Performance modeling of PageRank in large-scale systems

A case study

N.A. Doekemeijer

TUDelft

Performance modeling of PageRank in large-scale systems

A case study

by

N.A. Doekemeijer

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE.

Abstract

Graphs are a ubiquitous concept used for modeling entities and their relationships. Large graphs, present in a variety of domains, are often fundamentally difficult to process because of sheer size and irregular computation structure. In recent years, both academia and industry have committed to designing scalable solutions to efficiently process these graphs.

Next to processing large datasets in a distributed environment, a relatively new trend is to accelerate single node computation performance using heterogeneous platforms (for example, by leveraging the GPU as well as the CPU). However, the structure of the input graph can markedly influence the processing speed on a certain platform and it is unclear what would be the most efficient platform for execution given an input dataset.

In this thesis, we will analyze the performance of multiple PageRank implementations for diverse platforms. Using implementations for CPU (using OMP), GPU (using OpenCL and CUDA), and heterogeneous environments (using StarPU and MPI), we will characterize platform performance in relation to the structure of the input dataset. Finally, we will propose and evaluate a performance model for PageRank that takes into account traits of the input graph.

Student number:	4103890	
Public Defence:	August 26, 2020	
Thesis committee:	prof. dr. ir. H. J. Sips (chair)	Delft University of Technology
	dr. ir. A. L. Varbanescu (supervisor)	University of Amsterdam
	dr. ir. P. Pawełczak	Delft University of Technology
	dr. ir. J. Hidders	Birkbeck University of London

TU Delft

Preface

With this thesis, my time at TU Delft comes to an end. Over the years, I've had the privilege to meet, learn from, and work with a lot of talented people. I want to thank a few people in particular.

First, I want to thank Ana Lucia Varbanescu, my supervisor, for her guidance and support throughout the years. You are a great inspiration and I'm glad that we were able complete this thesis at long last.

Second, I want to thank Ernst van der Hoeven and Hans van den Bogert for their enjoyable company and thoughtful insights. You have made my time working in the Distributed Systems department a memorable experience. Similarly, I want to thank Elric, Laurens, Laurent, Maria, Martijn, Nir, Otto, Paul, Rob, Steffan, Wing, and everyone else from *MSc Lab 7.240*.

Finally, I want to thank my family for their endless love, support, and patience.

Niels Doekemeijer
Delft, August 14, 2020

Contents

1 Introduction **1**
1.1 Problem Statement . . . 2
1.2 Research Questions . . . 2
1.3 Approach . . . 3
1.4 Main Contributions . . . 3
1.5 Document Structure . . . 4

2 Background and Related Work **5**
2.1 Graph Theory . . . 5
2.2 Complex Networks . . . 6
2.3 PageRank . . . 8
2.4 Related Work . . . 9

3 Design and Implementation **11**
3.1 Methodology . . . 11
3.2 Graph-Processing Considerations . . . 12
3.2.1 Timing Model . . . 12
3.2.2 Execution Model . . . 13
3.2.3 Communication Model . . . 14
3.2.4 Data Partitioning . . . 15
3.3 Platform Considerations . . . 15
3.3.1 CPU Traits . . . 16
3.3.2 GPU Traits . . . 16
3.3.3 Heterogeneous Environments . . . 17
3.4 Design and Implementation of Parallel PageRank . . . 17
3.4.1 Data Representation for Graph Topology . . . 18
3.4.2 Data Representation for Computational State . . . 19
3.4.3 Data Partitioning . . . 19
3.4.4 Computation Kernels . . . 20
3.4.5 CPU Implementations . . . 21
3.4.6 GPU Implementations . . . 22
3.4.7 Heterogeneous Implementations . . . 23
3.5 Overview . . . 24

4 Device Performance Evaluation **27**
4.1 Methodology . . . 27
4.2 Datasets . . . 28
4.2.1 Synthetic Graphs . . . 28
4.2.2 Real-world Graphs . . . 30
4.3 Kernel Performance . . . 31
4.3.1 Graph Difficulty . . . 32
4.3.2 Workload . . . 33
4.3.3 Hardware Cache . . . 35
4.3.4 Data Flow . . . 36
4.4 Data Transfer . . . 37
4.5 Performance Modeling . . . 38
4.6 Main Findings . . . 39

5 Heterogeneous Performance Evaluation **41**
5.1 Methodology 41
5.2 Accelerator Performance 42
5.2.1 Heterogeneous Execution 42
5.2.2 Workload 43
5.2.3 Task Granularity 44
5.3 Cluster Performance 45
5.3.1 Execution Stages 46
5.3.2 Stream Partitioning 47
5.4 Scalability Analysis 48
5.5 Main Findings 49

6 Conclusion and Future Work **51**
6.1 Conclusion 51
6.2 Future Work 52

Bibliography **55**

1 Introduction

Graph theory provides an abstract model for entities and their relationships in the form of graphs. Formally, a graph (or network) consists of a set of vertices (entities) and a set of edges (links that connect vertices). In many fields of science, this system is used to model information. For example, the Internet is a collection of web pages connected by hyperlinks. A social network is a graph of people connected by friendship. The brain is a network of neurons. Transport networks, food webs, metabolic pathways can all be intuitively mapped to a graph representation.

Recent examples of hot research topics related to graph theory and graph processing are data mining, machine learning, and pattern recognition [43]. In these domains, one of the main challenges is the sheer size and velocity of input data. Early in 2011, the number of active Facebook users was 721 million with 68.7 billion friendship edges between them [57]. In 2020, the number of active users has grown to 2.6 billion [18]. The largest publicly available hyperlink graph of the World Wide Web, extracted from a crawl in 2012, contains over 3.5 billion pages and 128.7 billion links [46].

In computer science, time and space are two dimensions of a fundamental tradeoff; efficiently processing large quantities of data is an inherently difficult challenge (if data is stored compressed, it takes less space, but access time increases). One way to efficiently process the increasingly large datasets, is by distributing work over multiple processing units. However, this makes the design and implementation of graph-processing systems much more difficult.

In general, graph computations are data-driven [30, 41, 42]. This, in combination with the irregular structure of graphs, leads to a poor data locality and a varying degree of parallelism. While low-level implementations of graph algorithms allow for specific architectural and algorithmic optimization, they are subject to substantial (repeated) implementation effort. For example, loading a large graph dataset into memory, selecting optimal data structures, and implementing basic (but efficient) graph iteration are non-trivial tasks shared by most graph processing implementations. To speed up processing, it can be interesting to use accelerators in a system or distribute workload over a cluster. This brings extra challenges regarding synchronization, data consistency, and reliability. Again, non-trivial effort that would have to be repeated for each low-level implementation.

General-purpose large-scale batch processing frameworks such as MapReduce [13] and Spark [64] provide a certain level of abstraction (for fault tolerance, coordination, parallelism, etc.), but are not optimized for working with graph data and graph algorithms. Representing graphs and graph algorithms still requires a substantial implementation effort [48], which resulted in specialized graph-processing frameworks built on top of general-purpose frameworks ([31, 32, 49, 63]). Google, the organization that introduced MapReduce in 2004, recognized that existing general-purpose systems were ill-suited for graph processing and introduced Pregel [44] in 2010, a framework for graph processing that increases performance and usability compared to general systems.

Next to processing large datasets in a distributed environment, accelerating single node computation performance using heterogeneous platforms is a promising trend. However, because processing large-scale graphs involves data-driven computations, performance characteristics differ per system architecture and graph topology. So far, there has been little effort to model these performance characteristics for graph algorithms on heterogeneous architectures. A lot of (future) graph-processing frameworks would benefit from relying on such models to improve workload distribution that matches nodes' capabilities.

In this work, we present a case study of multiple implementations of one graph algorithm for a variety of architectures, in order to model performance and guide workload distribution between processors in heterogeneous environments. For this case study, we have decided to use PageRank [50] as main subject. By its nature, PageRank is a memory-intensive algorithm, which makes it interesting for investigating the influence of graph topology on performance.

1.1. Problem Statement

Due to the abundance of present-day graph computing problems and the ever-growing volume and velocity of data acquisition, academia and industry have committed a significant amount of effort into simplifying large-scale graph processing. For an extensive overview and comparison of over 80 large-scale graph frameworks released in recent years, we refer the reader to a survey by the authors [14].

The compelling next step, one that few frameworks have made so far, is to efficiently exploit an environment with heterogeneous processing capabilities. Hardware and memory topology does not have to be uniform in a cluster; some machines may have a newer generation of hardware, hardware with different processing capabilities, or some machines may be better connected than others. For example, Surfer[9] tries to partition the input graph based on available bandwidth between machines, while TOTEM[23] processes high-degree vertices on the CPU and offloads the low-degree vertices to the GPU.

In order to better guide the distribution of tasks and data between workers in a heterogeneous environment, more research is needed. Ideally, we would like to be able to predict the performance of a given workload and select the best computing infrastructure. However, the data-driven nature of graph computations, and varying topologies of graph datasets make this a non-trivial problem.

1.2. Research Questions

The main goal of this work is to provide models to support the distribution of a graph-processing workload in a large-scale heterogeneous system. This thesis proposes a three-stage research process driven by the following research questions:

RQ1 *How to evaluate the graph-processing performance characteristics of a processing unit?*

In a heterogeneous environment, it is unclear what would be the best allocation of resources to optimally process a given graph-processing workload. To this end, we need to evaluate the performance characteristics of a processing unit in order to characterize and compare computing devices. Moreover, we must understand and characterize the performance impact of dataset properties, in the context of vertex-centric graph processing, on device performance.

RQ2 *How to model the graph-processing performance of a processing unit with respect to dataset topology?*

The data-driven aspect of graph computations adds a layer of complexity when modeling performance in graph-processing systems. Therefore, we must identify the graph metrics that indicate complexity, and assess to what degree they correlate to processing performance. Finally, the goal is to model the *graph-processing performance* of a processing unit with respect to a dataset topology, in order to more accurately predict its performance for a given workload.

RQ3 *How to take into account the graph-processing performance characteristics of each processing unit in a heterogeneous system when distributing a workload?*

The optimal composition of heterogeneous devices might differ depending on the graph-processing workload. In some cases, it might even be beneficial to use a single device over a composition of devices. To maximize performance, we must understand how to use the graph-processing performance characteristics of each processing unit in a heterogeneous system to effectively distribute a graph-processing workload.

1.3. Approach

Addressing RQ1, we outline the general design decisions for graph-processing frameworks and propose a structural benchmarking method that determines the relative performance impact of each decision. Following this approach, we use PageRank as example algorithm for an in-depth case study of graph-processing performance on the CPU and the GPU.

Using a mix of real-world and synthetic datasets, we benchmark 20 custom PageRank kernels with gradually varying design decisions to evaluate the impact of each design decision. Addressing RQ2, we work towards modeling the kernel performance by determining the main performance indicators and correlating these to topological graph metrics of the input dataset.

Finally, addressing RQ3, we benchmark our kernels in a controlled heterogeneous setting and determine the speedup after expanding system resources. We relate this to our earlier findings and work towards systematically optimizing a workload distribution guided by the graph-processing performance characteristics of each processing unit in a heterogeneous system.

1.4. Main Contributions

The main contributions of this thesis are:

C1. (Conceptual) A structural benchmarking strategy aimed at measuring the impact of each design decision in a graph-processing framework.

C2. (Experimental) An in-depth evaluation of graph-processing performance characteristics of CPUs and GPUs using a PageRank case-study.

C3. (Experimental) A definition and evaluation of a graph-processing performance model that takes into account the topological structure of an input dataset.

C4. (Experimental) An in-depth evaluation of graph processing scalability in a heterogeneous environment using a PageRank case-study.

C5. (Software Artifact) 20 PageRank kernels with varying graph processing design decisions for CPU and GPU devices. 10 additional PageRank kernels specifically targeting heterogeneous environments.

C6. (Software Artifact) A benchmarking suite that facilitates structurally measuring the graph-processing performance characteristics of devices in a heterogeneous environment.

1.5. Document Structure

The remainder of this document is organized as follows. Chapter 2 briefly discusses the main concepts and related work. We provide a succinct introduction to graph theory and graph computing and define relevant terminology before discussing how this work relates to previous work. Chapter 3 addresses RQ1 and proposes a structural benchmarking method. Using this method, we design multiple PageRank kernels targeting various platforms. Chapters 4 and 5 study the performance of these kernels and address RQ2 and RQ3 respectively; chapter 4 proposes and evaluates a device performance model, chapter 5 evaluates performance and scalability in a heterogeneous setting. Finally, chapter 6 concludes this thesis and proposes directions for future work.

To readers who are interested in our main performance findings, we recommend reading sections 4.6 and 5.5. An overview of PageRank kernels is available in section 3.5. To readers who are interested in performance modeling, we recommend reading sections 4.5 and 5.4. To readers who want to learn more about our benchmarking suite, we recommend reading chapter 3 and sections 4.1 and 4.2.

2 Background and Related Work

In this chapter we introduce the main concepts needed to understand the remainder of this work. Moreover, we provide a brief overview of related work, specifically focusing on workload distribution in heterogeneous systems.

2.1. Graph Theory

Graph theory provides an abstract model for entities and their relationships in the form graphs. Formally, a graph (or network) consists of a set of *vertices* (entities) and a set of *edges* (links that pair vertices). Simply said, graph theory is the study of points and lines. Figure 2.1 illustrates the common way of visualizing graph data, with letters denoting vertices.

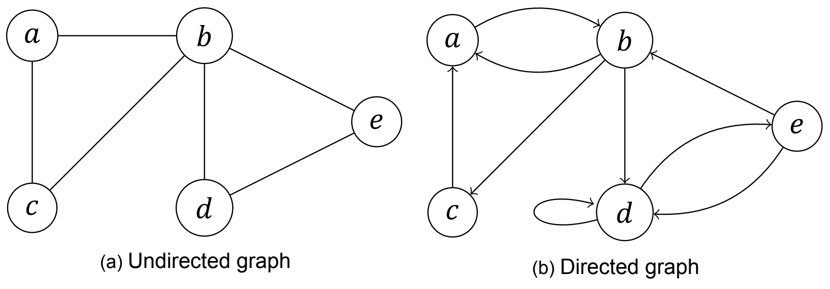


Figure 2.1: Common way of representing undirected (2.1a) and directed (2.1b) graphs.

Edges may be directed or undirected. Undirected edges indicate a two-way relationship, which means that the edge can be traversed in both directions (e.g., a collaboration graph). Directed edges represent a one-way relationship and can only be traversed in one direction (e.g., a citation graph). As undirected graphs can be simply mapped to a directed equivalent (by replacing each undirected edge with two directed edges), working with directed graphs for algorithms, proofs, and computation frameworks is more common.

We refer to the number of edges directly connected to a vertex as the vertex degree. For directed graphs, we can further distinguish between incoming and outgoing degree. The degree distribution models the probability that a selected vertex has exactly a given number of edges.

Graph topology refers to the arrangement of nodes and edges. Some examples of regular arrangements are: a mesh (every vertex is connected to every other vertex), a star (every vertex has a single edge towards the center vertex), or a ring (every vertex has a single edge towards the next vertex). However, most real-world networks do not follow a regular pattern and have a non-trivial topology, these networks are often referred to as Complex Networks.

2.2. Complex Networks

This section will briefly introduce complex networks, for a more extensive introduction the reader is referred to Wang and Chen [59]. In graph theory, a complex network is a graph with non-trivial topology, often observed in networks and models based on real-world phenomena. Among mathematicians, a well-known example of such a network is the Erdős collaboration graph (figure 2.2). It indicates the relation between authors of mathematical publications and their "collaboration distance" to Paul Erdős [28] (an influential mathematician who, among a broad number of other fields, contributed to graph theory).

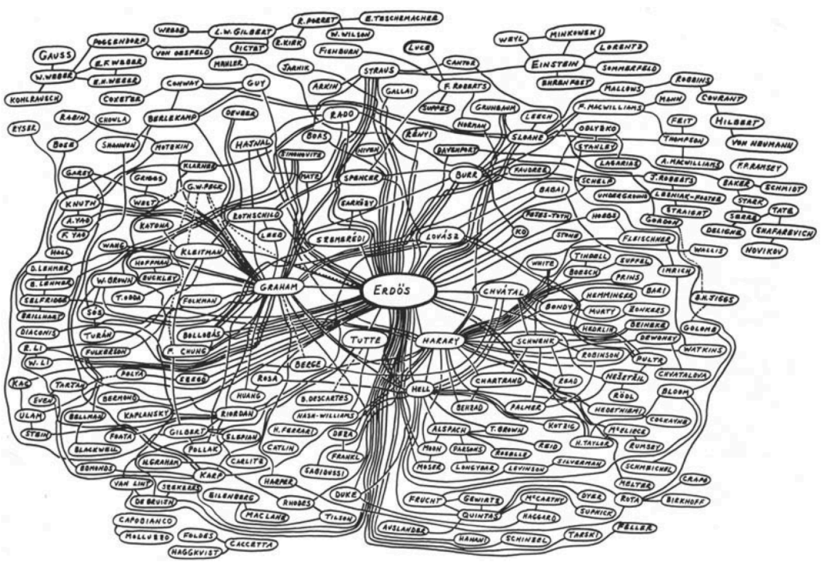


Figure 2.2: Illustration of a small part of the Erdős collaboration graph. The original dataset tracks over 400,000 authors. Courtesy of Easley and Kleinberg [15].

The Erdős collaboration graph has some properties that are typical of real-world networks. For example, while the average number of collaborators is 3, there is a small amount of authors that greatly exceed this number (e.g., Erdős himself has directly collaborated with over 500 people). The distribution of edges indicates a preferential attachment towards a small amount of authors. Next to that, the median path length between any two authors is very short. Even though the dataset consists of over 400,000 authors, the longest path is only 26 hops. These are phenomenons not seen in random graphs, where each possible edge between any two vertices has the same probability of existing.

Solé and Valverde [55] present a way of classifying graph topologies based on randomness, heterogeneity and modularity (figure 2.3). The authors note that most real-world complex networks can be characterized as highly heterogeneous, irregular, hierarchical graphs.

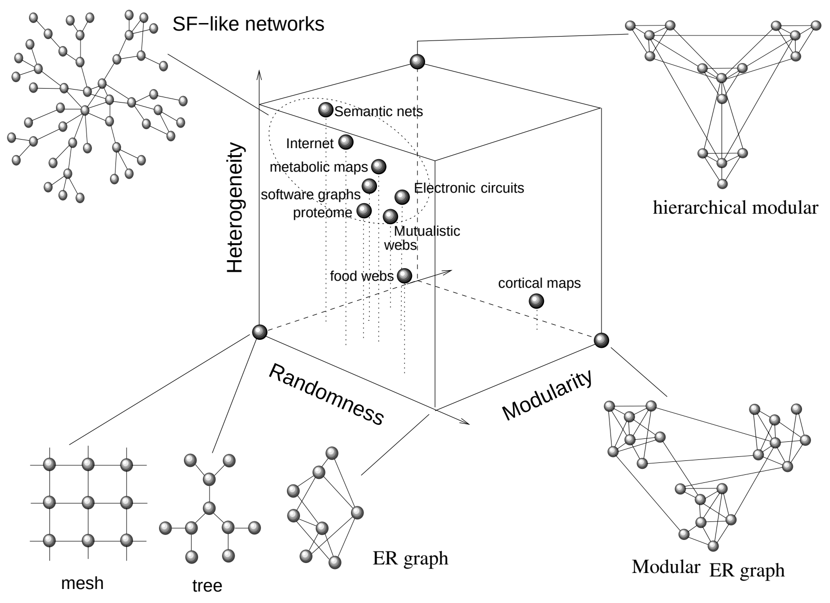


Figure 2.3: Zoo of complex networks. A classification of several graph types based on randomness, heterogeneity and modularity. Courtesy of Solé and Valverde [55].

