ACCELERATING IRREGULAR APPLICATIONS ON PARALLEL HYBRID PLATFORMS

by

Abdullah Gharaibeh
B.Sc., Jordan University of Science and Technology, 2005
M.A.Sc, The University of British Columbia, 2009

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate and Postdoctoral Studies
(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)

May 2015

Abstract

Future high-performance computing systems will be hybrid; they will include processors optimized for sequential processing and massively-parallel accelerators. Platforms based on Graphics Processing Units (GPUs) are an example of this hybrid architecture, they integrate commodity CPUs and GPUs. This architecture promises intriguing opportunities: within the same dollar or energy budget, GPUs offer a significant increase in peak processing power and memory bandwidth compared to traditional CPUs, and are, at the same time, generally-programmable.

The adoption of GPU-based platforms, however, faces a number of challenges, including the characterization of time/space/power tradeoffs, the development of new algorithms that efficiently harness the platform and abstracting the accelerators in a generic yet efficient way to simplify the task of developing applications on such hybrid platforms.

This dissertation explores solutions to the abovementioned challenges in the context of an important class of applications, namely irregular applications. Compared to regular applications, irregular applications have unpredictable memory access patterns and typically use reference-based data structures, such as trees or graphs; moreover, new applications in this class operate on massive datasets.

Using novel workload partitioning techniques and by employing data structures that better match the hybrid platform characteristics, this work demonstrates that significant performance gains, in terms of both time to solution and energy, can be obtained when partitioning the irregular workload to be processed concurrently on the CPU and the GPU.

Preface

I was the leader of all the research work presented here: proposing the key ideas, performing most or all the design, implementation and evaluation work, and leading the publication writing effort.

The research presented in this dissertation have been either published or submitted for publication. The following is a list of publications related to each chapter (listed in reverse chronological ordered):

- **Chapter 3.** The research presented in this chapter was published in three publications. The author of this dissertation is the leader and main contributor to this project from the idea, to system design and development, to evaluation and paper writing.
  - *(i)* **Abdullah Gharaibeh**, Elizeu Santos-Neto, Lauro Beltrão Costa and Matei Ripeanu, *The Energy Case for Graph Processing on Hybrid CPU and GPU Systems*, IEEE Workshop on Irregular Applications: Architectures & Algorithms (IA3) in conjunction with SC13, Denver, Colorado USA, November 2013 *(30% acceptance rate)*.
  - *(ii)* **Abdullah Gharaibeh**, Lauro Beltrão Costa, Elizeu Santos-Neto and Matei Ripeanu, *On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest*, IEEE International Parallel & Distributed Processing Systems (IPDPS 2013), Boston, MA, May 2013 *(21% acceptance rate)*.
  - *(iii)* **Abdullah Gharaibeh**, Lauro Beltrão Costa, Elizeu Santos-Neto and Matei Ripeanu, *A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing*, IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques (PACT 2012). Minneapolis, MN September 2012 (19% acceptance rate).

- **Chapter 2.** This chapter was presented in two publications. I am the main contributor to this project from the idea, to system design, development and evaluation. I also led the paper writing effort.

  (iv) **Abdullah Gharaibeh** and Matei Ripeanu, *Accelerating Sequence Alignment on Hybrid Architectures*, Scientific Computing Magazine, February 2011.

  (v) **Abdullah Gharaibeh** and Matei Ripeanu, *Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance*, IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2010), New Orleans, Louisiana, November 2010 *(20% acceptance rate).*

Table of Contents

List of Tables

List of Figures

To My Parents, Wife, Brother and Sisters

Chapter 1

Introduction and Overview

Irregular applications, such as large-scale graph computations, are on the rise. They are characterized by complex memory access patterns, data dependent parallelism, and they typically use sparse data structures such as trees or graphs [Pingali et al. 2009]. Additionally, a challenge for these irregular applications is that they operate on massive datasets with heterogeneous structure, which are difficult to partition and load balance among processing elements.

High-impact applications with such characteristics exist in different domains, such as computational biology (e.g., genome assembly [Pop 2009; Nagarajan and Pop 2013]), social networks (e.g., social network analysis [Brandes 2001; Gupta et al. 2013]) and web analytics (e.g., ranking the web [Page et al. 1999]).

At the same time, current trends suggest that future high-performance computing systems (HPC) will be 'hybrid' [Hill and Marty 2008; Johnson et al. 2011; Catanzaro et al. 2010]. These systems include processors optimized for sequential processing and accelerators optimized for parallel processing.

One example of this hybrid architecture is a GPU-accelerated platform that integrates commodity processors (CPUs) and Graphics Processing Units (GPUs). Hybrid GPU-accelerated platforms promise intriguing opportunities: within the same dollar or energy budget these platforms offer a significant increase in peak processing power and memory bandwidth compared to traditional platforms. This is clearly demonstrated in the Green500[1] list, where GPU-accelerated supercomputers dominates the top ten spots of the latest list (November, 2014).

[1] http://www.green500.org/

In this context, my dissertation focuses on exploring the opportunities, design methodologies and middleware to improve the efficiency and, at the same time, reduce the complexity of harnessing commodity hybrid GPU-accelerated platforms to improve the performance of irregular, data-intensive applications.

This exploration is driven by two important irregular applications: graph processing (Chapter 2) and DNA sequence alignment (Chapter 3). In the context of these two applications, this thesis presents efforts along the following two high-level directions:

- *First, developing performance models based on the characteristics of the targeted applications, the workloads and the hybrid system.* The goal of these models is to assess, at low cost, the feasibility of accelerating large-scale irregular applications using hybrid platforms. Moreover, they serve as a tool to guide software optimization efforts, system configuration and provisioning.
- *Second, designing and developing methods and platforms to efficiently support irregular applications on hybrid platforms.* The heterogeneity of the processing elements (e.g., GPUs implement a different parallel processing model than CPUs and have much less memory) and the inherent irregularity of the targeted applications (e.g. scale-free graphs) require careful design and consideration of the employed data structures and workload partitioning techniques.

This work demonstrates that offloading irregular large-scale workloads to be processed on hybrid GPU-accelerated platforms offers significant performance and energy gains. This result is contrary to the common belief that the GPU's strict parallel model limits its support for such complex workloads. Equally important, this work shows that data-intensive applications benefit from GPU offloading although current GPU models are provisioned with limited onboard memory space and are typically connected to the host processor via high-latency interconnect.

The rest of this chapter presents the background of this research, the high-level research questions, the methodology and a summary of contributions. First, it discusses hybrid platforms in general and elaborates on the characteristics of commodity GPU-accelerated systems in specific (Section 1.1). Next, the chapter presents background related to the irregular applications targeted by this work (Section 1.2). The discussion follows with a high-level description of the research questions this thesis aims to address (Section 1.3). The chapter then discusses the challenges and opportunities of using hybrid GPU-accelerated platforms to improve the performance of irregular applications (Section 1.4), the high-level methodology followed by this work (Section 1.5) and a summary of the main contributions (Section 1.6). Finally, the chapter concludes with a presentation of the dissertation structure (Section 1.7).

1.1 Hybrid Platforms

Hybrid platforms combine two types of processing elements with different design goals. The first type of processing element is optimized for sequential processing and aims to minimize execution latency for individual execution threads. To support fast sequential processing, such processors operate at high frequency and employ complex optimizations to utilize instruction level parallelism (ILP), such as out of order execution, multi-instruction issue and sophisticated branch prediction techniques.

However, increasing the frequency and/or the number of cores of such processors has limits. Increasing the operating frequency results in a considerable increase in power consumption and heat dissipation to the degree where it becomes infeasible. Additionally, the employed optimizations noticeably increase the number of transistors used, hence placing limits on the number of processor cores that can be put on the die. Traditional multi-processors produced by Intel and AMD are examples of processing elements in this category.

The second processing element type is optimized for massively-parallel processing and aims to maximize throughput. To enable massive-parallelism and overcome the power and area limitations, the cores have a simple design (e.g., limited support for ILP optimizations). Moreover, to reduce power consumption, the cores typically operate at lower frequency compared to traditional multiprocessors cores. GPUs and Intel's Many Integrated Core (MIC) architecture are examples of processing elements in this category.

Combining the two types of processors to form a hybrid platform makes intuitive sense. Applications typically have both sequential phases that can be run by the fast sequential processor, and parallel phases that can be run by the massively-parallel processor. Moreover, as argued by Hill et al. [Hill and Marty 2008], compared to traditional multiprocessors, hybrid systems offer a better balance between performance and used resources (energy and area). Examples of such hybrid platforms include IBM's Cell Broadband Engine [Chen et al. 2007], AMD's Fusion architecture [Branover et al. 2012], Rigel [Johnson et al. 2011] and commodity systems that host both CPUs and commodity accelerators such as GPUs and MICs.

In particular, GPU-accelerated platforms gained wide-spread popularity in the high-performance and scientific computing community due to their ability to deliver high peak compute rate and memory bandwidth. Moreover, the advent of new development toolkits, such as Nvidia's Compute Unified Device Architecture (CUDA) and OpenCL, offer generic GPU programming models, hence extending the use of these powerful resources to diverse domains that require high computational performance and exhibit large opportunities for data parallelism. At the same time, the GPU's large and fast growing gaming market keeps GPU prices low compared to other accelerators

Experience to date with hybrid platforms powered by GPUs includes reports of significant speedups compared to traditional multicore systems in the same price

range [Hwu 2011; Kirk and Hwu 2010]. These reports ignited zealous debate [Vuduc et al. 2010; Lee et al. 2010] on CPU vs. GPU performance for various classes of applications and on the relative advantages of hybrid architectures.

The fact that GPUs are becoming mainstream in high-performance computing encouraged GPU vendors to evolve the GPU to support features required by the HPC community, such as improving the performance of double precision calculations, support for atomic operations and increasing the internal memory space [NVIDIA 2012]. Also, this motivated building large-scale GPU-accelerated systems: as of writing this thesis, a number of the first ten supercomputers in the Top500 supercomputer list are GPU-accelerated[2].

However, while pervious works focused on regular applications (such as linear algebra), little experience has been accumulated to date related to using hybrid GPU-accelerated platforms to improve the performance of irregular applications, especially the ones that process massive datasets. Indeed, the GPU's strict parallel model and limited onboard memory, among other challenging characteristics, makes it unclear if it is beneficial to offload part of the workload of an irregular application to be processed concurrently on the GPU. Section 1.4 discusses in detail the challenges and the opportunity.

1.2 Irregular Applications

While regular applications, such as dense matrix computations, perform structured computations and operate on easy to partition datasets, irregular applications perform data-dependent computations and operate on unstructured datasets [Pingali et al. 2009]. The fact that current computing systems are optimized for data locality (e.g., employing deep caching hierarchies) and regular computation (e.g.,

[2] www.top500.org

employing SIMD parallel models) makes it challenging to efficiently execute irregular applications on them.

This dissertation focuses on two important applications from this class. The first is the generic problem of graph processing (1.2.1), the second is DNA sequence alignment (1.2.2). The rest of this section discusses the importance and the specific characteristics of each of these two applications.

1.2.1 Graph Processing

Graphs are the core data structure for problems that span a wide set of domains, from social networks [Gupta et al. 2013], to genomics [Pop 2009], to business and information analytics [Iori et al. 2008]. In these domains, key to our ability to transform raw data into insights and actionable knowledge is the ability to process large graphs efficiently and at a reasonable cost.

Imagine, for example, an advertising system for an online social network with hundreds of millions of users. In this example, graph centrality algorithms, such as Betweenness Centrality, can be used to identify the influential actors in the network, which is of considerable importance in order to attract the attention of the largest possible audience to a brand [de Valck et al. 2009; Kiss and Bichler 2008].

Graph workloads and algorithms exhibit a number of key characteristics relevant to the goal of accelerating this application using GPU-accelerated platforms. These characteristics are summarized as follows:

- *Modest processing requirements per vertex in each round*. A typical graph algorithm processes a graph in rounds and, in each round, only a subset of vertices may be active and may be processed in parallel. For example, in Breadth-first Search (BFS), each vertex iterates over its neighbors attempting to set their depth. Similarly in PageRank (a ranking algorithm typically used to rank web pages [Page et al. 1999]), each vertex computes, in each iteration, a new rank by accumulating the ranks of its neighbors.

- *Poor locality.* The topology of many real-world graphs, such as online social networks and web graphs, makes it hard to find a memory layout with good locality, and hence the neighbors of a vertex are typically scattered in memory. Therefore, iterating over the neighbors of a vertex results in accesses that are not spatially local and renders graph computation memory latency bound.
- *Imbalanced workload distribution.* Many relevant real graphs have power-law degree distribution: a few vertices have many edges and most vertices have only one or a few edges. Examples of such graphs include social networks [Ahn et al. 2007], the Internet [Faloutsos et al. 1999], the World Wide Web [Barabási et al. 2000], financial networks [Iori et al. 2008], protein-protein interaction networks [Jeong et al. 2001], semantic networks [Steyvers and Tenenbaum 2005] and airline networks [Wang and Chen 2003]. The skewed distribution of edges in real-world graphs leads to imbalanced workload distribution across vertices, where the high-degree vertices imply heavier processing tasks.
- *Large memory footprint.* Efficient graph processing requires the whole graph to be present in memory, and large real graphs can occupy gigabytes to terabytes of space. For example, a snapshot of the current Twitter follower network has over 500 million vertices and 100 billion edges, and storing it requires at least 0.5TB of memory.

