is small. We can infer the appropriate configuration from the subgraph for performance and energy if the results are similar. Additionally, if the subgraph approach works well for out-of-memory graphs, this could provide a way to profile these graphs in memory, potentially reducing profiling time and space. Using vertex-induced subgraphs presents a promising avenue for improving graph processing efficiency and scalability.

3.5 Measurement

3.5.1 Energy Measurement

Another significant component of EEGraph is measuring the energy consumption of graph processing on GPUs through experimental measurements. Currently, EEGraph measures only the GPU's energy. Given the heterogeneous nature of our work, it would also be helpful to measure the system's overall energy efficiency (i.e., including the CPU). Energy consumption is equivalent to power over time or the integral of power and time. Therefore we can only measure the energy after the execution of a GPU program subject to the error associated with integration. Additionally, we must consider the possible noise from the power readings we obtain as a part of our energy measurements. Despite these known obstacles, our energy measurement software shows great feasibility in understanding the energy efficiency of graph processing algorithms.

To measure the energy efficiency, we spawn a CPU thread to start measuring the GPU's power consumption during the program. A header-only class retrieves the power through the NVIDIA Management Library (NVML) [25] [23]. NVML is an interface that allows the user to query device information, including power and clock

frequencies. A spawned CPU thread begins polling the device query for power consumption. The GPU idles for 5 seconds to reduce the noise from previous operations, consistent with established best practices for measuring energy. The power consumption is measured every 250 microseconds for large graphs and Subway. The power consumption is polled every 50 microseconds for small graphs and subgraphs as they have shorter execution times and require more precision. The GPU is polled continuously at these intervals until the GPU computation (graph processing) completes. At this point, a stopping CPU thread is spawned and raises a flag to signal the end of the computation. A five-second cooldown period is then enforced, with a poll occurring every ten milliseconds for large graphs and one millisecond for small graphs and subgraphs. Subway on small graphs would be polled as frequently as EEGraph small graphs, but the software is incompatible with a higher polling rate without significant alteration. Additionally, Subway's implementation of an asynchronous SSWP only works well with our energy measurements with significant modification, so we do not report the energy measurements for the asynchronous Subway implementation of SSWP. After all the power information is collected, EEGraph can finally calculate the energy consumption by integrating the power consumption over time using the trapezoidal rule, similar to the Riemann sums illustrated in Figure 3.1.

The energy measurement is sensitive to the error term associated with trapezoidal integration, device power management, and sampling rate. The error associated with trapezoidal integration is never truly zero but mitigated by frequent polling. The

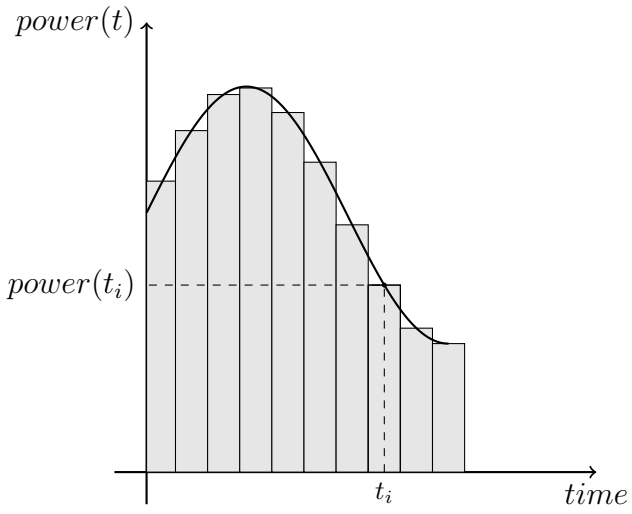

Figure 3.1: Energy Measurement Calculation

more frequent the polling, the more precise the energy measurement becomes, and the magnitude of the error essentially decreases. However, this quickly inflates the data space occupied by the power readings. Additionally, a more frequent polling rate becomes subject to irregularities in individual power readings. Small graphs with a short execution time are more subject to noise and integration error, but multiple runs and a shorter polling period mitigate these situations. Large graphs are impacted less by the possible noise as the longer execution time smoothes out the possible noise from the power readings. However, large graphs lose precision due to a less frequent polling rate to reduce the space considerations of the energy measurement. In future applications, it would be beneficial to dynamically calculate the optimal polling rate based on graph properties or expected execution times.

We provide an analysis of our software approach for GPU energy as current software-based approaches for GPU energy efficiency are considerably sparse. In fact, to

our knowledge, EEGraph implements one of the first software-based GPU energy measurement schemes. With our energy measurement, we consider different 'energy points,' shown in Table 3.1, to analyze the energy consumption for specific program sections, such as kernel execution or data transfer. EEGraph starts polling before initialization (point 0) but considers the total energy between the initialization (point 1) and the point before the device cooldown (point 4 for classic and point 3 for unified memory). The drawback of this approach is that the transfer back to the host is not strictly enforced in the unified memory approach. This situation occasionally results in unified memory implementation consuming less energy. However, these scenarios are rare in our analysis. Additionally, we compare the energy consumption (similar to a speedup calculation) relative to the Async-DD variant in the classic implementation and divide it by the energy consumption of the current variant of interest. Overall, our energy point and the power polling system give us a high-level experimental understanding of the contributing factors of GPU energy efficiency in the context of graph processing on GPUs.

Table 3.1
Relevant Energy Points

No.	**Classic Energy Pts**	**No.**	**UM Energy Pts**
0	Start of program (Warm Up)	0	Start of program (Warm Up)
1	Initial memory assignments	1	Initial memory assignments
2	Beginning of kernel	2	Beginning of kernel
3	End of kernel / Return transfer	3	End of kernel
4	End of transfer	4	Device cooldown
5	Device Cooldown		

3.5.2 Performance Measurement

Our performance measurement begins by tracking the earliest possible assignment to a common data structure in all our implementations. This common starting point for performance measurement allows for more effective measurement of overall execution time as the data transfer in unified memory implementations does not occur at a standard specified time. This performance measurement differs from Subway's approach, which does not track data transfer timing and needs to be more accurate, given the nuances of unified memory. We stop measuring time after the processing has finished in line with the standard practices in computer systems research. The performance is the execution time between points 1 and 4 in Table 3.1. Our performance analysis applies a speedup calculation relative to the Async-DD variant in the classic implementation (same as GPU energy). Our performance measurement aims to collect timing information on all relevant GPU activity considering the algorithm variants and memory paradigms.

Chapter 4

Evaluation

4.1 Software Environment

Our software EEGraph is a C++ software inspired by Tigr [24], Subway [29], and SEP-Graph [31]. We utilize the irregularity reduction and vertex-centric graph processing from Tigr [24]. Our software derives the algorithm variants from both Graph-SEP [31] and Subway [29]. EEGraph was developed with `CUDA 11.1`, `GCC 7.3.1`, and `CMAKE 3.20.0-rc4` with the 2014 C++ Standard, including 2011 C++ multi-threading for energy measurements. Ultimately this allows for a high-performance implementation of graph processing on GPUs on our NVIDIA GeForce RTX 3090 device.

4.2 Datasets

In our analysis of graph processing algorithms, we utilize a variety of datasets that represent real-world applications of graphs. The small set of datasets ranges from a few hundred thousand to a few million vertices and includes applications such as websites, social networks, autonomous systems, and road networks. The small datasets are edge lists that have been retrieved from The Stanford SNAP Project [15]. The Google datasets represent hyperlinks and webpages, while LiveJournal and Pokec are social networking sites [15]. Two more intriguing datasets are Skitter and Road-CA. Skitter is an internet topology-based autonomous systems graph and Road-CA is a network of roads in California [15]. We also utilize large datasets to analyze out-of-memory graph processing on GPUs. They contain millions to billions of edges and vertices from Friendster and Twitter social networks. The large graphs have been retrieved from the SNAP Project and the Konect Project [15] [14]. All these datasets have a variety of sizes, diameters, maximum out-degrees, and average degrees. It also is worth noting that both the large and small graphs express the power law of real-world graphs with many high-degree vertices in a small portion of the graph. Therefore we process a given graph from a source vertex within the first 20 percent of vertices in an input graph and deliberately choose from this percentage of vertices in our subgraph generation. More detailed information about the datasets can be observed in Tables 4.1 and Table 4.2.

Name	Vertices	Edges	Diam.	Max Deg.	Avg Deg.	Deg. First 20%	Deg. Rem 80%
Google	875713	5105039	21	456	5.83	1802.84	5.56468
LiveJournal	4837571	68993773	16	20293	14.26	38.768	8.09881
Road-CA	1965206	2766607	849	12	2.82	4532.07	2.80466
Skitter	1696415	11095298	25	35387	13.08	3333.42	2.63238
Pokec	1632803	39622564	11	8763	24.27	3291.73	13.3984

Table 4.1
In-Memory Graph Datasets from SNAP Project[15]

Name	Vertices	Edges	Diam.	Max Degree	Avg Deg.	Deg. First 20%	Deg. Rem 80%
Friendster	131216732	1806067135	32	3615	13.76	28.97	10.84
Twitter-MPI	52579682	1963263821	18	779958	74.677	123.969	15.6812
Twitter WWW	41652230	1468365182	23	770155	70.506	109.157	16.7769

Table 4.2
Large Datasets from SNAP [15] and KONECT [14]

4.3 In-Memory Graph Processing

Graph datasets are available in many different sizes. Therefore, we first evaluate graphs and their associated vertex-induced subgraphs that fit into GPU memory. Then we can determine the effects of the algorithm variants and memory resources utilized. The in-memory graphs used in our evaluation have various properties and diverse use cases. We evaluate each graph's execution time and GPU energy. The retrieved measurements are then normalized to the Async-DD variant in the classic implementation as speedup calculations for performance and the energy equivalent of speedup for energy. We then summarize the underlying relationships between graph processing configurations and real-world in-memory graphs.

4.3.1 Performance and GPU Energy Consumption of In-Memory Graphs

Case Study of In-Memory Graphs

The performance of processing the Google dataset is presented for the classic, unified memory, Subway implementations, and each algorithm variant. Under the classic memory configuration, the data-driven variants perform the best in processing this dataset as shown in Figure 4.1 and Table 4.3. Despite the high number of kernel iterations, the data-driven variants' performance is attributed to the low maximum node activation shown in Table 4.4. Additionally, PR has a slightly higher maximum node activation in Table 4.4 which explains the close competition between the variants for that algorithm. The Sync-TD variant performs best for most algorithms using the unified memory implementation (Table 4.3). All the variants exhibit the best performance under the classic implementation rather than the unified memory or Subway implementation (Table 4.3). Since the Google graph dataset is relatively small, the energy consumption is more sensitive than the large graphs, but the energy results are still observed in Table 4.3. The data-driven variants are the most energy-efficient using unified memory, and both the classic and unified memory implementations consume less energy than the Subway implementation.