The randomness dimension illustrates how much randomness is involved in modeling the network. This is done by comparing how similar the graph and its characteristics are to a uniformly random edge distribution of the same size. The Erdős–Rényi (ER) model [17], on the end of this axis, is used to model fully random graphs. The model dictates that for a set of vertices, there exists an edge between any two vertices with a given probability.

Graph heterogeneity distinguishes between degree distributions; a homogeneous distribution evenly scatters all edges among the vertices (such as with the Erdős–Rényi model), while in a heterogeneous distribution some vertices have a degree that significantly differs from the average. For heterogeneous graphs, preferential attachment causes a few hubs to be connected to most of the edges, which causes a power-law degree distribution. These networks are called scale-free (SF) networks and can be recognized by fitting a power-law function (or fitting a linear function on logarithmic axes).

From degree distribution alone, it is difficult to observe graph modularity. Graph modularity refers to the extent to which vertices can be divided into separate clusters (modules), where connectivity with vertices within the cluster is considerably stronger than to those in other clusters. The average local clustering coefficient can be used to quantify graph modularity. For each vertex, this coefficient is given by the proportion of links between the vertices within its neighborhood divided by the number of links that could possibly exist between them.

Figure 2.4 compares the degree distribution and clustering coefficient of three graphs of different heterogeneity and modularity and shows that degree distribution and average clustering coefficient vary accordingly. The clustering coefficient chart `Cc` implies that the low-degree nodes belong to very dense sub-graphs and, combined with chart `Cb`, we can observe that those sub-graphs are connected to each other through hubs.

The Erdős collaboration graph is an example of a scale-free real-world complex network. In this graph, most authors are connected via a short path even though no direct edge exists between the two. It also has a high clustering coefficient (significantly higher

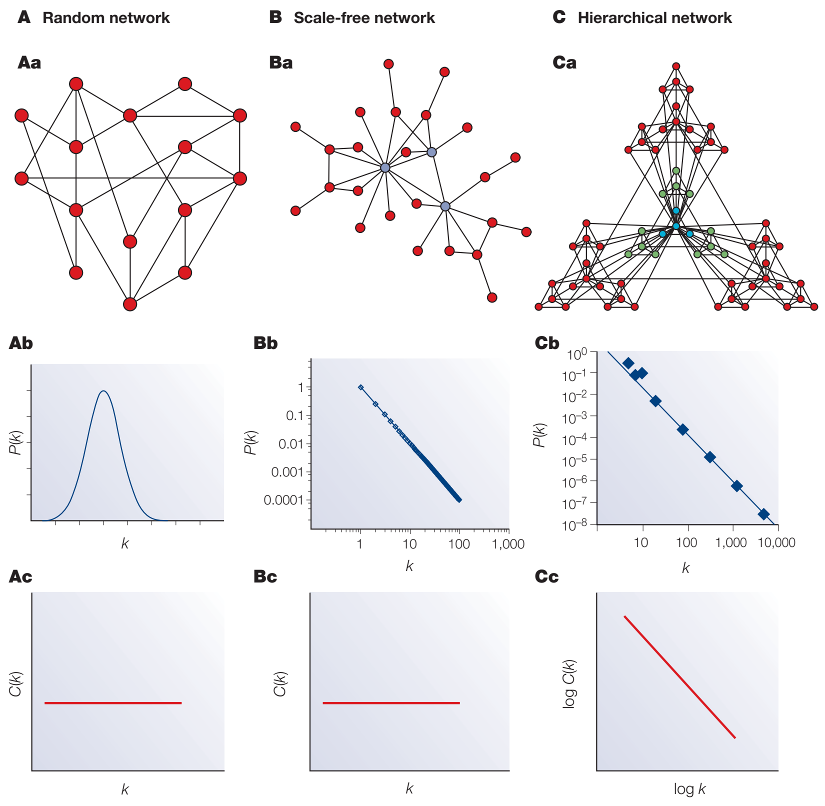


Figure 2.4: Comparison of (a) topology, (b) degree distribution, and (c) average clustering coefficient for (A) a random network, (B) a scale-free network, and (C) a hierarchical scale-free network. Note the logarithmic scales in `Bb`, `Cb`, `Cc`. Courtesy of Barabasi and Oltvai [5].

than its corresponding random graph), which means there is a tendency to form communities (for example, authors from the same department). Combined, these two characteristics are referred to as the small-world effect.

Watts and Strogatz [60] show that many real-world graphs exhibit the small-world effect and that this class of graphs should be classified somewhere between regular and random. Regular meshes are clustered, but have a high average path length. Random graphs have a short average path length, but do not show clustering.

The highly heterogeneous, irregular structure of complex networks make them inherently difficult to process efficiently in parallel. As graph algorithms are often data-driven, a scale-free degree distribution makes load-balancing difficult. On top of that, realizing a minimum graph cut with equal-sized subsets is an NP-complete problem [21], so partitioning to minimize communication is non-trivial as well.

2.3. PageRank

PageRank [50], named after its author, is a well known graph algorithm and is often used for demonstrating and testing graph-processing frameworks [14]. Although several accuracy and performance improvements have been proposed for the algorithm, we focus on the original algorithm as we are more concerned with the fundamental graph-processing challenges it exhibits than the actual output.

Traditionally meant for estimating the relative importance of a web page based on the number and quality of incoming hyperlinks, PageRank has proven its worth in more general settings as well (e.g., researcher impact, spam detection and trust networks [25]). As computation considers an entire dataset at once, it is a primary example of offline graph analytics. Contrary to online graph analytics (or graph querying), which desires fast response times, emphasis lays on high throughput.

Equation (2.1) shows the function for calculating the rank of a single vertex. V refers to the set of vertices, $E_{in}(x)$ refers to the set of vertices that have an edge pointed towards vertex x, and $D_{out}(x)$ refers to the degree of outgoing edges for vertex x.

$$PR(v_i, s) = \frac{1-d}{|V|} + d \times \sum_{w_i \in E_{in}(v_i)} \frac{PR(w_i, s-1)}{D_{out}(w_i)} \tag{2.1}$$

This equation models a *random surfer*; a visitor randomly following links on every visited page. For every page, the resulting rank represents the likelihood that any random surfer ends up on that particular page. In that sense, PageRank outputs a probability distribution where a higher value represents a more dominant (important) page. Each rank will have a value between 0 and 1 and the sum of all ranks equals 1. The damping factor d represents the chance that the random surfer will keep on following links. When no link is followed, surfing is assumed to continue on any other page.

Although the algorithm is simple in terms of theoretical description, PageRank exhibits one of the intrinsic challenges of graph processing well: computation is data-driven and parallelization is non-trivial. A second intrinsic challenge is the recursive nature of the equation. To practically approximate the result, evaluation ensues in iterations until the result converges.

We implement and study multiple PageRank implementations in order to gain insight into the graph-processing performance characteristics of different architectures. By its nature, PageRank is a more memory-intensive than computation-intensive algorithm, which makes it interesting for investigation of the influence of graph topology on performance. In essence, it can be regarded as the composition of two graph-processing building blocks: per vertex iteration over neighboring edges (information propagation) and a global reduction over all vertices (rank update and stop condition).

2.4. Related Work

With the Bulk Synchronous Parallel (BSP) paradigm, Valiant [58] introduced a simple model for designing and analyzing parallel systems. Key of the paradigm is that computation and communication proceed in synchronized iterations, referred to as supersteps. Cost of a superstep can then be determined using the sum of computation cost, communication cost, and the cost of a global barrier.

Williams and Parsons [61] recognize that BSP is not sufficient for heterogeneous parallel systems since it assumes all components have equal computation and communication abilities. The authors extend the model and introduce the Heterogeneous Bulk Synchronous Parallel (HBSP) framework.

HBSP takes into account the relative speed, bandwidth, and latency between all available components in a system. The concept of supersteps remains similar, but the cost calculation becomes a little more complex. In essence, it is the cost of the slowest worker (straggler) that determines the final cost, and the goal is to find a workload distribution that minimizes the final cost. BSP and HBSP abstractions can be used to reason about parallel graph-processing systems, but actual performance models will depend on (and vary between) algorithms and their implementations.

Using a simple BSP-based performance model for heterogeneous graph processing, Gharaibeh et al. [22] demonstrate that, theoretically, GPUs can be used to accelerate parallel graph applications. The relative computing power between processors is expressed through edges per second and communication is expressed through the number of partition-crossing edges (assumed to be an upper-bounded percentage of the number of total edges). In their demonstration, the authors use theoretical bounds of today's platforms for processing rates, memory space, and communication cost to show offloading 30% of a graph to the GPU can result in a theoretical speedup of up to $\sim 1.35\times$ for a reasonably sized dataset.

The authors also introduce a heterogeneous graph-processing framework called TOTEM that offloads part of the dataset to the GPU. The framework is used to verify the theoretical performance model (for PageRank and Breadth-First-Search) and demonstrate the real-world potential of GPUs as accelerators for graph processing. Dataset partitioning is done based on vertex degree; the (few) high-degree vertices are processed on the CPU, while the many low-degree nodes are offloaded to the GPU under the assumption that the GPU can process these more efficiently.

The performance model introduced by Gharaibeh et al. [22] shows potential, but ignores some graph-processing complexities in exchange for model simplicity. For example, the model assumes that the processing rate for a processor is constant and can be determined by a benchmark independent of the graph characteristics of the actual workload. In this work (inspired by the HBSP abstraction) we generalize and extend TOTEM's heterogeneous performance model to take into account the usage of multiple accelerators, as well as the graph characteristics of the dataset.

With StarPU, Augonnet et al. [1] take a more top-down approach to try and solve the problem of optimally scheduling tasks inside a heterogeneous system. Instead of working with performance models designed beforehand for the algorithm and datasets at hand, the StarPU framework provides a “black-box” task scheduler that uses execution heuristics from previous runs. In some cases, the authors note a speedup in a hybrid system that exceeds the sum of speed of the individual elements (because of the distribution of tasks with respect to the strengths of each processing unit). Currently, these scheduling heuristics are only based on the kernel, execution unit, and task size, which has proven not to be sufficient for graph processing. Our work focuses on finding better heuristics, such that future heterogeneous schedulers can also benefit graph algorithms.

3 Design and Implementation

In this chapter, we address RQ1: *how to evaluate the graph-processing performance characteristics of a processing unit?* To this end, section 3.1 proposes a novel benchmarking methodology on the basis of a PageRank case-study. Sections 3.2 and 3.3 discuss the design principles for such a benchmark.

Section 3.4 elaborates on implementation and describes each of our benchmark kernels in detail. Finally, section 3.5 provides an overview and succinctly lists all kernels.

3.1. Methodology

PageRank is a graph algorithm that is composed of two graph-processing building blocks: per-vertex iteration over neighboring edges (information propagation), and a global reduction over all vertices (rank update and stop condition). The actual computation is relatively simple, which makes PageRank well-suited to analyze the influence of graph topology on performance.

A lot of general-purpose graph-processing frameworks offer users a higher-level programming interface to implement graph applications. The “think like a vertex” (*TLAV*) paradigm (also referred to as *vertex-centric*), as introduced with the Pregel framework by Malewicz et al. [44], limits the scope of computation to a single vertex, thereby allowing fine-grained parallelization. As shown in the original paper, PageRank is one of the algorithms that maps elegantly to this model (algorithm 1 displays a simplified representation; note that Pregel does not allow for global gather/scatter operations, so a prefixed number of iterations coordinates the stop condition).

Algorithm 1 Pseudocode for Pregel’s Vertex-Centric PageRank

```
function ComputeVertexPageRank($M : Messages$)
    $Vertex.Value \leftarrow \frac{1-d}{|V|} + d\times$ Sum($M$)

    if $SuperStep < 30$ then
        for all $w_i \in Vertex.OutEdges$ do
            SendMessage($w_i, \frac{Vertex.Value}{Vertex.OutDegree}$)
        end for
    else
        VoteHalt( )
    end if
end function
```

Each iteration and for every vertex, the `ComputeVertexPageRank` program is scheduled for execution. Communication between vertices is message-based and synchronized (i.e., all messages sent at iteration i are processed in iteration $i+1$). This methodology of a conceptually parallel bulk execution of vertex-programs, followed by a global synchronization step between iterations (*supersteps*), derives from the Bulk Synchronous Parallel (BSP) parallelization pattern [58].

Nguyen et al. [49] show that, even though the higher-level programming interfaces are similar (i.e., vertex-centric), performance varies greatly between graph-processing frameworks, and the framework achieving the lowest runtime changes depending on the input dataset. Moreover, frameworks also diverge in their APIs and optimization strategies: a more restricted API allows for more lower-level framework optimizations that target data structures, data transfer, and task scheduling [14]. In order to compare hardware platforms on equal footing and in a structured manner, we propose the use of straightforward in-house PageRank kernel implementations with gradually varying design decisions. To allow for a certain extent of generalization, implementations should follow patterns generic to graph processing. Section 3.2 elaborates on the requirements and design decisions made regarding this aspect.

Next to "traditional" (multi-core) CPUs, we take GPUs into account. In big-data processing applications, GPUs are very often used as platform for offloading computation. In graph processing, too, the platform has proven its potential ([20, 23, 34, 67]). However, due to the intrinsic graph-processing challenges, implementing graph algorithms for the GPU is significantly more challenging. Section 3.3 discusses how the benchmark kernels should address the traits and implementation objectives per platform, as well as the challenges of heterogeneous environments, where a set of independent processors is exploited (e.g., exploiting distributed CPUs in a cluster or exploiting both the CPU and GPU on a single node).

3.2. Graph-Processing Considerations

McCune et al. [45] define TLAV-frameworks as software that supports iterative execution of user-defined vertex-programs over vertices of a graph. The authors identify four principle design decisions (interdependent *pillars*) that dictate how programs execute and utilize the underlying hardware: Timing, Communication, Execution Model, and Partitioning. However, the performance implications of different decisions for these components are unclear.

This section discusses, per graph-processing pillar, the requirements and design decisions (and subsequent implications) for our benchmark kernels. First, section 3.2.1 (Timing Model) elaborates on how vertex calculations are scheduled for execution. Section 3.2.2 (Execution Model) explains the decisions made regarding abstraction level and data flow. Section 3.2.3 (Communication Model) considers how program data is shared between entities. Finally, section 3.2.4 (Data Partitioning) focuses on the distribution of input data among workers.

3.2.1. Timing Model

The timing model portrays how vertices are ordered by the scheduler for computation. In the traditional BSP model, all entities are executed (conceptually, in parallel) in supersteps with a global synchronization barrier between supersteps. Computation follows a deterministic pattern, where calculation is performed on data from the previous iteration, and updates are exchanged between iterations. Per iteration, vertices can be scheduled in any (random) order, as the execution order does not affect the state of the program. This is *the synchronous timing model*.

Alternatively, *the asynchronous timing model* discards a global iteration barrier, allowing calculations to be performed on most recent data. In synchronous execution, each superstep takes as long as the computation time for the slowest vertex (also known as the *straggler* problem). For asynchronous execution, vertex execution order can be dynamically reorganized by the scheduler, which helps counter the problem of imbalanced workloads.

Figure 3.1 depicts the difference in execution flow for synchronous (upper part of the figure) and asynchronous timing models (lower part). Asynchronous execution shows more fine-grained synchronization without a global barrier. States update immediately and execution order is not fixed.

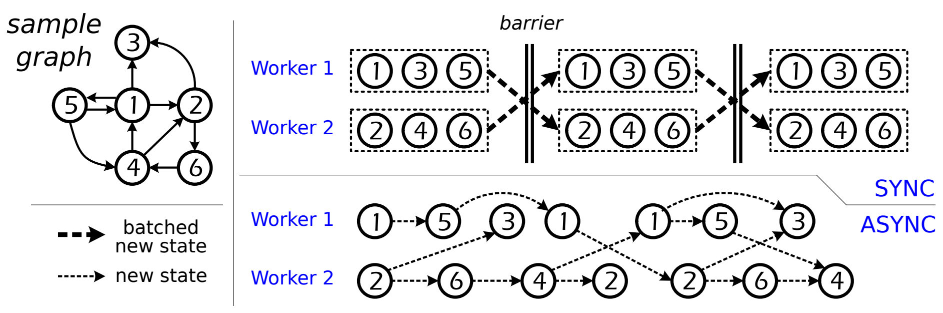


Figure 3.1: Visualization of the different execution flows for synchronous (upper part of the figure) and asynchronous timing models (lower part). Courtesy of Xie et al. [62].

PageRank is an algorithm with a high communication-to-computation ratio, which means it can benefit from optimizations to communication [62]. For synchronous implementations, communication patterns are predictable, so throughput can be improved through batch operations. Next to that, the nondeterministic nature of asynchronous execution makes performance modeling more difficult, so we restrict to implementing a **synchronous timing model**. We also assume all vertices to stay active during execution, as opposed to more intricate techniques that feature early termination for vertices that remain inactive ([40, 65]).

3.2.2. Execution Model

McCune et al. [45] define a framework's execution model as the style of algorithm implementation and flow of data. The authors differentiate between models using the number of distinct computation phases; instead of the single computation function used by Pregel (algorithm 1), a more general description of PageRank — using the Scatter-Gather model [52, 53, 56] (two phases) — would be: (1) summation of partial rank based on iteration over edges; (2) vertex rank update based on iteration over vertices.

The flow of data between vertex-programs can be characterized as either *push* or *pull*. In a push style flow, information flows from a vertex to its neighbors (for example, Pregel's message-based abstraction naturally maps to a push-based information flow). In a pull style flow, information flows in the reverse direction: a vertex reads data directly from its neighbors. Algorithm 2 illustrates a pull-based variant of algorithm 1.

Implementations are markedly different depending on the chosen execution model (e.g., several typical data structures for topology map to different execution models — see section 3.4.1). As it is unclear how these decisions impact performance [14, 49], we have decided to develop multiple benchmark kernels with different model parameters: both the **push** and the **pull** data flows are taken into account.

Algorithm 2 Pseudocode for pull-based Vertex-Centric PageRank

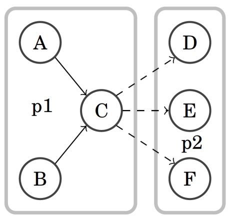
```
function ComputeVertexPageRank
    if $SuperStep < 30$ then
        $sum \leftarrow 0$
        for all $Neighbor \in Vertex.InEdges$ do
            $sum \leftarrow sum + \frac{Neighbor.Value}{Neighbor.OutDegree}$
        end for

        $Vertex.Value \leftarrow \frac{1-d}{|V|} + d \times sum$
    else
        VoteHalt( )
    end if
end function
```

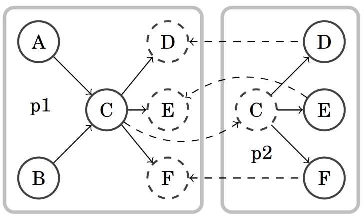
3.2.3. Communication Model

Parallel systems generally distinguish between two models of communication: shared-memory and message-passing. Although both models can emulate each other, the implemented communication model usually depends on the target platform, which natively implies a model — i.e., global address space systems rely on shared-memory communication, while distributed-memory systems rely on message-passing communication. In our case, we target distributed-memory systems and thus rely on message-passing.