1.2.2 Sequence Alignment

The second application this thesis focuses on is a key bioinformatics problem called sequence alignment (also known as 'read alignment') [Trapnell and Salzberg 2009]: a widely-used step in computational biology pipelines such as comparative genomics and genome assembly. Sequence alignment aims to find all occurrences of each sequence of a large set (millions to billions) of short sequences in another, much longer sequence, called the 'reference sequence'. In this context, sequences

are strings formed using the alphabet {A,C,G,T}, where those letters refer to what is formally called nucleotides.

The importance of this application comes from the fact that DNA sequencing, the biochemical process of determining the order of nucleotides in a DNA molecule, have taken major steps towards commoditization [Venter 2010]. Moreover, sequencing rates have drastically increased: almost 100 billion nucleotides per day per machine, which is 50,000 times faster than ten years ago [Venter 2010; Kaiser 2008; Abecasis et al. 2012].

This dramatic increase in sequencing rates shifted the bottleneck in the ability to generate new knowledge from sequencing (i.e., the biochemical process of generating raw data) to the sequence analysis pipeline (i.e., the computer analysis tools that extract knowledge from the raw data). In fact, there is an increasingly growing gap between sequence generation and analysis [McPherson 2009; Ward et al. 2013]. Accordingly, improving the performance of sequence analysis tools, such as sequence alignment, is becoming more critical.

One important sequence analysis tool is sequence alignment. The following list summarizes its main characteristics:

- *Memory bound.* The core of the sequence alignment problem is a basic substring matching operation: find a string of length *m* in another reference string of length *n*, where $n>>m$. This problem is memory bound as no significant number-crunching or floating point computation is performed.
- *Poor locality.* The typical approach to solve the sequence alignment problem is to pre-process the long reference string into a data structure that allows for efficient search. The queries are then streamed through the in-memory data structure (e.g., a "suffix tree" [Kurtz et al. 2004]). Since the different queries search different parts of the data structure (e.g., in the case of a suffix tree, different queries traverse different branches of the tree), memory access locality is hard to achieve.

- *Large memory footprint.* Depending on the species, the length of the genome reference sequence ranges from few million nucleotides (e.g., for *bacteria*), to few billion nucleotides (e.g., for *Homo sapiens*), to hundreds of billions nucleotides (e.g., for *Amoeba dubia*). Hence, the resulting search-efficient in-memory data structure is large.

1.3 Research Questions

This thesis explores opportunities to use commodity hybrid GPU-accelerated platforms to accelerate irregular applications. This section presents the high-level questions that guide this exploration (Sections 2.2 and 3.2 present more detailed research questions specific to the two applications this thesis is focused on):

- *Is it feasible to harness GPUs to accelerate irregular applications?* In particular, what are the general challenges to support processing data-intensive irregular applications on a single-node GPU-accelerated system?
- *How does combining two processing elements, traditional CPUs and GPUs, with different performance characteristics affects the design of irregular applications?* For example, for a given class of irregular problems, how does this combination impact the way the workload is partitioned and/or the choice of the core data structures?
- *What is the optimal balance between traditional and massively-parallel processing elements?* For example, for a fixed power or dollar budget, should one assemble a machine with four CPUs or the same performance can be obtained with one CPU and one GPU?

Addressing these questions is important in the context of current hardware trends: as the relative cost of energy continues to increase relative to the cost of silicon, future systems will host a wealth of different processing units. In this context, partitioning the workload and assigning the partitions to the processing

element where they can be executed most efficiently in terms of power or time becomes a key issue.

1.4 Opportunities and Challenges

Hybrid GPU-accelerated platforms bring a number of advantages, however utilizing such systems to accelerate irregular applications is challenging. The rest of this section discusses the opportunities (1.4.1) and challenges (1.4.2) of this platform-application matchup.

1.4.1 Opportunities

GPU-accelerated platforms offer a number of properties that one can potentially harness to improve the performance of irregular applications.

First, *a hybrid GPU-accelerated platform has the potential to cope with the heterogeneous structure of irregular workloads*. In particular, a platform that hosts both processing units optimized for fast sequential processing and units optimized for bulk processing matches well the heterogeneous structure of irregular workloads, which have variable levels of parallelism. For example, most graphs processed in practice have power-law degree distribution [Ahn et al. 2007; Barabási et al. 2000; Faloutsos et al. 1999; Iori et al. 2008; Jeong et al. 2001] where few vertices have many edges and many vertices have only one or few edges.

Second, *GPUs offer massive hardware multithreading that is able to hide memory access latency, a main bottleneck for irregular applications*. Current traditional multiprocessors are optimized for data locality and regular computations, which often lead to poor performance when processing irregular applications. GPUs, however, support orders of magnitude more in-flight memory requests while still performing useful work, and thus masking memory access latency. This is important to improve the performance of irregular applications, which depend heavily on data-dependent memory access patterns.

Third, *GPUs are commodity accelerators*. Therefore, GPU-accelerated platforms have the potential to provide cost effective solutions for the performance-hungry irregular applications. Within the same dollar (and power) budget, GPUs offer a significant increase in peak performance, and incorporate capabilities that turned them from dedicated graphics engines to a generally-programmable, highly-parallel processors featuring peak processing and memory bandwidth that exceed their CPU counterparts. This increase in GPUs' performance and programmability suggests that they have the potential to be cost-effective solutions for a broad range of performance-demanding problems.

1.4.2 Challenges

Large-scale irregular applications pose, however, a number of important challenges to GPU-accelerated platforms.

First, *the GPU execution model is significantly different, and arguably more complex, than that of traditional, multicore CPUs*. For example, GPUs offer massive parallelism in an execution model known as single-instruction multiple-thread (SIMT), which offers a tradeoff between performance and programming flexibility. In a SIMT model, a group of scalar threads executes the same instruction on multiple data items at each point in time; however, to enable programming flexibility, the model allows for the threads to diverge at the expense of reduced performance. Another example is that GPUs offer software controlled cache which, if the application uses to improve performance, requires explicit management of data movement between the memory and the cache.

Second, *past experience on performance-efficient data structures and workload partitioning techniques need to be reconsidered when porting applications to hybrid GPU-accelerated platforms*. This is because GPUs and CPUs offer different computational tradeoffs. On the one hand, GPUs offer an order of magnitude higher peak memory access bandwidth and peak computational

power compared to traditional multiprocessors. On the other hand, current GPUs have limited, often an order of magnitude lower, internal memory space. This challenge is magnified by the fact that the applications this work targets operate on massive datasets that significantly exceeds the memory space available in current GPU models. Examples of such datasets include social networks (billions of users and connections) and bioinformatics data (DNA sequences of billions of nucleotides long).

Third, *efficiently scheduling data transfers and finding a low coupling point that limits the overhead of CPU-GPU data transfers.* General purpose CPUs and GPUs are connected to separate memories with different characteristics: the CPUs' memory is designed to minimize latency, while the GPUs' onboard memory focuses on maximizing data throughput [Gelado et al. 2010]. Moreover, the two memory modules are typically connected via high latency I/O channels (e.g., PCI Express). Therefore, there is a need for explicit management of data transfers and consistency between the two memory spaces, which is critical for emerging irregular applications as they operate on massive datasets.

Finally, *balancing the system's hardware configuration (i.e., finding an optimal CPUs to GPUs ratio), from a performance or energy optimization perspective, is a design space that has been scarcely explored*. Today's hardware supports attaching multiple GPUs to existing compute nodes. However, it is not clear how systems, both hardware and software, scale with increasing number of GPUs. More importantly, dynamically detecting the optimal configuration for performance or energy utilization is still an open problem.

The next section presents the methodology I followed to realize the opportunities and overcome the challenges posed by utilizing GPU-based platforms to accelerate irregular applications.

1.5 Methodology

This work follows a top-down methodology: it is driven by two high-impact irregular applications, each with unique computational characteristics: graph processing (Chapter 2) and DNA sequence alignment (Chapter 3). The methodology consists of the following high-level steps:

- ***Performance modeling***. Both projects start by developing performance models to preliminary assess the feasibility of accelerating the application by offloading part of the computation to the GPU. The models take into account a number of key aspects such as the parallel processing model, the characteristics of the processing elements and the communication among these elements.
- ***Application and middleware design and prototyping***. Informed by the performance model and the characteristics of the application, I designed and developed prototypes optimized for GPU-based platforms. The design challenges include maximizing the utilization of the GPU's limited memory space, reducing CPU-GPU communication overhead and matching the workload with the processor it is allocated to. Specifically, I designed a generic graph processing engine named TOTEM, which supports a wide range of graph algorithms. In the second project, I designed and prototyped a new DNA sequence alignment tool named MUMMERGPU++.
- ***Prototype evaluation and model validation***. Using large-scale synthetic and real-world workloads, I performed extensive evaluation of the prototypes. The evaluation includes detailed analysis of the overheads and observed performance (e.g., using performance counters when applicable). It is important to stress here that I sought the largest available real-world workloads related to the two applications.

Note that in terms of chronological order I worked first on the sequence alignment problem then on graph processing. The sequence alignment project

served as an initial exercise to explore the characteristics and capabilities of the GPU to accelerate an irregular and data-intensive application. The experience obtained from this project was essential to the success of the graph processing project, which targeted a more generic, and potentially higher impact problem.

1.6 Summary of Contributions

This section summarizes the contributions of this work. While the contributions presented here are high-level, Section 2.3 and Section 3.4 discuss in detail the contributions in the context of the two applications driving this thesis, graph processing and sequence alignment, respectively.

The contributions of this work are summarized as follows:

- *Demonstrate that hybrid GPU-accelerated platforms can improve the performance, in terms of both time to solution and energy, of irregular applications* (Sections 2.8, 2.9, 2.10, 2.11, 2.12, 3.9 and 3.9.4). While there is no shortage of work that shows the ability of GPU-accelerated platforms to improve the performance of regular applications, this dissertation provides evidence that such hybrid platforms can also accelerate a more challenging class of applications, namely irregular applications. In fact, to the best of my knowledge, this is the first work to explore and demonstrate the benefits of partitioning large-scale graph workloads to be processed concurrently on the CPU and the GPU.
- *Performance models that assess the feasibility of accelerating large-scale irregular problems on hybrid GPU-accelerated platforms* (Section 2.5 and Section 3.8). The models take into account only a small number of key aspects such as the parallel processing model, the characteristics of the processing elements, and the properties of the communication channel among these elements. The models support the intuition that keeping the communication

overhead low is critical for efficient processing on GPU-based systems and it prompted the exploration of compression techniques to reduce these overheads.

- *Novel low-cost workload assignment and design techniques customized for processing on hybrid platforms* (Section 2.9 and Section 3.7). Since the abovementioned optimizations eliminate communication as a major bottleneck, this work proposes partitioning and design strategies that aim to improve the performance of the computation phase. In the case of graph processing, these strategies aim to partition the graph such that the workload assigned to the bottleneck processing element utilizes well the element's strengths. In the same spirit, in the sequence alignment case, this work explores employing data structures that better matches the hybrid system's characteristics.
- *Key optimizing techniques to reduce communication overheads* (Section 2.6 and Section 3.7). This work designs and evaluates two main techniques to reduce communication overheads. First, in the context of graph processing, this work shows that aggregating at the source processor messages targeted to the same remote destination vertex significantly reduces communication overheads. Moreover, in the context of sequence alignment, this work shows that trading-off higher computational complexity for a more compact in-memory representation increases overall performance by reducing data transfer overheads. In both cases, an important consequence is that the computation phase becomes the dominating overhead.
- *Comparison with other platforms* (Section 2.13 and Section 3.9). TOTEM favorably compares with other graph processing platforms, such as Galois [Nguyen et al. 2013], Ligra [Shun and Blelloch 2013] and PowerGraph [Gonzalez et al. 2012]. For example, the performance of TOTEM on a modest one CPU socket and one GPU hybrid setup speeds up the performance by more than 2x compared to the best performance achieved by state-of-the-art frameworks on a shared-memory machine. Moreover, MUMMERGPU++

achieves, on realistic workloads, significant speedups compared to previous highly optimized CPU [Kurtz et al. 2004] and GPU-based [Schatz et al. 2007; Trapnell and Schatz 2009] implementations.

- *Open-source software artifacts* (Section 2.6 and Section 3.7). This work resulted in two main software artifacts that embed the ideas presented in this thesis. First, TOTEM: a generic graph processing framework for GPU-accelerated platforms. TOTEM enables efficiently using all CPU and GPU cores on a given node all while limiting the development complexity. In addition to the techniques discussed previously, TOTEM applies a number of algorithm-agnostic optimizations that lead to performance improvements. Second, MUMMERGPU++: a fully compatible GPU port of the widely used sequence alignment tool MUMMER [Kurtz et al. 2004]. MUMMERGPU++ is part of Nvidia's bioinformatics benchmark and has been used as a benchmark for many research projects on GPU hardware design (e.g. [Fung and Aamodt 2011; Rhu and Erez 2012]).

1.7 Dissertation Structure

The rest of this dissertation presents the two main projects that I conducted to explore accelerating irregular applications using GPU-based systems. Chapter 2 presents the effort to efficiently process large-scale graphs, while Chapter 3 discusses the effort to efficiently process DNA sequence alignment computation on GPU-accelerated systems. Each chapter discusses in detail the performance model, the system design and evaluation on synthetic and/or real-world workloads. Finally, Chapter 4 presents a summary of the dissertation and highlights its impact.

Chapter 2

Efficient Large-Scale Graph Processing on Hybrid Platforms

The increasing scale and wealth of inter-connected data, such as those accrued by social network applications, demand the design of new techniques and platforms to efficiently derive actionable knowledge from large-scale graphs. However, large real-world graphs are famously difficult to process efficiently. Not only they have a large memory footprint, but also most graph algorithms entail memory access patterns with poor locality, data-dependent parallelism and a low compute-to-memory access ratio. To complicate matters further, most real-world graphs have a highly heterogeneous node degree distribution (i.e., they are scale-free), hence partitioning these graphs for parallel processing and simultaneously achieving access locality and load-balancing is difficult.