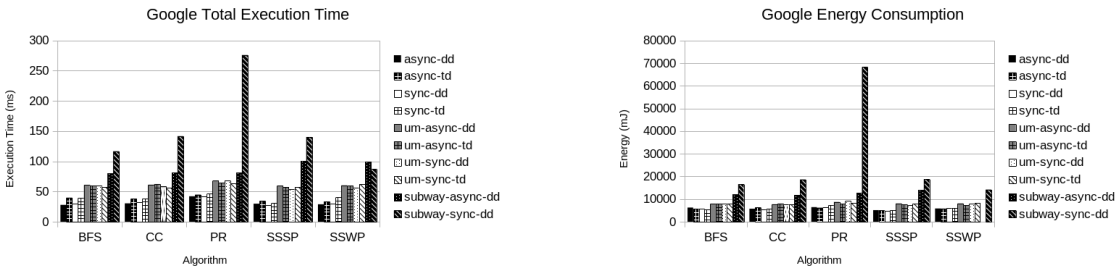

Figure 4.1: Google Execution Time (ms) and Energy Consumption (mJ)

Google Performance Relative to Async-DD										
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	**1.00**	0.70	0.94	0.71	0.46	0.47	0.47	**0.48**	0.35	0.24
CC	**1.00**	0.80	0.96	0.79	0.50	0.49	0.52	**0.55**	0.38	0.22
PR	1.00	0.94	**1.02**	0.91	0.62	**0.66**	0.62	**0.66**	0.52	0.15
SSSP	1.00	0.87	**1.07**	0.97	0.50	0.52	**0.55**	0.52	0.30	0.21
SSWP	**1.00**	0.86	0.95	0.71	0.48	0.48	**0.51**	0.46	0.29	0.33
Google Energy Relative to Async-DD										
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	1.00	1.11	1.10	**1.15**	**0.80**	0.78	**0.80**	0.78	0.52	0.38
CC	1.00	0.91	**1.04**	0.99	0.74	0.72	0.76	0.74	0.49	0.31
PR	1.00	**1.04**	1.02	0.88	0.74	**0.82**	0.70	0.76	0.50	0.09
SSSP	1.00	1.02	**1.06**	0.99	0.63	0.67	**0.68**	0.63	0.36	0.27
SSWP	1.00	**1.01**	0.97	0.94	0.72	**0.80**	0.74	0.70	X	0.41

Table 4.3
Google Normalized Performance and Energy

	Max Active		Iterations			
	async-dd	sync-dd	async-dd	async-td	sync-dd	sync-td
BFS	22.31	3.61	31	22	31	21
CC	33.92	3.61	21	20	31	21
PR	35.31	35.37	28	29	28	29
SSSP	22.31	3.61	31	22	31	21
SSWP	22.33	3.59	31	21	31	21

Table 4.4
Google Maximum Percentage of Active Nodes & Kernel Iterations

For the LiveJournal graph dataset, the Sync-DD variant performs best for most of the algorithms besides PR and CC, where the topologically driven variants perform well as shown in Figure 4.2 and Table 4.5. In Table 4.6 we can observe that PR has high iterations and node activation for this dataset, explaining the topological variants' performance. Therefore we can infer that a large frontier of vertices is updated for PR. This behavior is observed in both unified memory and classic implementations (Table

4.5). Energy consumption is minimized for the Sync-DD variant in both the unified memory and classic implementation for LiveJournal (Table 4.5). Additionally, Async-TD and Async-DD often perform similarly. Thus the algorithm variants' relationships with GPU energy reflect the relationships with performance.

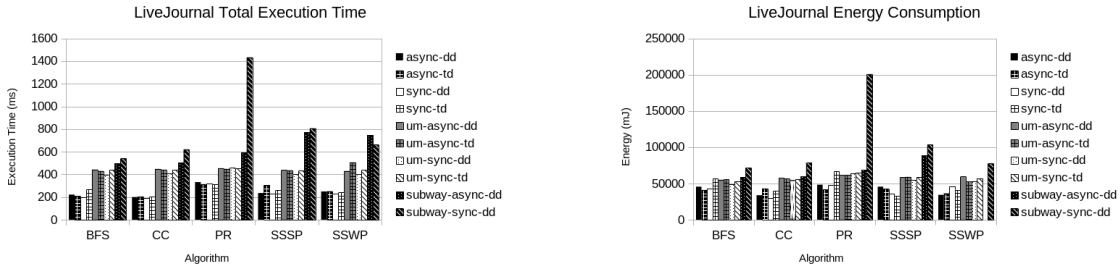

Figure 4.2: LiveJournal Execution Time (ms) and Energy Consumption (mJ)

LiveJournal Performance Relative to Async-DD										
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	1.00	1.06	**1.11**	0.83	0.50	0.51	**0.56**	0.51	0.44	0.41
CC	1.00	0.98	**1.04**	0.98	0.45	0.45	**0.49**	0.45	0.39	0.32
PR	1.00	**1.06**	1.03	**1.06**	0.73	**0.74**	0.72	0.73	0.56	0.23
SSSP	1.00	0.77	**1.02**	0.90	0.54	0.54	**0.59**	0.55	0.31	0.29
SSWP	1.00	0.99	**1.08**	1.03	0.58	0.49	**0.6**2	0.57	0.33	0.38
LiveJournal Energy Relative to Async-DD										
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	1.00	**1.11**	1.06	0.80	0.83	0.82	**0.92**	0.86	0.77	0.64
CC	1.00	0.79	**1.14**	0.84	0.59	0.60	**0.62**	0.61	0.57	0.43
PR	1.00	**1.16**	1.01	0.73	**0.78**	**0.78**	0.76	0.74	0.70	0.24
SSSP	1.00	1.07	**1.27**	1.37	0.78	0.77	**0.83**	0.78	0.51	0.44
SSWP	**1.00**	0.94	0.74	0.83	0.58	**0.65**	**0.65**	0.60	X	0.44

Table 4.5
LiveJournal Normalized Performance and Energy

	Max Active		Iterations			
	async-dd	sync-dd	async-dd	async-td	sync-dd	sync-td
BFS	33.56	33.52	11	9	11	9
CC	64.64	33.90	9	9	11	9
PR	64.40	64.70	43	43	43	43
SSSP	33.56	33.52	11	9	11	9
SSWP	33.56	33.52	11	8	11	8

Table 4.6
LiveJournal Maximum Percentage of Active Nodes & Kernel Iterations

The Pokec dataset is a social media graph that is similar to LiveJournal. The Sync-DD variant performs well for almost every algorithm for classic and unified memory

implementations as shown in Figure 4.3 and Table 4.7. Overall, each variant performs similarly in all algorithms, with the classic implementation outperforming the others (Table 4.7). The performance trends are similar to the energy consumption trends for Pokec, with the Sync-DD variant often consuming the least energy (Table 4.7). Additionally, the topological variants sometimes consume less energy due to all of the variants resulting in similar execution times and the reduced memory footprint of the topological variants when processing this dataset (Table 4.7). Another conclusion is that this is due to the moderate maximum node activation shown in Table 4.8. Overall the unified memory and Subway implementations have longer execution times and, therefore, more energy consumption than the classic variants for the Pokec dataset (Table 4.7).

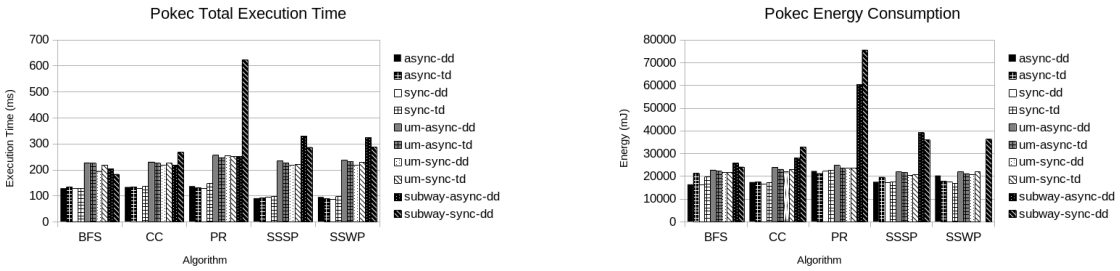

Figure 4.3: Pokec Execution Time (ms) and Energy Consumption (mJ)

Road-CA is included in our analysis as road networks are popular in graph processing due to their unique properties and direct application to navigation systems. The Subway implementation performs poorly on this graph, as shown in Figure 4.4. Otherwise, Sync-DD performs best for BFS, CC, and SSWP, and Async-DD performs best for SSWP and PR as shown in Figure 4.5 and Table 4.9. The classic implementation

Pokec Performance Relative to Async-DD										
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	1.00	0.96	**1.01**	0.99	0.56	0.57	**0.66**	0.58	0.63	0.70
CC	1.00	0.99	**1.04**	0.97	0.58	0.59	**0.61**	0.59	0.61	0.50
PR	1.00	1.05	**1.06**	0.93	0.53	**0.55**	0.53	0.54	0.54	0.22
SSSP	**1.00**	0.98	0.95	0.92	0.38	0.40	**0.41**	0.40	0.27	0.31
SSWP	1.00	1.06	**1.08**	0.97	0.40	0.41	**0.43**	0.41	0.29	0.33
Pokec Energy Relative to Async-DD										
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	1.00	0.77	**1.00**	0.83	0.72	0.74	**0.75**	**0.75**	0.63	0.68
CC	1.00	0.99	**1.04**	1.01	0.72	0.75	**0.79**	0.75	0.61	0.53
PR	1.00	**1.05**	1.00	0.98	0.90	0.94	0.94	**0.95**	0.37	0.29
SSSP	1.00	0.89	**1.02**	0.99	0.80	0.80	**0.85**	0.84	0.45	0.49
SSWP	1.00	1.14	1.15	**1.19**	0.92	0.95	**0.98**	0.92	X	0.56

Table 4.7
Pokec Normalized Performance and Energy

	Max Active		Iterations			
	async-dd	sync-dd	async-dd	async-td	sync-dd	sync-td
BFS	32.23	29.57	9	7	10	7
CC	66.19	74.37	9	7	12	7
PR	58.86	74.80	34	33	34	39
SSSP	46.17	30.29	10	8	10	8
SSWP	33.14	31.00	10	6	10	6

Table 4.8
Pokec Maximum Percentage of Active Nodes & Kernel Iterations

outperforms the Subway and unified memory implementations (Table 4.9). However, for BFS, CC, and SSSP, the unified memory performs similarly or better than some of the classic variants (Table 4.9). The sparse nature of this graph, inferred from the high iterations and low maximum node activation in Table 4.10, encourages the use of a data-driven graph processing framework regardless of the memory implementation. The performance is similar to the energy consumption, with the Sync-DD variant consuming a small amount of energy and the least for most of the algorithms under the unified memory implementation (Table 4.9). Subway consumes a large amount of energy proportional to the previously mentioned performance on this dataset (Table 4.9). The energy consumption for Road-CA highlights the potential performance

and energy consumption savings for unified memory on a small sparse graph. Additionally, it shows a relationship between performance and GPU energy for unique graphs.