However, in terms of vertex-centric graph processing, the communication model defines the *abstraction* method for sharing information between vertex-programs. Figure 3.2 depicts the communication pattern between two workers for the message-passing and shared-memory abstractions. The vertices with dashed outline in figure 3.2b represent so-called *ghost-vertices*; one worker is assigned ownership of the vertex, while other workers use replicas that are synchronized after each iteration.



(a) Message-Passing



(b) Shared-Memory

Figure 3.2: Comparison of TLAV communication models in distributed-memory systems for vertices (A–F) partitioned over two workers (p1–p2). Courtesy of McCune et al. [45].

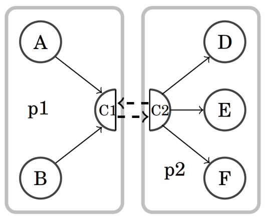


Figure 3.3: Communication pattern with vertex-cut [45].

For PageRank, we use a **shared-memory** abstraction for two reasons. First, because pull-based information flow is only possible with this model. Second, because it allows for an optimized push-based information flow: message-passing with sender-side message aggregation (locally aggregate messages in a ghost-vertex, then globally aggregate the values of ghost-vertices in their respective origin — resulting in the communication pattern depicted in figure 3.3).

3.2.4. Data Partitioning

Partitioning refers to the way in which data is divided between workers. Data volume is an intrinsic challenge of large-scale graph processing: the data exceeds the capacity of a single node, so a straightforward solution is to split the input over distributed memory. Traditionally, a good data distribution optimizes for equal processing time, while minimizing communication between workers. However, realizing a minimum graph cut with equal-sized subsets is an NP-complete problem [21]. In practice, with METIS[33] algorithms being the industrial standard [45], it is common to use pragmatic heuristics for graph partitioning. We use **METIS** as one of the methods for data partitioning.

METIS provides *edge-cut* partitioning, i.e., nearly equal size clusters of disjoint vertices, while minimizing the number of edges that span clusters (see figure 3.2a). For graphs with a power-law degree distribution, this partitioning might result in unbalanced workloads due to the imbalance in number of edges between clusters [26]. To counter this, *vertex-cut* partitioning balances edges between clusters, while minimizing the number of vertices that span clusters (figure 3.3). Blocks are balanced based on vertex size.

Partitioning data with METIS is a time-intensive and memory-intensive preprocessing operation. For that reason, we also implement a streaming **block-based** partitioning method. Figure 3.4 illustrates the methodology: 3.4a depicts an example graph as adjacency matrix where non-empty entries represent edges, 3.4b displays a sample edge-cut partitioning (vertices A–C are placed on worker $p1$, D–F on $p2$), and 3.4c shows a sample vertex-cut partitioning (non-diagonal blocks work with replicas of vertices).

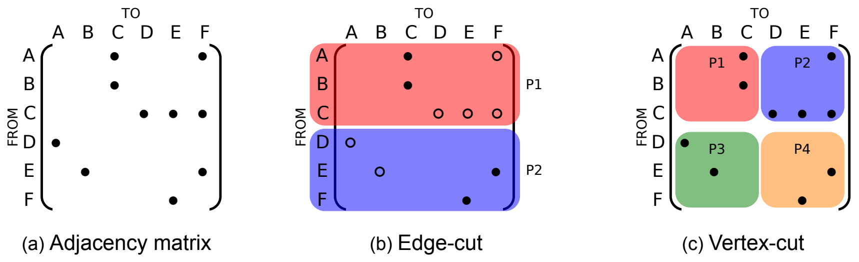


Figure 3.4: Visualization of block-based edge-cut and vertex-cut partitioning of an adjacency matrix. A dot in the adjacency matrix represents the existence of edge $(row \rightarrow col)$. In (b), a hollow dot represents an edge to another partition. In (c), non-diagonal blocks work with vertex replicas.

Note that this block-based partitioning method is naive, and does not approximate an optimal vertex-cut; blocks are simply balanced using the number of vertices. However, it does allow for more structured communication patterns, as the set of ghost-vertices is defined by the row and column of a block, thereby allowing utilization of (possibly optimized) data-broadcasting operations.

3.3. Platform Considerations

This section elaborates on the design objectives, and the decisions we made, regarding our parallel PageRank implementations for different platforms. As discussed in section 3.1, we target (compositions of) CPU and GPU platforms. To take maximum advantage of a platform, it is important to take its hardware architecture and performance traits into account. Section 3.3.1 and section 3.3.2 discuss these traits for CPU and GPU, respectively, and section 3.3.3 considers the heterogeneous aggregation of platforms where computation or memory access capabilities are unbalanced between processors.

3.3.1. CPU Traits

The parallelization technique used by multi-core CPU architectures can be classified as Multiple Instruction, Multiple Data (MIMD) under Flynn's taxonomy [19]. That is, different CPUs (cores) are working independently, and thus typically are executing different instructions and are accessing different memory locations at any given instant.

At the time of writing, modern commodity multi-core processors have up to 32 cores. Intel, in their guide to developing multi-threaded applications, recommend not using more software threads than the number of processing cores in the system [10]. For our application, this means that each software thread will execute a batch of vertex programs. To avoid false sharing, a cache consistency issue where two processors write to the same cache line, each thread should ideally write to consecutive block(s) of memory.

Nowadays, it is commonplace for multi-core CPUs to support SIMD instructions (Single Instruction, Multiple Data) in order to improve throughput for vector operations. Intel's Advanced Vector Extensions (AVX) instructions are able to process 256 bit registers. To benefit from these instructions, operations on consecutive memory are a prerequisite.

To summarize, the primary design objectives for graph processing on CPUs are: **limit the amount of threads** and operate on **consecutive blocks of memory** if possible.

3.3.2. GPU Traits

Contrary to CPUs, which are designed to execute a limited amount of threads at once, GPUs focus on highly parallel applications by processing thousands of threads concurrently. GPUs possess many more execution units than CPUs, but generally operate on a lower frequency. Execution follows a Single Instruction, Multiple Data (SIMD) pattern: a group of threads (warp) is physically executed in parallel on cores of a streaming multiprocessor (SMP), one instruction per cycle per warp.

This SIMD model differs from the model offered by the CPU SIMD intrinsics described in section 3.3.1. NVIDIA labels their parallel programming model for GPUs as Single Instruction, Multiple Threads (SIMT) [38]. With SIMT, each core is assigned its own thread with a corresponding context, which means that, although the executed instruction is similar for all threads in a warp, context may vary (i.e., registers, memory access, or flow path). This enables parallel processing of constructs that cannot be expressed with CPU SIMD intrinsics, such as indirect memory access and execution of conditional branches.

Another architectural difference between GPUs and CPUs is the limited implementation of memory caches on GPUs. CPUs are optimized for minimal latency by utilizing large caches for, e.g., instruction pre-fetching, out-of-order execution, and operations on memory. GPUs are optimized for maximal throughput, and try to hide latency through heavy multi-threading. Whenever a warp is waiting for memory access, the hardware-based scheduler can switch execution to another warp (time-slicing principle). This encourages the use of a large number of light-weight threads, preferably a multiple of the number of cores in an SMP.

Grouping of threads into warps is not only relevant to computation, but also to memory loads and stores. The SMP is able to coalesce memory requests issued by threads of a warp into as few transactions as possible. A single memory transaction can address up to 128 bytes, so threads in a warp should ideally work with neighboring memory in order to maximize bandwidth efficiency [11].

Exploiting the full capacity of the GPU for parallel PageRank is non-trivial. For vertex-centric programs, one of the most important performance challenges is keeping all cores busy as much as possible (i.e., we aim for high core occupancy). Whenever threads in a warp diverge, for example through conditional branches or loops of unequal number of iterations, only a subset of the cores is able to execute their instruction. A straightforward vertex-centric application would map the computation of **a single vertex to a single**

thread, but data-driven computation (i.e., edge iteration) in combination with imbalanced input graphs (many real-world graphs exhibit power-law degree distributions) results in thread divergence and therefore low occupancy.

To increase (theoretical) occupancy, another way of parallelizing PageRank is to divide the work of **a single vertex program over a full warp**, so each core iterates over part of the vertex edges. As added benefit, this iteration pattern allows for memory coalescing over edge data as well as vertex data. However, this does add synchronization overhead and results in low occupancy when the input graph has a lower average degree than the number of threads in a warp.

Liu and Schmidt [39] propose a lightweight work distribution algorithm in order to further combat workload imbalance. By atomically increasing a counter, **each warp dynamically determines the vertices to be processed**. We implement all three methods of work distribution.

To summarize, a primary design objective is to operate on **consecutive memory blocks** as much as possible. Secondly, work should be split into a large number of light-weight threads that spread load equally among threads in a warp, in order to **keep core occupancy high** and maximize data throughput. Finally, a CPU manages the coordination and invocation of GPU programs (kernels) in a system, but it is idle during kernel execution. This opens opportunities for sharing the workload between the two devices, thereby using a heterogeneous computing paradigm.

3.3.3. Heterogeneous Environments

Heterogeneous computing refers to the use of more than one kind of processor in a system, with the purpose of increasing performance or energy efficiency. We will refer to heterogeneous environments as environments with dissimilar processors (i.e., different clock speed or performance characteristics) or environments that exhibit Non-Uniform Memory Access (NUMA) characteristics (i.e., distributed CPUs or usage of external memory, where communication latency and bandwidth depends on location). For the sake of simplicity, we assume each independent processor to have an independent pool of local memory, and communication between separate processors to occur through message-passing.

Balancing workloads in heterogeneous environments is challenging, especially with data-driven computations such as PageRank and other graph-processing algorithms. In this regard, it is interesting to know the graph-processing performance characteristics of different platforms. With the ambition of defining these characteristics, we implement PageRank targeting only a CPU or a GPU, but also heterogeneous mixes, such as a **cluster** of multi-core CPUs and **CPU+GPU** on a single node.

As inter-worker communication is markedly more expensive than local memory operations (higher latency / lower bandwidth), a design objective for these implementations is to **optimize communication** between workers, for example, by combining messages and performing batch operations.

3.4. Design and Implementation of Parallel PageRank

Taking into account the design considerations from sections 3.2 and 3.3, this section will now discuss the design and implementation of our PageRank benchmark kernels. First, sections 3.4.1 to 3.4.3 explain the design and usage of data structures. Next, sections 3.4.5 to 3.4.7 explain implementation details for CPU, GPU, and heterogeneous platforms, respectively.

3.4.1. Data Representation for Graph Topology

Effectively storing graph topology is one of the most important aspects of high-performance graph processing. Mapping unique vertex identifiers to ascending indices allows efficient storage of vertex data in vectors. Edge data, however, is less straightforward. Adjacency matrices (figure 3.4a) allow for efficient querying of vertex adjacency, but are memory-inefficient for sparse graphs (i.e., graphs with low average degree). The number of edges between two vertices is also limited to one per direction.

As real-world graphs are generally sparse, and PageRank performs edge iteration (rather than edge querying), data storage using adjacency matrices is not a viable option. Depending on the type of edge iteration, we prefer different data representations for topology. Following is the list of storage formats we use, with descriptions and use cases.

From	**A**				**…**	**F**		
To	**A**	**B**	**C**	**D**	**…**	**D**	**E**	**F**
Edge	f	f	t	f	…	f	t	f

(a) Adjacency Matrix — $\Theta(|V|^2)$
Edge value is **t**rue or **f**alse

Edge	**0**	**1**	**2**	**3**	**…**	**7**	**8**	**9**
From	A	A	B	C	…	E	E	F
To	C	F	C	D	…	B	F	E

(b) Adjacency List — $\Theta(2 \times |E|)$

Vertex	**A**	**B**	**C**	**D**	**E**	**F**	**∅**
Idx	0	2	3	6	7	9	10

Edge	**0**	**1**	**2**	**3**	**…**	**7**	**8**	**9**
To	C	F	C	D	…	B	F	E

(c) Compressed Sparse Row — $\Theta(|V| + |E|)$

Vertex	**A**	**B**	**C**	**D**	**E**	**F**	**∅**
Idx	0	1	2	4	5	7	10

Edge	**0**	**1**	**2**	**3**	**…**	**7**	**8**	**9**
To	D	E	A	B	…	A	C	E

(d) Compressed Sparse Column — $\Theta(|V| + |E|)$

Figure 3.5: Partial memory layout for different graph topology representations of figure 3.4a with corresponding storage requirements.

Adjacency List

Storing edges in a list (dubbed *adjacency list*, figure 3.5b) prevents waste of memory and enables trivial edge iteration. Contrary to adjacency matrices, where vertex addition/removal is expensive as it requires a matrix resize, topology modification is relatively cheap.

However, performing vertex calculations through (the more fine-grained) concurrent iteration of edges requires synchronization before modifying vertex data (e.g., memory locks or atomic operations). Secondly, there is no easy way of finding and iterating over edges belonging to a single vertex without sorting the list first.

For GPUs, which are optimized for fine-grained parallelism, this is a viable representation of data. Edge vectors will be accessed consecutively, but the potential for coalescing (atomic) requests to vertex data depends on the ordering of the adjacency list.

Compressed Sparse Row

Sorting and creating an index for one of the two vectors of an adjacency list facilitates per-vertex edge iteration, and reduces storage requirements further (assuming $|V| < |E|$). Creating an index for the to-vertex list corresponds to compressing the adjacency matrix by storing non-empty cells consecutively in row-major order; hence, this format is referred to as “Compressed Sparse Row” (CSR, figure 3.5c).

Edge addition/removal is more expensive than with previous data structures. For *push-based* PageRank, however, an algorithm that works with outgoing edges and does not alter graph topology, this is a straightforward approach to storage. Note that push-based modification of vertex data does require atomic operations.

Compressed Sparse Column

Using Compressed Sparse Column storage (CSC, figure 3.5d) implies creating an index for the from-vertex vector in an adjacency list. CSC has the same storage requirements as CSR, but facilitates iteration of reversed edges. For *pull-based* PageRank, this is the storage format of choice. Atomic operations are not required, as each vertex-program propagates read-operations over edges and only modifies values associated to itself.

3.4.2. Data Representation for Computational State

For PageRank, the main computational state is expressed as one floating point value per vertex. Information flows through the system over graph edges. Because vertex identifiers are mapped to numerical indices, vertex state can be stored as a vector of floating point numbers. Messages can be trivially combined through mathematical addition, so we also store message state in a vector with one floating point value per (destination) vertex. When exclusive data access cannot be guaranteed and messages from multiple threads target the same vertex, atomic addition is required.

For pull-based PageRank, no (explicit) messages are sent between vertices. However, to guarantee synchronous execution of the algorithm without executing all vertex programs concurrently (which is physically impossible), an immutable copy of vertex values is used as read-only source of data. In practice, this means that the destination vector of the current iteration becomes the source vector of the next iteration.

3.4.3. Data Partitioning

As described in section 3.2.4, partitioning data over distributed memory is a necessity for processing large input data. A low-overhead, autonomous, and naive way of partitioning data is by using a hash function to determine the location of a vertex (edge-cut) or edge (vertex-cut). This allows graphs to be loaded independently by workers in a single-pass (streaming) fashion.

Illustrated by figure 3.4, our hash function simply takes a vertex identifier into account and partitions data in a round-robin fashion. Besides low overhead, this partitioning enables relatively straightforward ghost-vertices management, because ghost-data is also placed consistently throughout memory. Within a partition, data is stored with one of the previously mentioned data structures for graph topology.

Alternatively, we distribute data using a METIS partitioning. METIS aims to provide equally sized partitions with a minimal number of partition-crossing edges. This partitioning is (pre)processed once per graph, and can be reused for multiple applications. Workers then autonomously load their part of the graph based on a provided vertex mapping file (note: METIS only provides edge-cuts).

Again, workers use one of the previously mentioned data structures for local topology storage. Additionally, each worker maintains a dedicated list of inter-worker edges to synchronize ghost-vertices efficiently. Compared to block-based partitioning, communication volume is reduced. However, as memory access requires one step of indirection, management overhead is increased.

3.4.4. Computation Kernels

Following the Scatter-Gather model, PageRank is a composition of two building blocks: (1) computation of a partial rank based on iteration over edges; (2) vertex rank update based on aggregated information from the scattering phase (section 3.2.2). These building blocks will be referred to as the *computation kernels* of PageRank, and are implemented according to graph-processing considerations discussed in section 3.2.

Algorithm 3 Generalized PageRank Computation
Ⓢ Inter-worker synchronization task
Ⓒ PageRank computation task

```
function ComputePageRank($W$: Workers; $(V, E)$: Graph)
    for all $w \in W$ in parallel do
        $(V', E', deg) \leftarrow$ load_local_graph($V$, $E$, $w$)

        $src_v \leftarrow \frac{1}{|V|}$ for $v \in V'$
        $dst_v \leftarrow \frac{1}{|V|}$ for $v \in V'$

        repeat
            Ⓢ synchronize_ghost_vertices($origin \rightarrow ghost, update$)      ▷ Scatter
            Ⓒ $dst \leftarrow$ pagerank_edge_traversal($src, deg, V', E'$)       ▷

            Ⓢ synchronize_ghost_vertices($origin \leftarrow ghost, sum$)         ▷ Gather
            Ⓒ $err_w \leftarrow$ pagerank_vertex_apply($src, dst, DAMPING, V'$)  ▷
            Ⓢ $err \leftarrow \sum_{w \in W} err_w$                               ▷

            SWAP($src, dst$)
        until $err < EPSILON$

        Ⓢ $result \leftarrow$ collect_result($src$)
    end for

    return $result$
end function
```

Algorithm 3 presents a generalized Parallel PageRank computation. Note that the manner of kernel execution and data synchronization is dependent on the target platform and execution model. Every inter-worker synchronization task (depicted with Ⓢ) implies a computation barrier. For edge-cut data, all edges for a vertex are placed on the same worker, so only *one* ghost-data synchronization step is required; pull-based computation style requires the first synchronization (replication) step, while push-based requires the second (aggregation).

Note that we implement the actual PageRank algorithm using convergence detection, rather than Pregel's simplified algorithm which uses a preset number of iterations (resulting in another synchronization step). We also make sure vertices without outgoing edges ("endpoints") distribute their value over the full set of vertices, as per the original algorithm (not depicted in pseudocode). This guarantees an output vector with sum 1.0.

Algorithm 4 Push-Based Traversal Kernel

```
1: for all $v \in V'$ do
2:     for all $e \in v.OutEdges(E')$ do
3:         $dst_e \leftarrow dst_e \cup \frac{src_v}{deg_v}$
4:     end for
5: end for
6: return $dst$
```

Algorithm 5 Pull-Based Traversal Kernel

```
1: for all $v \in V'$ do
2:     for all $e \in v.InEdges(E')$ do
3:         $dst_v \leftarrow dst_v \cup \frac{src_e}{deg_e}$
4:     end for
5: end for
6: return $dst$
```

Algorithms 4 and 5 show reference PageRank edge traversal kernels for push and pull-based flows, respectively. Although the computation structure is similar between the two, memory access patterns are markedly different. Implementation details differ per platform and will be discussed in following sections.