This work starts from the hypothesis that hybrid platforms (e.g., GPU-accelerated systems) have both the potential to cope with the heterogeneous

The research presented in this chapter resulted in the following publications:

(i) **Abdullah Gharaibeh** *et al.*, The Energy Case for Graph Processing on Hybrid CPU and GPU Systems, IEEE Workshop on Irregular Applications: Architectures & Algorithms (IA^3) in conjunction with SC13, Denver, Colorado USA, November 2013 (*30% acceptance rate*).

(ii) **Abdullah Gharaibeh** *et al.*, On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest, IEEE International Parallel & Distributed Processing Systems (IPDPS), Boston, MA, May 2013 *(21% acceptance rate)*.

(iii) **Abdullah Gharaibeh** *et al.,* A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing, IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques (PACT). Minneapolis, MN September 2012 (*19% acceptance rate*).

structure of scale-free graphs and to offer a cost-effective platform for high-performance graph processing.

This research assesses the above hypothesis and presents an extensive exploration of the opportunity to harness hybrid GPU-accelerated platforms to process large scale-free graphs efficiently. In particular, *(i)* this work presents a performance model that estimates the achievable performance on hybrid platforms; *(ii)* informed by the performance model, I designed and developed **TOTEM** – a processing engine that provides a convenient environment to implement graph algorithms on hybrid platforms; *(iii)* this work shows that further performance gains can be extracted using partitioning strategies that aim to produce partitions that matches the strengths of the processing element it is allocated to, finally, *(iv)* it demonstrates the performance advantages of the hybrid system through a comprehensive evaluation that uses real and synthetic scale-free workloads (as large as 16 billion edges), multiple graph algorithms that stress the system in various ways, and a variety of hardware configurations.

2.1 Context

Processing large-scale graphs efficiently and at a reasonable cost is key to many domains. However, a major challenge when processing large graphs is their memory footprint: efficient graph processing requires the whole graph to be present in memory, and large real graphs can occupy gigabytes to terabytes of space. For example, a snapshot of the current Twitter follower network has over 500 million vertices and 100 billion edges, and requires at least 0.5TB of memory. As a result, the most commonly adopted solution to cost-efficiently process large-scale graphs is to partition them and use shared-nothing cluster systems [Malewicz et al. 2010; Gonzalez et al. 2012].

However, today more efficient solutions are affordable: it is feasible to assemble single-node[3] platforms that aggregate 100s of GB to TBs of RAM and massive computing power [Gupta et al. 2013; Rowstron et al. 2012; Shun and Blelloch 2013] all from commodity components and for a relatively low budget. Compared to clusters, single-node platforms are easier to program, and promise better performance and energy efficiency for a large class of real-world graph problems. In fact such single-node graph processing platforms are currently being used in production: for example, Twitter's 'Who To Follow' (WTF) service, which uses the follower network to recommend connections to users, is deployed on a single node [Gupta et al. 2013].

Despite these recent advances, single-node platforms still face a number of performance challenges. First, graph algorithms have low compute-to-memory access ratio, which exposes fetching/updating the state of vertices (or edges) as the major overhead. Second, graph processing exhibits irregular and data-dependent memory access patterns, which lead to poor memory locality and reduce the effectiveness of caches and pre-fetching mechanisms. Finally, many real-world graphs have a highly heterogeneous vertex degree distribution (i.e., they have power-law degree distribution and are commonly named "scale-free") [Barabási 2003; Barabási et al. 2000; Jeong et al. 2001; Iori et al. 2008], which makes dividing the work among threads for access locality and load-balancing difficult.

In this context, two reasons (summarized here and detailed in Section 2.4) support the intuition that single-node GPU-accelerated platforms may be an appealing platform for high-performance, low-cost graph processing: First, GPUs bring massive hardware multithreading that is able to mask memory access latency

[3] In this dissertation, node is used to refer to processing elements (i.e., machines, processors), while vertex is used to refer to the graph element.

– the major barrier to performance for this class of problems. Second, a hybrid system that hosts processing units optimized for fast sequential processing and units optimized for bulk processing matches well the heterogeneous structure of the many scale-free graphs that need to be processed in practice.

2.2 Research Questions

This work investigates the feasibility and the comparative advantages of supporting graph processing of scale-free graphs on hybrid, GPU-accelerated nodes. The following research questions guide this investigation:

Q1. *Is it feasible to efficiently combine traditional CPU cores and massively parallel processors (e.g., GPUs) for graph processing?* In particular, what are the general challenges to support graph processing on a single-node GPU-accelerated system?

Q2. Assuming that a low-level engine can efficiently process large graphs on hybrid nodes, *what would an abstraction that aims to simplify the task of implementing graph algorithms look like*?

Q3. *How should the graph be partitioned to efficiently use both traditional CPU cores and GPU(s)?* More specifically, are there low-complexity partitioning algorithms that generate partitions that match well the individual strengths of CPUs and GPUs?

Q4. *Is it energy-efficient to partition the graph to be processed concurrently on a GPU and a CPU?* While GPUs are known for their energy efficiency when processing regular, compute intensive workloads (such as matrix computations), it is unclear whether this can be preserved for irregular, memory-bound problems like graph processing.

Addressing these questions is important to inform the design of graph-workload partitioning solutions that aim to optimally harness hybrid computing

platforms. In the context of current hardware trends, as the cost of energy continues to increase relative to the cost of silicon, future systems will host a wealth of different processing units. In this hardware landscape, the key issue will become partitioning the workload and assigning the partitions to (possibly, a subset of) the existing processing elements where the workload can be executed most efficiently in terms of power, energy, or time.

2.3 Contributions and Chapter Structure

This work demonstrates that partitioning large scale-free graphs to be processed concurrently on hybrid CPU and GPU platforms offers significant performance and energy gains. Moreover, this work defines the class of partitioning algorithms that will enable best performance on hybrid platforms: these algorithms should focus on shaping the workload to best match the bottleneck processing engine (rather than on minimizing communication overheads). Finally, this work experiments with a few partitioning solutions from this class, analyze the observed performance, and propose guidelines for when they should be used. In more detail, the contributions are:

- *A performance model* (Section 2.5) to assess the feasibility of accelerating large-scale graph processing by offloading a graph partition to the GPU. The model is agnostic to the exact graph processing algorithm, and it takes into account only a small number of key aspects such as the parallel processing model, the characteristics of the processing elements, and the properties of the communication channel among these elements. The model supports the intuition that keeping the communication overhead low is crucial for efficient graph processing on hybrid systems and it prompts exploring the benefits of message aggregation to reduce these overheads.

- *TOTEM[4]: an open-source graph processing engine for GPU-accelerated platforms* (Section 2.6). TOTEM efficiently uses all CPU and GPU cores on a given node, while limiting the development complexity. Guided by the performance model, TOTEM applies a number of algorithm-agnostic optimizations that lead to performance improvements. One key optimization this work introduces is reducing communication overhead by over an order of magnitude by aggregating messages at the source processor.
- *Insights into key performance overheads* (Section 2.8). Using TOTEM's abstractions, a number of graph processing algorithms have been implemented that stress the hybrid system in different ways. This work demonstrates that the gains predicted by the model are achievable in practice when offloading a random partition to the GPUs. Moreover, this work shows that the optimizations applied by TOTEM significantly reduce the overheads to communicate among the processing elements, and that the computation phase becomes the dominating overhead.
- *Low-cost partitioning strategies tailored for processing scale-free graphs on hybrid systems* (Section 2.9). Since the applied optimizations eliminate communication as a major bottleneck, this work focuses on partitioning strategies that aim to reduce the computation bottleneck. These strategies aim utilize the heterogeneity in scale-free graphs to produce partitions such that the workload assigned to the bottleneck processing element exploits well the element's strengths. The proposed partitioning strategies are informed by vertex-connectivity, and lead to super-linear performance gains with respect to the share of the workload offloaded to the GPU.

[4] The code can be found at: http://netsyslab.ece.ubc.ca

- *Detailed evaluation of the impact of possible partitioning strategies* (Section 2.9.3, Section 2.10 and Section 2.11). The reasons for the observed performance impact is explained in detail (e.g., using hardware counters and pseudocode analysis) using large-scale, real-world and synthetic graphs of different sizes (from 2 to 16 billion edges) and various hardware configurations. The experiments include different graph algorithms that stress the platform in various ways. To the best of my knowledge, this is the first work to evaluate graphs as large as 1 billion vertices and 16 billion edges on a single-node commodity machine.
- *Application evaluation* (Section 2.10). Using large real-world workloads, this work demonstrates that the gains offered by the GPU-accelerated platform hold for key applications: ranking web pages using PageRank, finding the main actors in a social network using Betweenness Centrality algorithm and computing point-to-point shortest paths in a network using Single-Source Shortest Path algorithm.
- *Comparison with other frameworks* (Section 2.13). TOTEM favorably compares with other frameworks including Galois, Ligra and PowerGraph: the performance of TOTEM on a modest one CPU socket and one GPU hybrid setup speeds up the performance by more than 2x compared to the best performance achieved by state-of-the-art frameworks on a shared-memory machine with four high-end CPU sockets.
- *Guidelines* (Section 2.15). The results presented in this work allow putting forward a number of guidelines related to the opportunity and the supporting techniques required to harness hybrid systems for large-scale graph processing problems. Notably, the guidelines describe which partitioning strategy to use given a workload and an algorithm.

The importance of this work comes from all these contributions. Firstly, the performance model not only encourages the design of GPU-offloading techniques, but can guide hardware purchase and software design decisions for various classes of graph-related problems. Secondly, this work is the first to demonstrate the feasibility of using, in parallel, all CPU and GPU processors of a hybrid platform for a key class of irregular problems: graph processing. Finally, the processing engine that resulted from this work, TOTEM, offers an efficient and easy to use environment to develop graph applications that can benefit from acceleration.

2.4 Opportunities and Challenges

Section 2.1 discussed the general challenges of single-node graph processing. This section details the opportunities and challenges brought by GPU acceleration in this context.

The opportunities: GPU-acceleration has the potential to offer the key advantage of massive, hardware-supported multithreading. In fact, current GPUs not only have much higher memory bandwidth than traditional CPU processors, but can mask memory access latency as they support orders of magnitude more in-flight memory requests through hardware multithreading.

Additionally, properly mapping the graph-layout and the algorithmic tasks between the CPU(s) and the GPU(s) holds the promise to exercise each of these computing units where they perform best: CPUs for fast sequential processing (e.g., for the few high degree nodes of a power-law graph) and GPUs for the bulk parallel processing (e.g., for the many low-degree nodes).

In particular, this work focuses on harnessing the heterogeneity of vertex degree distribution in scale-free graphs. For example, the few high-degree vertices can be processed by the CPU, while the many low-degree ones can be processed on the GPU. While this limits the scope of this work, it still benefits various high-

impact applications as many real-world graphs are scale-free. Examples of such graphs include social networks [Kwak et al. 2010], the Internet [Faloutsos et al. 1999], the World Wide Web [Barabási et al. 2000], financial networks [Iori et al. 2008], protein-protein interaction networks [Jeong et al. 2001], and airline networks [Wang and Chen 2003] to mention few.

The challenges: Large-scale graph processing poses two major challenges to hybrid systems. First, the large amount of data to be processed, and the need to communicate between processors put pressure on two scarce resources: the GPUs' on-board memory and the host to GPU transfer bandwidth. Intelligent graph representation, partitioning and allocation to compute elements are key to reduce memory pressure, limit the generated PCI Express bus transfer traffic and efficiently harness each processing element in an asymmetrical platform.

Second, to achieve good performance on GPUs, the application must, as much as possible, match the SIMD computing model. As graph problems exhibit data-dependent parallelism, traditional implementations of graph algorithms lead to low memory access locality. Nevertheless, GPUs are able to hide memory access latency via massive hardware multithreading that, with careful design of the graph data structure and thread assignment, can reduce the impact of these factors.

Finally, mapping high-level abstractions (e.g., vertex-centric processing) and APIs to facilitate application development to the low-level infrastructure while limiting the efficiency loss, is an additional challenge.

2.5 Modeling Hybrid Systems' Performance

The model aims to provide insights to answer the following question: *Is it beneficial to partition the graph and process it on both the host and the GPU (compared to processing on the host only)?*

It is worth stressing that the goal is a simple model that captures the key characteristics of a GPU-accelerated platform, highlights its bottlenecks, and helps reasoning about the feasibility of offloading. I deliberately steer away from a complex (though potentially more accurate) model, the evaluation validates this choice.

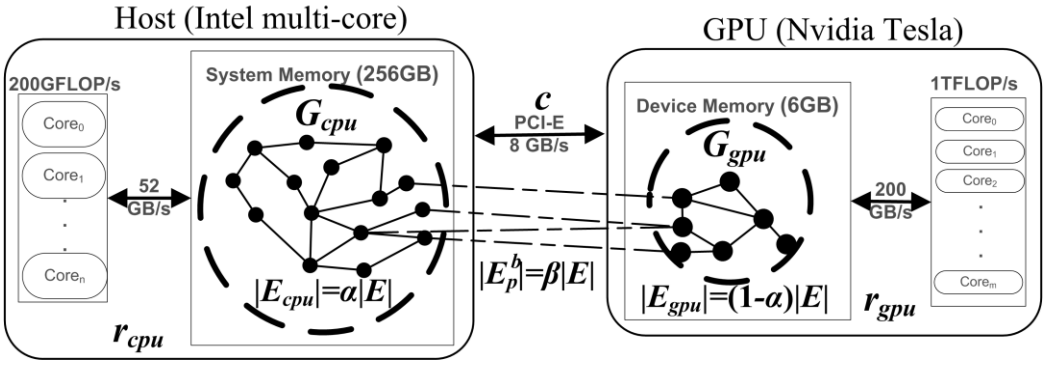

Figure 1: An illustration of the model, its parameters, and their values for today's state-of-the-art commodity components.

r_{cpu} r_{gpu} *Processing rates on the CPU and GPU*
c *Communication rate between the host and GPU*
α *Ratio of the graph edges that remain on the host*
β *Ratio of edges that cross the partition*

2.5.1 Notations and Assumptions

Let $G = (V, E)$ be a directed graph, where V is the set of vertices and E is the set of directed edges; $|V|$ and $|E|$ represent their respective cardinality. Also, let $P = \{CPU, GPU\}$ be the set of processing elements of a hybrid node (Figure 1). While the model can be easily generalized to a mix of multiple CPUs and GPUs; for the sake of simplicity, a setup with only two processing units is used here.