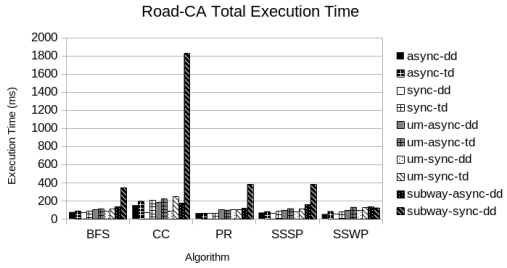

Figure 4.4: Road-CA Execution Time (ms)

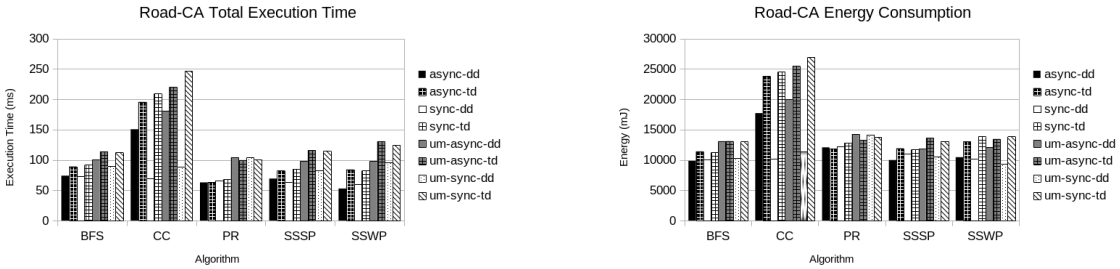

Figure 4.5: Road-CA Execution Time (ms) and Energy Consumption (mJ) without Subway

Road-CA Performance Relative to Async-DD										
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	1.00	0.83	**1.01**	0.81	0.74	0.65	**0.83**	0.66	0.54	0.21
CC	1.00	0.77	**2.15**	0.72	0.83	0.68	**1.70**	0.61	0.86	0.08
PR	**1.00**	0.99	0.95	0.93	0.60	**0.63**	0.60	0.62	0.54	0.16
SSSP	1.00	0.84	**1.09**	0.82	0.70	0.60	**0.84**	0.60	0.43	0.18
SSWP	**1.00**	0.63	0.88	0.64	0.54	0.40	**0.55**	0.42	0.38	0.43
Road-CA Energy Relative to Async-DD										
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	**1.00**	0.86	0.98	0.88	0.75	0.75	**0.96**	0.76	0.53	0.23
CC	1.00	0.74	**1.74**	0.72	0.89	0.69	**1.56**	0.66	0.80	0.09
PR	1.00	**1.02**	0.98	0.94	0.85	**0.91**	0.85	0.88	0.73	0.17
SSSP	**1.00**	0.84	0.91	0.85	0.84	0.73	**0.95**	0.76	0.47	0.21
SSWP	1.00	0.80	**1.02**	0.75	0.87	0.78	**1.12**	0.75	X	0.60

Table 4.9
Road-CA Normalized Performance and Energy

The Skitter dataset results show that many variants process the dataset at similar

	Max Active		Iterations			
	async-dd	sync-dd	async-dd	async-td	sync-dd	sync-td
BFS	8.47	0.08	164	163	164	163
CC	29.52	0.08	629	626	164	626
PR	42.30	41.99	85	85	85	85
SSSP	0.39	0.08	164	163	164	163
SSWP	0.39	0.08	164	163	164	163

Table 4.10
Road-CA Maximum Percentage of Active Nodes & Iterations

speeds shown in Figure 4.6 and Table 4.11. The performance is similar due to the moderate node activation, even with relatively small iterations presented in Table 4.12. Sync-DD is generally the best-performing variant in most algorithms for both the classic and unified memory implementation (Table 4.11). Furthermore, the classic implementation outperforms the other implementations in terms of performance. The Skitter dataset shows similar results to the performance results for GPU energy (4.11). The Sync-DD variant is often the best-performing variant. However, Async-DD and Sync-TD are more energy efficient for some of the algorithms, particularly for unified memory implementation, which is attributed to the previously observed similar execution times. However, we still conclude that the classic implementation consumes less energy than the other implementations for the Skitter dataset.

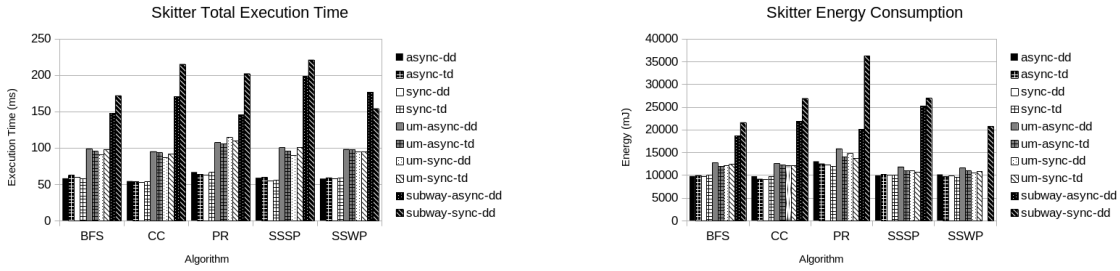

Figure 4.6: Skitter Execution Time (ms) and Energy Consumption (mJ)

Skitter Performance Relative to Async-DD										
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	**1.00**	0.93	0.97	**1.00**	0.59	0.61	**0.64**	0.59	0.39	0.34
CC	1.00	1.01	**1.02**	1.01	0.57	0.58	**0.63**	0.59	0.32	0.25
PR	1.00	1.04	**1.06**	1.00	0.62	**0.63**	0.58	0.61	0.46	0.33
SSSP	1.00	0.99	**1.06**	**1.06**	0.59	0.61	**0.65**	0.58	0.30	0.27
SSWP	**1.00**	0.98	1.00	0.99	0.59	0.59	**0.61**	**0.61**	0.33	0.37
Skitter Energy Relative to Async-DD										
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	1.00	0.98	**1.00**	0.97	0.77	**0.82**	0.81	0.79	0.52	0.45
CC	1.00	1.06	**1.07**	1.00	0.77	0.80	0.80	**0.81**	0.45	0.36
PR	1.00	1.04	1.05	**1.09**	0.82	0.93	0.88	**0.94**	0.65	0.36
SSSP	**1.00**	0.98	0.98	0.99	0.84	0.90	0.91	**0.92**	0.39	0.37
SSWP	1.00	1.05	1.03	**1.06**	0.87	0.92	**0.96**	0.94	X	0.49

Table 4.11
Skitter Normalized Performance and Energy

	Max Active		Iterations			
	async-dd	sync-dd	async-dd	async-td	sync-dd	sync-td
BFS	52.92	32.52	5	4	5	4
CC	84.56	32.48	4	4	7	4
PR	56.54	52.56	12	10	11	10
SSSP	55.13	32.52	5	4	5	4
SSWP	55.13	32.52	5	4	5	4

Table 4.12
Skitter Maximum Percentage of Active Nodes & Kernel Iterations

.

Summary of In-Memory Graph Processing

Evaluating our classic, unified memory, and Subway implementations leads to several conclusions about configurations for graph processing on GPUs for in-memory graphs. For performance, the synchronous data-driven variant performs well on most graphs as shown in the previous tables. This variant is similar to Tigr, which is considered state-of-the-art [24]. The Sync-DD variant is commonly the best-performing algorithm variant for both performance and energy. The Sync-DD variant exhibits the balancing of the GPU's preference for threads to perform similar operations and the effectiveness of node activation in many of the datasets. Our results suggest that asynchronous

execution often leads to an imbalance of thread work, and topology driven tends to be at a disadvantage to data-driven computations for graph problems explaining why Async-TD often has slower execution times and higher energy consumption. However, the Sync-DD variant is only sometimes the best variant for each dataset. Additionally, the trend with the Sync-DD variant is only observed in the traversal and components algorithms. PageRank tends to have different algorithms narrowly outperforming one another due to the difference in initial node activation and its iterative evaluation of the nodes. Therefore our work confirms our hypothesis that the performance often depends on the graph algorithm's nature and the input graph's properties.

Algorithm	**UM-Google**	**Sub-Google**	**UM-LJ**	**Sub-LJ**	**UM-Pokec**	**Sub-Pokec**	**UM-Road**	**Sub-Road**	**UM-Skitter**	**Sub-Skitter**	
BFS	2.07	2.86	1.99	2.50	1.52	1.43	1.23	1.88	1.57	2.54	
CC	1.83	2.65	2.10	2.62	1.71	1.70	1.19	2.37	1.64	3.22	
PR	1.53	1.96	1.43	1.89	1.92	1.96	1.59	1.86	1.69	2.32	
SSSP	1.94	3.61	1.73	3.33	2.43	3.18	1.29	2.55	1.63	3.59	
SSWP	1.96	3.02	1.74	2.85	2.50	3.28	1.83	2.33	1.64	2.67	
Overall											**Total**
GEOMEAN	1.86	2.77	1.78	2.60	1.98	2.18	1.40	2.18	1.63	2.83	2.07
MAX	2.07	3.61	2.10	3.33	2.50	3.28	1.83	2.55	1.69	3.59	3.61

Table 4.13
Classic Speedup Over UM and Subway

Algorithm	**UM-Google**	**Sub-Google**	**UM-LJ**	**Sub-LJ**	**UM-Pokec**	**Sub-Pokec**	**UM-Road**	**Sub-Road**	**UM-Skitter**	**Sub-Skitter**	
BFS	1.44	2.20	1.20	1.44	1.34	1.48	1.05	1.88	1.23	1.93	
CC	1.38	2.13	1.84	2.02	1.32	1.69	1.11	2.17	1.32	2.40	
PR	1.27	2.05	1.48	1.66	1.11	2.85	1.12	1.40	1.15	1.69	
SSSP	1.54	2.90	1.65	2.67	1.20	2.10	1.06	2.12	1.08	2.54	
SSWP	1.26	2.44	1.53	2.26	1.22	2.13	0.91	1.72	1.11	2.18	
Overall											**Total**
GEOMEAN	1.37	2.33	1.53	1.96	1.23	2.00	1.05	1.83	1.18	2.12	1.60
MAX	1.54	2.90	1.84	2.67	1.34	2.85	1.12	2.17	1.32	2.54	2.90

Table 4.14
Classic Energy Consumption Improvement Over UM and Subway

In summary, classic implementation shows a 2.07 mean speedup and a 3.61 maximum speedup over the best-performing variants for unified memory and Subway for in-memory graphs shown in Table 4.13. The energy efficiency of small graphs follows in Table 4.14. The classic implementations have a 1.60 mean energy consumption

improvement and a 2.90 max energy consumption improvement. Small graphs are not necessarily the primary target of Subway, and unified memory serves as a better comparison in Table 4.15 and Table 4.16. Our unified memory implementation shows a 1.39 mean speedup and a 2.20 max speedup over the best-performing variants over Subway and shows a 1.62 mean energy improvement and 2.57 maximum energy improvement. Ultimately, our results show improved performance and energy consumption when analyzing a variety of configurations for graph processing algorithms. In the following section, we will strategically construct subgraphs that confirm the best variants for each dataset and algorithm.

Algorithm	**Subway-Google**	**Subway-LJ**	**Subway-Pokec**	**Subway-Road**	**Subway-Skitter**	
BFS	1.38	1.26	0.95	1.53	1.62	
CC	1.45	1.25	0.99	1.99	1.96	
PR	1.28	1.32	1.02	1.17	1.38	
SSSP	1.86	1.93	1.31	1.97	2.20	
SSWP	1.54	1.63	1.31	1.28	1.63	
Overall						**Total**
GEOMEAN	1.49	1.46	1.10	1.55	1.39	1.39
MAX	1.86	1.93	1.31	1.99	2.20	2.20

Table 4.15
UM Speedup Over Subway

Algorithm	**Subway-Google**	**Subway-LJ**	**Subway-Pokec**	**Subway-Road**	**Subway-Skitter**	
BFS	1.53	1.20	1.11	1.80	1.57	
CC	1.54	1.10	1.28	1.95	1.81	
PR	1.62	1.12	2.57	1.25	1.46	
SSSP	1.88	1.62	1.75	2.01	2.35	
SSWP	1.94	1.47	1.75	1.88	1.96	
Overall						**Total**
GEOMEAN	1.69	1.29	1.62	1.75	1.58	1.62
MAX	1.94	1.62	2.57	2.01	2.35	2.57

Table 4.16
UM Energy Consumption Improvement Over Subway

4.3.2 Performance and GPU Energy Consumption for Subgraphs of In-Memory Graphs

Case Study of Individual In-Memory Subgraphs

Google is a small graph where the cost subgraph generation exceeds the utility of the subgraph. This relationship is evident with the processing of the Google dataset being relatively inaccurate for the classic subgraph implementation. However, there still may be a use case for a very small subgraph. The classic implementation has a small overall error but often does not align with the complete graph as shown in Figure 4.7 and Table 4.17. The unified memory implementation has aligned performance with the entire Google graph on BFS, CC, and PR (Table 4.17). However, SSSP and SSWP yielded an error rate of 23 percent for the unified memory implementation (Table 4.18). The energy consumption for the Google subgraphs is similar to the execution time for the Google subgraphs (Table 4.17). The classic implementation has accurate energy consumption for all algorithms besides BFS, while the unified memory implementation is either perfectly aligned or significantly misaligned compared to the complete graph (Table 4.18). Overall, the subgraph energy results are slightly better than the execution time results with a ten percent error rate (Table 4.18).