Algorithm 6 Vertex Apply Kernel

```
1: for all $v \in V'$ do
2:     $dst_v \leftarrow \frac{1-DAMPING}{|V|} + DAMPING \times dst_v$
3:     $err \leftarrow err + |src - dst|$
4: end for
5: return $err$
```

Algorithm 6 shows the reference kernel for PageRank's vertex-value update. This kernel's implementation is relatively straightforward on all platforms, as its memory access patterns and computation structure are familiar.

3.4.5. CPU Implementations

Ideally, each kernel is programmed once, independent of its target platform. However, different platforms demand different optimization strategies. High-level cross-platform parallelization Application Programming Interfaces (APIs) exist, but they inherently provide a lesser ability of fine-tuning. Not knowing the effect of these tradeoffs in the context of graph processing, we implement all PageRank kernels for CPUs using three conceptually different interfaces: OpenMP, OpenCL, and MKL. The programming language of choice is `C11`, as it is the common denominator between all target APIs. This section will briefly discuss the concepts behind each implementation.

OpenMP

OpenMP is an open standard that defines a collection of compiler directives and library routines for straightforward parallelization of shared-memory `C/C++/Fortran` programs. The standard is well-established and adopted by most popular open source and commercial `C/C++` compilers.

The PageRank kernels using OpenMP are similar to their sequential counterpart, only adding explicit threading directives for guided data management and parallel execution, but a grasp of basic concurrency concepts is required to optimally place directives and guarantee proper execution without race conditions.

A single "`omp parallel for`" directive before algorithm 3 line 2 ensures threaded execution. The number of threads (workers) is statically set to the number of local graph partitions and the compiler is guided to vectorize inner loops using the "`omp simd`" directive (e.g., algorithm 6 line 1). Inter-worker synchronization is achieved through explicit barriers, atomic operations, and reduction directives.

OpenCL
OpenCL is an open standard for the development of parallel programs across a wide range of processors and hardware accelerators. It defines a set of library routines and a kernel programming language (subset of the `C` programming language) that enable the execution of arbitrary user programs (kernels) on varying processors (such as CPUs, GPUs, and FPGAs). Rather than relying on compiler adoption, this standard relies on vendor adoption. Every supporting vendor provides an OpenCL implementation that is optimized for the target platform. Compilation of user kernels is done at runtime and fully tailored for the executing hardware.

The PageRank kernels using OpenCL are implemented as vertex-centric OpenCL functions and enqueued for execution using vendor heuristics for work distribution (with `clGetKernelWorkGroupInfo`). OpenCL event wait lists are used for kernel synchronization.

MKL
The Intel Math Kernel Library (MKL) is a collection of vendor-optimized math routines. It can be used for implementing graph-processing algorithms in their algebraic formulation. MKL trades the ability of fine-tuning performance for a programming interface with a higher level of abstraction. Instead of allowing parallelization of arbitrary user constructs, the library offers a set of basic composable high-performance computation kernels. These kernels are specifically optimized for Intel processors, and support the CSR/CSC data structures discussed in section 3.4.1.

It should be noted that the MKL cannot be used to implement arbitrary custom kernels and, due to its limited set of available kernels, only a few graph algorithms can benefit from using this library. However, this PageRank implementation can be used as a performance reference point for the platform.

3.4.6. GPU Implementations
For the GPU, we use three different programming interfaces with concepts corresponding to their CPU counterparts: CUDA, OpenCL, and cuSPARSE. This section will briefly discuss the concepts behind each implementation.

CUDA
The CUDA (Compute Unified Device Architecture) framework is a parallel programming framework by NVIDIA that enables the use of CUDA-enabled (NVIDIA) GPUs as accelerators for general purpose processing. It is the most commonly used interface for writing arbitrary user programs for GPUs. User kernels are annotated with specific syntax and can be intertwined with “traditional” code. Contrary to OpenCL, user kernels are compiled beforehand and target a specific CUDA (hardware) version.

Compared to OpenCL, the PageRank kernels are implemented quite similarly. However, CUDA exposes more low-level GPU capabilities useful for graph processing, most notably more atomic operations, which allows for more straightforward/optimized implementations. Again, we use vendor heuristics for work distribution (with `cudaOccupancyMaxPotentialBlockSize`).

OpenCL
Although OpenCL targets both CPUs and GPUs and implementations can be reused, we implement additional kernels to exploit the GPU execution characteristics as described in section 3.3.2. More specifically, instead of having each execution unit process one vertex, we process one vertex per warp.

cuSPARSE
The CUDA Sparse Matrix (cuSPARSE) library is part of the CUDA framework and provides composable GPU-accelerated computation kernels for sparse data structures. The API is similar to MKL, but targets NVIDIA hardware rather than Intel's. Just as the MKL, cuSPARSE cannot be used to implement arbitrary custom kernels, so this PageRank implementation will be used as a performance reference point for the platform.

3.4.7. Heterogeneous Implementations

When targeting heterogeneous platforms, writing software becomes notably more difficult. Taking into account the performance characteristics of each platform and non-uniform inter-worker communication times make data partitioning and task scheduling non-trivial problems, especially in the area of graph processing.

We can observe the performance characteristics of each platform by benchmarking the respective specialized implementations, but we need to take into consideration the overhead of orchestrating different devices as well. We use two markedly different programming interfaces to be able to model these overheads: the low-level MPI and the high-level StarPU.

MPI
OpenMPI is a low-level Message Passing Interface (MPI) library that enables a cluster of processors to communicate through messages. The best way of transport is selected transparently (i.e., in order of preference, shared-memory, InfiniBand, or Ethernet). The library also offers intrinsics for coordination, such as global barriers and topology management.

For each node in the cluster, already-made kernels for the platform can be reused, while OpenMPI takes care of the data synchronization. Partitioning and task scheduling (selecting the best device for a task) is done manually.

StarPU
StarPU is a framework that allows programmers to exploit all processors in a heterogeneous system (i.e., all CPUs and GPUs), while abstracting away the task scheduling and data synchronization parts. For each "task", users attach one or multiple kernels (CPU, OpenCL, and CUDA are supported), and StarPU automatically schedules the task on the best suited device, and migrates data accordingly. It does this based on task size and heuristics from on earlier runs. In this regard, StarPU should automatically learn to execute a set of tasks optimally over time.

For the StarPU implementations, we reuse the CPU and GPU kernels. We let StarPU model performance based on task size, and execute tasks accordingly (this is known as StarPU's `dequeue model data aware ready` or `dmdar` scheduler). However, StarPU does not partition data to execute simultaneously on all processors, so we partition the input data manually, and create a StarPU task for each partition in order to create the possibility of parallel execution on multiple devices.

3.5. Overview

In order to evaluate the graph-processing performance characteristics of a processing unit, we propose a structural benchmarking method based on the four pillar design principles of graph-processing frameworks defined by McCune et al. [45]. To examine different platforms, we follow a bottom-up approach and prefer in-house PageRank implementations over preexisting graph-processing frameworks and libraries. This allows for a more structured comparison and a better understanding of the performance characteristics of each platform. The design requirements for each benchmark kernel, based on both graph-processing considerations (section 3.2) and platform considerations (section 3.3), are summarized in listing 3.1.

1. Must return correct result (not an optimized approximation).
2. Must work with directed graphs.
3. Must follow TLAV model (section 3.2):
   (a) synchronous timing model for deterministic execution.
   (b) shared-memory communication abstraction.
   (c) push or pull execution model depending on topology data structure.
   (d) edge-cut or vertex-cut data partitioning depending on topology data structure.
4. Should optimize for target architecture (section 3.3):
   (a) CPU kernels should limit the amount of threads and operate on consecutive memory blocks.
   (b) GPU kernels should exploit a large number of light-weight threads and operate on consecutive memory blocks.
   (c) heterogeneous implementations should reuse CPU/GPU kernels while taking care of data synchronization and scheduling.
5. Should use fitting programming interface for each platform (section 3.4).

Listing 3.1: Requirements for benchmark kernels.

Table 3.1 provides an overview of all kernel implementations. For each platform, we use multiple programming interfaces. First, we use the most commonly used interface for each platform: OpenMP for CPU, CUDA for GPU. Second, we use OpenCL to target both CPU and GPU with the same implementation. Finally, we use a vendor-optimized library for a performance reference point: MKL for CPU, cuSPARSE for GPU.

Kernels mostly differ on granularity, i.e., the the amount of work performed by a single execution unit. For each kernel, we implement both the push and pull execution modes (with CSR and CSC memory layouts respectively); this makes for a total of 30 kernels. For source code, the reader is referred to github.com/nielsAD/hgb.

Platform	Interface	Kernel	Granularity	Implementation notes
CPU	OpenMP	`OMP1`	edge-cut block	Parallel pragma for (outer) vertex loop, SIMD pragma for (inner) edge loop.
		`OMP2`	vertex-cut block	Use parallel reduction to merge vertex progress.
		`OMP3`	vertex-cut range	Thread cuts 2 vertices at most (range start/end), processes all edges of vertices inbetween.
CPU/GPU	OpenCL	`OCL1`	vertex	Use `clGetKernelWorkGroupInfo` to determine work-group size.
		`OCL2`	warp	Constant warp size of 32.
GPU	CUDA	`CUD1`	vertex	Use `cudaOccupancyMaxPotentialBlockSize` to determine block size.
		`CUD2`	warp	Vary warp size between 2 and 32 based on average degree.
		`CUD3`	warp	Use atomically increasing global counter to distribute vertices among warps.
HET	StarPU	`SPU1`	edge-cut block	Provide all `OMP`, `OCL`, `CUD` kernels as codelets. Use regression perfmodel, DMDA scheduler.
		`SPU2`	vertex-cut block	Use `STARPU_REDUX` to merge vertex progress.
	MPI	`MPI1`	edge-cut block	Use `OMP` or `CUD` kernels on each worker. Use `MPI_Allgather` to synchronize ghost-data.
		`MPI2`	vertex-cut block	Use `MPI_Reduce` to merge vertex progress.
		`MPI3`	METIS partition	Use `MPI_Alltoallv` to synchronize ghost-data.
CPU	MKL	`MKL`	graph	Vendor-optimized reference implementation. Internals unknown.
GPU	cuSPARSE	`CSP`	graph	Vendor-optimized reference implementation. Interals unknown.

Table 3.1: Overview of our in-house PageRank kernels for each platform.

4 Device Performance Evaluation

In this chapter, we address RQ2: *how to model the graph-processing performance of a processing unit with respect to dataset topology?* For this, we conduct an in-depth case study of the graph-processing performance characteristics of CPUs and GPUs. Sections 4.1 and 4.2 introduce our evaluation methodology, metrics, and datasets.

Sections 4.3 and 4.4 discuss our results. Consequently, we propose a graph-processing performance model that takes into account dataset topology in section 4.5. Finally, we conclude this chapter with our main findings in section 4.6.

4.1. Methodology

We start by evaluating CPUs and GPUs independently, with the goal of making performance observations for graph processing in general. In order to accurately compare and model performance in multi-architectural graph-processing systems, we introduce a new benchmarking suite, which is publicly available at github.com/nielsAD/hgb and can easily be repurposed for other architectures and graph-processing building blocks.

Popular graph libraries that deal with implementing data structures, dataset loading, preprocessing, and synthetic graph generation are generally not well-suited for large-scale and/or high-performance graph processing purposes. For example, iGraph [12] uses double-precision floating point data types in data structures where integers are preferred. Boost Graph Library [54] (including its parallel [27] derivative) uses nested containers to store edge lists (resulting in noncontinuous memory allocation), while NetworkX [29] and SNAP [36] trade in performance for flexibility, by using hash maps to store data.

Our benchmarking library exposes generic, lightweight, accelerator-ready graph data structures (section 3.4.1). Datasets can be loaded from common formats (e.g., the METIS graph format [33], the Matrix Market Exchange format [7], compressed (un)directed adjacency lists), or generated using one of the synthetic large-scale graph generators. A (distributed) partitioning interface is available for both block-based and edge-cut partitioning (using METIS). Additionally, the library provides visualization tools for results and datasets.

In this chapter, we focus purely on the performance of PageRank edge traversal on isolated devices. Before execution, all datasets are converted to directed graphs, and stripped of any unconnected vertices. Benchmark timings are averaged over 10 iterations, after dataset initialization and a warm-up iteration. All experiments are run on commodity hardware of the DAS-5 cluster [3], described in table 4.1 and compiled with software outlined in table 4.2. An overview of the examined kernels is available in section 3.5. Specifically, we focus on the OpenMP (OMP), OpenCL (OCL), and CUDA (CUD) kernels.

We defined our main metric of interest to be traversed Edges Per Second (EPS). Measuring throughput, this metric normalizes execution time against dataset size ($|E|$), thereby scaling results to a common scale that allows for performance comparison between different datasets.

	CPU	GPU
Architecture	Intel Haswell	Nvidia Maxwell
Model	Xeon E5-2630v3	GTX TITAN X
Core Frequency	2.4 GHz	1.0 GHz
Core Count	8	3072
Memory	32 GB	12 GB
Launch Year	2014	2015

Table 4.1: Experiment hardware specification.

Software	Version	Options
GCC	4.9.3	`-march=native -Ofast`
CUDA	10.0.130	`-Xcompiler "-march=native" -O3 -use_fast_math`
MKL	11.2	`-DMKL_ILP64`
CentOS	7.4.1708	

Table 4.2: Software and corresponding compilation options used for experiments.

4.2. Datasets

We benchmark using a mix of synthetic and real-world datasets. The advantage of using synthetic graphs is that datasets can be generated to arbitrary sizes, while maintaining a similar structure. Real-world datasets are included for verification purposes.

4.2.1. Synthetic Graphs

We make use of five types of synthetic datasets, with degree distributions ranging from regular to scale-free. For the sake of reproducibility, all graphs are generated using a predefined random seed. Figure 4.1 provides an overview, with visualizations for example graphs of equal size.

Regular

In a regular graph (REG), each vertex has the same constant degree. To generate these graphs, we built a simple generator where we connect each vertex to its k consecutive neighbors, resulting in a graph where the in-degree, out-degree, and memory access patterns are the same for each vertex.

Erdős–Rényi

Erdős and Rényi [16] (ER) describe a random graph model where each edge from a fully connected graph is included with equal probability, independent from every other edge. This results in a binomial degree distribution, with an average degree corresponding to $\frac{|E|}{|V|}$ and relatively small tails on either side.

ER graphs have an inherently low clustering coefficient and no heavy-tailed degree distribution. However, the balanced edge distribution in combination with randomized memory access patterns make them interesting subjects for analysis.

Triangular Erdős–Rényi
By keeping the lower triangular of the adjacency matrix of an Erdős–Rényi random graph (TER), i.e., by removing all edges (v_1, v_2) where $(v_1 > v_2)$, we generate a graph with semi-uniform degree distribution. This introduces workload imbalance into ER graphs, although not heavy-tailed.

Preferential Attachment
By assigning each vertex an exponentially decaying preference of attachment (PA), we generate a random graph with small diameter and log-normal degree distribution. This is a heavy-tailed distribution, but not quite as “unbalanced” as power-law.

Kronecker
Finally, we use the Kronecker (KRO) random graph model, described by Leskovec et al. [37], to generate hierarchical scale-free graphs with a power-law degree distribution. In this model, edges are distributed recursively among four equal-sized partitions with unequal probability.

Using the Graph500 [47] initial matrix, the dataset is scaled up to desired size by repeatedly applying the Kronecker product. In our benchmark, we consider this model to be the most accurate approximation of real-world complex graphs.

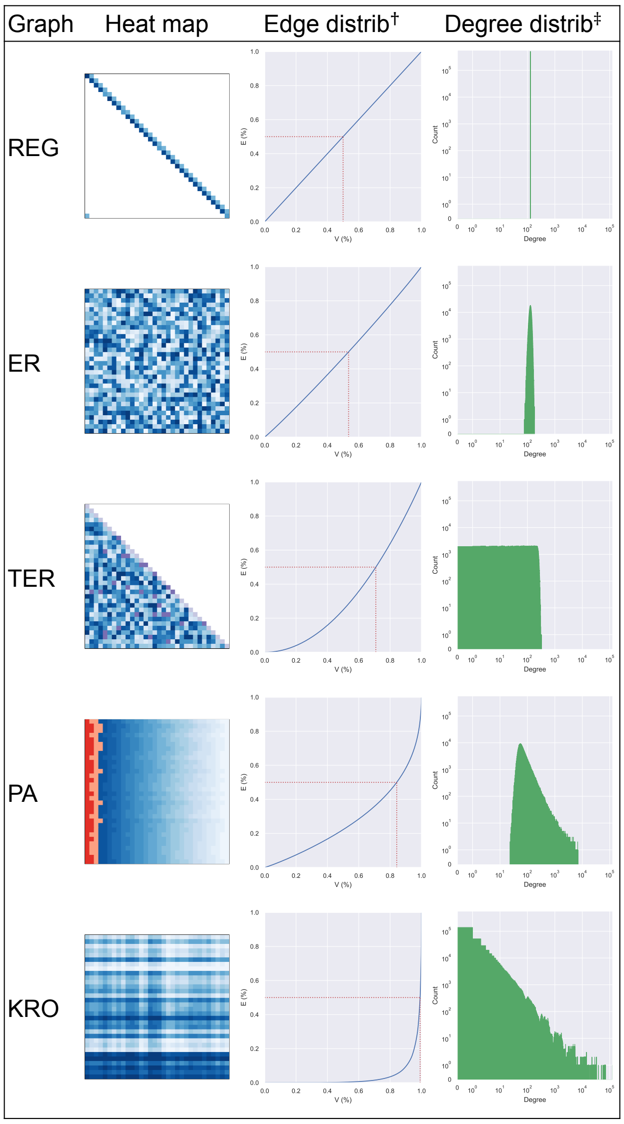


Figure 4.1: Edge distribution of synthetic benchmark graphs.
† Dotted red line marks the percentage of vertices needed to process 50% of edges.
‡ Graph sizes are similar, axes are logarithmic and shared.

4.2.2. Real-world Graphs

Table 4.3 gives an overview of all used real-world benchmark datasets. We use a selection of graphs with varying density and an average degree ranging from 2 to 102. The street (OSM) and numerical (BUB) graphs are quite regular, while the other datasets are heavy-tailed. Figure 4.2 shows that the citation (CIT) graph stands out due to the large number of nodes with only incoming edges, while the autonomous system (AS) and social (SOC) graphs display the longest tails.