The model makes the following assumptions:

(i) Each processing element has its own local memory. The processing elements are connected by a bidirectional interconnect with communication rate c measured in edges per second (E/s) – this is a reasonable unit as the time complexity of a large number of graph algorithms depends on the number of

edges in the graph. But, the same model can be recast in terms of vertex-centric algorithms by normalizing by the number of vertices instead of edges.

(ii) Once the graph is partitioned, the GPU processes its partition faster. This is because: first, GPUs have a higher graph processing rate than CPUs (based on published results [Hong et al. 2011a; Hong et al. 2011b], which I validated independently); second, GPUs have significantly less memory than the host, which limits the size of the offloaded partition.

(iii) The model assumes the overheads of scheduling the workload (e.g., partitioning the graph) and gathering the results produced by each processor are negligible compared to the algorithm's processing time.

2.5.2 The Model

Under the assumptions stated in the previous section, the time to process a partition of G, $G_p = (V_p, E_p) \subseteq G$ on a processing element p is given by:

$$t_p(G_p) = \frac{\left|E_p^b\right|}{c} + \frac{\left|E_p\right|}{r_p} \quad (1)$$

where r_p is the processing rate of processor p (in edges/s), and $E_p^b \subseteq E_p$ represents the subset of *boundary edges* – edges where either the source or the destination vertex is *not* located in p's local memory.

Equation 1 estimates the time required to process a partition as a combination of the time it takes to communicate possible updates through boundary edges (communication phase) plus the time it takes to process the edges in that given partition on processor p (computation phase). Intuitively, the higher the processing rate of a processing element, the lower is the processing time. Similarly, the less communication a processing element needs to access the edges of the other partition, the lower is the processing time.

Building on Equation 1, the makespan[5] of a graph workload *G* on a given hybrid node with a set of processing elements *P* can be defined as follows:

$$m_P(G) = \max_{p \in P} \{t_p(G_p)\} \quad (2)$$

The intuition behind Equation 2 is that the performance of a parallel system is limited by its slowest component. Since, as discussed before, the model assumes that the host processes its partition slower than the GPU (assumption *ii*), resulting that the time spent on processing the CPU partition is always higher than that of the GPU partition (i.e., $t_{cpu}(G_{cpu}) > t_{gpu}(G_{gpu})$).

Hence, the improvement brought by processing a graph on a hybrid platform (compared to processing it on the host only) can be calculated by Equation 3, as follows:

$$s(G) = \frac{t_{cpu}(G)}{m_P(G)} = \frac{t_{cpu}(G)}{t_{cpu}(G_{cpu})} = \frac{|E|/r_{cpu}}{|E^b_{cpu}|/c + |E_{cpu}|/r_{cpu}} \quad (3)$$

where $t_{cpu}(G)$ is the time it takes to process G, the whole graph, on the CPU, while $t_{cpu}(G_{cpu})$ is the time it takes to process on the CPU the partition assigned to it as a result of partitioning the graph to be processed on both the CPU and the GPU.

To understand the gains resulted from moving a portion of the graph to the GPU, Equation 3 is rewritten by introducing two parameters that characterize the '*quality'* of the graph partition. Let α be the share of edges (out of the total number of graph edges $|E|$) assigned to the host, similarly let β be the percentage of *boundary* edges (i.e., the edges that cross the partition). Introducing these parameters, the speedup can be expressed as follows:

[5] Makespan: the completion time of a graph processing task [Pinedo 2012].

$$s(G) = \frac{|E|/r_{cpu}}{\beta |E|/c + \alpha |E|/r_{cpu}} = \frac{c}{\beta r_{cpu} + \alpha c} = \frac{1}{\frac{\beta \cdot r_{cpu}}{c} + \alpha} \qquad (4)$$

As expected, Equation 4 predicts that a high host-accelerator interconnect communication rate, *c*, improves the speedup. In fact, if *c* is set to infinity, the speedup can be approximated as $1/\alpha$. This is intuitive, as in this case the communication overhead becomes negligible compared to the time spent on processing the CPU's share of edges, and the speedup becomes proportional with the offloaded portion of the graph.

2.5.3 Setting the Model's Parameters

Figure 1 presents an illustration of the model with "reasonable" values for its parameters for a state-of-the-art commodity hybrid platform. They are discussed in turn below:

- *Communication rate* (*c*) is directly proportional to the interconnect bandwidth and inversely proportional to the amount of data transferred per edge. The GPU is typically connected to the host via a PCI Express bus. Latest GPU models support PCI Express Gen 3.0, which has a measured transfer bandwidth of 12GB/sec. If the data transferred per edge is assumed to be 4-byte value (e.g., the "*distance*" in Breadth-first Search or the "*rank*" in PageRank), the transfer rate *c* becomes 3 Billion E/s – or BE/s.
- *CPU's processing rate (r_{cpu})* depends on the CPU's characteristics, the graph algorithm and implementation, and the graph topology. The assumption is that a CPU-only implementation is available and can be run on the machine to obtain r_{cpu}. This is a reasonable assumption as one typically starts off by implementing a CPU version of the algorithm.

- *Percentage of boundary edges (β)* depends on the way the graph is partitioned between the processing elements. In the worst case, all edges cross the partition. Random partitioning leads to an average β=50%.
- *The share of the graph that stays on the CPU (α)* is configurable, but is constrained by the memory space available on the processing elements (for example, larger memory on the GPU allows for offloading a larger partition, hence smaller α).

Figure 2 shows the speedup predicted by the model (Equation 4) for different values of α, while varying the CPU processing rate (left plot) and the percentage of boundary edges (right plot). The values used for the CPU processing rate are informed by the best reported graph processing rates in the literature [Nguyen et al. 2013] for state-of-the-art commodity single-node machines.

The figure indicates that as the CPU processing rate increases (higher r_{cpu}, left plot) or for a graph partition that leads to larger percentage of boundary edges (higher β, right plot), the speedup decreases. This is because the communication overhead becomes more significant.

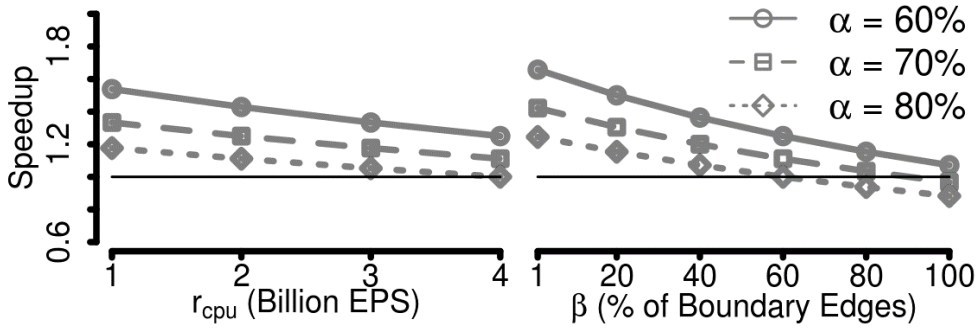

Figure 2: Predicted speedup (values below one indicate slowdown). Left: varying the CPU's processing rate (β is set to 5%). Right: varying the percentage of boundary edges (r_{cpu} is set to 1 BE/s). The communication rate is 3 BE/s.

Nonetheless, the figure indicates that offloading part of the graph to be processed in parallel on the GPU can be beneficial. In particular, if β, is kept low

(below 40% in Figure 2 (right)) the model predicts speedups. The figure also presents a hypothetical worst case where all of the edges are boundary edges (e.g., a bipartite graph where the partition cuts each edge). Even in this case, and due to the high communication rate *c*, a slowdown is predicted only for $\alpha > 70\%$.

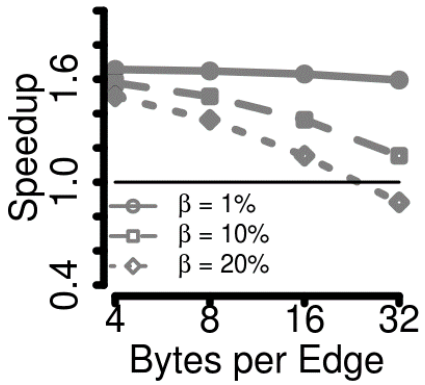

Figure 3: Predicted speedup while varying the volume of transferred data per edge (α is set to 60% and r_{cpu} to 1 BE/s).

Finally, Figure 3 demonstrates the effect of the amount of transferred data per edge on the predicted speedup. As expected, the speedup drops as the amount of transferred data is doubled. However, if β is kept low, the model predicts tangible speedups even when tripling the size of data transferred per boundary edge. To this end, the next section discusses how to keep β low for scale-free graphs, the focus of this work.

2.5.4 Summary

With parameters set to values that represent realistic scenarios, the model predicts speedups for the hybrid platform, even when using naïve random partitioning. Hence, *it can be beneficial to explore this opportunity in more depth by prototyping an engine to partition graphs and process them on a GPU-accelerated platform.*

The next sections discuss an algorithm-agnostic technique to reduce the impact of boundary edges (Section 2.6), and presents the design of a graph processing engine for hybrid platforms (Section 2.7). The discussion then proceeds by showing

that the model offers good accuracy (Section 2.8), demonstrates the advantages of advanced partitioning techniques for a set of graph processing algorithms, workloads, and processing platforms (Sections 2.9-2.11), compares with the performance of state-of-the-art graph processing frameworks (Section 2.13), and evaluates the energy footprint of the hybrid platform (Section 2.12).

2.6 Reducing the Impact of Boundary Edges

This section presents an efficient technique that minimizes β, *i.e.,* the percentage of boundary edges for scale-free graphs and a wide range of graph algorithms.

In particular, the section explores the opportunity to aggregate messages sent from multiple vertices residing in one processing element to a single vertex residing on the other. The intuition behind this optimization is that the power-law nature of scale-free graphs leads to a topology where multiple edges from the same partition point to the high-degree vertices on the other partition and thus enable message aggregation.

Note that aggregation is employed in cluster-based graph processing frameworks [Malewicz et al. 2010] to reduce the communication overhead between partitions residing in different nodes. However, this technique is more effective in the single hybrid node platform this work targets because the expected number of partitions (e.g., two for a system with one GPU) is significantly lower than in the case of a distributed system with hundreds of compute nodes (i.e., hundreds of partitions).

To highlight the benefit of aggregation, this section compares how much communication would happen with and without aggregation when using a naïve random partitioning algorithm. Figure 4 shows β resulted from two- and three- way partitioning, representing setups with one and two GPUs respectively, for real (Twitter and UK-WEB) and synthetic graphs (RMAT28 and UNIFORM28). The graphs are described in detail in section (Section 2.8); for now, the relevant

characteristic that differentiates them is the degree distribution: real-world and RMAT28 graphs are scale-free and have skewed degree distribution, while UNIFORM28 has a uniform distribution.

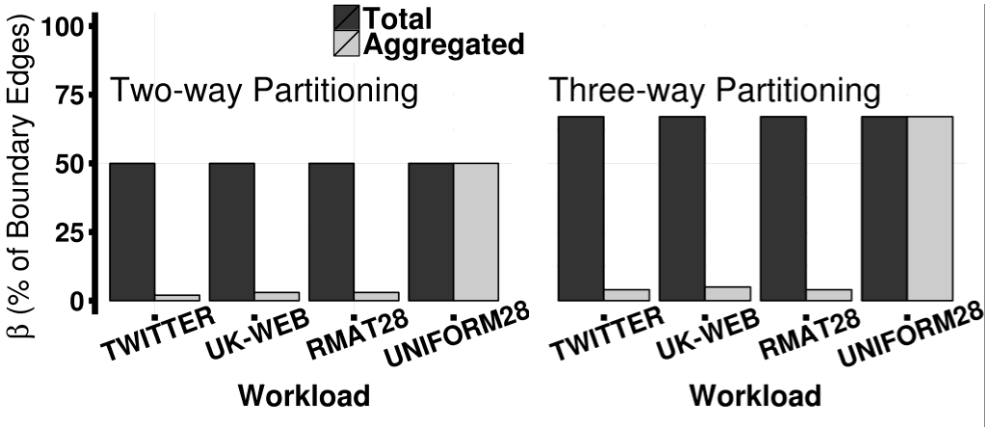

Figure 4: The impact of aggregation. Resulted ratio of edges that cross partitions (*β*) with and without aggregation for two real-world graphs (Twitter and UK-WEB), one synthetic scale-free graph (RMAT28), and one synthetic graph with uniform node degree distribution (UNIFORM28).

The figure shows that aggregation significantly reduces β (to less than 5%) for the graphs with skewed distribution. The worst case input is an Erdős-Renyi random graph [Erdős and Rényi 1960], where an edge exists with independent random probability, and the resultant graph has uniform edge degree distribution.