The LiveJournal subgraphs present promising results for classic and unified memory

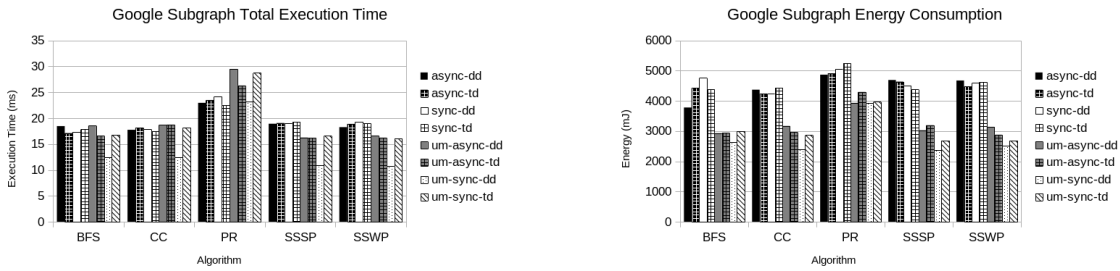

Figure 4.7: Google Subgraph Execution Time (ms) and Energy Consumption (mJ)

Google Subgraph Performance Relative to Async-DD								
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	1.00	**1.08**	1.06	1.04	1.00	1.11	**1.48**	1.10
CC	1.00	0.98	0.99	**1.02**	0.95	0.95	**1.43**	0.98
PR	1.00	0.98	0.95	**1.02**	0.78	0.87	**0.99**	0.80
SSSP	**1.00**	**0.99**	1.00	0.98	1.16	1.17	**1.73**	1.14
SSWP	**1.00**	0.97	0.94	0.96	1.10	1.12	**1.70**	1.14
Google Subgraph Energy Relative to Async-DD								
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	**1.00**	0.85	0.79	0.86	1.29	1.29	**1.43**	1.26
CC	1.00	**1.03**	**1.03**	0.98	1.38	1.48	**1.82**	1.52
PR	**1.00**	0.99	0.96	0.93	**1.24**	1.13	**1.24**	1.23
SSSP	1.00	1.01	1.04	**1.07**	1.55	1.47	**1.98**	1.74
SSWP	1.00	**1.05**	1.02	1.01	1.49	1.62	**1.86**	1.74

Table 4.17
Google Subgraph Normalized Performance and Energy

	Speedup			Energy		
	Classic	UM	Overall	Classic	UM	Overall
BFS	0.08	0.00	0.04	0.14	0.00	0.07
CC	0.02	0.00	0.01	0.00	0.00	0.00
PR	0.02	0.00	0.01	0.01	0.11	0.06
SSSP	0.01	0.57	0.29	0.03	0.51	0.27
SSWP	0.03	0.58	0.31	0.00	0.24	0.12
Average	0.03	0.23	0.13	0.04	0.17	0.10

Table 4.18
Google Subgraph Selected Relative Speedup & Energy Difference vs Full Selection

implementations. For performance, the execution times do not necessarily align as presented in Figure 4.8 and Table 4.19. However, the subgraphs are only slightly misaligned with the complete graph, and many variants perform similarly (Tables 4.19 and 4.20). The unified memory is slightly misaligned for PR but is otherwise entirely aligned with the results of the complete graph for execution time (Tables

4.19 and 4.20). The energy measurements for the subgraph follow the performance measurements for the LiveJournal subgraphs also shown in Tables 4.19 and 4.20. The classic implementation only sometimes selects the most energy-efficient variant of the complete graph (Table 4.19). However, overall the most energy-efficient variant of the subgraphs is quite similar to the most energy-efficient variant in the complete graph (Table 4.19). The unified memory implementation of the subgraph aligns precisely with the unified memory implementation of the whole graph and only has a misalignment on PageRank (Table 4.19). Remarkably, it selects the two most energy-efficient variants for SSWP that are also selected by the complete graph of LiveJournal (Table 4.19). The subgraphs generated for LiveJournal represent the energy consumption of the complete graph with an average difference of one percent shown in Table 4.20.

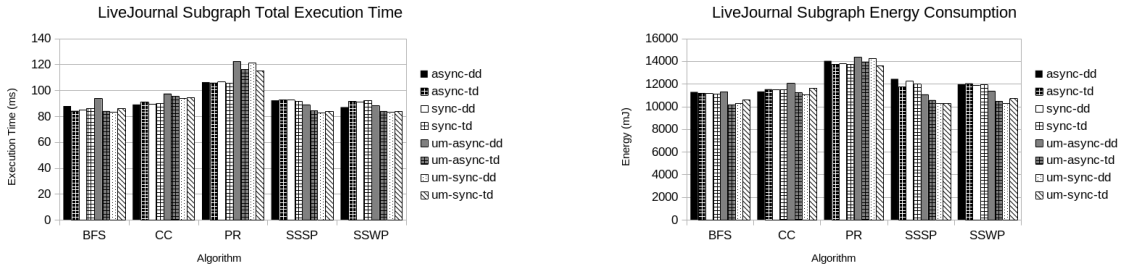

Figure 4.8: LiveJournal Subgraph Execution Time (ms) and Energy Consumption (mJ)

Pokec subgraphs also replicated the associated full graph well. Both classic and unified memory implementations are often aligned with the full graph or almost aligned with the full graph for performance and energy as shown in Table 4.21. Interestingly, the

LiveJournal Subgraph Energy Relative to Async-DD								
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	1.00	**1.04**	1.03	**1.02**	0.94	1.04	**1.05**	1.02
CC	**1.00**	0.98	0.99	0.99	0.91	0.93	**0.95**	0.94
PR	1.00	**1.00**	1.00	**1.01**	0.87	0.91	0.88	**0.92**
SSSP	1.00	**0.99**	1.00	**1.01**	1.04	1.09	**1.11**	1.10
SSWP	**1.00**	0.95	**0.96**	0.94	0.98	**1.04**	**1.04**	1.04
LiveJournal Subgraph Energy Relative to Async-DD								
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	1.00	1.01	1.01	**1.02**	1.00	1.11	**1.09**	1.06
CC	**1.00**	0.98	0.98	0.98	0.94	1.01	**1.02**	0.97
PR	1.00	**1.02**	1.02	1.02	0.98	1.01	0.98	**1.03**
SSSP	1.00	**1.06**	1.01	1.04	1.12	1.18	**1.21**	1.21
SSWP	1.00	0.99	**1.01**	1.00	1.05	**1.14**	**1.16**	1.12

Table 4.19
LiveJournal Subgraph Normalized Performance and Energy

	Speedup			Energy		
	Classic	UM	Overall	Classic	UM	Overall
BFS	0.01	0.00	0.01	0.01	0.00	0.00
CC	0.01	0.00	0.00	0.02	0.00	0.01
PR	0.00	0.01	0.00	0.00	0.03	0.01
SSSP	0.01	0.00	0.01	0.05	0.00	0.02
SSWP	0.04	0.00	0.02	0.01	0.00	0.00
Average	0.01	0.001	0.01	0.01	0.005	0.01

Table 4.20
LiveJournal Subgraph Selected Relative Speedup & Energy Difference vs Full Selection

unified memory implementation of the subgraphs has significantly shorter execution times than the classic implementations of the subgraphs (Table 4.21). The error was only one percent overall for classic and unified implementations of the subgraph processing for performance and energy for the Pokec dataset (Table 4.22).

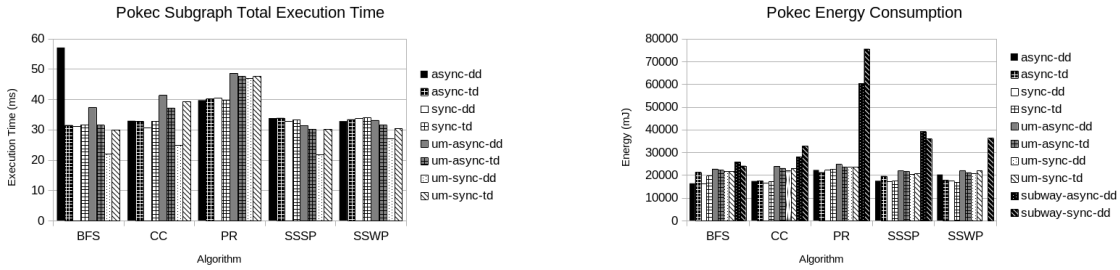

Figure 4.9: Pokec Subgraph Execution Time (ms) and Energy Consumption (mJ)

Pokec Subgraph Performance Relative to Async-DD								
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	1.00	1.81	**1.83**	1.80	1.53	1.81	**2.58**	1.90
CC	1.00	1.00	**1.07**	1.00	0.80	0.88	**1.31**	0.84
PR	**1.00**	0.99	0.98	**1.00**	0.82	0.83	**0.85**	0.83
SSSP	1.00	1.00	**1.03**	1.01	1.08	1.12	**1.55**	1.11
SSWP	**1.00**	0.98	0.97	0.96	0.99	1.04	**1.21**	1.08
Pokec Subgraph Energy Relative to Async-DD								
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	1.00	0.94	**0.96**	0.96	1.15	1.30	**1.69**	1.28
CC	1.00	1.08	**1.15**	1.06	1.08	1.12	**1.50**	1.08
PR	1.00	0.98	0.94	**1.02**	0.99	**1.07**	1.02	1.01
SSSP	1.00	1.03	**1.12**	1.02	1.22	1.37	**1.47**	1.33
SSWP	1.00	1.05	0.97	**1.09**	1.21	1.36	**1.50**	1.43

Table 4.21
Pokec Subgraph Normalized Performance and Energy

	Speedup			Energy		
	Classic	UM	Overall	Classic	UM	Overall
BFS	0.00	0.00	0.00	0.00	0.00	0.00
CC	0.00	0.00	0.00	0.00	0.00	0.00
PR	0.02	0.01	0.02	0.05	0.06	0.05
SSSP	0.03	0.00	0.02	0.00	0.00	0.00
SSWP	0.03	0.00	0.01	0.00	0.00	0.00
Average	0.02	0.003	0.01	0.01	0.012	0.01

Table 4.22
Pokec Subgraph Selected Relative Speedup & Energy Difference vs Full Selection

The unique behavior of the full Road-CA graph was captured well by our generated subgraphs shown in Table 4.23. The Sync-DD variant often had the best performance for the complete graph (Table 4.23). The subgraph selects this variant or a variant with similar performance for both the classic and unified memory implementations (Table 4.23). The subgraph results for PageRank were slightly misaligned with the complete graphs for both classic and unified memory subgraphs (Table 4.23). The classic implementations had difficulty replicating the results of the complete graph for energy, but the unified memory implementation exhibited the same energy differences as the complete graph (Table 4.23). For energy, the Road-CA subgraphs resulted in an error of two percent (Table 4.24). We captured the nuances of processing the Road-CA graphs with our generated subgraphs.