Name	Dataset	Class	$\lvert V\rvert$	$\lvert E\rvert$	$\overline{D}$	δ_D	d	P_{50}	P_{95}	P_{100}
OSM	europe_osm [2]	Street	50.91	108.11	2.12	0.48	0.04	2	3	13
BUB	hugebubbles-00020 [2]	Numerical	21.20	63.58	3.00	0.03	0.14	3	3	3
CIT	cit-Patents [35]	Citation	3.78	16.52	4.38	7.78	1.16	2	16	770
EDU	wb-edu [24]	Web	9.45	55.31	5.85	20.62	0.62	2	20	3841
WIKI	wikipedia-20070206 [24]	Web	3.52	45.01	12.81	33.18	3.63	2	50	7061
AS	as-Skitter [35]	Topology	1.70	22.19	13.08	136.86	7.68	5	37	35455
SOC	soc-LiveJournal1 [35]	Social	4.84	68.48	14.13	36.00	2.92	4	57	20292
WWW	arabic-2005 [8]	Web	22.74	631.15	27.75	78.79	1.22	15	82	9905
COL	coPapersCiteseer [2]	Collab	0.43	32.07	73.88	101.27	173.45	39	257	1188
HOL	hollywood-2009 [8]	Collab	1.11	112.75	101.83	275.31	91.51	29	421	11467

Table 4.3: Real-world dataset statistics. $|V|$ – number of vertices (10^6). $|E|$ – number of edges (10^6). $\overline{D}$ – average degree. δ_D – degree standard deviation. d – link density (10^{-6}). P_{00} – degree percentile.

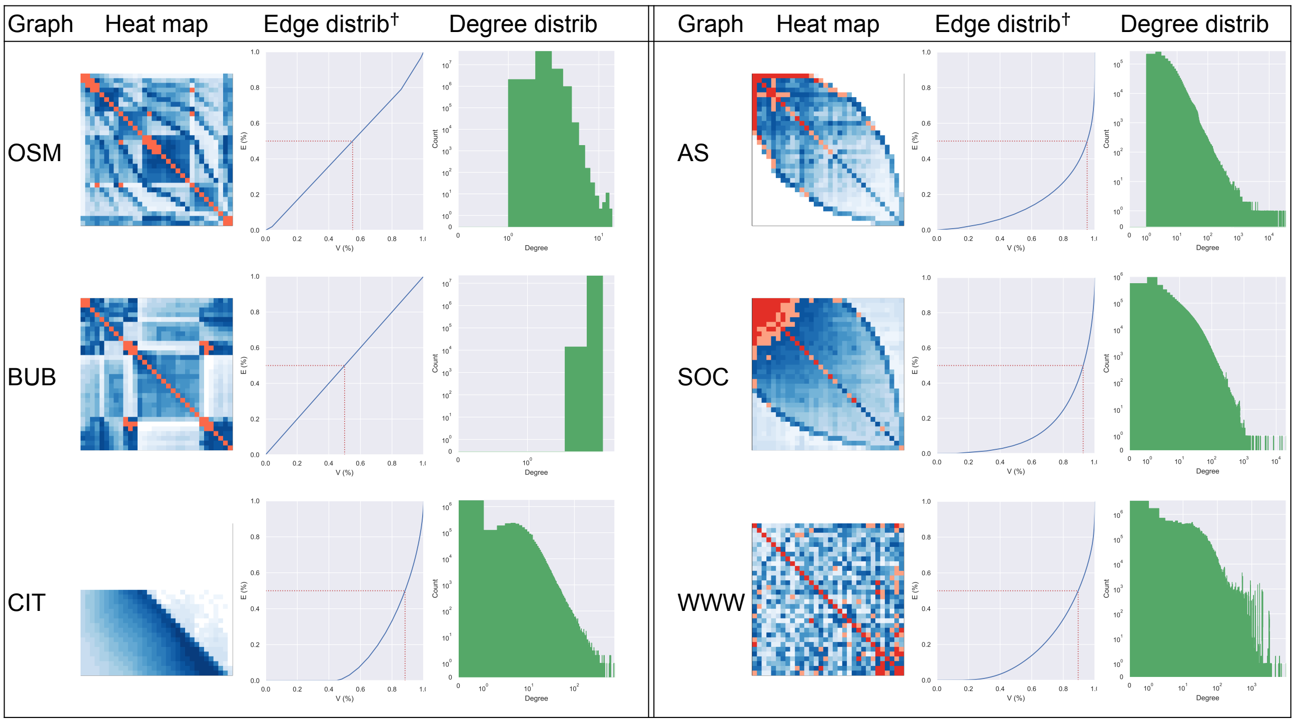


Figure 4.2: Continued on next page.

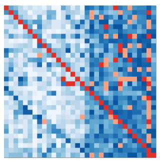
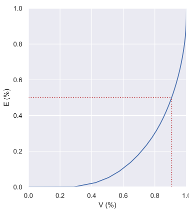
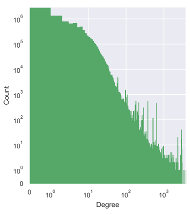
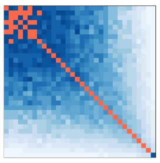
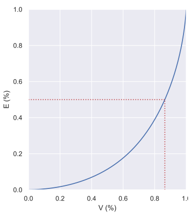
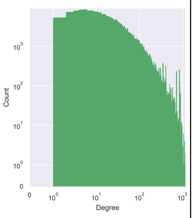
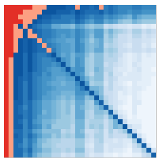
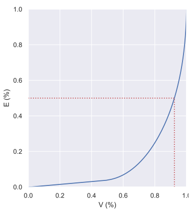
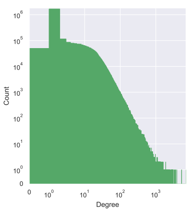
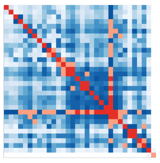
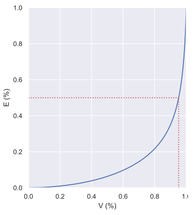
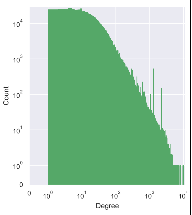
Graph	Heat map	Edge distrib†	Degree distrib	Graph	Heat map	Edge distrib†	Degree distrib
EDU				COL			
WIKI				HOL			

Figure 4.2: [continued] Edge distribution of real-world datasets.
† Dotted red line marks percentage of vertices needed to process 50% of edges.

4.3. Kernel Performance

Figure 4.3 shows the performance for each kernel over all datasets in traversed edges per second. At first glance, it is clear that results vary significantly; the OMP3 kernel ranges between 0.3 and 9.3 billion edges per second ($\mu = 1.5$, $\sigma = 1.8$), CUD3 between 0.9 and 57.1 ($\mu = 8.7$, $\sigma = 10.4$). The results are a clear indication that using a mean EPS number per kernel is not suited for accurate performance modeling.

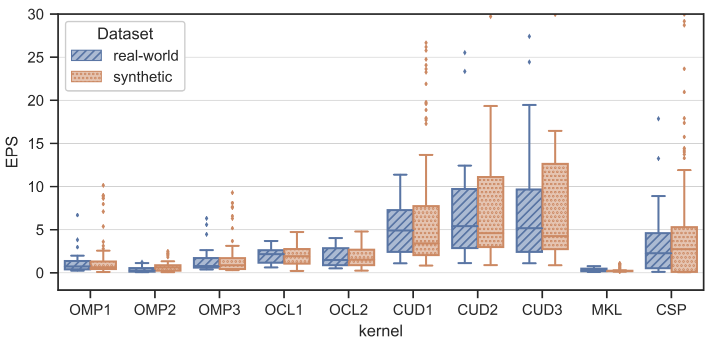


Figure 4.3: Performance in 10^9 traversed edges per second over all datasets, push and pull data flow. We have 20 real-world and 240 synthetic data points per box plot. There are 15 outliers above $30 * 10^9$ EPS cut off, all REG graphs. OCL kernels are executed on the GPU.
Synthetic graphs: $\left\{|V| \simeq 1M, \overline{D} \in (1, 2, 4, 8, 16, 32, 64, 128)\right\} \cup \left\{|V| \in (250k, 500k, 1M, 2M, 4M), \overline{D} \simeq 32\right\}$.

We can make three general observations from the data in figure 4.3. First off, GPU kernels perform better than CPU kernels overall, but their performance varies more between datasets. OpenCL kernels seem to be lacking in performance in comparison to CUDA. This can be attributed to (1) OpenCL missing some essential feature support (e.g., atomic floating point operations); (2) better optimized CUDA kernels (e.g., dynamic warp size); (3) compiler optimizations. However, the performance difference is quite significant for similar implementations. Executing OCL1 on CPU (not depicted in the figure) does give similar performance to OMP1 as expected. While OpenCL is, in theory, an ideal framework for heterogeneous programming, we find that writing programs for OpenCL is more convoluted than for its platform-specific counterparts, and using naive implementations seemingly results in a significant performance penalty.

Secondly, figure 4.3 confirms the relation between our real-world and synthetic datasets: the synthetic datasets range from best-case to worst-case scenario. Measured performance for synthetic datasets is indicative for the performance of real-world datasets, but not necessarily representative. Using only synthetic benchmarks will give proper performance boundaries, but extrapolating conclusions for real-world scenarios should be done with caution. Especially the better-performing synthetic graphs, such as REG, might not be representative for real-world scenarios.

Finally, we note that our kernel implementations are on par or outperform the vendor-optimized implementations MKL and CSP.

4.3.1. Graph Difficulty

To gain better insight into a kernel's performance characteristics, it is clear we have to examine results on a per-graph basis rather than yielding to averages. Figure 4.4 shows the performance of the best performing kernel for each device per dataset.

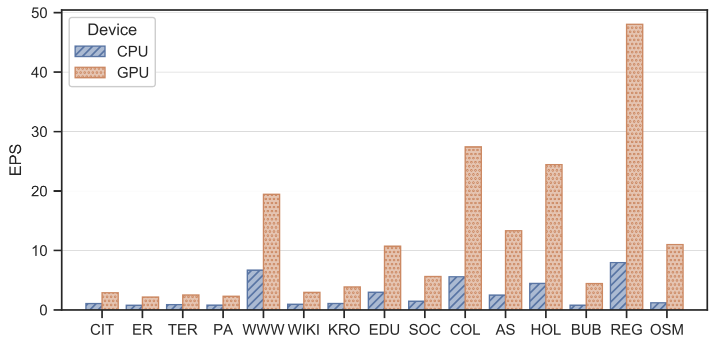


Figure 4.4: Performance of best performing kernel in 10^9 traversed edges per second. Datasets on horizontal axis are ordered based on speedup of GPU over CPU. Synthetic graphs: $\left\{|V| \simeq 4M, \overline{D} \simeq 32\right\}$.

A few datasets stand out immediately: WWW, COL, HOL, and REG graphs perform very well on both the CPU and the GPU. This is an interesting observation, as these graphs do not necessarily have a similar arrangement. Table 4.3 shows that these graphs have the highest average degree among the real-world datasets. However, our synthetic datasets all have the same average degree, but do not display comparable performance, so this cannot be the only indicator.

Figures 4.1 and 4.2 show that the best-performing graphs have a lot of edges centered around the diagonal of the heat map, indicating relatively high clustering. It is likely that hardware caches have a considerable impact on performance here. We also observe that ER is among the worst-performing graphs, even though the workload for each vertex is normally distributed (and thus roughly similar). Erdős–Rényi graphs are known to exhibit low clustering, which adds to the suspicion of caches having a substantial impact.

Figure 4.4 displays a performance disparity between datasets, with some graphs being inherently more difficult to process than others. Figure 4.5 shows that relative performance between the CPU and the GPU exhibits additional disparity. The figure shows that the GPU consistently performs better, but speedup ranges between $2.7\times$ and $9.2\times$.

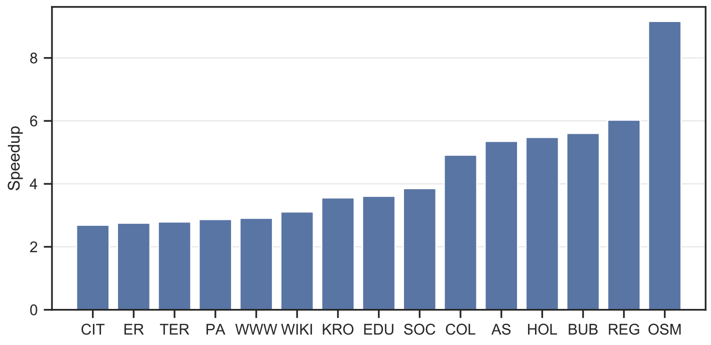


Figure 4.5: Speedup of the best GPU kernel compared to the best CPU kernel for all datasets. Average: $4.3\times$

The relatively high speedup for the OSM and BUB graphs on the GPU — and low speedup for CIT — could possibly be attributed to low warp divergence due to a very narrow degree distribution. As expected, the GPU thrives when most vertices have the same amount of edges.

There does not seem to be a relation between speedup and performance on either the CPU or the GPU; a high performance on the CPU does not necessarily guarantee higher (or lower) speedup on GPU. This confirms the suggestion that there is no universal graph difficulty hierarchy, but rather that such a hierarchy is bound to device and kernel.

4.3.2. Workload

To examine the impact of data size on device performance, figure 4.6 shows the kernel performance for synthetic graphs with varying vertex set size and varying edge set size. In general, we observe a slight increase in performance with growing average degree for most kernels. This can likely be attributed to the usage of SIMD instructions on CPU and better warp occupancy on GPU. REG graphs stand out by benefiting the most from having more edges per vertex, while other graphs stabilize around a lower limit, probably a memory bottleneck.

We observe a slight decrease in performance when varying the vertex set size while keeping the average degree constant. For all GPU kernels, there is a tipping point around the same mark ($|V| \gtrapprox 2^{19}$) after which performance drops significantly. Again, hardware caches seem to have notable influence here. REG graphs are designed to be cache-friendly and exhibit stable performance independent of the vertex set size.

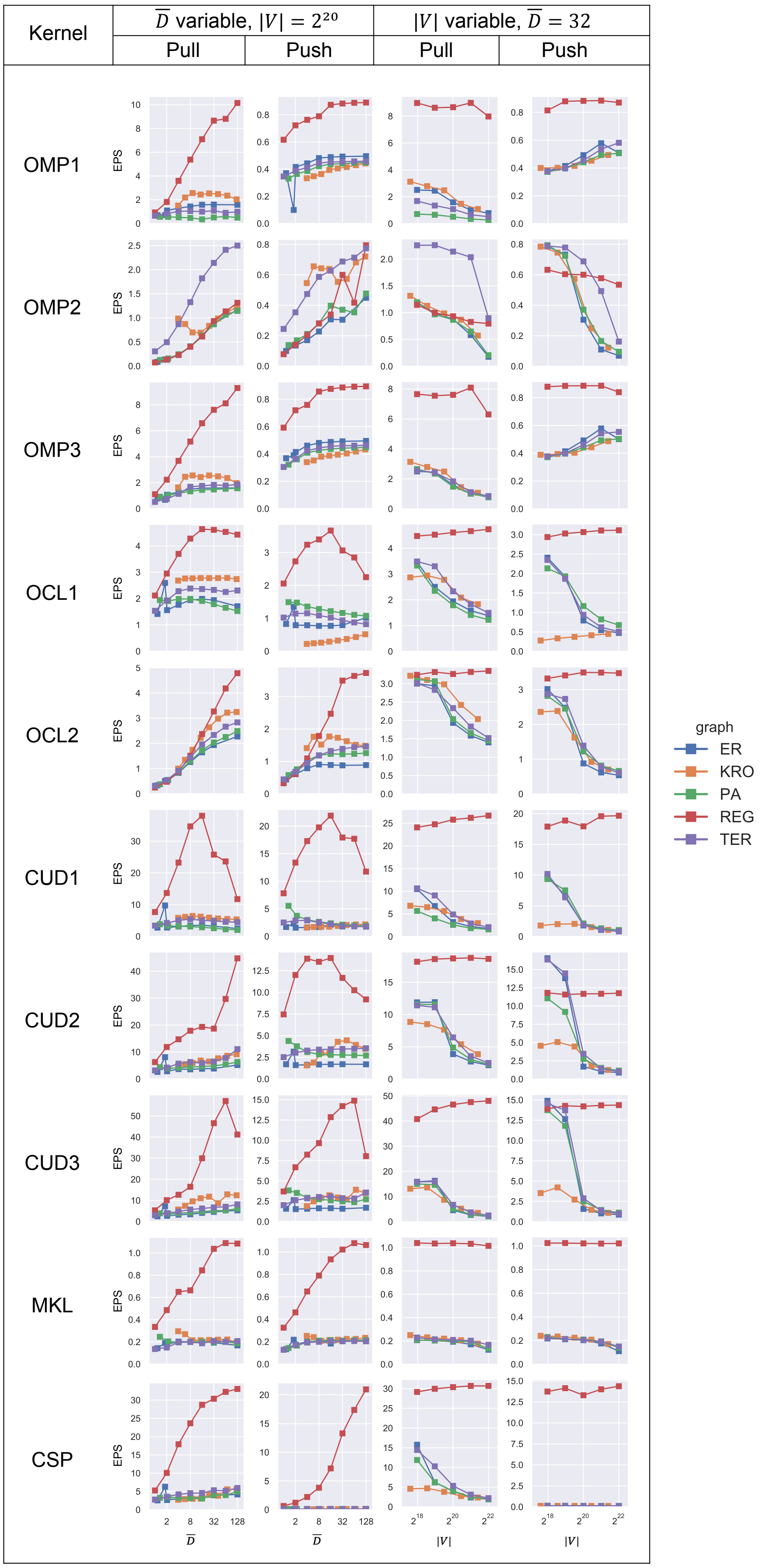


Figure 4.6: Kernel performance in 10^9 traversed edges per second for synthetic datasets with varying sizes. Note that x-axes are similar, y-axes are not.

4.3.3. Hardware Cache

After examining the well-performing datasets in sections 4.3.1 and 4.3.2, we suspect hardware caches have a significant influence on kernel performance. Table 4.4 shows the GPU cache hit rate for CUD3 on several graphs, and confirms a correlation between cache hit rate and edge processing speed; ER graphs display a cache hit rate of around 15% compared to 63% for REG graphs. In all cases, most warp stalls are caused by delayed memory operations.

Kernel	EPS	Cache Hit Rate	Memory Stalls†	Warp Efficiency‡
ER	2.1	14.64%	91.90%	62.15%
OSM	9.3	50.00%	87.84%	100.00%
WWW	19.5	63.29%	83.79%	58.27%
REG	48.1	62.84%	79.17%	79.17%

Table 4.4: GPU cache hit rate and warp efficiency for CUD3 pull kernel.
† Percentage of warp stalls occurring because a memory operation cannot be performed.
‡ Ratio of the average active threads per warp.

By introducing variable stride to the synthetic REG datasets, we can reduce its cache efficiency and benchmark the effects of cache hit ratio on kernel performance. Adding stride to only the destination vertex of an edge is a structured permutation of REG, which means that the altered dataset depicts the exact same graph as the original, but neighboring vertices will be scattered throughout memory as stride increases. Figure 4.7 shows performance for OMP3 and CUD3 while varying stride. As stride increases (and cache hit rate decreases), edge processing speed drops significantly for both kernels, indicating that cache efficiency is an important performance indicator for both CPU and GPU.

In push-based execution, higher cache hit rate increases the risk of false sharing, where different threads modify memory on the same cache line, thereby invalidating the cache line and forcing a reload. However, performance on CPU is also negatively affected by increasing the dataset stride, while performance on GPU remains stable.