However, as discussed before, most graphs processed in practice have power-law degree distribution, thus this optimization is useful in practice. Figure 5 shows the degree distribution of the two real-world graphs used in the experiments, and clearly demonstrating the skewed degree distribution among the vertices.

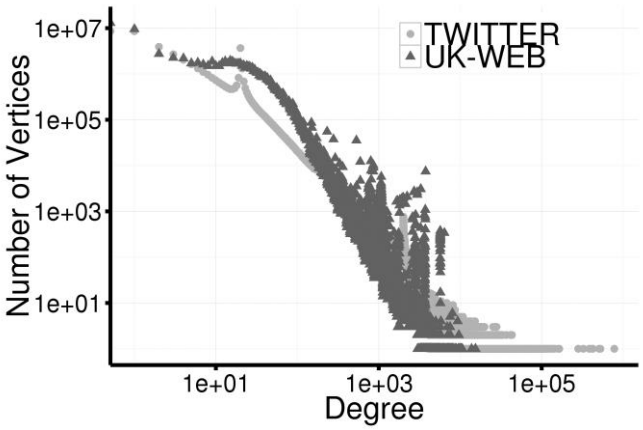

Figure 5: Degree distribution of two instances of real-world graphs (Twitter and UK-Web). Note that the plot is presented in log-log format.

Finally, it is important to mention that aggregation works for algorithms where it is possible to reduce, at the source partition, into one value the values sent to the same remote vertex. Although some graph algorithms cannot benefit from aggregation (e.g., triangle counting), a wide range of graph algorithms has this characteristic. For example, the "visited" status in BFS, minimum "distance" in SSSP, and the "rank" sum in PageRank.

2.7 Totem: A Graph Processing Engine for Hybrid Platforms

To enable application programmers to leverage hybrid platforms, I designed TOTEM – a graph processing engine for hybrid and multi-GPU single-node systems. This section presents TOTEM's programming model (Section 2.7.1 and Section 2.7.2), its implementation (Section 2.7.3), and a discussion of its design trade-offs (Section 2.7.4).

2.7.1 Programming Model

TOTEM adopts the Bulk Synchronous Parallel (BSP) computation model [Valiant 1990], where processing is divided into rounds – *supersteps* in BSP terminology.

Each superstep consists of three phases executed in order: in the *computation phase,* each processing unit executes asynchronously computations based on values stored in their local memories; in the *communication phase*, the processing units exchange the messages that are necessary to update their statuses before the next computation unit starts; finally, the *synchronization phase* guarantees the delivery of the messages. Specifically, a message sent at superstep *i* is guaranteed to be available in the local memory of the destination processing unit only at superstep *i* +1.

Adopting the BSP model allows to circumvent the fact that the GPUs are connected via the higher-latency PCI Express bus. In particular, batch communication matches well BSP, and enables TOTEM to hide (some of) the bus latency. In more detail, TOTEM performs each of these phases as follows:

- *Computation phase.* TOTEM initially partitions the graph and assigns each partition to a processing unit. In each compute phase, the processing units work in parallel, each executing a user-specified kernel on the set of vertices that belongs to its assigned partition.
- *Communication phase.* TOTEM enables the partitions to communicate via boundary edges. The engine stores messages sent to remote vertices in local buffers that are transferred in the communication phase to the corresponding remote partitions. As the performance model shows, reducing communication overhead is paramount to improve performance. The engine achieves such reduction by aggregating at the source processor messages targeted to the same remote destination vertex (as discussed in Section 2.6). The aggregation is performed based on a user-provided callback. Note that the *synchronization phase* is performed implicitly as part of the communication phase.
- *Termination.* The engine terminates execution when all partitions vote to finish (through a user-defined callback) in the same superstep. At this point, the engine invokes another user-specified callback to collect the results from all partitions.

2.7.2 A Programmer's View

A programmer prepares TOTEM to execute a graph algorithm by providing a number of callback functions that are executed at different points in the BSP execution cycle.

The TOTEM framework itself is essentially in charge of implementing the callback API and orchestrating these calls. This hides some of the inherent complexity of developing for a hybrid platform as TOTEM offers a common data representation, abstracts the communication through boundary edges, and hides various low-level optimizations that target the hybrid platform. For example, TOTEM optimizes the data layout to increase access locality, enables transparent and efficient communication between the processing elements, and provides abstractions to handle transparently boundary edges).

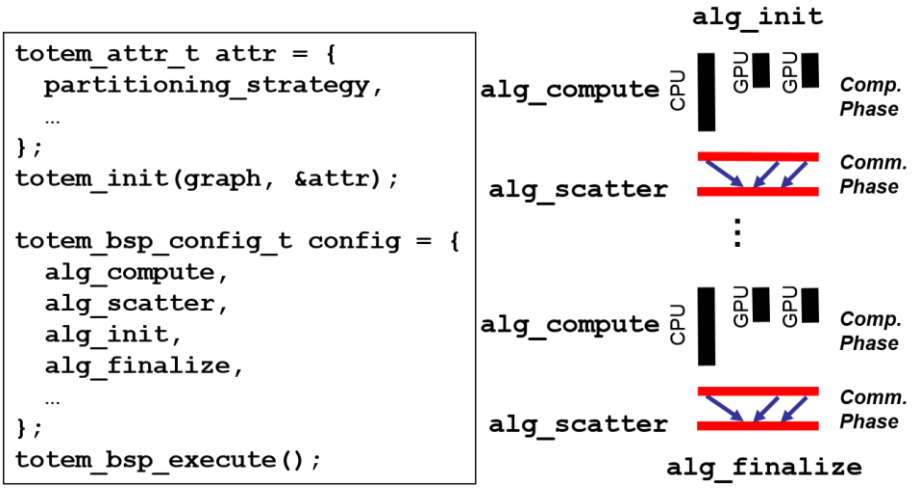

Figure 6: A simplified TOTEM configuration and how an algorithm callbacks map to the BSP phases.

Figure 6 shows a simplified implementation of a graph algorithm using TOTEM (Appendix A presents in detail and with extensive comments how each of these callbacks looks for implementing BFS). TOTEM loads the graph and creates one partition for the host and a partition for each GPU. TOTEM accepts a number of

attributes, most notably is the graph partitioning strategy (discussed in Section 2.9) and the size of each partition. The BSP engine is configured with the algorithm-specific callbacks provided by the user. The `alg_init` callback allows allocating algorithm-specific state (such as the 'level' array in BFS or the 'rank' array in PageRank), the `alg_compute` callback performs the core computation of the algorithm, while `alg_scatter` callback defines how a message received from a boundary edge updates a vertex's state (e.g., update the vertex's state with the sum of the two in the case of PageRank, or the minimum in SSSP). The `alg_finalize` callback enables the framework to release state allocated at initialization. All callbacks are invoked per partition in each BSP round.

Note that each callback has access to the entire graph state stored on the processing element where it executes: this is a programming paradigm that has recently been dubbed "think like a graph" (as opposed to "think like a vertex") [Tian et al. 2013]

Finally, the current version of Totem requires the programmer to provide CPU and GPU versions of these callbacks. While this offers the flexibility to choose the parallel implementation that best suits each processing element, it entails an extra effort. However, recently, new programming models have been proposed to address this problem, most notably is the OpenACC [OpenACC 2012] standard which defines directives that can be used to annotate C/C++ or Fortran programs for expressing parallelism for both accelerators and traditional multiprocessors. Implementations of the OpenACC standard have also become available [Lee and Vetter 2014]. This will allow implementing a single callback for both CPU and GPU partitions, hence improving programmers' productivity.

2.7.3 TOTEM Design and Implementation

TOTEM is open-source, and is implemented in C and CUDA. While a number of aspects related to TOTEM's design and implementation are worth discussing, for

brevity we discuss only two: the data structures used to represent the graph and communication via boundary edges.

2.7.3.1 Graph representation and Additional Data Structures to Support Partitioning

Graph partitions are represented as Compressed Sparse Rows (CSR) in memory [Barrett et al. 1994], a space-efficient graph representation that uses $O(|V| + |E|)$ space. Figure 7 shows an example of a two-way partitioning setup. The arrays *V* and *E* represent the CSR data structure. In each partition, the vertex IDs span a linear space from zero to $|V_p|$-1. A vertex ID together with a partition ID represents a global ID of a vertex. A vertex accesses its edges by using its ID as an index in *V* to fetch the start index of its neighbors in *E*.

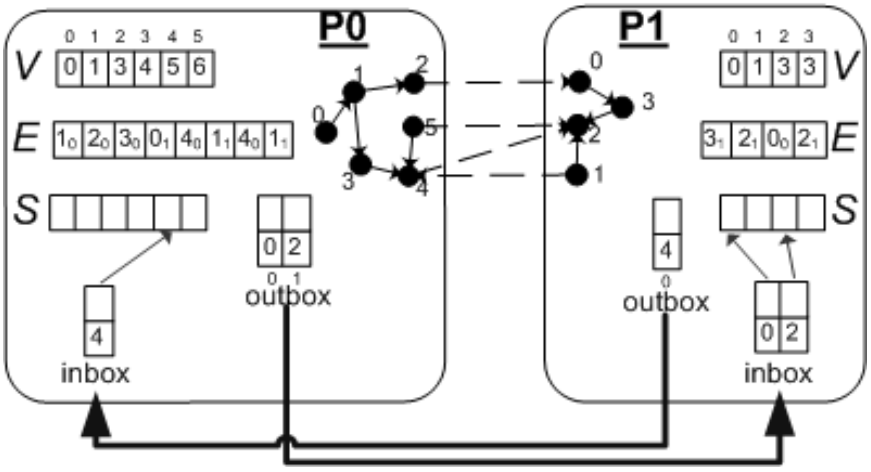

Figure 7: An illustration of the graph data structure and the communication infrastructure in a two-way partitioning setup.

The array E stores the destination vertex of an edge, which includes the partition ID (shown in the figure as subscripts) encoded in the high-order bits. In the case of boundary edges, the value stored in E is not the remote neighbor's ID, rather it is an index to its entry in the outbox buffer (discussed later). To simplify state management, a vertex in a directed graph has access only to its outgoing edges,

which is sufficient for most graph algorithms (undirected edges can be represented as two directed edges, one in each direction).

The array S represents the algorithm-specific local state for each vertex, it is of length $|V_p|$, and is indexed using vertex IDs. A similar array of length $|E_p|$ can be used if the state is required per-edge rather than per-vertex.

The processing of a vertex typically consists of iterating over its neighbors. A neighbor ID is fetched from E, and is used to access S for local neighbors, or the outbox buffer for the remote ones. Typically, accessing the state of a neighbor (either in S or in the outbox buffer) is done via atomic operations as multiple vertices may simultaneously try to update the state of a common neighbor.

To improve pre-fetching, the set of neighbors of each vertex in E are ordered such that the local edges are processed first (entails accessing S), and then the boundary edges (entails accessing the outbox buffers).

To improve pre-fetching, the set of neighbors of each vertex in *E* are sorted and are placed such that the local edges are processed first (entails accessing *S*), and then the boundary edges (entails accessing the outbox buffers).

2.7.3.2 Communication via boundary edges

A challenge for a graph processing engine for hybrid setups is keeping the cost of communication low. TOTEM addresses this problem by using local buffers and user-provided aggregation callbacks. Messages sent via boundary edges in the computation phase of a superstep are temporarily buffered and, if possible, aggregated in these buffers then transferred in the communication phase.

TOTEM maintains two sets of buffers for each processing unit (Figure 7). The outbox buffers have an entry for each remote neighbor, while the inbox buffers have an entry for each local vertex that is remote to another partition. An in/outbox buffer is composed of two arrays: one maintains the remote vertex ID and the other stores the messages.

The outbox buffer in a partition is symmetric to an inbox buffer in another. Therefore, in the communication phase, only the message array is transferred. Once transferred, TOTEM uses the user-defined aggregation function to update the remote neighbors' state in the *S* array at the remote partition with the new values. Similar to *E*, the entries in the inbox buffers are sorted by vertex IDs to improve pre-fetching and cache efficiency when doing the update.

Finally, note that TOTEM allows for two way communication via the boundary edges: a vertex can either "*push*" updates to its neighbors, or "*pull*" (i.e., read) the neighbors state to update its own value. This is a necessary feature for some graph algorithms (e.g., Betweenneess Centrality) and an optimization for others (e.g., PageRank).

2.7.3.3 Summary of Other Optimizations

In addition to the two main optimizations discussed previously: using compressed graph representation to reduce memory footprint and aggregating messages sent over boundary edges, TOTEM employs a number of other optimizations. They have been discovered through an iterative exploration process and provide sizeable gains. The following list summarizes these optimizations:

- *Improving data access locality*. A vertex can have local or remote neighbors. While local neighbors' state can be accessed directly (e.g., via the state array *S* in Figure 7), accessing remote neighbors' state must be done via the outbox buffers. To improve access locality, processing a vertex edges is done by processing its local neighbors first (which requires for all of them accessing the local state array), and then the remote neighbors (which requires for all of them accessing the outbox buffer).
- *Improving load balancing between the CPU and the GPU for large graphs.* The GPU's limited memory space constrains the size of the offloaded partition. This is a major challenge when targeting multi-billion scale graphs. To enable

offloading a larger partition to the GPU, TOTEM allows allocating part of the state on host memory and map it into the GPU's address space. The tradeoff is extra communication overhead. TOTEM reduces this overhead restricting the use of mapped memory to allocate the part of the state that is (*i*) read-only, and (*ii*) can be accessed sequentially in batches. Section 2.11 evaluates multi-billion scale graphs, and elaborates more on this optimization.