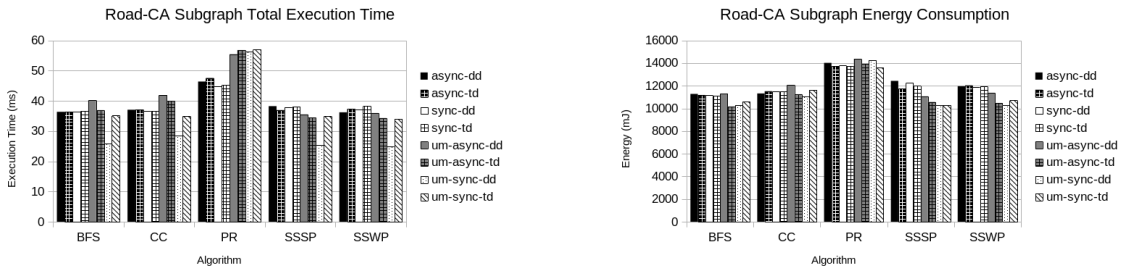

Figure 4.10: Road-CA Subgraph Execution Time (ms) and Energy Consumption (mJ)

Road-CA Subgraph Performance Relative to Async-DD								
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	1.00	1.00	**1.00**	0.99	0.90	0.98	**1.41**	1.04
CC	1.00	1.00	**1.01**	1.01	0.88	0.93	**1.30**	1.06
PR	1.00	0.98	**1.03**	1.02	**0.84**	0.82	0.82	0.81
SSSP	1.00	**1.04**	1.01	1.00	1.08	1.11	**1.50**	1.10
SSWP	**1.00**	0.97	0.98	0.95	1.01	1.05	**1.45**	1.06
Road-CA Subgraph Energy Relative to Async-DD								
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	1.00	1.05	**1.05**	1.05	1.20	1.34	**1.93**	1.44
CC	**1.00**	0.99	0.95	0.72	0.83	0.83	**1.12**	0.85
PR	1.00	0.98	1.00	**1.05**	0.98	**1.01**	0.97	1.01
SSSP	**1.00**	0.96	0.96	0.94	1.19	1.22	**1.69**	1.23
SSWP	**1.00**	0.94	0.98	0.90	1.25	1.23	**1.70**	1.26

Table 4.23
Road-CA Subgraph Normalized Performance and Energy

	Speedup			Energy		
	Classic	UM	Overall	Classic	UM	Overall
BFS	0.00	0.00	0.00	0.05	0.00	0.03
CC	0.00	0.00	0.00	0.05	0.00	0.02
PR	0.03	0.02	0.027	0.07	0.00	0.04
SSSP	0.03	0.00	0.01	0.00	0.00	0.00
SSWP	0.00	0.00	0.00	0.02	0.00	0.01
Average	0.01	0.00	0.008	0.04	0.00	0.02

Table 4.24
Road Subgraph Selected Relative Speedup & Energy Difference vs Full Selection

The full skitter graph dataset had similar performance across each algorithm variant, which was challenging to capture with our subgraph generation as shown in Figure 4.11 and Table 4.25. The classic subgraphs had an error of two percent, with the worst alignment observed in BFS for the classic subgraphs presented in Table 4.26. However, the unified memory subgraphs were aligned perfectly with the complete

graph of unified memory performance results (Table 4.25). Cumulatively the selection was only off by one percent for both kinds of implementations for performance (Table 4.26). However, the error was six percent and five percent for the classic and unified memory variants, respectively, for energy (Table 4.26).

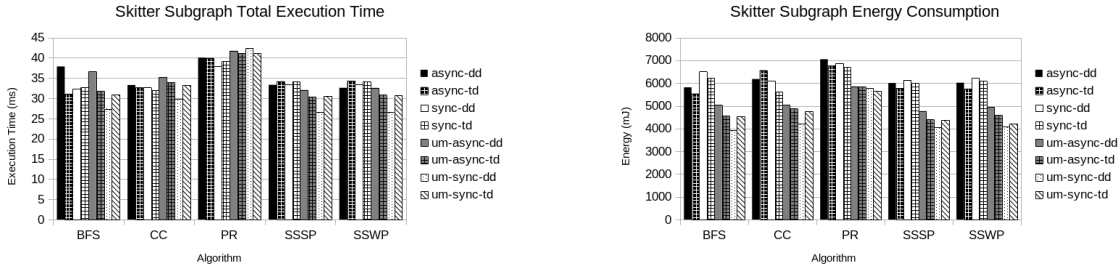

Figure 4.11: Skitter Subgraph Execution Time (ms) and Energy Consumption (mJ)

Skitter Subgraph Performance Relative to Async-DD								
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	1.00	**1.22**	1.17	1.16	1.03	1.19	**1.38**	1.22
CC	1.00	1.02	1.02	**1.04**	0.94	0.98	**1.11**	1.00
PR	1.00	1.00	**1.06**	1.02	0.96	**0.98**	0.95	0.97
SSSP	1.00	0.97	**1.00**	0.97	1.04	1.10	**1.25**	1.09
SSWP	**1.00**	0.95	0.97	0.95	1.00	1.06	**1.22**	1.06
Skitter Subgraph Energy Relative to Async-DD								
	async-dd	async-td	sync-dd	sync-td	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	1.00	**1.05**	0.89	0.93	1.15	1.28	**1.48**	1.28
CC	1.00	0.94	1.01	**1.10**	1.23	1.27	**1.46**	1.30
PR	1.00	1.04	1.03	**1.05**	1.20	1.21	1.22	**1.25**
SSSP	1.00	**1.04**	0.98	1.00	1.26	1.36	**1.48**	1.37
SSWP	1.00	**1.05**	0.96	0.99	1.22	1.31	**1.47**	1.42

Table 4.25
Skitter Subgraph Speedup Over Async-DD

	Speedup			Energy		
	Classic	UM	Overall	Classic	UM	Overall
BFS	0.06	0.00	0.03	0.16	0.00	0.08
CC	0.02	0.00	0.01	0.09	0.16	0.13
PR	0.00	0.00	0.00	0.00	0.00	0.00
SSSP	0.00	0.00	0.00	0.00	0.11	0.05
SSWP	0.00	0.00	0.00	0.06	0.00	0.03
Average	0.02	0.00	0.01	0.06	0.05	0.06

Table 4.26
Skitter Subgraph Selected Relative Speedup & Energy Difference vs Full Selection

.

Summary of In-Memory Subgraph Processing

The performance and energy of the generated subgraphs for in-memory graphs showed similarities to their associated complete graphs. The overall error between the full graphs and subgraphs was three percent and four percent for the relative execution times and energy shown in Table 4.27 and Table 4.28, respectively. The worst-performing algorithm for both execution time and performance was SSSP. Notably, in Table 4.29, the subgraph generation times are large relative to the overall execution times in the complete graphs. Also in Table 4.29 and Table 4.30 the space reduction relative to the complete graph is quite significant. The memory footprint is significantly reduced whether the sum of vertices and edges is considered or the memory size is considered. The memory size (Table 4.30) is the calculated size of the graph structure size in the program and depends on the systems data sizes for C++ programs. The space ratio refers to the comparison of the sum of vertices and edges for both the complete graph and the subgraph. Thus the cost of generating subgraphs for small graphs often outweighs the overall utility. This relative cost is especially true for the subgraphs for relatively small complete graphs for datasets like Google. However, suppose the subgraph generation is essential in a given workflow (i.e., post-execution profiling). Then our results show that selectively choosing vertex-induced subgraphs can replicate their full graph counterparts for graph processing on GPUs

for in-memory graphs.

Algorithm	Google	LiveJournal	Pokec	Road-CA	Skitter	Cumulative
BFS	0.04	0.01	0.00	0.00	0.03	0.01
CC	0.01	0.00	0.00	0.00	0.01	0.01
PR	0.01	0.00	0.02	0.02	0.00	0.04
SSSP	0.29	0.01	0.02	0.01	0.00	0.07
SSWP	0.31	0.02	0.01	0.00	0.00	0.05
AVERAGE	0.13	0.01	0.01	0.01	0.01	0.03

Table 4.27
Overall Subgraph Speedup Difference for Small Graphs

Algorithm	Google	LiveJournal	Pokec	Road-CA	Skitter	Cumulative
BFS	0.07	0.00	0.00	0.03	0.08	0.03
CC	0.00	0.01	0.00	0.02	0.13	0.03
PR	0.06	0.01	0.05	0.04	0.00	0.05
SSSP	0.27	0.02	0.00	0.00	0.05	0.05
SSWP	0.12	0.00	0.00	0.01	0.03	0.04
AVERAGE	0.10	0.01	0.01	0.02	0.06	0.04

Table 4.28
Overall Subgraph Energy Difference for Small Graphs

	Vertices	Edges	Sub Vertices	Sub Avg Edges	Classic Gen. (ms)	UM Gen. (ms)	Space Ratio
Google	875713	5105039	45821	12,111.00	32.62	91.69	.0097
LiveJournal	4837571	68993773	242378	716,351.60	376.38	1,117.57	.0130
Pokec	1965206	2766607	81640	215,379.00	161.78	477.74	.0628
Road-CA	1696415	11095298	98564	26,871.20	32.05	98.31	.0098
Skitter	1632803	39622564	84820	105,544.25	60.23	182.02	.0046

Table 4.29
Subgraph Size and Generation Time

	Sub Mem Size (KB)	Mem Size (KB)	Mem Ratio	Sub Avg Time (ms)	Avg Time (ms)	Time Ratio
Google	1,021.06	61260.479	0.017	18.677	62.473	0.30
LiveJournal	13,433.80	827925.287	0.0162	93.563	419.249	0.22
Pokec	4,549.77	33199.295	0.137	34.985	198.167	0.18
Road	2,018.88	133143.587	0.015	38.67	160.083	0.24
Skitter	2,899.35	475470.779	0.006	33.946	98.825	0.34

Table 4.30
Subgraph Memory Size and Average Execution Times

4.4 Out-of-Memory Graph Processing

As graph applications continue to grow, larger graph datasets are becoming a necessary area of focus. Thus we evaluate large graphs that do not fit into GPU memory

and evaluate the effects of our variant and memory configurations in a unified memory implementation and a strategic out-of-memory transfer implementation (Subway). The graphs used in our evaluation are social media graphs with varying sizes. We evaluate these graphs' execution times and GPU energy. The collected measurements are then normalized with respect to the unified memory Async-DD variant in the same manner as the in-memory graphs. Then we can observe the effectiveness of the out-of-memory solutions for graph processing on GPUs.

4.4.1 Performance and GPU Energy Consumption of Out-of-Memory Graphs

Case Study of Individual Out-of-Memory Graphs

Twitter-MPI is a large, directed asymmetric graph dataset that contains Twitter follower data based on a 2009 snapshot with users as vertices and following relationships as edges [14]. The Sync-DD variant performs the best on all algorithms besides PageRank for Twitter-MPI. In PageRank, Async-DD and Sync-TD perform the best, with the other variants having short execution times. Our unified memory graph processing for Twitter-MPI is significantly better than the Subway implementation. The energy consumption follows the performance for processing Twitter-MPI. Sync-DD has the best energy consumption for most algorithms, but Sync-TD has the best

energy consumption for PR. However, this is also the case for the execution time for PR. Additionally, Subway performs best for BFS. Low energy consumption may be a side effect of Subway's state-of-the-art memory management for out-of-memory graph processing [29]. Despite this, our unified memory approach is generally more energy efficient than Subway.