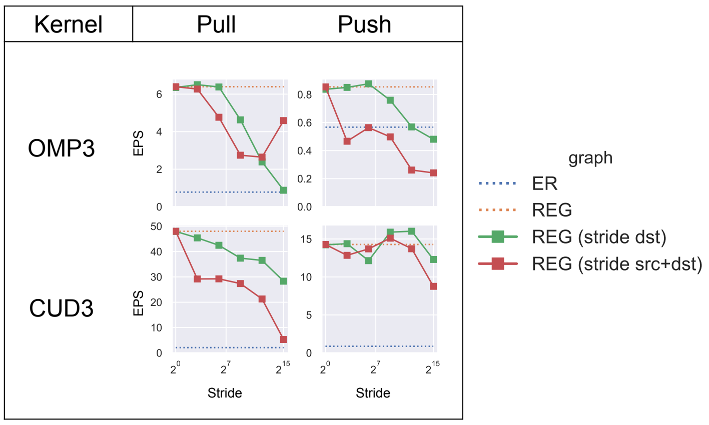


Figure 4.7: Kernel performance in 10^9 traversed edges per second for REG graphs with varying stride. Graph size: $\left\{|V| \simeq 4M, \overline{D} \simeq 32\right\}$.

Cache behavior is difficult to model due to the growing complexity of modern processors, which feature optimization techniques such as out-of-order execution, pipelined execution, and speculative execution; they all interact with the hardware caches. Figure 4.8 plots performance against a simple dataset cache hit ratio estimation. The estimation is based on sequential kernel execution with 256KB LRU cache and 64B cache lines (based on the CPU's secondary cache size).

It is interesting to note that even a grossly simplified estimation shows a clear linear correlation between cache hit ratio and performance.

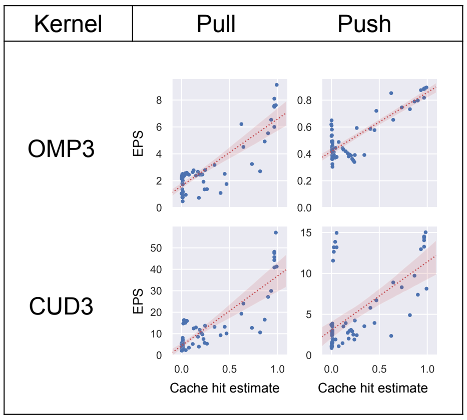


Figure 4.8: Kernel performance in 10^9 traversed edges per second for estimated cache hit ratio with linear trend line and 95% confidence interval. Includes all synthetic and real-world datasets.

4.3.4. Data Flow

Figure 4.6 shows a big performance difference in favor of pull-based data flow for all kernels. This is to be expected, as push-based kernels require synchronization (and thus, incur additional overhead) to ensure exclusive access while sending neighbor data. However, a common push-based optimization is to only process the set of "active" vertices. That is, the set of vertices that has been sent a message in the previous iteration. In contrast, pull-based kernels are more efficient but unaware of any neighbor updates, so there might be an overhead in checking for changes.

In our kernel implementations, we process 100% of the vertices in every iteration as a worst-case scenario. Figure 4.9 shows that for push-based execution to be competitive with pull-based, the size of active vertices would have to be less than 27% on average for the CPU and 40% for the GPU. For PageRank, this condition is often reached after a limited number of iterations.

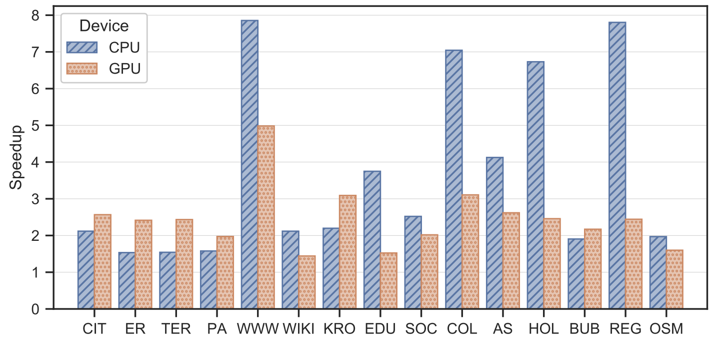


Figure 4.9: Speedup for pull over push data flow. Average: CPU $3.7\times$, GPU $2.5\times$

For the CPU, pull-based execution is considerably faster for the WWW, COL, HOL, and REG datasets. These are the same graphs that stood out in figure 4.4 as well-performing datasets. However, this performance appears to be mostly limited to pull-based execution. In that sense, pull-based execution performance does not correlate to push-based performance. Switching to push-based execution on GPU has slightly less of an impact.

4.4. Data Transfer

GPU kernels operate on device memory rather than host memory, which means there is an additional data transfer required to load the dataset in the device memory. Although a single kernel execution is faster on the GPU, the additional transfer overhead might result in faster algorithm completion on the CPU. Figure 4.10 depicts the data transfer overhead in terms of kernel iterations. That is, by dividing the data transfer time to the iteration time on each device, we effectively quantify the data transfer in number of executed iterations. The figure indicates that most datasets can be transferred before a single CPU kernel iteration can complete.

However, looking at the cost relative to the computation time on the GPU, it is clear that data transfer can take up a significant chunk of time. For example, data transfer will take up the majority of algorithm completion time for REG graphs if there are less than 16 iterations required for convergence.

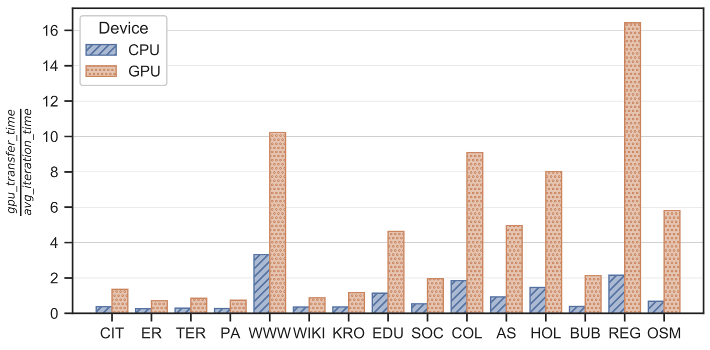


Figure 4.10: GPU data transfer overhead in terms of kernel iterations on the CPU or the GPU.

Figure 4.11 illustrates the GPU vs CPU speedup when taking into account the initial GPU data transfer cost. Execution on CPU is occasionally faster for a small number of iterations, but transfer cost is amortized as the number of convergence iterations grows.

The number of convergence iterations is dependent on algorithm and dataset, but it is safe to assume offloading computation to GPU is generally worth it. After 5 iterations, GPU performs similarly or better than CPU for all datasets.

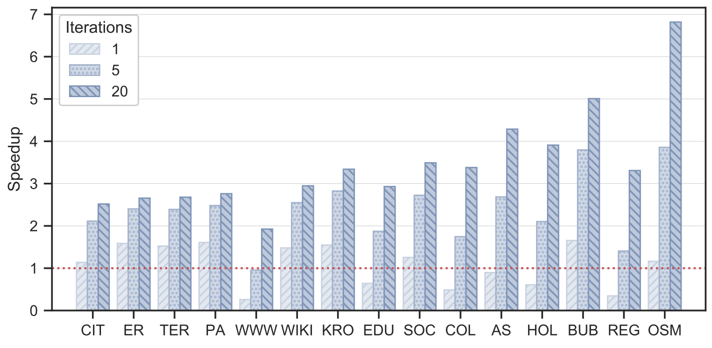


Figure 4.11: Theoretical speedup for best GPU kernel over best CPU kernel after taking into account initial data transfer cost and number of convergence iterations.

4.5. Performance Modeling

A simple model to predict algorithm runtime T for graph G on device d is illustrated in equation (4.1), where M_d is the transfer cost, S is the number of supersteps, and P_d represents the processing speed for device d in edges per second (assuming $|E_G| \gg |V_G|$).

Assuming P_d is a linear function that takes the input graph as parameter, we propose the estimation of P_d with equation (4.2), where EPS_d is the lower bound processing speed of device d, EPS'_d the (additional) cache-optimized processing speed, and $H(G)$ represents the cache hit rate (as estimated in section 4.3.3).

$$T_d(G) = M_d(G) + \sum_{s=1}^{S} \frac{|E_G|}{P_d(G)} \tag{4.1}$$

$$P_d(G) = EPS_d + H(G) \times EPS'_d \tag{4.2}$$

$H(G)$ is notably determined by how a dataset is instantiated in memory. Ideally, this value would be defined in terms of graph metrics. The local clustering coefficient ($\overline{C}$) and assortativity coefficient (r) capture the nature of connectivity between vertices, and are primary candidates to estimate cache-friendliness of a graph. Table 4.5 shows the Pearson correlation coefficients between a selection of graph metrics, the estimated cache hit ratio, and actual kernel performance. The local clustering coefficient stands out as only graph metric with a strong linear correlation to kernel performance.

	$H(G)$	EPS_{OMP3}	EPS_{CUD3}
$H(G)$	–	0.86	0.86
$\overline{C}$	0.84	0.75	0.76
r	0.48	0.24	0.28
P_{50}	0.15	0.42	0.33
$\|V\|$	0.11	−0.06	−0.05
$\|E\|$	0.16	0.31	0.12

Table 4.5: Pearson correlation coefficients between various graph metrics and kernel performance over all datasets. All values are statistically significant ($p < 0.01$), values < -0.70 and > 0.70 indicate a strong linear correlation.

To measure accuracy of the performance model, we compare approximated runtime against measured runtime. To determine EPS_d and EPS' for different cache-hit estimators, we fit the performance model using linear regression with ten-fold cross-validation.

Table 4.6 lists the mean approximation error for performance predictors based on estimated cache hit rate (P_H), local clustering coefficient (P_C), and device mean EPS (P_M). Some form of device mean processing speed is regularly used as naive predictor in graph processing applications ([1, 22]). However, without taking dataset characteristics into account, such a predictor is highly inaccurate with a mean relative error of 189% for the GPU kernel and 92% on CPU.

Using the predictor based on local clustering coefficient improves accuracy by 97% percent points for GPU and 13% on CPU, but the resulting error ranges are still significant.

	MRE		MSE	
	OMP3	CUD3	OMP3	CUD3
P_H	43%	76%	42%	114%
P_C	79%	92%	100%	124%
P_M	92%	189%	139%	251%

Table 4.6: Mean Relative Error (MRE) and Mean Squared Error (MSE) for performance predictors.

Figure 4.12 illustrates the mean relative error values for predicted OMP3 kernel performance. For P_H, datasets with low link density (EDU, BUB, OSM) stand out negatively. For these datasets, vertex-related processing time is not necessarily amortized by the (usually dominating) edge-related workload, resulting in an overestimation of performance.

P_C is more accurate than the naive P_M predictor for most datasets. We suspect that predictors can be made more accurate by using additional graph metrics (such as dataset size, average degree, or degree distribution skewness). Machine learning might be an interesting approach here, but this is out of the scope of this study.

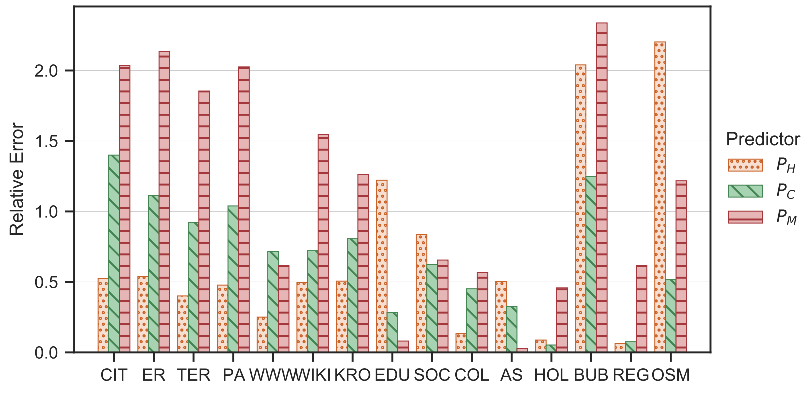


Figure 4.12: Relative Error for predicted OMP3 kernel performance per dataset.

4.6. Main Findings

We make the following observations regarding the performance of graph processing on CPUs and GPUs:

1. **Memory latency is the main graph processing bottleneck.**

   On both the CPU and the GPU, cache-friendly graphs perform considerably better, indicating that memory access latency is the main graph-processing performance bottleneck. Performance consequences due to workload imbalance are less impactful and can be mitigated by the programmer up to a certain extent (i.e., bisected vertex-cut on CPU and dynamic work distribution on GPU).

2. **Graph topology heavily impacts performance.**

   Most processors provide small fast caches backed up by larger, slower caches. The larger caches have better hit rates but higher access latency. Graph processing is intrinsically cache-unfriendly, as computations follow the structure of the input dataset, resulting in irregular memory accesses and performance characteristics.

Caches impact graph processing in two ways: data locality (spatial) and data reuse (temporal). For both, effort can be made to improve cache usage. First, spatial locality may be improved for graphs that display a high clustering coefficient (a tendency to form local communities), by placing data for local communities close together in memory.

Second, temporal locality may be improved for graphs that display preferential attachment, as a lot of vertices will access a subset of the same data. A Least Recently Used (LRU) cache-line replacement policy (as is common for hardware caches) will be beneficial for temporal reuse in this case.

Research effort has been made to reorder graphs in order to optimize cache usage [4, 6, 66].

3. **Graph metrics that quantify clustering are good indicators of performance.**

   Some datasets are inherently more difficult to process than others, due to their topology; cache hit ratio has a strong linear correlation with kernel performance. Modeling kernel performance is non-trivial, but graph metrics that quantify the level of node clustering can be used approximate cache-hit ratio and thereby increase predictor accuracy.

4. **GPU kernels perform better than CPU kernels.**

   GPU kernels generally provide better throughput than CPU kernels. To a certain extent, memory latency effects are mitigated by heavy multi-threading and time-slicing. Kernel implementations are more complex for GPU, but our naive CUDA implementation (CUD1) holds up remarkably well compared to more device-optimized kernels. Data transfer takes up a significant portion of execution time, but this overhead is amortized over convergence iterations.

5. **Push-based algorithms have significant synchronization overhead.**

   Pull-based data flow should be preferred as push-based execution is significantly slower for all evaluated datasets. The tradeoff between performance and awareness of neighbor updates becomes beneficial when it allows for algorithmic optimizations that quickly reduce the number active vertices by more than half.

5 Heterogeneous Performance Evaluation

In this chapter, we address RQ3: *how to take into account the graph-processing performance characteristics of each processing unit in a heterogeneous system when distributing a workload?* For this, we study the performance of PageRank in a heterogeneous environment. Section 5.1 elaborates on our methodology and evaluation metrics.

Sections 5.2 and 5.3 discuss our results. In section 5.4, we put our results into perspective and provide a theoretical scalability analysis. Finally, we conclude this chapter with our main findings in section 5.5.

5.1. Methodology

After a thorough performance evaluation of edge traversal on isolated devices in chapter 4, this chapter will focus on complete execution of the PageRank algorithm using a heterogeneous amalgamation of devices. Specifically, we examine the performance of CPU+GPU on a single node, and distributed CPUs in a cluster.

Our goal is to determine the impact of using multiple heterogeneous nodes on the performance of PageRank. To this end, we will quantify this impact using speedup, as a quantifier of the relative improvement in runtime after expanding system resources.

Benchmark timings are averaged over 2 algorithm runs (of 16 PageRank iterations) after one warm-up run. All experiments are run on the DAS-5 cluster [3]; the hardware platforms are described in table 4.1, and the software (versions) are outlined in tables 4.2 and 5.1. An overview of the examined kernels is available in section 3.5. Specifically, we focus on the heterogeneous StarPU (SPU) and OpenMPI (MPI) kernels, and their relative performance to the single device OpenMP (OMP) and CUDA (CUD) kernels.

Software	Version	Options
Metis	5.1.0	`-seed=12345`
StarPU	1.3.1	`STARPU_SCHED=dmdar STARPU_CALIBRATE=0 STARPU_PREFETCH=1`
OpenMPI	1.10.3	FDR InfiniBand (theoretical throughput of up to 8 GB/s)

Table 5.1: Software and corresponding flags/options used for our large-scale heterogeneous platforms experiments.

5.2. Accelerator Performance

Figure 5.1 shows the completion time ratio of running the complete PageRank algorithm on GPU compared to CPU, using the best performing kernels for each device. While pure PageRank computation is an average of $4.3\times$ faster on the GPU than on the CPU (figure 4.5), offloading the complete algorithm reduces the average speedup to $3.4\times$. As discussed in section 4.4, most of the performance regression can be attributed to the additional cost of the initial data transfer.

Thus, when GPUs are available, it makes sense to offload the PageRank execution to these accelerators. However, assuming a heterogeneous CPU+GPU system can be used to its full potential, there is an additional theoretical speedup of $\frac{3.4+1}{3.4} \approx 1.3\times$ obtainable by utilizing both devices simultaneously. Note that a lower GPU to CPU speedup ratio indicates a higher potential gain.

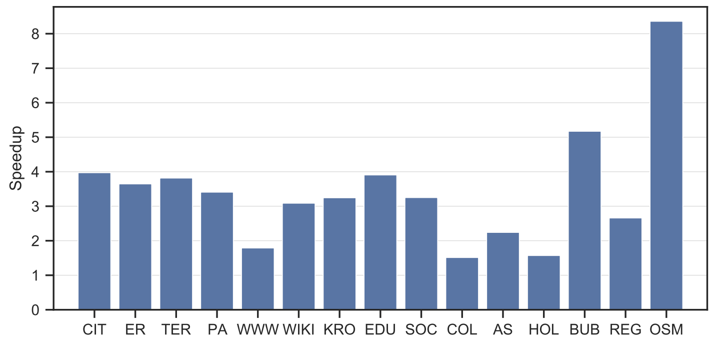


Figure 5.1: Speedup for algorithm execution using best GPU kernel over best CPU kernel. Average: $3.4\times$ Synthetic graphs: $\left\{|V| \simeq 4M, \overline{D} \simeq 32\right\}$.

5.2.1. Heterogeneous Execution

Figure 5.2 shows the PageRank speedup when using StarPU to schedule our device kernels in a heterogeneous CPU+GPU environment. SPU1 achieves an average additional speedup of $1.2\times$ over the best performing single device kernel, while SPU2 decreases overall performance more often than it increases it.

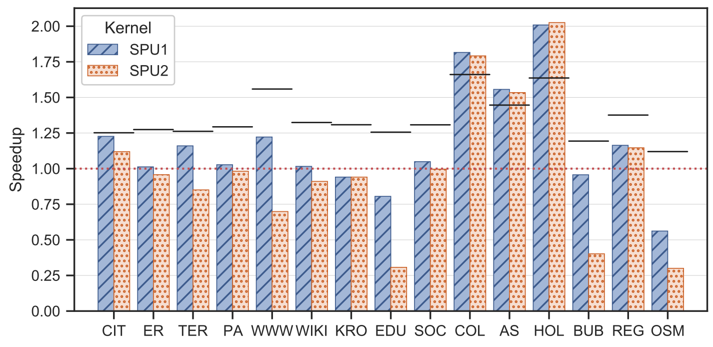


Figure 5.2: Speedup for SPU kernels over best OMP/CUD kernel. The black lines denote the theoretical speedup for the dataset. Average SPU1: $1.2\times$, SPU2: $1.0\times$

Although the heterogeneous kernels do manage to increase performance, most datasets do not achieve their theoretical speedup potential. We consider several reasons for this gap. First, the overhead of StarPU itself: as StarPU uses on-the-fly performance modeling, based on heuristics to allocate resources, there is a scheduling overhead.