- *Improving load balancing across GPU threads.* Early work on graph processing on GPUs employed parallelism across vertices [Harish et al. 2007]; however, this approach creates load-imbalance among threads and can lead to GPU underutilization since some vertices, in particular the high-degree ones, require more work than others. To address this problem, Hong et al. [Hong et al. 2011a] propose to parallelize processing not only across vertices, but also across the edges of a vertex. Hong et al. do this by statically allocating a block of threads for each vertex to process its edges in parallel. Although this approach improves performance, it does not completely address the problem: the fact that threads were being statically allocated results in some vertices being assigned more threads than they require (e.g., vertices with a degree less than the configured value), while others will be assigned less threads. This is especially an issue for scale-free graphs where the degree varies considerably across vertices. TOTEM addresses this problem by using a new feature introduced recently by CUDA: dynamic parallelism, which allows a GPU kernel to create work from within the GPU. TOTEM employs this feature to create dynamically launch kernels based on vertex degree for each group of vertices with similar degree, and hence improving GPU utilization.

  In most cases, this speedup does not translate to a performance gains for the hybrid system because the CPU is usually the bottleneck processor (as it will be discussed in Section 2.8.2), however this optimization allows the GPU to run

faster to idle, and hence reducing energy consumption (Section 2.12 presents an evaluation of energy consumption).

- *Hiding communication overhead by overlapping communication with computation*. For example, if the GPU finishes processing its partition faster than the CPU does, the GPU will start copying its output buffer to the CPU's input buffer while the CPU still processing its partition, and vice versa. Double buffering techniques enable such an optimization.

2.7.4 Design Trade-offs

There are two main trade-offs in the current TOTEM implementation that are worth discussing. First, the graph representation (CSR) used makes it expensive to support updates to the graph structure during algorithm execution (e.g., creation of new edges or vertices). This is a tradeoff, as CSR enables a lower memory footprint and efficient iteration over the graph's elements (vertices and edges), which are essential for performance. Any other graph data structure that enables mutable graphs will have to have some form of dynamic memory management (e.g., linked lists), which is costly to support, particularly on GPUs.

Nevertheless, a large and important class of applications is based on static graphs. For example, many graph-based applications in social networks [Gupta et al. 2013; Wang et al. 2013] and web analytics [Malewicz et al. 2010] are performed on periodic snapshots of the system's state, which is typically maintained in storage efficient, sometimes graph-aware, indexing systems [Curtiss et al. 2013; Barroso et al. 2003].

The second limitation is related to the way communication is performed. During the communication phase of each superstep, the current implementation copies the whole outbox buffer of a partition to the inbox buffer of a remote partition assuming that there is a message to be sent via every edge between the two partitions. This is efficient for algorithms that communicate via *each* edge in

every superstep, such as PageRank. However, this is an overhead for algorithms that communicate only via a selective set of edges in a superstep (e.g., in the level-synchronized BFS algorithm, at a given superstep, only the vertices in the *frontier* communicate data via their outgoing edges). Additional compression techniques can be employed to lower the communication volume.

2.8 Evaluating the Model's Accuracy and Processing Overheads

This section aims to address the following questions: First, *how does TOTEM performance compare to that predicted by the model?* Answering this question allows us to validate the model and understand, for each use case, how much room is possibly left for optimizations.

Second, *which phase (computation or communication) and processing element (CPU or GPU) the bulk of time is spent?* Such profiling identifies the bottlenecks in the system, and guides the quest for better performance.

Table 1: Testbed characteristics: two Xeon 2560 processors and two GeForce Kepler Titan GPUs, connected via PCI-E 3.0 bus.

Characteristic	Sandy-Bridge (Xeon 2650)	Kepler (Geforce Titan)
Number of Processors	2	2
Cores / Processor	8	14
Core frequency (MHz)	2000	800
Hardware Threads / Core	2	192
Hardware Threads / Processor	16	2688
Last Level Cache / Processor (MB)	20	2
Memory / Processor (GB)	128	6
Memory Bandwidth / Processor (GB/s)	52	288
Thermal Design Power / Processor (Watt)	95	250

Testbed characteristics. The machine used in the experiments is provisioned with recent CPU and GPU models as of writing this thesis (Table 1). The two processing

elements are representative for their categories and support different performance attributes. On the one hand, GPUs have significantly larger number of hardware threads, higher memory access bandwidth, and support a larger number of in-flight memory requests. On the other hand, the CPU cores are clocked at over double the frequency, and have access to roughly one order of magnitude larger memory and cache.

Table 2: Workloads used throughout the chapter. The synthetic RMAT graphs were generated using the Recursive MATrix (RMAT) graph generation model [Chakrabarti et al. 2004], which generates graphs with skewed degree distribution. The following parameters were used to generate the RMAT graphs: (A,B,C) = (0.57, 0.19, 0.19) and an average degree of 16. The synthetic UNIFORM graphs were generated using Erdős–Rényi graph generation model [Erdős and Rényi 1960], which generates graphs with uniform degree distribution. A graph is classified as "Small" if it fits the memory of a GPU, or as "Large" if it does not.

Scale	Workload	|V|	|E|	Memory (GB)
Large	Twitter [Cha et al. 2010]	52M	1.9B	7,689
	UK-Web [Boldi et al. 2008]	105M	3.7B	14,666
	RMAT27	128M	2.0B	8,704
	RMAT28	256M	4.0B	17,048
	RMAT29	512M	8.0B	36,864
	RMAT30	1,024M	16.0B	73,728
	UNIFORM28	256M	4.0B	17,048
Small	RMAT25	32M	512M	2,176
	UNIFORM25	32M	512M	2,176

Benchmarks. The evaluation tests five graph algorithms with different characteristics: Breadth-first Search (BFS), Betweeness Centrality (BC), PageRank, Single-Source Shortest Paths (SSSP) and Connected Components (CC). The details of the algorithms and their implementations are discussed in later sections. However, one difference between the algorithms is worth mentioning here: BFS uses a summary data structure, particularly a bitmap, to increase the utilization of the cache, while the other algorithms do not.

Workloads. The evaluation in this section is focused on an instance of Graph500 workload, RMAT28 graph[6] (Table 2). The memory footprint of this workload is large compared to the space available on a single GPU (~4 times larger), yet it allows us to explore offloading ratios as low as 50% when using a second GPU.

Time Measurements. For all experiments in this and the following sections, we measure the time to execute the algorithm only. The time to load and partition the graph is not included when calculating the processing rate of an algorithm. Separating the algorithm processing time from the time spent on pre-processing the graph is common [Nguyen et al. 2013] as the pre-processing time is considered an amortized cost. Note that the Graph500 challenge also adopts this approach, where only the algorithm's processing time is used for ranking.

Evaluation Metrics. While this section reports speedups when comparing with a host-only execution, later sections report TEPS as a performance metric. Similar to the Graph500 benchmark, the corresponding TEPS for BFS is calculated by dividing the sum of the degrees of the visited vertices by the time. The way TEPS is calculated for SSSP and BC is similar. For SSSP, the number of edges traversed is calculated by summing the degrees of the vertices that have a non-infinite distance; for BC, a non-zero score, with the difference being that for BC the number of traversed edges is multiplied by two as the algorithm has backward and forward propagation phases (see Section 2.10.2 for details regarding the BC algorithm). For connected components, TEPS is calculated by dividing the number of edges in the graph by the time. Finally, for PageRank, the corresponding TEPS is computed by

---

[6] The synthetic graphs are described by the log base 2 of the number of vertices (e.g., RMAT30 graph has 2^{30} vertices). Unlike in the Graph500 challenge (www.graph500.org), our graphs are directed (as generated by the model).

dividing the number of edges in the graph by the time per PageRank iteration (in each iteration, each vertex accesses the state of all its neighbors). The TEPS metric has the advantage that it can allow a (rough) comparison between runs of the same algorithm or implementation on different workloads.

Data collection and notations. For each data point, here and in later evaluation sections, the plots show the average over 64 runs. Error bars present the 95% confidence interval, in most cases, are too narrow to be visible.

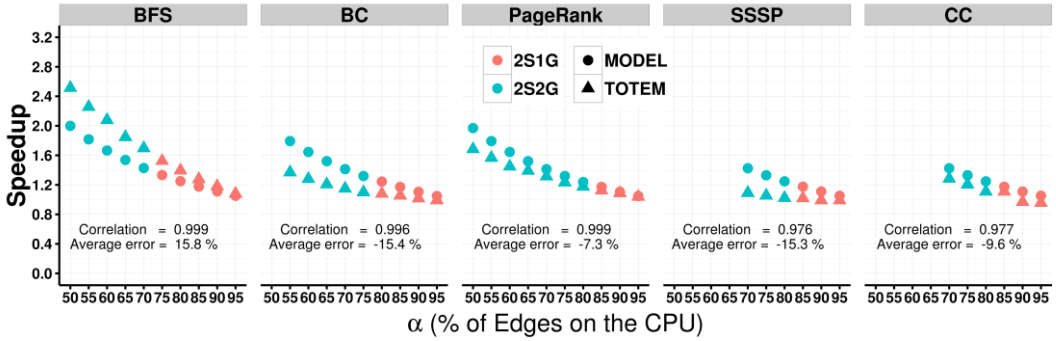

Figure 8: Predicted (circles) and achieved (triangles) speedup for RMAT28 graph while varying the percentage of edges assigned to the CPU partition (using random partitioning). The plot shows the results while using one (2S1G) and two (2S2G) GPUs. Having a second GPU allows offloading more edges. Note that the start point on the x-axis represents the minimum percentage of edges that needs to be kept on the host due to GPU space constraints. Also, note that due to different memory space requirements, the point at which a second GPU needs to be used is different for each algorithm. Pearson's correlation coefficient [Lee Rodgers and Nicewander 1988] **is reported on each plot - this is a value in the range [1,-1] where 1 is total positive correlation and 0 is no correlation.**

The different hardware configurations used in the experiments are presented in the following notation: ***xS yG***, where *x* is the number of CPU sockets (processors) used, while *y* represents the number of GPUs. For example, "*2S1G*" refers to processing the graph on two CPU sockets and one GPU.

2.8.1 Totem and the Performance Model

This section first compares the speedup predicted by the model and the one achieved by TOTEM. Figure 8 shows the speedup while varying α, the percentage of edges left on the CPU for the different graph algorithms. Note that the figure shows the speedup while using one (2S1G) and two (2S2G) GPUs. Table 3 presents a summary of the correlation coefficients and average errors for all other workloads.

Table 3: Average error and correlation between the predicted speedup by the model and the achieved one by Totem for all algorithms and large scale-free workloads. The results for an RMAT30 graph are missing for SSSP and CC because of memory space constraints (SSSP requires additional memory space to store the edge-weights while CC doubles the number of edges as it operates on undirected graphs).

Algorithm	Workload	Correlation	Avg. Err.
BFS	RMAT27	0.99	6%
	RMAT28	0.99	16%
	RMAT29	0.99	6%
	RMAT30	0.99	11%
	Twitter	0.99	-1%
	UK-WEB	0.99	-25%
PageRank	RMAT27	0.99	4%
	RMAT28	0.99	-7%
	RMAT29	0.97	4%
	RMAT30	0.99	8%
	Twitter	0.93	10%
	UK-WEB	0.98	-8%
BC	RMAT27	0.99	-13%
	RMAT28	0.99	-15%
	RMAT29	0.99	-10%
	RMAT30	0.99	-3%
	Twitter	0.99	-11%
	UK-WEB	0.99	-5%
SSSP	RMAT27	0.98	-20%
	RMAT28	0.97	-15%
	RMAT29	0.99	-8%
	Twitter	0.88	-22%
	UK-WEB	0.97	-4%
CC	RMAT27	0.98	-25%
	RMAT28	0.97	-10%
	RMAT29	0.99	-1%
	Twitter	0.98	-7%

Observe the following: First, the achieved speedup has strong positive correlation with the one predicted by the model for all algorithms and with low average error. Second, the model under-predicts BFS performance. This is because, for BFS, offloading to the GPU not only reduces the amount of work that the CPU needs to do, but also improves the CPU processing rate due to improved cache hit ratio: the bitmap used by BFS

becomes smaller and hence fits better the cache. This effect is not captured by the model.

The latter observation is important as it suggests that carefully choosing the part of the graph to be offloaded to the GPU may lead to superlinear speedups due to cache effects. This premise is evaluated in more detail in Section 2.9 where different partitioning strategies are explored that aim to further increase the chance of achieving superlinear speedups.

Finally, it is worth mentioning that similar accuracy holds for other workloads. Moreover, this accuracy also holds for a different, older generation, hardware platform. The results are published in [Gharaibeh et al. 2012], and are not presented here for brevity.

2.8.2 Overhead Analysis

To understand the phase (computation or communication) and processing element (CPU or GPU) on which the bulk of time is spent, this section examines the breakdown of the total execution time. Figure 9 shows the percentage of time spent on each phase for BFS while processing RMAT28 graph.

Two points are worth discussing. First, the GPU processes its partition at a faster rate, and, as a result processing the CPU partition always remains the main bottleneck. The GPU is 2 to 20 times faster. This indicates that the assumption that the GPU finishes its processing first holds in practice.

Second, the CPU-GPU communication overhead is significantly lower than the computation, even when using two GPUs. This is due to aggregating boundary edges and to the high bandwidth of the PCI Express bus.

Note that the two other algorithms, BC and PageRank, exhibited the exact same behavior, moreover these results were observed on all other workloads.