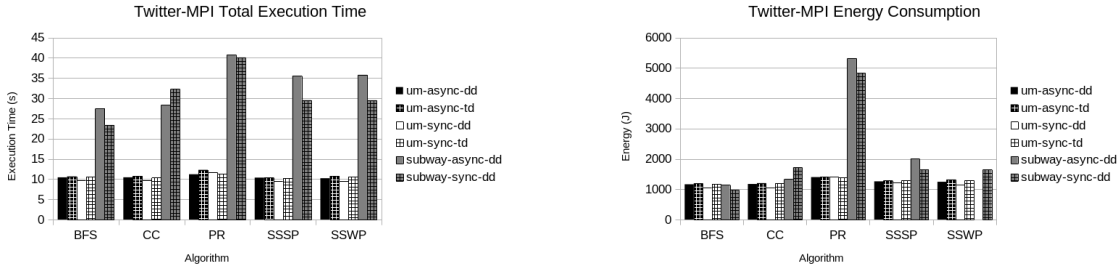

Figure 4.12: Twitter-MPI Execution Time (ms) and Energy Consumption (mJ)

Twitter-MPI Speedup Relative to Async-DD						
	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	1.00	0.98	**1.08**	0.99	0.38	0.45
CC	1.00	0.97	**1.08**	1.01	0.37	0.32
PR	**1.00**	0.91	0.96	**1.00**	0.27	0.28
SSSP	1.00	1.00	**1.08**	1.01	0.29	0.35
SSWP	1.00	0.95	**1.07**	0.96	0.29	0.35
Twitter-MPI Energy Relative to Async-DD						
	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	1.00	0.98	**1.09**	0.98	1.02	1.18
CC	1.00	0.98	**1.11**	0.98	0.88	0.68
PR	1.00	0.99	0.98	**1.01**	0.26	0.29
SSSP	1.00	0.98	**1.04**	0.97	0.63	0.76
SSWP	1.00	0.94	**1.08**	0.96	X	0.76

Table 4.31
Twitter-MPI Normalized Performance and Energy

Friendster is a graph of the social network by the same name, representing social communities of users [15]. The results for the Friendster dataset show that Sync-DD has the shortest execution time for the unified memory implementation. Likewise, the unified memory graph processing outperforms Subway. Sync-DD has the least

energy consumption, and our unified memory implementation is generally more energy efficient than Subway. However, some Subway variants perform better than the worst-performing unified memory variants.

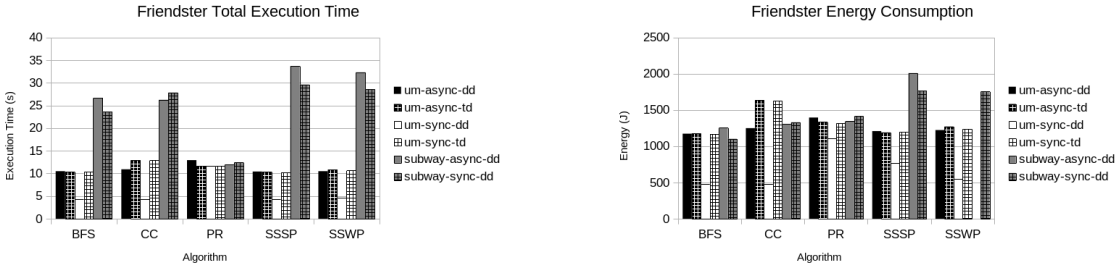

Figure 4.13: Friendster Execution Time (ms) and Energy Consumption (mJ)

Friendster Speedup Relative to Async-DD						
	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	1.00	1.01	**2.42**	1.02	0.39	0.44
CC	1.00	0.84	**2.49**	0.85	0.42	0.39
PR	1.00	1.11	1.10	**1.12**	1.08	1.04
SSSP	1.00	1.01	**2.39**	1.02	0.31	0.35
SSWP	1.00	0.97	**2.29**	0.99	0.32	0.37
Friendster Energy Relative to Async-DD						
	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	1.00	1.00	**2.44**	1.00	0.94	1.06
CC	1.00	0.76	**2.59**	0.77	0.96	0.94
PR	1.00	1.05	**1.26**	1.06	1.04	0.99
SSSP	1.00	1.02	**1.56**	1.01	0.60	0.68
SSWP	1.00	0.96	**2.23**	0.99	X	0.70

Table 4.32
Friendster Normalized Performance and Energy

Twitter-WWW is another large dataset based on a snapshot of Twitter. The Sync-DD variant continues to be the best for performance on this dataset. Additionally, the unified memory generally outperforms Subway for all algorithms. The energy consumption is also the best for Sync-DD for unified memory. Additionally, Subway consumes comparable or less energy for BFS, similar to the results of the Twitter-MPI dataset. As with the Twitter-MPI dataset, this may result from Subway's quality

memory management. Our results show that the unified memory implementation consumes less energy than Subway.

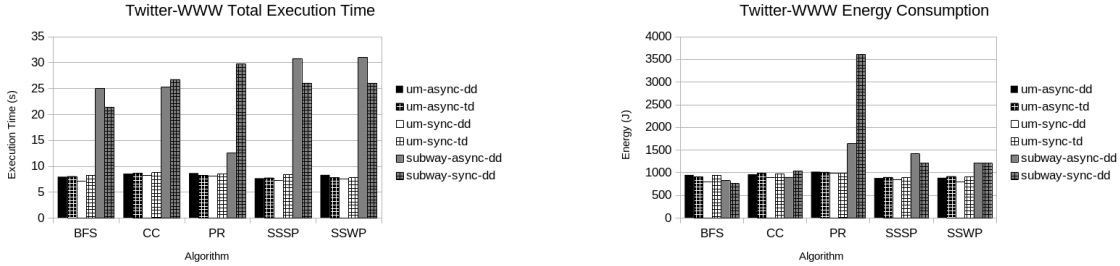

Figure 4.14: Twitter-WWW Execution Time (ms) and Energy Consumption (mJ)

Twitter-WWW Performance Relative to Async-DD						
	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	1.00	0.99	**1.10**	0.96	0.32	0.37
CC	1.00	0.99	**1.04**	0.97	0.34	0.32
PR	1.00	1.05	**1.07**	1.01	0.69	0.29
SSSP	1.00	0.99	**1.04**	0.90	0.25	0.29
SSWP	1.00	1.06	**1.10**	1.07	0.27	0.32
Twitter-WWW Energy Relative to Async-DD						
	um-async-dd	um-async-td	um-sync-dd	um-sync-td	subway-async	subway-sync
BFS	1.00	1.04	**1.17**	1.00	1.15	1.24
CC	1.00	0.97	**1.07**	0.98	1.07	0.92
PR	1.00	1.01	**1.03**	1.03	0.62	0.28
SSSP	1.00	0.98	**1.04**	0.99	0.62	0.73
SSWP	1.00	0.96	**1.11**	0.97	0.73	0.73

Table 4.33
Twitter-WWW Normalized Performance and Energy

.

Summary of Out-of-Memory Graph Processing

The processing of large out-of-memory graphs with our unified memory implementation and Subway implementation confirms the conclusions from the small graph processing. Typically, the synchronous data-driven variant balances the GPU's preference for evenly distributed thread-level load balancing and node activation. With the Sync-DD variant, unified memory tends to outperform Subway on large out-of-memory graphs despite this being a primary focus of Subway. However, they focus on out-of-memory processing primarily from a memory perspective. This improved memory approach occasionally results in Subway having better GPU energy consumption than our unified memory implementation on the Twitter datasets. The inclusion of the CPU energy would best determine if this relationship is true, as Subway potentially has more CPU-side organization present. However, large graphs express a clearer relationship between performance and energy as they smooth out the irregularities associated with our energy measurement scheme. Generally, our GPU energy results align with the performance results. Most importantly, EEGraph's unified memory implementation shows improvement over Subway.

Overall the unified memory implementation improves upon Subway. It exhibits a mean speedup of 3.3 and a maximum speedup of 6.80 compared to Subway for out-of-memory graph processing on our GPU system shown in Table 4.34. Additionally,

our unified memory implementation shows a 1.63 mean GPU energy improvement and 3.49 maximum GPU energy improvement over Subway, shown in Table 4.35. In the following section, we observe vertex-induced subgraphs for out-of-memory graphs as a practical in-memory profiling for large graph datasets.

Algorithm	**Subway-TMPI**	**Subway-FS**	**Subway-TWWW**	
BFS	2.42	5.45	2.98	
CC	2.94	5.99	3.08	
PR	3.58	1.03	1.55	
SSSP	3.09	6.80	3.56	
SSWP	3.11	6.24	3.44	
Overall				**Total**
GEOMEAN	3.00	4.27	2.81	3.30
MAX	3.58	6.80	3.56	6.80

Table 4.34
Out-Of-Memory UM Speedup Over Subway

Algorithm	**Subway-TMPI**	**Subway-FS**	**Subway-TWWW**	
BFS	0.92	2.29	0.94	
CC	1.26	2.71	1.00	
PR	3.49	1.22	1.66	
SSSP	1.36	2.28	1.43	
SSWP	1.44	3.19	1.53	
Overall				**Total**
GEOMEAN	1.51	2.23	1.28	1.63
MAX	3.49	3.19	1.66	3.49

Table 4.35
Out-Of-Memory UM Energy Improvement Over Subway

4.4.2 Performance and GPU Energy Consumption for Subgraphs of Out-of-Memory Graphs

Case Study of Individual Out-of-Memory Subgraphs

The Twitter-MPI subgraphs were excellent at exhibiting similar performance behavior as the full Twitter-MPI graph. As with the complete graph, Sync-DD performed the best in all but PageRank. For the PageRank algorithm, Async-TD was chosen rather

than Async-DD. Despite this, PageRank's error was only two percent, with the overall average error being 0.4 percent for execution time. The GPU energy measurements followed the performance results for the Twitter-MPI subgraphs. Sync-DD was the best energy consumption for the subgraphs for all algorithms besides PageRank. The subgraph variant that had the best energy consumption for PageRank was Async-TD which had the shortest execution time for PageRank. Cumulatively, the energy consumption error was 0.2 percent.

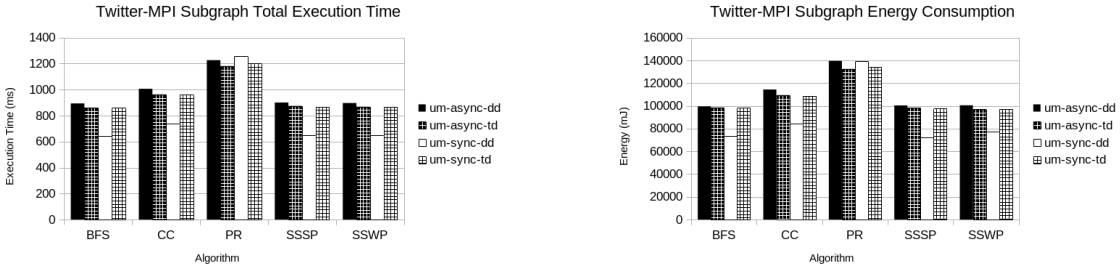

Figure 4.15: Twitter-MPI Subgraph Execution Time (ms) and Energy Consumption (mJ)

Twitter-MPI Subgraph Performance Relative to Async-DD				
	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	1.00	1.04	**1.39**	1.04
CC	1.00	1.05	**1.36**	1.05
PR	1.00	**1.04**	0.98	1.02
SSSP	1.00	1.03	**1.38**	1.04
SSWP	1.00	1.03	**1.39**	1.03
Twitter-MPI Subgraph Performance Relative to Async-DD				
	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	1.00	1.01	**1.36**	1.01
CC	1.00	1.05	**1.35**	1.05
PR	1.00	**1.05**	1.00	1.04
SSSP	1.00	1.02	**1.39**	1.03
SSWP	1.00	1.04	**1.30**	1.03

Table 4.36
Twitter-MPI Subgraph Normalized Performance and Energy

Like Twitter-MPI, the full graph of Friendster had the best performance and energy consumption with the Sync-DD variants of each algorithm except PageRank.