Second, the inaccuracy of StarPU's performance modeling: as established in section 4.5, kernel performance is heavily influenced by dataset topology, and a naive predictor based on mean throughput only can exhibit error values of up to 189%.

Third, the additional overhead of ghost-vertex data synchronization: between convergence iterations, workers need to synchronize state. There appears to be a correlation between link density and speedup; table 4.3 shows that COL, AS, HOL are the most dense datasets, while EDU, BUB, OSM are the most sparse datasets. For sparse datasets, the overhead of data synchronization increases relatively, so a decrease in performance might indicate a communication bottleneck. Especially for SPU2, which increases communication volume to accommodate for vertex-cut data partitioning, this effect is visible.

For three datasets (COL, AS, HOL), heterogeneous execution exceeds the theoretically expected speedup. As workers are processing smaller data chunks, performance numbers may vary (for example, due to better caching). Similarly, the relative cost of data transfer may be smaller when only processing a subset of data.

Figure 5.3 shows the runtime speedup when adding a second GPU device to the system. Performance improvements are surprisingly marginal. Using CPU+2GPU, we expect a theoretical average speedup of $1.8\times$ over CPU+GPU, but we only achieve an average speedup of $1.1\times$ for SPU1 and $1.3\times$ for SPU2. Overall, the larger datasets (WWW,BUB,OSM) appear to benefit from the extra processing power to a certain extent. However, the additional transfer and orchestration overhead results in sub-linear speedup.

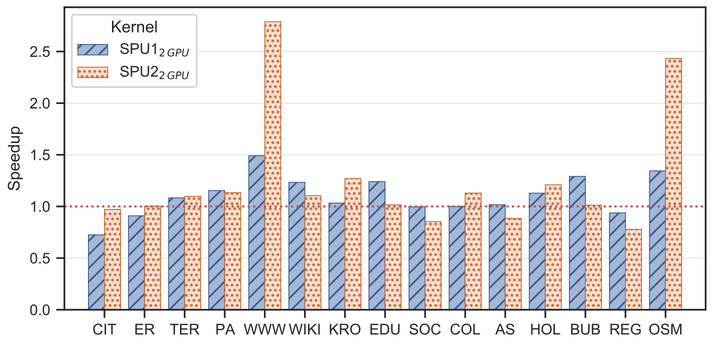


Figure 5.3: Speedup of SPU kernels when adding one extra GPU device (thus, running on CPU + 2GPUs). Average SPU1: $1.1\times$, SPU2: $1.3\times$

5.2.2. Workload

To examine the impact of data size on performance, figure 5.4 shows PageRank performance for synthetic graphs with varying vertex set size and varying edge set size. We observe a strong correlation between growing average degree and performance. This is in line with the observation in figure 5.2, that denser graphs perform better. For these graphs, vertex-specific processing time is amortized by the dominating edge workload. However, for sparse graphs, the vertex-related processing time might be non-negligible.

Similarly, the StarPU kernels start to outperform OMP/CUD kernels for high average degrees ($|\overline{D}| \gtrapprox 32$). As communication volume is limited by the number of vertices (because messages to the same destination can be merged before transfer), this is an indication that communication overhead is indeed the main bottleneck.

We observe no significant change in performance when varying the vertex set size while keeping the average degree constant. This indicates that communication overhead is not caused by the initial dataset transfer, but rather the additional communication needed to synchronize state between convergence iterations.

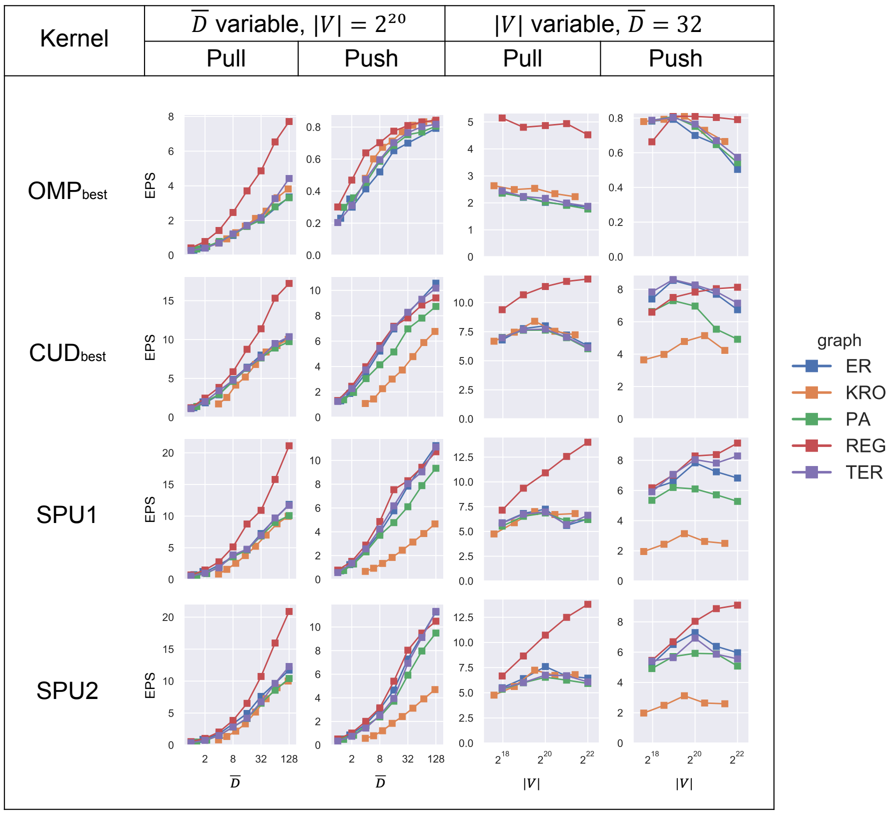


Figure 5.4: Algorithm performance in 10^9 traversed edges per second for synthetic datasets with varying sizes. Note that x-axes are similar, y-axes are not.

5.2.3. Task Granularity

As StarPU does not split input data, we manually divide the dataset into blocks of equal size, and create a StarPU task for each block. Note that the number of tasks we create has a direct impact on the task size - thus, changing the number of tasks will automatically control task granularity. Fine-grained task submission allows for a more gradual work distribution between devices, but increases the scheduling and synchronization overhead. Figure 5.5 shows the performance of SPU1 with varying task granularity.

We observe that there is no task granularity that works best for all graphs. Although no dataset performs best with 256 tasks, a granularity of 4/16/64 tasks results in best performance for 5 datasets each; submitting 16 tasks might be a good middle ground. This indicates that the overhead of StarPU itself is limited. No matter the granularity, around 84% of all tasks are scheduled on the GPU.

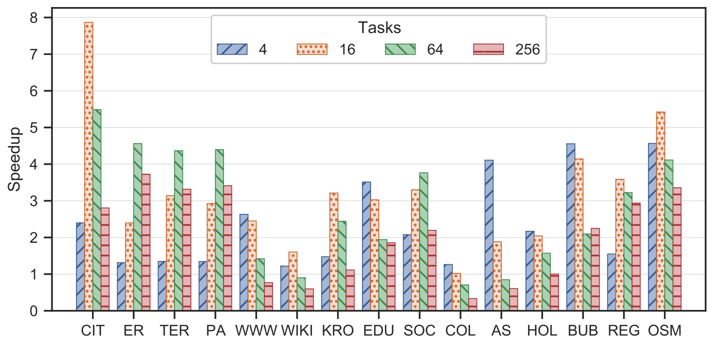


Figure 5.5: Speedup of the SPU1 kernel with varying task granularity compared to submitting a single StarPU task.

5.3. Cluster Performance

StarPU's task scheduling is asynchronous and nondeterministic, complicating systematic performance analysis. To further analyze synchronization overhead and its impact on scalability, we prefer to use MPI in a distributed system. Homogeneous workers use OMP3 with explicit MPI synchronization barriers, resulting in deterministic algorithm execution and predictable communication patterns.

Figure 5.6 shows the completion time ratio of running the complete PageRank algorithm with an increased the number of workers. Even though performance increases, we mostly observe sub-linear speedups. For the BUB and OSM datasets, MPI3 manages to achieve near linear speedups.

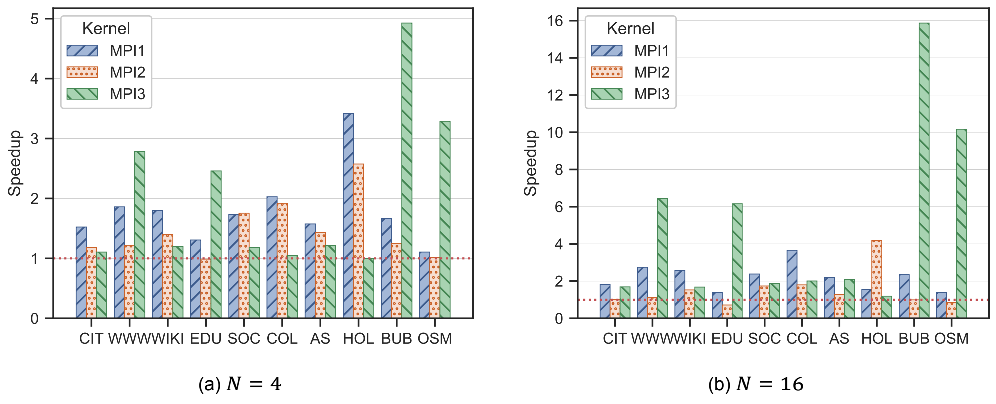


Figure 5.6: Scaling MPI kernels with different number of nodes (N). Speedup is calculated using $N = 1$ as reference.

As listed in table 5.2, the datasets that do well on MPI3 (WWW, EDU, BUB, OSM) all have $\beta < 1\%$ partition-crossing edges. Where MPI1 and MPI2 use straightforward block-based partitioning (edge-cut and vertex-cut respectively), MPI3 uses a METIS minimum graph-cut to minimize communication.

The structured communication patterns for block-based partitioning enables the use of optimized broadcast operations, while a minimal edge-cut partitioning requires additional bookkeeping overhead. This is why MPI1 and MPI2 generally perform better, but MPI3 scales better for datasets with a limited number of partition-crossing edges. However, it should be noted that preprocessing a good data partitioning using METIS takes up a significant amount of resources, most notably time.

Dataset	4-way		16-way	
	t	β	t	β
CIT	$23.7s$	~4%	$37.7s$	~8%
WWW	$76.1s$	~0%	$67.3s$	~0%
WIKI	$72.2s$	~3%	$93.2s$	~7%
EDU	$13.4s$	~0%	$15.5s$	~0%
SOC	$59.8s$	~3%	$77.3s$	~7%
COL	$1.6s$	~1%	$1.4s$	~1%
AS	$6.1s$	~1%	$7.0s$	~3%
HOL	$19.8s$	~1%	$22.2s$	~3%
BUB	$30.3s$	~0%	$27.1s$	~0%
OSM	$50.4s$	~0%	$53.0s$	~0%

Table 5.2: Duration (t) of METIS partitioning and percentage of partition-crossing edges (β) in result.

5.3.1. Execution Stages

Figure 5.7 displays the relative duration of different algorithm stages during the execution of MPI3. We observe that edge traversal and state synchronization take up the majority of runtime, while algorithm initialization and vertex update runtime are negligible. State synchronization (communication) is the main bottleneck and correlates to the number partition-crossing edges (table 5.2).

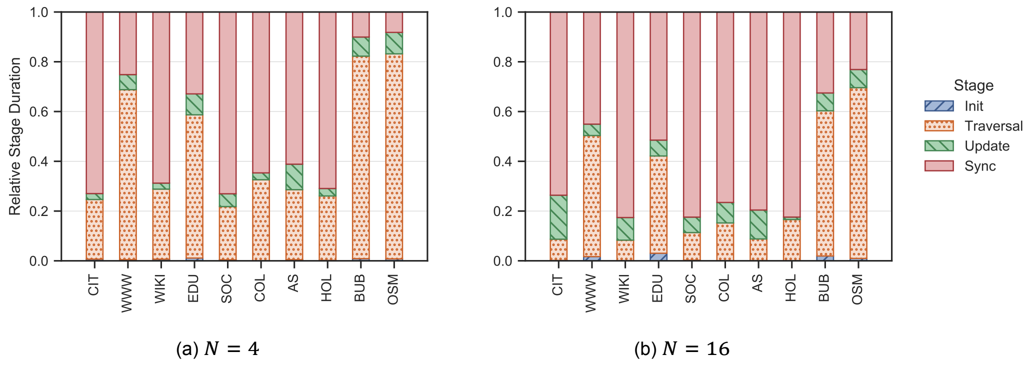


Figure 5.7: Relative duration of different stages of PageRank for MPI3.

For datasets that scale well in a 4-way partitioning (such as WWW), communication takes up to ~25% of runtime, while for lesser performing datasets (such as SOC) it takes up the majority (~75%) of runtime. These ratios might differ depending on how computation-heavy a graph algorithm is. As PageRank is more memory-intensive, these observations provide an upper bound for the synchronization overhead.

Based Amdahl's law, we can estimate that maximum speedup is rather limited. Equations (5.1) and (5.2), where N is the number of cores and p the relative duration of communication, project the maximum speedup for SOC and WWW to be $5.3\times$ and $16\times$ respectively, and this is assuming communication overhead stays relatively similar when increasing the number of workers. In reality, communication will likely increase as well, which make these projections optimistic. Addtionally, due to diminishing returns, a disproportionate number of workers is required to approach these speedup numbers in practice.

$$S_{SOC} = N \times \frac{1}{1-p} \approx 4 \times \frac{1}{1-0.25} \approx 5.3 \tag{5.1}$$

$$S_{WWW} = N \times \frac{1}{1-p} \approx 4 \times \frac{1}{1-0.75} \approx 16 \tag{5.2}$$

5.3.2. Stream Partitioning

As shown in table 5.2, using METIS for partitioning is relatively time consuming. Ideally, a dataset can be distributed on the fly with minimal preprocessing. We compare the impact of partitioning quality with two arbitrary one-pass partitioning strategies: chunk-based and randomized vertex distribution.

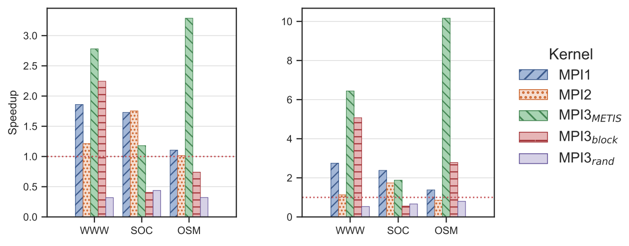
Table 5.3 lists the percentage of partition-crossing edges for a subset of our datasets. As expected, random partitioning results in a high percentage of partition-crossing edges. Quality of chunk-based partitioning primarily relies on the ordering of input data. For the selected datasets, the strategy results in a quality between METIS and randomized distribution.

Dataset	4-way		16-way	
	Chunk	Rand	Chunk	Rand
WWW	~0%	~6%	~0%	~13%
SOC	~9%	~13%	~22%	~37%
OSM	~6%	~64%	~8%	~90%

Table 5.3: Percentage of partition-crossing edges for alternative partitioning strategies.

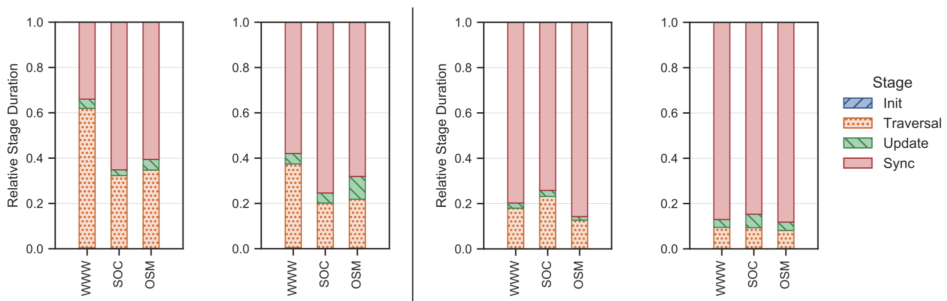
Figure 5.8 shows the change in scalability when applying alternative partitioning strategies, while figure 5.9 shows the relative duration of each stage. In line with earlier findings, speedup decreases, because the synchronization overhead increases with the number of partition-crossing edges.

Minimizing communication overhead is important, but difficult to achieve with a one-pass partitioning strategy. Static datasets can reuse partitioning results, while dynamic datasets can explore incremental or multi-pass partitioning strategies. If no good partitioning strategy is available, it might be preferable to fall back to the structured block-based work distribution from MPI1/MPI2.



(a) $N = 4$ (b) $N = 16$

Figure 5.8: Scaling MPI kernels with different partitioning strategies. Speedup compared to $N = 1$.



(a) Chunk $N = 4$ (b) Chunk $N = 16$ (c) Random $N = 4$ (d) Random $N = 16$

Figure 5.9: Relative duration of different stages of PageRank for different MPI3 partitioning strategies.

5.4. Scalability Analysis

Building upon the device performance model proposed in section 4.5, an approximate model to predict algorithm runtime T_w for graph partition G_p on worker w is illustrated in equation (5.3), where M_w is initialization cost, S number of supersteps, P_w edge processing speed, $E_i \leftrightarrow E_j$ the subset of boundary edges between partitions i and j, and $C_{i,j}$ the corresponding communication rate.

Applying this model to a cluster of workers W, where every worker processes exactly one graph partition, the algorithm runtime T is approximated using equation (5.4).

$$T_w(G_p) = M_w(G_p) + \sum_{s=1}^{S} \left(\frac{|E_p|}{P_w(G_p)} + \sum_{i=1}^{N} \frac{|E_p \leftrightarrow E_i|}{C_{w,i}} \right) \tag{5.3}$$

$$T(G) = \max_{w \in W} \left(M_w(G_w)\right) + \sum_{s=1}^{S} \left(\max_{w \in W} \left(\frac{|E_w|}{P_w(G_w)} \right) + \max_{w \in W} \left(\sum_{v \in W} \frac{|E_v \leftrightarrow E_w|}{C_{v,w}} \right) \right) \tag{5.4}$$

Assuming that the initialization cost is amortized as the number of supersteps grows, scalability is primarily limited by the ratio of local computation to global communication. Ideally, computation cost dominates. Equation (5.5) uses the ratio of partition-crossing edges β to estimate general communication cost in this relation. Consequently, the maximum ratio for which computation exceeds communication cost is defined in equation (5.6).

$$\frac{|E|}{P} \geq \frac{\beta * |E|}{C} \tag{5.5}$$

$$\beta < \frac{C}{P} \tag{5.6}$$

Communication rate C is bounded by the interconnection speed between workers. For accelerators, this is the system bus latency and bandwidth, while for distributed workers this is network latency and bandwidth. Table 5.4 lists the theoretical and tested throughput rates for different platforms (10G ethernet listed for reference). For both PCIe (GPU) and InfiniBand (distributed CPUs), we are able to achieve around ~74% of the theoretical throughput.

Using these throughput rates, we can approximate a limit for partition-crossing edges such that communication cost does not exceed a specified ratio. For example, for a maximum theoretical $4\times$ speedup (1:3 ratio) for OMP3 workers using InfiniBand, a partitioning with (at least) $\beta < 0.18$ is required. For a $16\times$ speedup, this limit decreases to $\beta < 0.04$.