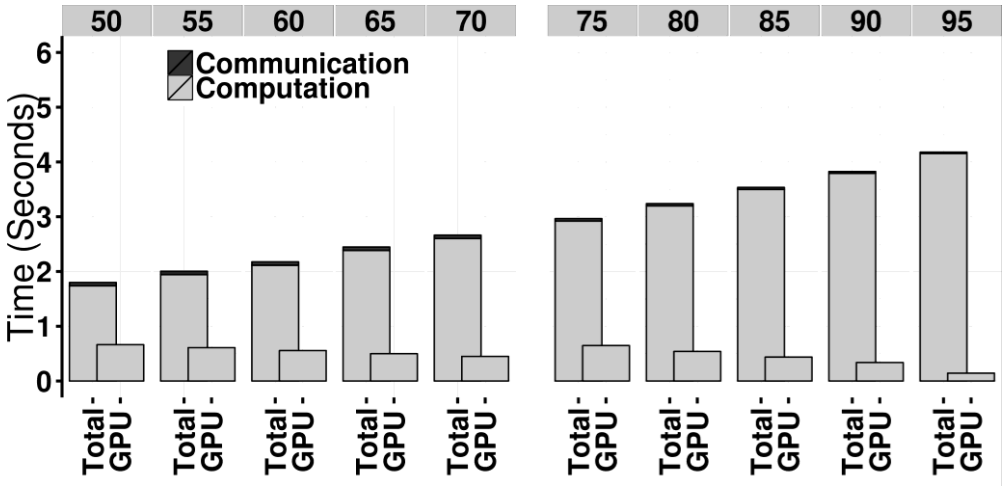

Figure 9: Breakdown of BFS execution time for the RMAT28 graph (the same data points in Figure 8). *Left*: using two GPUs (2S2G). *Right*: using one GPU (2S1G). The Total bar refers to the total execution time (i.e., the makespan). The Computation portion of the Total bar refers to the time of the bottleneck processor (the CPU in all cases). The GPU bar refers to the portion of Computation time where the GPU was busy.

The fact that communication is not a bottleneck has important consequences: rather than focusing on minimum cuts when partitioning the graph to reduce communication (a pre-processing step that, generally, is prohibitively expensive), an effective partitioning strategy should focus on reducing computation.

To this end, the next section explores the impact of various graph partitioning strategies and workload allocation schemes on the performance of graph algorithms on a hybrid system. Particularly, the focus is on investigating low-cost partitioning techniques that *generate workload that match well the strength of the processing element they are allocated to.*

2.9 Graph Partitioning for Hybrid Systems

This section presents the set of requirements for effective partitioning strategies for hybrid systems (Section 2.9.1), discusses (Section 2.9.2) and evaluates (Section 2.9.3) the proposed degree-based partitioning strategy.

2.9.1 Partitioning Strategy Requirements

An effective graph partitioning strategy must have the following characteristics:

- *Has a low space and time complexity*. Processing large-scale graphs is expensive in terms of both space and time; hence partitioning algorithms with time complexity higher than linear or quasilinear are impractical.
- *Handles scale-free graphs*. Many important graphs in different domains present skewed vertex degree distributions. Therefore, the partitioning strategy must be able to handle the severe workload imbalance associated with such graphs.
- *Handles large (billion-edge scale and larger) graphs*. The amount of memory offered by single-node systems is considerably large. For instance, 256GB on the evaluation machine used in this study is enough to fit a graph with one billion vertices and 16 billion edges (i.e., scale 30 in Graph500 terminology).
- *Minimizes algorithm's execution time by reducing computation (rather than communication)*. The BSP model divides processing into computation and communication phases. The focus is on partitioning strategies that reduce the computation time. Note that this approach is in sharp contrast to previous work on graph partitioning for distributed graph processing, as they focus on minimizing the time spent on communication (e.g., by minimizing the edge-cut between partitions) [Chamberlain 1998]. The evaluation in the previous section (Section 2.8) provides the intuition that supports this choice: message aggregation and batch communication (assisted by the high bandwidth of the PCI Express bus that typically connects discrete GPUs) can significantly reduce the communication overhead for concurrent graph processing (or similar applications, as the optimizations are application agnostic) on hybrid systems, which makes computation rather than communication the bottleneck.

2.9.2 Partitioning by Degree Centrality

I propose to partition the graph by degree centrality, placing the high-degree vertices in one type of processor and the low-degree ones in the other type. Our hypothesis is that this simple and low-cost partitioning strategy brings tangible performance benefits while meeting the solution requirements.

The motivation behind this intuition is twofold. First, dividing a scale-free graph using the vertex degree as the partition criterion produces partitions with significantly different levels of parallelism that match those of the different processing elements of the hybrid system. Second, such a partitioning strategy produces partitions that are more homogenous in terms of vertex connectivity compared to the original graph, resulting in a more balanced workload within a partition. This is important to maximize the utilization of a processor's cores, especially for the GPU because of its strict parallel computation model.

Partitioning the graph based on vertex degree is low cost in terms of computational and space complexity. One way to classify the low and high degree vertices is by sorting, with time complexity O(|V|log|V|). In practice, one can improve the running time even further by using partial sorting (i.e., finding the degree values that divide the graph into the desired partitions), which takes linear O(|V|) time complexity [Chambers 1971]. Regarding space complexity, these manipulations require O(|V|) of additional space, which represent the permuted vertex ids after sorting (or partial sorting). Once the vertices are placed in the required order, the edges of each vertex can be read from disk according to the new order. This is a moderate space cost as the size of scale-free graphs is typically dominated by the number of edges.

2.9.3 Evaluation

This section builds on the previous evaluation (Section 2.8), which was based on random partitioning. In particular using an instance of the Graph500 benchmark,

this section presents experiments that highlights the effect of partitioning the graph based on vertex connectivity (Section 2.9.3.1), and explains the reasons behind the observed performance by each partitioning strategy using performance counter statistics and psedu-code analysis (Section 2.9.3.2)

2.9.3.1 Highlighting the Effect of Partitioning

BFS is used to evaluate the partitioning strategies. Three partitioning strategies are compared: RAND, HIGH, and LOW. RAND divides the graph randomly. The other two strategies are based on degree centrality: HIGH divides the graph such that the highest degree vertices are assigned to the CPU, and LOW divides the graph such that the lowest degree vertices are assigned to the CPU.

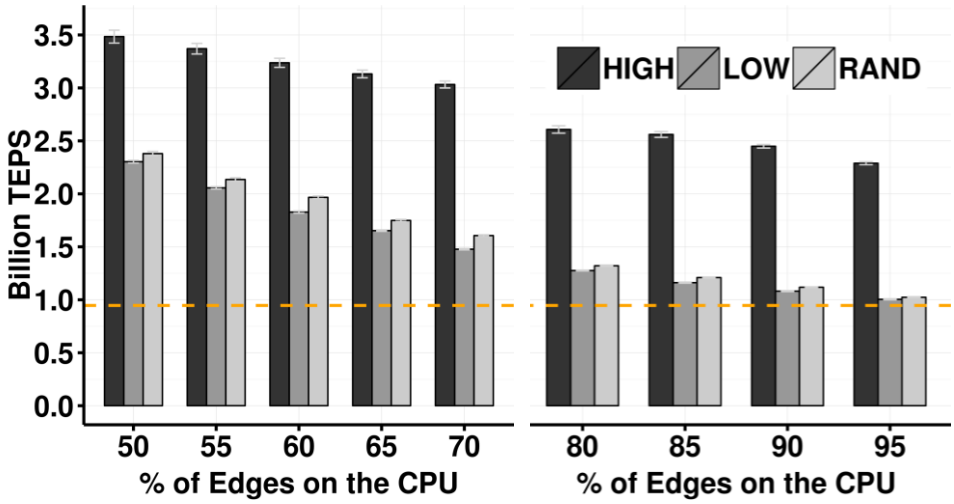

Figure 10: BFS traversal rate (in billions of traversed edges per second - TEPS) for the RMAT28 graph and different partitioning algorithms while varying the percentage of edges placed on the CPU. *Left*: two GPUs (2S2G); *Right*: one GPU (2S1G). The performance of processing the whole graph on the host only (2S) is shown as a dashed line.

Figure 10 shows BFS traversal rate in billions Traversed Edges Per Second (TEPS) for the RMAT28 workload (|V|=256M, |E|=4B, see Table 2). Note that the

graph is too large to fit entirely on one or two GPUs and, thus, the host must keep at least 80% and 50% of the graph's edges, respectively.

In this figure, the x-axis represents the share of the edge array assigned to the CPU partition (after the vertices in the vertex-array have been ordered by degree). For example, consider the 80% data point and HIGH partitioning. The high-degree vertices are assigned to the host until 80% of the edges of the graph and their corresponding vertices are placed on the host. The remaining vertices and their edges are placed on the GPU. Similarly, in the case of LOW partitioning, the low-degree vertices are assigned to the host until it holds 80% of the graph's edges.

The figure reveals a significant performance difference generated by the various partitioning schemes. In particular, assigning the high-degree nodes to the CPU results in superlinear speedup with respect to the share of the graph offloaded for processing on the GPU. For example, offloading 50% of the graph to be processed on the GPUs offers 2.8x speedup. A question that arises from this analysis is: *What are the causes for this observed performance difference?*

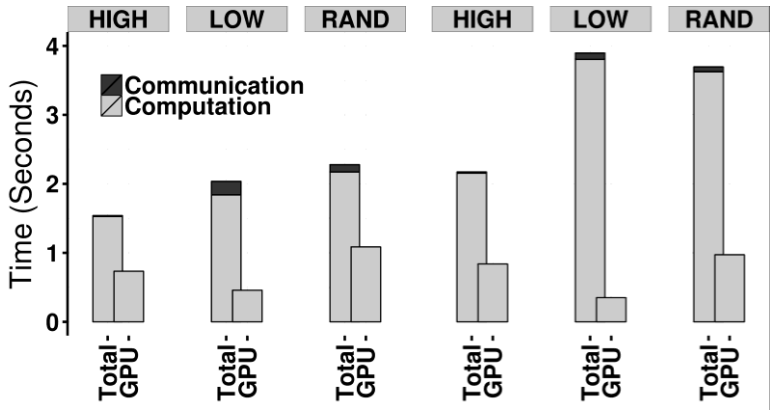

Figure 11: Breakdown of execution time for an RMAT28 graph. *Left*: using two GPUs (2S2G) and 50% of the edges are assigned to the CPU. *Right*: using one GPU (2S1G) and 80% of the edges are assigned to the CPU. The "Computation" bar refers to the computation time of the bottleneck processor (the CPU in this case).

2.9.3.2 Explaining the Performance Difference

Figure 11 presents the breakdown of execution time for two of the data points presented in Figure 10: the 50% and 80% data points, which represent the maximum partition size that can be offloaded to two and one GPU(s), respectively. The breakdown shows that the hybrid system's performance is *bottlenecked by the CPU regardless of the partitioning scheme*, even when offloading 50% of the edges to be processed on the GPUs. This happens because of two reasons: (i) the GPU has a higher processing rate; and (ii) the communication overhead is negligible compared to the computation phase. Based on these two observations, the rest of this section focuses on the effect of graph partitioning strategies on CPU performance.

```
1     BFS(Partition partition, int level){
2      bool done = true;
3      parallel for vertex in partition.vertices{
4       if (vertex.level != level) continue;
5        for (neighbour in vertex.neighbours){
6         if (!partition.visited.isSet(n)){
7          if (partition.visited.atomicSet(n)){
8           neighbour.level = level + 1;
9           done = false;
10     }}}}
11     return done;
12    }
```

Figure 12: Pseudocode of the level-synchronous BFS compute kernel. The kernel is invoked in each round for each partition. The algorithm terminates when all partitions in the same round return true.

Figure 12 lists the pseudocode for the BFS kernel. Hong et al. [Hong et al. 2011b] showed that this implementation has superior performance over typical queue-based approaches. In order to reduce main memory traffic, the algorithm uses a bit-vector (lines 6 and 7 in Figure 12) to mark the vertices that have already been visited, thus avoiding fetching their state from main memory.

Chhugani et al. [Chhugani et al. 2012] showed that a cache-resident "visited" bit-vector is critical for BFS performance on the CPU, and that the performance

significantly drops for large graphs as the bit-vector becomes larger. For the RMAT28 workload, the size of the "visited" bit-vector is 32MB (i.e., a bit array that represents the 256M vertices) and it is only a little smaller than the total amount of last level cache (LLC) on the two CPU sockets, which is 40MB.

To evaluate the cache behavior, Figure 13 shows the LLC cache miss rate (left) and the percentage of main memory accesses (right) for the different partitioning schemes. Depending on the partitioning strategy, the "visited" vector is differently distributed between the host and the accelerator. Thus, to better understand the profiling data in Figure 13, Figure 14 shows the percentage of vertices assigned to the CPU due to graph partitioning. The two figures highlight the strong correlation between $|V_{cpu}|$ and the cache miss rate.

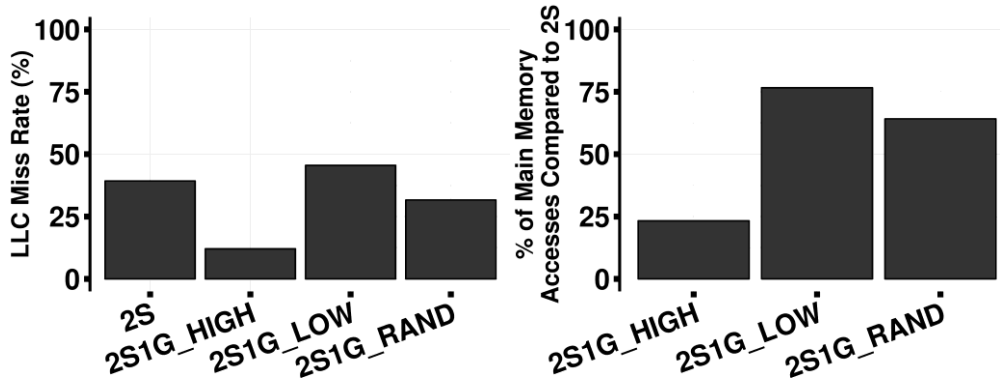

Figure 13: Performance counter statistics gathered when running BFS on an RMAT28 graph for a CPU-only configuration (2S), and a hybrid configuration using one GPU (2S1G) when 80% of the edges are assigned to the CPU. *Left*: LLC miss rate (the lower the better), computed as 100×(LLC_MISS /LLC_REFS). *Right*: the percentage of main memory accesses on the host compared to processing the whole graph on the host (the lower the better), computed as 100×(LLC_MISS_{2S1G}/ LLC_MISS_{2S}).