	Performance	Energy
BFS	0.00	0.00
CC	0.00	0.00
PR	0.02	0.01
SSSP	0.00	0.00
SSWP	0.00	0.00
Average	0.004	0.002

Table 4.37
Twitter-MPI Subgraph Selected Difference vs Full Selection

The subgraphs of Friendster were correct on every variant, with no wrong selections relative to the complete graphs. Since both energy and performance have no misselections, the tables for the subgraph errors are omitted. Overall, our subgraph results perfectly align with the performance results for the entire graph for the Friendster dataset.

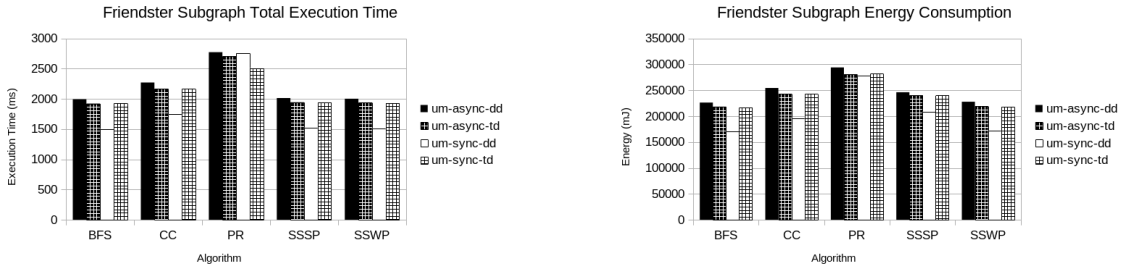

Figure 4.16: Friendster Subgraph Execution Time (ms) and Energy Consumption (mJ)

Friendster Subgraph Performance Relative to Async-DD				
	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	1.00	1.04	**1.33**	1.03
CC	1.00	1.05	**1.30**	1.05
PR	1.00	1.03	1.01	**1.11**
SSSP	1.00	1.04	**1.32**	1.04
SSWP	1.00	1.03	**1.32**	1.04
Friendster Subgraph Energy Relative to Async-DD				
	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	1.00	1.04	**1.33**	1.04
CC	1.00	1.05	**1.30**	1.05
PR	1.00	1.05	**1.06**	1.04
SSSP	1.00	1.02	**1.18**	1.03
SSWP	1.00	1.04	**1.33**	1.04

Table 4.38
Friendster Subgraph Normalized Performance and Energy

Twitter-WWW subgraphs also replicated their full graph counterpart well. In the full graph, Sync-DD performed best for all algorithms besides PageRank. PageRank was the only misalignment between the subgraphs and the full graph. Thus, PageRank's error is five percent for the subgraphs' execution time. Cumulatively, this yields an error rate of one percent for the subgraphs' performance. The energy measurement results for the subgraphs of Twitter-WWW follow the subgraph results for performance. Sync-DD has the best energy consumption for the full graph and the best energy consumption for the subgraph in all algorithms besides PageRank. The error for PageRank for energy is four percent. Overall the subgraphs for Twitter-WWW yield an error rate of 0.8 percent for energy for the subgraphs of Twitter-WWW.

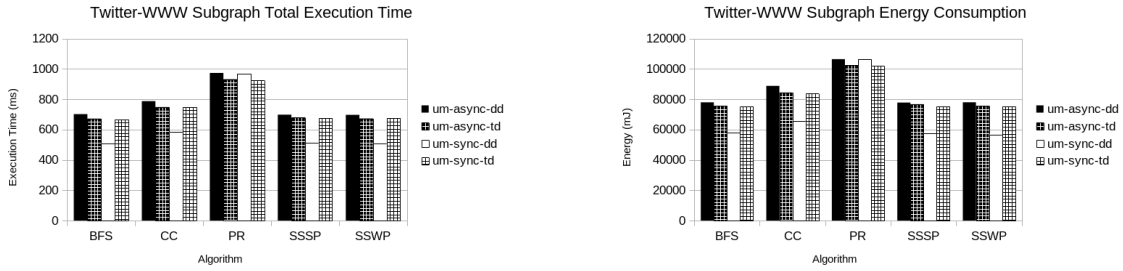

Figure 4.17: Twitter-WWW Subgraph Execution Time (ms) and Energy Consumption (mJ)

Twitter-WWW Subgraph Performance Relative to Async-DD				
	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	1.00	1.04	**1.38**	1.05
CC	1.00	1.05	**1.35**	1.05
PR	1.00	1.04	1.00	**1.05**
SSSP	1.00	1.03	**1.37**	1.04
SSWP	1.00	1.04	**1.37**	1.03
Twitter-WWW Subgraph Energy Relative to Async-DD				
	um-async-dd	um-async-td	um-sync-dd	um-sync-td
BFS	1.00	1.03	**1.35**	1.03
CC	1.00	1.05	**1.35**	1.06
PR	1.00	1.04	1.00	**1.04**
SSSP	1.00	1.01	**1.35**	1.03
SSWP	1.00	1.03	**1.37**	1.03

Table 4.39
Twitter-WWW Subgraph Normalized Performance and Energy

	Performance	Energy
BFS	0.00	0.00
CC	0.00	0.00
PR	0.05	0.04
SSSP	0.00	0.00
SSWP	0.00	0.00
Average	0.010	0.008

Table 4.40
Twitter-WWW Selected Difference vs Full Selection

.

Summary of Out-of-Memory Subgraph Processing

Large out-of-memory graphs can be difficult to process in their entirety, but our results show that using subgraphs can accurately represent their behavior for performance and energy. The observed subgraphs can represent the full graph and can be generated in a reasonable amount of time and a significant reduction in space as shown in Table 4.43 and Table 4.44. As with the in-memory subgraphs, the memory size (Table 4.44) is a calculation of the graph structure size in the program and may vary depending on the system and does not include the program control data. The space ratio refers to the comparison of the sum of vertices and edges for both the complete graph and the

subgraph. While there may be some slight errors in the subgraph, the overall error rate is low, with only a .4% error for performance and .3% error for GPU energy shown in Table 4.42 and Table 4.41. Therefore, a selective vertex-induced subgraph can be a viable option for representing large out-of-memory graphs, making out-of-memory graph processing more manageable than processing the entire graph.

Algorithm	Twitter-MPI	Friendster	Twitter-WWW	Cumulative
BFS	0.00	0.00	0.00	0.00
CC	0.00	0.00	0.00	0.00
PR	0.02	0.00	0.04	0.02
SSSP	0.00	0.00	0.00	0.00
SSWP	0.00	0.00	0.00	0.00
AVERAGE	0.00	0.00	0.01	0.004

Table 4.41
Overall Subgraph Speedup Difference for Large Graphs

Algorithm	Twitter-MPI	Friendster	Twitter-WWW	Cumulative
BFS	0.00	0.00	0.00	0.00
CC	0.00	0.00	0.00	0.00
PR	0.01	0.00	0.04	0.017
SSSP	0.00	0.00	0.00	0.00
SSWP	0.00	0.00	0.00	0.00
AVERAGE	0.00	0.00	0.01	0.003

Table 4.42
Overall Subgraph Energy Difference for Large Graphs

	Vertices	Edges	Sub Vertices	Sub Avg Edges	UM Gen Time (ms)	Space Ratio
Friendster	131216732	1806067135	1248361	391394.5	24877.33	.00085
Twitter-MPI	52579682	1963263821	525796	1160106.25	27543.55	.00084
Twitter-WWW	41652230	1468365182	416522	656032	32757.3	.00071

Table 4.43
Subgraph Size and Generation Time for Out-Of-Memory Graphs

	Sub Mem Size (KB)	Mem Size (KB)	Mem Ratio	Sub Avg Time	Avg Time	Time Ratio
Friendster	135913.5	21672805.631	0.006	918.392	17795.73	0.05
Twitter-MPI	66501.0	23559165.863	0.003	2063.001	47608	0.04
Twitter-WWW	49524.6	17620382.195	0.003	716.931	13872.551	0.05

Table 4.44
Subgraph Memory Size and Average Execution Time for Out-Of-Memory Graphs

Chapter 5

Related Work

5.1 GPU-side Graph Processing

In computer systems research, performance often precedes more subtle characterizations like energy efficiency. Hence much of the research for graph processing for GPU systems focuses on performance. There are several approaches to improving the performance of graph problems.

The non-traditional approaches for graph processing performance on GPUs focus on hardware and compilation. GraphPEG takes a low-cost hardware-based approach by enhancing resource utilization, memory bandwidth, and load balancing [19]. There are also compiler-oriented approaches such as G2, which extends the graph processing

compiler GraphIt by adding GPU implementations of graph algorithms [3].

Most of the GPU performance-oriented work for graph processing takes a software perspective. Xu et al. observe execution characteristics of graph applications of GPU architectures by simulating GPU behavior and running on real hardware to suggest optimizations for this problem space [35]. Several softwares (SwarmGraph [11], SAGE [30], XBFS [5]) tackle the issue of memory efficiency. These implementations are successful as they effectively take advantage of the unique organization of GPUs for graph processing. An even more comprehensive approach is SIMD-X which leverages several optimizations to manage irregularities associated with graph processing. The optimizations employed by SIMD-X include an Active-Compute-Combine strategy that better identifies node activity, a just-in-time task list for better load balancing and kernel fusion to address deadlock issues and greatly improve upon previous works [17]. Implementing SIMD-X is not as lightweight as some softwares, but it addresses relevant bottlenecks in graph processing.

A few GPU-based graph processing works are especially relevant to our implementation. The first is Tigr which implements a lightweight virtual graph transformation to reduce the effects of degree irregularity and improves the efficiency of in-memory graphs [24]. We adopt the virtual transformation from Tigr to apply a vertex-centric graph processing design effectively. Another influence on EEGraph is

Gunrock. Gunrock implements a synchronous, data-driven, and vertex-centric parallel graph processing abstraction that achieves better performance and expressiveness [33]. Alternatively, Groute is an asynchronous multi-GPU graph processing software [2]. Recently, research has found that both Gunrock and Groute offer viable use cases. SEP-Graph built on these works by creating software that processes graphs by dynamically switching difference configurations of the pairs of critical parameters: synchronization, message passing, and node activation [31]. We apply the thrust of SEP-Graph by analyzing pairs of critical parameters. However, our work also focuses on energy efficiency and out-of-memory graph processing.