Platform	Throughput		EPS		β		
	Bus	Bench	P	C	$2\times$	$4\times$	$16\times$
PCIe 3.0	16 GB/s	11.7 GB/s	8.7*	1.5	17%	6%	1%
InfiniBand	8 GB/s	5.9 GB/s	1.5**	0.8	53%	18%	4%
10G Ethernet	1.25GB/s	–	1.5**	0.2	10%	3%	1%

Table 5.4: Approximated maximum ratio β of partition-crossing edges for specified speedup factor.
P – mean computation rate (10^9 edges per second). C – estimated communication rate (10^9 edges per second).
*Mean CUD3 performance. **Mean OMP3 performance.

As shown in table 5.2, achieving such quality data distributions might be difficult, if not impossible. Using METIS, three of our real-world datasets cannot be partitioned in 16 parts such that $\beta < 0.04$, while a majority of datasets exceed the theoretical PCIe speedup threshold of $\beta < 0.01$. Using (suboptimal) on-the-fly partitioning strategies, these limits might be even more difficult to accomplish.

Most real-world graphs exhibit limited theoretical scalability, even with optimal (METIS) data distribution. In that regard, it might be more beneficial to increase scalability by minimizing communication volume through optimization of the graph algorithm. For example, by only sending differential updates or by limiting the set of active vertices. However, such optimizations are largely algorithm-dependent.

5.5. Main Findings

We make the following observations regarding the performance of heterogeneous graph processing:

1. **Communication is the major graph processing scalability bottleneck.**

   For edge iteration in large-scale graph-processing systems, the main scalability bottleneck is the synchronization overhead. Communication of ghost-vertex data takes up an increasing proportion of the runtime, resulting in sub-linear speedups for most evaluated datasets. Maximum theoretical speedup is limited by the ratio of computation to communication cost. Computation-heavy algorithms will scale better as relatively more work can be parallelized.

2. **Graph partitioning heavily impacts scalability.**

   The number of partition-crossing edges indicates communication volume and directly correlates to performance. Decreasing the number of partition-crossing edges improves scalability. However, realizing a minimum graph cut with equal-sized subsets is an NP-complete problem, which makes optimal data distribution non-trivial. Maximum theoretical speedup is rather low for the majority of tested real-world graphs due to the high percentage of partition-crossing edges, even with an optimal data distribution.

3. **Block-based partitioning can provide an alternative to minimum graph cuts.**

   Using METIS to realize a minimum graph cut for optimal data distribution is not realistic, as the preprocessing cost would significantly exceed algorithm runtime. However, using an arbitrary one-pass partitioning strategy such as random vertex distribution can be detrimental to performance.

   For static datasets, expensive preprocessing cost can be amortized by reusing the minimum graph cut for multiple algorithm runs. For dynamic datasets, research effort has been made to efficiently distribute datasets using incremental or multi-pass partitioning heuristics ([9, 23]). For communication-heavy graph algorithms, straightforward block-based partitioning can achieve decent performance by exploiting optimized broadcast operations in distributed environments.

4. **Accelerators can increase graph-processing performance.**

   Accelerators can significantly improve graph-processing performance. When possible, it makes sense to offload algorithm execution to GPUs. For a single device, costly initialization overhead is amortized over multiple convergence iterations.

   In heterogeneous scenarios, where one or multiple accelerators are used as part of a larger system, benefits are smaller, as an increased disparity between computation and communication rates make GPUs more susceptible to communication bottlenecks.

The extent to which an additional device in the system can result in speedup is determined by the (additional) cost of synchronization. For GPUs, this means it will be difficult to tap into the full processing potential for a lot of real-world graphs.

6 Conclusion and Future Work

Graphs are an ubiquitous concept used for modeling entities and their relationships. Although abundant in present-day computing challenges, large graphs are fundamentally difficult to process because of their irregular computation structure. In heterogeneous distributed systems, it is unclear what would be the optimal partitioning of a graph-processing workload, as the input dataset has a significant impact on device performance.

In this work, we have contributed models to support the distribution of a graph-processing workload in a large-scale heterogeneous system. To achieve this, we have designed a novel graph-processing performance benchmark and used PageRank for a case study to analyze the influence of graph topology on performance and scalability.

In this chapter, we conclude this thesis by presenting answers to our research questions and discussing directions for future work.

6.1. Conclusion

We conclude this thesis by answering the research questions posed in section 1.2:

RQ1 *How to evaluate the graph-processing performance characteristics of a processing unit?*

In a heterogeneous environment, it is unclear what would be the best allocation of resources to optimally process a given workload. To determine and compare the graph-processing performance characteristics of each processing unit, we have proposed a structural benchmarking strategy (contribution C1) based on the four pillar design principles of "think like a vertex" graph-processing frameworks defined by McCune et al. [45].

In chapter 3, we have discussed the design considerations and implications for each pillar and how they apply to different computing architectures. Using the PageRank algorithm as case study, we have designed and implemented 30 custom kernels targeting CPUs, GPUs, and heterogeneous environments (contribution C5). Each kernel gradually alters a single pillar in order to evaluate the impact of each design decision.

Furthermore, we have introduced a graph processing benchmarking suite (contribution C6) to facilitate structural evaluation of graph-processing performance characteristics in a heterogeneous environment. Five different types of synthetic graphs are used to evaluate the impact of workload size, while maintaining similar connectivity characteristics, and ten real-world datasets are included to put results into perspective.

RQ2 *How to model the graph-processing performance of a processing unit with respect to dataset topology?*

PageRank is the composition of two common graph-processing building blocks. First, an information scattering stage that iterates over all edges in the dataset. Second, a global gathering stage that updates vertex values and evaluates a stop condition. As PageRank is more memory-intensive than computation-intensive, it is an ideal algorithm to analyze the impact of graph topology on performance.

In chapter 4, we have conducted an in-depth case study of the graph-processing performance characteristics of CPUs and GPUs using our in-house PageRank kernels (contribution C2). One of our main findings is that hardware caches have a significant impact on kernel performance.

We have proposed a graph-processing performance model that takes into account the topological structure of an input dataset (contribution C3). We estimate cache hit rate using the local clustering coefficient, a graph metric that quantifies node clustering.

Compared to mean processing speed, a naive predictor that is commonly used as performance predictor, we are able to reduce the average predicted runtime error from 189% down to 92% for GPU kernels, and from 92% to 79% for CPU kernels.

RQ3 *How to take into account the graph-processing performance characteristics of each processing unit in a heterogeneous system when distributing a workload?*

In chapter 5, we have conducted an in-depth evaluation of graph processing scalability in heterogeneous systems (contribution C4). First, using StarPU for a heterogeneous amalgamation of CPU and GPU devices. Second, using MPI for a homogeneous cluster of CPUs to further analyze the impact of synchronization overhead when scaling the number of workers.

In both scenarios, we have found communication overhead to be the main scalability bottleneck. When adding a worker in a heterogeneous system, we determine the maximum theoretical speedup by the ratio of device processing speed (estimated using RQ2), communication speed (estimated using interconnection bandwidth), and communication volume (estimated using the number of partition-crossing edges).

In this equation, the quality of data partitioning (communication volume) has a major impact on performance. In our findings, maximum theoretical speedup is limited for the majority of tested real-world graphs by the high percentage of partition-crossing edges, even with optimal (METIS) data distribution.

Our work contributes towards a decision model that can optimize distribution of workloads in a heterogeneous system, but more research is needed towards predicting the quality of graph partitioning and the corresponding communication volume.

6.2. Future Work

We propose the following directions for future work:

1. **Extend performance evaluation to include more devices.**

   In this work, our performance evaluation focused on a single generation of commodity CPU and GPU hardware, but it might be interesting to evaluate the graph-processing performance characteristics of other devices in a similar matter. Co-processors such as Field-Programmable Gate Arrays (FPGA) or the Xeon Phi are possible targets. Similarly, ARM CPUs (based on the RISC instruction set) are seeing a rise in popularity, and might exhibit different performance characteristics.

As architectures develop over time, so might respective performance characteristics. In that regard, it will also be interesting to evaluate performance over different generations of hardware in order to better predict performance for future generations of hardware.

2. **Extend performance evaluation to include more partitioning strategies.**

   As shown in section 5.3.2, dataset distribution and the resulting number of partition-crossing edges is vital to scalability of large-scale graph processing system. In this work, we focused on evaluating boundaries: that is, we used METIS minimum graph cut as a best-case scenario, and random vertex distribution in a worst-case scenario.

   To get a better understanding of the performance impact of partitioning strategies currently used by various graph-processing frameworks, more research is needed. Extending our benchmarks with more heuristic partitioning strategies would allow for a better grasp on the subtleties of data distribution.

3. **Apply performance evaluation to other graph algorithms.**

   PageRank is an ideal algorithm to analyze the influence of graph topology on performance, but it might not necessarily be representative for the entire class of graph algorithms. To relate our work to other algorithms, more research is needed.

   First, we suggest community detection algorithms such as label propagation[51] and weakly connected components as interesting next research subjects, as they are similar to PageRank, but more demanding in computation and communication respectively.

   Second, evaluating another building block such as breadth-first search might put our findings into perspective with regards to more lightweight iterative algorithms.

4. **Improve performance model using additional graph metrics.**

   Performance modeling of a graph processing system is non-trivial. In section 4.5, we have demonstrated that graph metrics that quantify the level of node clustering can be used to increase prediction accuracy. We suspect that better accuracy can be achieved by taking extra graph metrics in account, such as dataset size, average degree, or degree distribution skewness.

   In that regard, multiple linear regression or more advanced machine learning techniques are likely candidates for better performance models, but attention should be paid to the risk of overfitting.

5. **Integrate device performance model with StarPU.**

   StarPU uses a naive performance predictor heuristic based on mean throughput, which (as demonstrated in section 4.5) can exhibit up to 189% relative error for our GPU kernels. Using our proposed performance model, one should be able to improve StarPU's performance predictor, and thereby its task scheduling quality.

   Calculating the necessary graph metrics for our performance model on-the-fly might induce significant extra cost. Integrating the model into StarPU is an easy way to test viability in a real system.

6. **Design a decision model to distribute a workload in a heterogeneous system.**

   Combining our performance model from section 4.5 and our scalability insights from section 5.4, one should be able to design a decision model for optimal allocation of resources for graph processing in a heterogeneous environment.

   In such a decision model, the number of partition-crossing edges is likely to be a fundamental variable. It is unclear whether or not this variable can be accurately predicted before partitioning. Ideally, this value would be expressed in terms of graph metrics, similar to our performance model.

Bibliography

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[2] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 10th DIMACS implementation challenge - graph partitioning and graph clustering. http://www.cc.gatech.edu/dimacs10/index.shtml, 2011.

[3] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016.

[4] Vignesh Balaji and Brandon Lucia. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 203–214. IEEE, 2018.

[5] Albert-Laszlo Barabasi and Zoltan N Oltvai. Network biology: understanding the cell's functional organization. *Nature reviews genetics*, 5(2):101–113, 2004.

[6] Scott Beamer, Krste Asanovic, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *2015 IEEE International Symposium on Workload Characterization*, pages 56–65. IEEE, 2015.

[7] Ronald F. Boisvert, Ronald F. Boisvert, and Karin A. Remington. *The matrix market exchange formats: Initial design*. US Department of Commerce, National Institute of Standards and Technology, 1996.

[8] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[9] Rishan Chen, Xuetian Weng, Bingsheng He, and Mao Yang. Large graph processing in the cloud. In *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, page 1123, New York, New York, USA, 2010. ACM Press. ISBN 9781450300322. doi: 10.1145/1807167.1807297.

[10] Intel Corporation. Developing Multithreaded Applications: A Platform Consistent Approach. Technical report, February 2005.

[11] Nvidia Corporation. CUDA v10 Toolkit Documentation, 2019.

[12] Gabor Csardi, Tamas Nepusz, et al. The igraph software package for complex network research. *InterJournal, Complex Systems*, 1695(5):1–9, 2006.

[13] Jeffrey Dean and Sanjay Ghemawat. MapReduce. *Communications of the ACM*, 51 (1):107, January 2008. ISSN 00010782. doi: 10.1145/1327452.1327492.

[14] Niels Doekemeijer and Ana Lucia Varbanescu. A Survey of Parallel Graph Processing Frameworks. Technical report, PDS Technical Report PDS-2014-003, December 2014.

[15] David Easley and Jon Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

[16] Paul Erdős and Alfréd Rényi. On random graphs I. *Publ. Math. Debrecen*, 6:290–297, 1959.

[17] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[18] Facebook Inc. Facebook Reports First Quarter 2020 Results, 2020.

[19] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[20] Zhisong Fu, Michael Personick, and Bryan Thompson. MapGraph. In *Proceedings of Workshop on GRAph Data management Experiences and Systems - GRADES'14*, pages 1–6, New York, New York, USA, 2014. ACM Press. ISBN 9781450329828. doi: 10.1145/2621934.2621936.

[21] Michael R. Garey, David S. Johnson, and Larry Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing - STOC '74*, pages 47–63, New York, New York, USA, 1974. ACM Press. doi: 10.1145/800119.803884.

[22] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT '12*, page 345, New York, New York, USA, 2012. ACM Press. ISBN 9781450311823. doi: 10.1145/2370816.2370866.

[23] Abdullah Gharaibeh, Lauro Beltrao Costa, Elizeu Santos-Neto, and Matei Ripeanu. On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 851–862. IEEE, May 2013. ISBN 978-1-4673-6066-1. doi: 10.1109/IPDPS.2013.37.

[24] David F. Gleich. Web matrices and wikipedia matrices. http://www.cise.ufl.edu/research/sparse/aux/Gleich/, 2007.

[25] David F. Gleich. Pagerank beyond the web. *SIAM Review*, 57(3):321–363, 2015.

[26] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and C Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 17–30, 2012.

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

[28] Jerrold W Grossman. *Erdos Number Project*. Jerrold W. Grossman., 2002.

[29] Aric Hagberg, Pieter Swart, and Daniel Schult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[30] Bruce Hendrickson and Jonathan W. Berry. Graph Analysis with High-Performance Computing. *Computing in Science & Engineering*, 10(2):14–19, March 2008. ISSN 1521-9615. doi: 10.1109/MCSE.2008.56.

[31] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *2009 Ninth IEEE International Conference on Data Mining*, pages 229–238. IEEE, December 2009. ISBN 978-1-4244-5242-2. doi: 10.1109/ICDM.2009.14.

[32] U Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. Gbase: an Efficient Analysis Platform for Large Graphs. *The VLDB Journal*, 21 (5):637–650, June 2012. ISSN 1066-8888. doi: 10.1007/s00778-012-0283-9.

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

[34] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. CuSha. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing - HPDC '14*, pages 239–252, New York, New York, USA, 2014. ACM Press. ISBN 9781450327497. doi: 10.1145/2600212.2600227.

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

[36] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8 (1):1, 2016.

[37] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11(Feb):985–1042, 2010.

[38] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2):39–55, 2008.

[39] Yongchao Liu and Bertil Schmidt. Lightspmv: faster csr-based sparse matrix-vector multiplication on cuda-enabled gpus. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 82–89. IEEE, 2015.

[40] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A New Framework for Parallel Machine Learning. Technical report, June 2010.

[41] Yucheng Low, Danny Bickson, Joseph E. Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab. *Proceedings of the VLDB Endowment*, 5(8):716–727, April 2012. ISSN 21508097. doi: 10.14778/2212351.2212354.

[42] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(01):5–20, March 2007. ISSN 0129-6264. doi: 10.1142/S0129626407002843.

[43] Michael W. Mahoney, LekHeng Lim, and Gunnar E. Carlsson. Algorithmic and statistical challenges in modern largescale data analysis are the focus of MMDS 2008. *ACM SIGKDD Explorations Newsletter*, 10(2):57, December 2008. ISSN 19310145. doi: 10.1145/1540276.1540294.

[44] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel. In *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, page 135, New York, New York, USA, 2010. ACM Press. ISBN 9781450300322. doi: 10.1145/1807167.1807184.

[45] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.

[46] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. Graph Structure in the Web — Revisited: A Trick of the Heavy Tail. In *Proceedings of the companion publication of the 23rd international conference on World wide web companion*, pages 427–432, 2014. ISBN 9781450327459. doi: 10.1145/2567948.2576928.

[47] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[48] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, pages 439–455, New York, New York, USA, 2013. ACM Press. ISBN 9781450323888. doi: 10.1145/2517349.2522738.

[49] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, pages 456–471, New York, New York, USA, 2013. ACM Press. ISBN 9781450323888. doi: 10.1145/2517349.2522739.

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

[51] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76 (3):036106, 2007.

[52] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, pages 472–488, New York, New York, USA, 2013. ACM Press. ISBN 9781450323888. doi: 10.1145/2517349.2522740.

[53] Julian Shun and Guy E. Blelloch. Ligra. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '13*, page 135, New York, New York, USA, 2013. ACM Press. ISBN 9781450319225. doi: 10.1145/2442516.2442530.

[54] Jeremy Siek, Andrew Lumsdaine, and Lie-Quan Lee. *The Boost graph library: user guide and reference manual*. Addison-Wesley, 2002.

[55] Ricard V. Solé and Sergi Valverde. Information theory of complex networks: on evolution and architectural constraints. In *Complex networks*, pages 189–207. Springer, 2004.

[56] Philip Stutz, Abraham Bernstein, and William Cohen. Signal/Collect: Graph Algorithms for the (Semantic) Web. In Peter F. Patel-Schneider, Yue Pan, Pascal Hitzler, Peter Mika, Lei Zhang, Jeff Z. Pan, Ian Horrocks, and Birte Glimm, editors, *The*

Semantic Web–ISWC 2010, volume 6496 of *Lecture Notes in Computer Science*, pages 764–780. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-17745-3. doi: 10.1007/978-3-642-17746-0.

[57] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the Facebook Social Graph. *arXiv preprint arXiv:1111.4503*, November 2011.

[58] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990. ISSN 00010782. doi: 10.1145/79173.79181.

[59] Xiao Fan Wang and Guanrong Chen. Complex networks: small-world, scale-free and beyond. *IEEE circuits and systems magazine*, 3(1):6–20, 2003.

[60] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

[61] Tiffani L. Williams and Rebecca J. Parsons. The heterogeneous bulk synchronous parallel model. In *International Parallel and Distributed Processing Symposium*, pages 102–108. Springer, 2000.

[62] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo Chen. SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation. Technical report, 2013.

[63] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX. In *First International Workshop on Graph Data Management Experiences and Systems - GRADES '13*, pages 1–6, New York, New York, USA, 2013. ACM Press. ISBN 9781450321884. doi: 10.1145/2484425.2484427.

[64] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark : Cluster Computing with Working Sets. In *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[65] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Maiter: An Asynchronous Graph Processing Framework for Delta-Based Accumulative Iterative Computation. *IEEE Transactions on Parallel and Distributed Systems*, 25(8):2091–2100, August 2014. ISSN 1045-9219. doi: 10.1109/TPDS.2013.235.

[66] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302. IEEE, 2017.

[67] Jianlong Zhong and Bingsheng He. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, June 2014. ISSN 1045-9219. doi: 10.1109/TPDS.2013.111.