5.2 Heterogenous Graph Processing

Heterogenous solutions to graph processing has been a growing focus in recent years. This focus is due to the necessary linkage between the CPU and GPU. The connection between the CPU and GPU is critical as the pairing represents the complete system used by a given software. Understanding this system can provide the broadest perspective on performance and energy efficiency. Heterogenous implementations have also become more of a focus due to the space constraints of the GPU. Despite some advantages, the GPU has more limited memory resources than the CPU. This limitation generates the "out-of-memory" problem for processing graphs larger than the GPU memory. To address this, NVIDIA has developed the unified memory feature

[26] that we will discuss further in subsequent chapters as it has been applied in EEGraph. Due to the increasing size of input graphs, EEGraph includes heterogeneous computing features.

Heterogenous graph processing has been applied to a variety of subtopics. CHAI is a set of benchmark programs created to help developers understand the trade-offs of heterogenous computing features [7]. Some works have focused on specific graph processing problems. GRUS targets the effects of graph complexities on graph processing with unified memory enhancements and execution optimizations [32]. TC-Stream explores out-of-memory triangle counting algorithms with performance improvements [10]. Xia et al. explore out-of-memory all pairs shortest path problems on GPU and develops a configuration selector [34]. Li et al. implement a transmission-focused heterogeneous parallel graph processing method. Their improvements are from multiple subgraph processing that more effectively exhausts the vertex values during execution as opposed to the single subgraph approach in other implementations [16]. Subway also uses subgraph generation and Tigr for out-of-memory parallel graph processing implementations [29] [24]. Subway's subgraph generation processes the most active vertices[29]. These explicit subgraph generation schemes are common in out-of-memory parallel graph processing. Therefore we compare the unified memory configuration of EEGraph to Subway for large graphs for both performance and energy.

5.3 Energy Efficiency of GPU Computations

Exploring the energy efficiency of different configurations for graph processing on GPUs is a primary objective of EEGraph. However, the current literature on specific problems and their energy efficiency is limited. Mittal et al. establish methods and improvements for GPU energy efficiency [21]. They primarily observe the relationship between CPU, GPU, and FPGA energy consumption and their power management settings [21]. Their work broadly establishes the energy efficiency and feasibility of GPUs but does not comment on specific types of problems applied to GPU platforms. GreenGPU achieves energy savings with a CPU-GPU distributed workload for several specific sample problems [20]. However, they use a hardware approach that physically monitors the power consumption of components. While this is perhaps the most effective way of measuring the energy consumption of the entire system, it is unrealistic in scenarios where users cannot access physical devices. Jiao et al. apply a similar hardware-based approach to analyze various computing and memory intensity levels on GPUs [12]. Both of the previously mentioned hardware-based approaches offer insight into the energy efficiency of GPUs. However, we implement a software-based measurement of energy consumption similar to Navarro et al. [4]. Their work focused on the energy efficiency and performance of triangle-dropping algorithms found in visual applications concerning mobile GPUs. They monitor energy using an external software Teapot [4] [1].

Research on the energy efficiency of GPUs has increased in recent years, but to our knowledge, there is limited energy efficiency research for graph processing on GPUs. Therefore we perform a software-based approach for measuring the energy consumption for graph processing on GPUs. Then we can experimentally characterize energy efficiency for different configurations of these algorithms in addition to performance.

Chapter 6

Conclusion

This work presents a successful graph processing software for both in-memory and out-of-memory graphs. Despite the success of our findings, there are several avenues for improvement for EEGraph. The energy measurement portion of EEGraph, to our knowledge, is the first lightweight software-based GPU energy measurement. However, in future work, it is also best to measure the CPU energy as much of the application of EEGraph includes heterogeneous graph processing. Optimizing the polling rate would be a beneficial addition to EEGraph as well. Another optimization would be the subgraph sizes to determine the smallest viable subgraphs for our profiling. Additionally, the evaluations of more algorithms to either confirm or contest the variant selections of both the small and large graphs. Finally, there is the potential for more configurable parameters, such as pull-based updates. Pull-based updates

have shown viability for performance, but most existing approaches, including EEGraph, use a push-based update scheme. Depending on the algorithm, a push-based scheme's effectiveness may not be optimal. Our software could also evaluate other graph algorithms. As mentioned in Chapter 2, we only cover a few prevalent graph processing algorithms. It would also be helpful to evaluate different algorithms, such as triangle counting or finite-state machine transitions. Despite these potential improvements, our graph processing software improves upon recent works and provides critical insight.

Our evaluation details the effectiveness of EEGraph for graph processing for both performance and energy efficiency. EEGraph includes several graph processing algorithms with different configurations for update synchronization, node activation, and memory paradigms. Additionally, our software has shorter processing times and consumes less GPU energy than a state-of-the-art graph processing software Subway for large and small graphs. For in-memory graphs, the best configuration of EEGraph outperforms the best configuration of Subway, with an average speedup of 2.08 and a maximum speedup of 3.61. EEGraph is also 1.60 more energy efficient and maximally 2.90 more energy efficient on the GPU device than Subway. For out-of-memory graphs using a unified memory, EEGraph has a 3.30 average speedup and a maximum speedup of 6.80 over Subway. EEGraph is 1.63 more times energy efficient than Subway on average and has a maximum energy improvement of 3.49. Therefore, EEGraph offers a significant improvement over existing frameworks for graph

processing.

EEGraph also explores vertex-induced subgraphs for in-memory and out-of-memory graphs for both execution time and GPU energy. On average, the small datasets' subgraphs deviate from the complete graph by 3.3% for performance and 4% for GPU energy. The large datasets' subgraphs have an error of .4% and .3% for performance and energy, respectively. The subgraphs generated in this work greatly reduce the space constraints of graph processing for post-execution analysis. The subgraph performance encourages using our subgraphs to represent the complete graph for profiling the performance and energy of graph processing algorithms and variants on the GPU.

References

[1] Arnau, J.-M., Parcerisa, J.-M., and Xekalakis, P. Teapot: A toolset for evaluating performance, power and image quality on mobile graphics systems. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (New York, NY, USA, 2013), ICS '13, Association for Computing Machinery, p. 37–46.

[2] Ben-Nun, T., Sutton, M., Pai, S., and Pingali, K. 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* (2017), PPoPP '17, Association for Computing Machinery.

[3] Brahmakshatriya, A., Zhang, Y., Hong, C., Kamil, S., Shun, J., and Amarasinghe, S. Compiling graph applications for gpus with graphit. *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (2021).

[4] Corbalán-Navarro, D., Aragón, J. L., Anglada, M., Parcerisa, J.-M., and González, A. Triangle dropping: An occluded-geometry predictor for energy-efficient mobile gpus. *ACM Transactions on Architecture and Code Optimization 19*, 3 (2022), 1–20.

[5] Gaihre, A., Wu, Z., Yao, F., and Liu, H. Xbfs: exploring runtime optimizations for breadth-first search on gpus. *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (2019).

[6] Garg, C., and Sakharnykh, N. Improving gpu memory oversubscription performance, Aug 2022.

[7] Gomez-Luna, J., Hajj, I. E., Chang, L.-W., Garcia-Flores, V., de Gonzalo, S. G., Jablin, T. B., Pena, A. J., and Hwu, W.-m. Chai: Collaborative heterogeneous applications for integrated-architectures. *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2017).

[8] Heidari, S., Simmhan, Y., Calheiros, R. N., and Buyya, R. Scalable graph processing frameworks: A taxonomy and open challenges. *ACM Comput. Surv. 51*, 3 (jun 2018).

[9] Hirschberg, D. S., Chandra, A. K., and Sarwate, D. V. Computing connected components on parallel computers. *Communications of the ACM 22*, 8 (1979), 461–464.

[10] Huang, J., Wang, H., Fei, X., Wang, X., and Chen, W. Tcstream: Large-scale graph triangle-counting on a single machine using gpus. *IEEE Transactions on Parallel and Distributed Systems* (2021).

[11] Ji, Y., Liu, H., and Huang, H. H. Swarmgraph: Analyzing large-scale in-memory graphs on gpus. *2020 IEEE 22nd International Conference on High-Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)* (2020).

[12] Jiao, Y., Lin, H., Balaji, P., and Feng, W. Power and performance characterization of computational kernels on the gpu. *2010 IEEE/ACM Int'l Conference on Green Computing and Communications; Int'l Conference on Cyber, Physical and Social Computing* (2010).

[13] Keith, W. J., and Kreher, D. Lecture notes from combinatorics and graph theory, October 2013.

[14] Kunegis, J. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion* (2013), pp. 1343–1350.

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

[16] Li, X. Scaling up large-scale graph processing for gpu-accelerated heterogeneous systems. *CoRR abs/1806.00762* (2018).

[17] Liu, H., and Huang, H. H. SIMD-X: Programming and processing of graph algorithms on GPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (July 2019), USENIX Association.

[18] Lumsdaine, A., Gregor, D., Hendrickson, B., and Berry, J. Challenges in parallel graph processing. *Parallel Processing Letters 17*, 01 (2007), 5–20.

[19] Lü, Y., Guo, H., Huang, L., Yu, Q., Shen, L., Xiao, N., and Wang, Z. Graphpeg: Accelerating graph processing on gpus. *ACM Transactions on Architecture and Code Optimization* (2021).

[20] Ma, K., Li, X., Chen, W., Zhang, C., and Wang, X. Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. *2012 41st International Conference on Parallel Processing* (2012).

[21] Mittal, S., and Vetter, J. S. A survey of methods for analyzing and improving gpu energy efficiency. *ACM Computing Surveys 47*, 2 (2014), 1–23.

[22] Nguyen, D., Lenharth, A., and Pingali, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, Association for Computing Machinery, p. 456–471.

[23] Nicely, M. Nvml examples. `https://github.com/mnicely/nvml_examples`, 2022.

[24] Nodehi Sabet, A. H., Qiu, J., and Zhao, Z. Tigr: Transforming irregular graphs for gpu-friendly graph processing. *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (2018).

[25] NVIDIA Developer. *Nvidia Management Library (NVML).* NVIDIA, Jan 2021.

[26] NVIDIA Documentation Hub. *Cuda C++ Programming Guide.* NVIDIA, 2020.

[27] Onder, S. Computer architecture lecture 12, 2021.

[28] Page, L., Brin, S., Motwani, R., and Winograd, T. The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford InfoLab, 1999.

[29] Sabet, A. H., Zhao, Z., and Gupta, R. Subway. *Proceedings of the Fifteenth European Conference on Computer Systems* (2020).

[30] Sha, M., Li, Y., and Tan, K.-L. Self-adaptive graph traversal on gpus. *Proceedings of the 2021 International Conference on Management of Data* (2021).

[31] Wang, H., Geng, L., Lee, R., Hou, K., Zhang, Y., and Zhang, X. Sep-graph: Finding shortest execution paths for graph processing under a hybrid framework on gpu. *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (2019).

[32] Wang, P., Wang, J., Li, C., Wang, J., Zhu, H., and Guo, M. Grus: Toward unified-memory-efficient high-performance graph processing on gpu. *ACM Transactions on Architecture and Code Optimization 18*, 2 (2021), 1–25.

[33] Wang, Y., Pan, Y., Davidson, A., Wu, Y., Yang, C., Wang, L., Osama, M., Yuan, C., Liu, W., Riffel, A. T., and et al. Gunrock. *ACM Transactions on Parallel Computing* (2017).

[34] Xia, Y., Jiang, P., Agrawal, G., and Ramnath, R. Scaling and selecting gpu methods for all pairs shortest paths (apsp) computations. *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2022).

[35] Xu, Q., Jeon, H., and Annavaram, M. Graph processing on gpus: Where are the bottlenecks? *2014 IEEE International Symposium on Workload Characterization (IISWC)* (2014).