On the one hand, RAND and LOW partitioning strategies produce a CPU partition with a large number of vertices leading to a large "visited" vector comparable in size to that of the original graph. Therefore the LLC miss rate

changes only slightly when compared to processing on the CPU only: improved for RAND due to lower $|V_{cpu}|$, and worsened for LOW due to the added overhead of handling boundary edges (i.e., edges with source and destination vertices reside on partitions that are assigned to different processors). However, Figure 13 (right) shows that both these strategies still reduce the number of main memory accesses – as a consequence of offloading part of the graph to the GPU, resulting in an overall performance improvement by the hybrid system.

On the other hand, due to the power-law degree distribution of the graph, the CPU partition produced by the HIGH strategy has two orders of magnitude fewer vertices for the same number of edges. This results in a more cache friendly CPU workload, and leads to significant improvement in the CPU processing rate (the main bottleneck in the system).

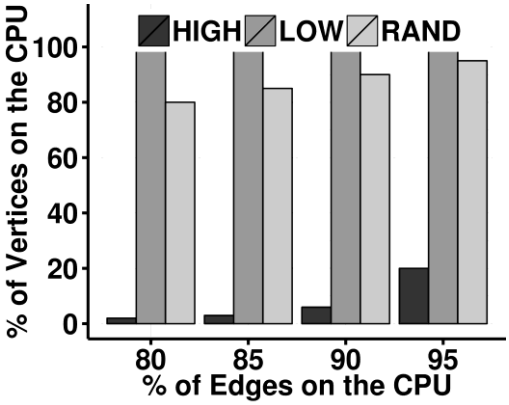

Figure 14: Percentage of vertices placed on the CPU for RMAT28 graph while varying the percentage of edges assigned to the partition, and for various partitioning strategies.

With the HIGH partitioning strategy, offloading as little as 5% of the edges to the GPU offers 2x speedup compared to processing the graph on the CPU only, and up to 2.5x speedup when offloading 25% of the edges. This demonstrates that although GPUs have limited memory, they can significantly improve performance. This is because GPUs are able to efficiently handle the sparser part of the graph as

they rely on massive multi-threading rather than caches to hide memory access latency.

2.9.3.3 Same Exploration for a Smaller Graph

This section discusses the effects of partitioning on a relatively small scale graph. Figure 15 shows BFS traversal rate for an RMAT25 graph (|V|=32M, |E|=512M, see Table 2), which is almost an order of magnitude smaller than the one used in the previous experiments. A smaller graph creates two implications: it enables offloading a larger partition to the GPU (in this case, the graph fits entirely in the GPU memory); and, for BFS, a graph with a small number of vertices improves the cache hit rate of the algorithm (in this case, the bit-vector size is 4MB, and fits the LLC cache better than the RMAT28 graph).

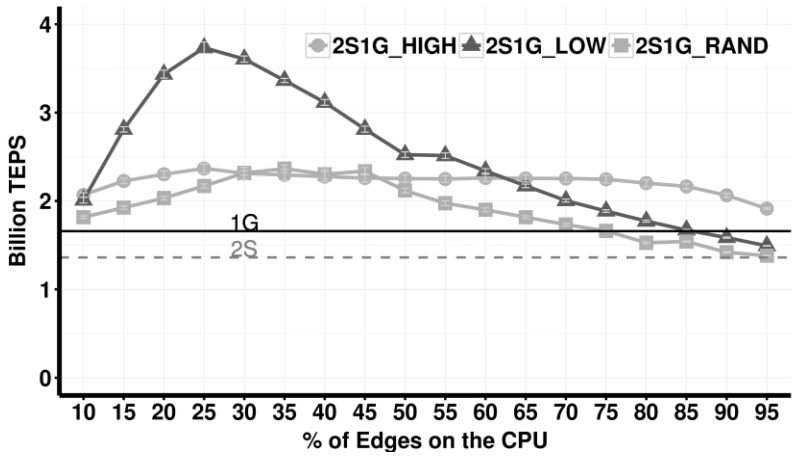

Figure 15: BFS traversal rate for the RMAT25 graph and different partitioning algorithms on a 2S1G hybrid configuration. Note that the graph is small enough to fit in the memory of a single GPU, hence the performance of processing the whole graph on the GPU only is shown as a straight line labelled 1G. The performance of processing the whole graph on the host only is also shown as a dashed line labelled 2S.

In the right end of the figure, where the CPU partition is larger than the GPU partition, CPU processing is the bottleneck and the performance of the three strategies exhibits behavior similar to the one observed for the larger scale graph

where HIGH partitioning offers the best performance. However, the relative improvement is not as prominent as on the larger graph because, for this workload, the "visited" bit-vector is already small enough to fit the cache compared to the larger, RMAT28 graph.

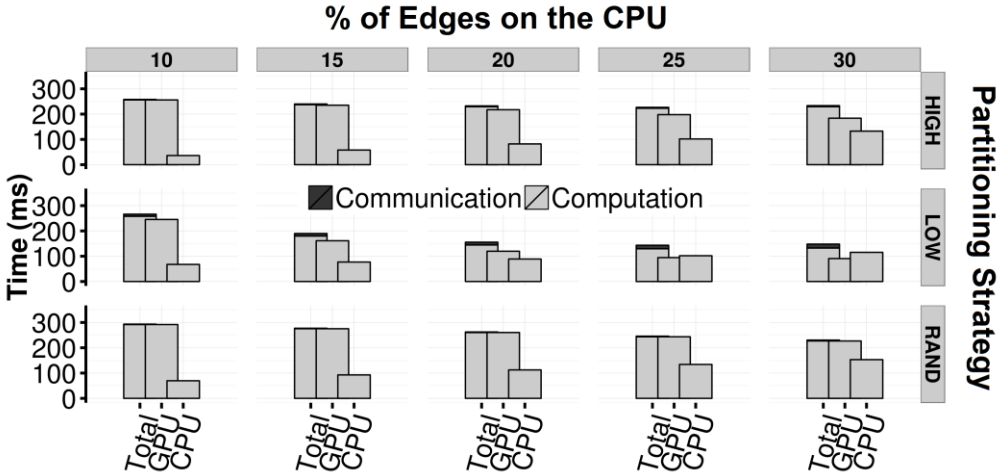

Figure 16: Breakdown of execution time for an RMAT25 graph on a 2S1G hybrid configuration while varying the partitioning strategy and the percentage of edges assigned to the CPU. The "Computation" bar refers to the computation time of the bottleneck processor. The "GPU" and "CPU" partitions execution times are shown alongside the "Total" execution time. This allows demonstrating which processor is the bottleneck for different configurations: the bottleneck processor is the one that is closer to the computation time in the "Total" bar.

To understand the behavior of the hybrid system when most of the graph is processed on the GPU (the left end of Figure 15), Figure 16 shows the breakdown of execution time for the first five data points. When partitioning the graph using RAND and HIGH strategies while keeping only a small percentage of the edges on the CPU, the result is a larger GPU partition with similar characteristics to the original graph. As Figure 16 shows, GPU processing is the bottleneck for both of these strategies and the gain brought by the hybrid system is proportional to the part of the graph processed concurrently on the CPU. Note that, for both strategies, the

performance improves up to a point where the load is more balanced between the two processing units. After that, it drops as the CPU partition becomes the bottleneck as discussed previously.

In the case of LOW partitioning strategy, the resulting large GPU partition is vastly denser than those of the other two strategies. A denser graph leads to better locality, which the GPU is able to leverage efficiently because the associated "visited" bit-vector is small and fits the limited available cache (this has been confirmed by collecting performance counter statistics for the GPU's L2 cache hit rate, which are not shown here for brevity). Hence, the GPU processing rate is much faster when using LOW compared to the other two strategies for the same percentage of offloaded edges. In fact, the GPU performance is significantly more efficient to the degree that the bottleneck shifts to the CPU when increasing the CPU's share of edges to only 25%.

2.9.3.4 The Effect of Vertex Degree Distribution

As discussed previously, many real-world graphs are scale-free. The fact that these graphs have skewed vertex degree distribution (*i*) guided the choice of the partitioning strategies (Section 2.9.2) in this work, and (*ii*) facilitated the aggregation optimization (Section 2.6), which aims to reduce the communication overhead.

To quantify the effect of vertex degree distribution on the above mentioned aspects, the performance of the hybrid system is evaluated using a case that is the worst input for TOTEM's optimizations: random graphs with uniform degree distribution. The graphs were generated using the Erdős–Rényi graph generation model [Erdős and Rényi 1960]. The model generates edges with equal probability of setting an edge between any two vertices, independently of the other edges.

Figure 17 shows BFS traversal rate for a small scale, UNIFROM25 graph. The figure highlights that, when the graph has a uniform degree distribution the hybrid

system performs almost the same irrespective of the partitioning strategy as all strategies produce partitions with similar characteristics. Compared to processing the whole graph on the GPU, the hybrid system performance offers slight improvement that is proportional to the size of the partition kept on the CPU (the left side of the figure). This improvement diminishes when the CPU partition becomes large enough to make the CPU the bottleneck processor (the right side of the figure).

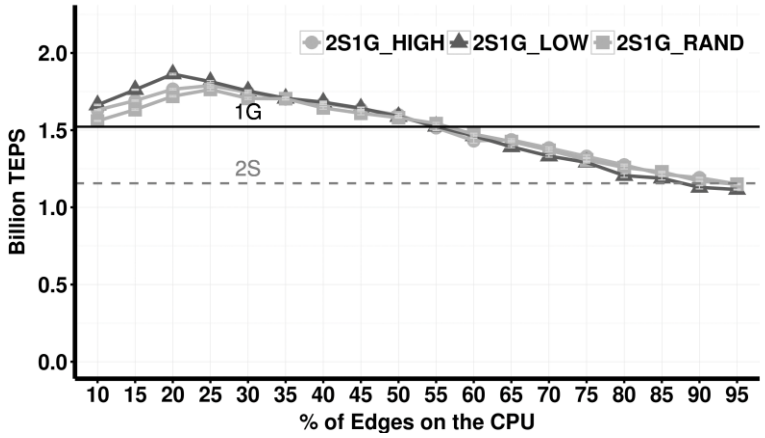

Figure 17: BFS traversal rate for the UNIFORM25 graph and different partitioning algorithms on a 2S1G hybrid configuration. Note that the graph is small enough to fit in the memory of a single GPU, hence the performance of processing the whole graph on the GPU only is shown as a straight line labelled 1G. The performance of processing the whole graph on the host only is also shown as a dashed line labelled 2S.

Figure 18 (left) shows the system's performance for the larger UNIFORM28 graph on a hybrid 2S1G configuration. Unlike the RMAT workload and similar to the performance of the smaller graph above, all partitioning strategies perform similarly. Moreover, there is marginal gain from processing part of the graph on the GPU that is proportional to the offloaded partition.

Finally, Figure 18 (right) shows the breakdown of execution time for the case where the largest partition possible is placed on the GPU (i.e., 80% of the edges

were kept on the CPU). The figure shows that the CPU is the bottleneck processor, and that communication is not a major overhead due to the ability of TOTEM to overlap a major part it (the messages sent from the GPU to the larger CPU partition) with the computation of the CPU partition.

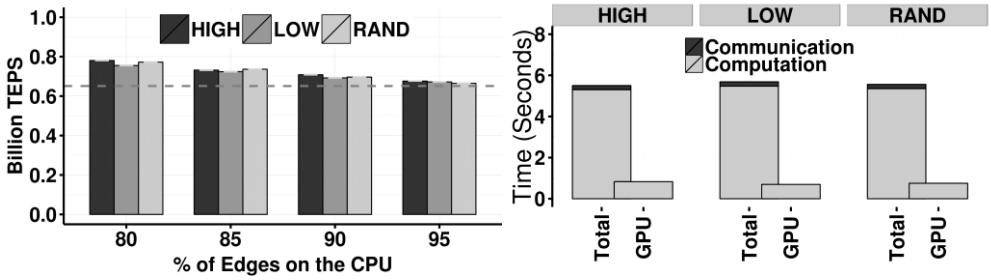

Figure 18: BFS performance on a UNIFORM28 graph on a hybrid 2S1G configuration. Left: traversal rate. Right: breakdown of execution time for the 80% data point. The performance of running the whole graph on the CPU (2S) is shown as a dashed line.

2.10 Extending the Application Set

This section focuses on the following two questions: *Do the performance gains offered by the hybrid system on BFS extend to more complex applications? How do the partitioning strategies influence performance in such settings?*

To answer these questions, this section presents two additional applications implemented using TOTEM: ranking web pages using PageRank (Section 2.10.1) and finding the main actors in a social network using Betwenness Centrality (Section 2.10.2).

2.10.1 Ranking Web Pages

PageRank [Page et al. 1999] is a fundamental algorithm used by search engines to rank web pages. This section presents an evaluation of PageRank on the UK-WEB workload [Boldi et al. 2008], a crawl of over 100 million pages from the .uk domain, and 3.7 billion directed links among the pages.

```
1  PageRank(Partition partition) {
2   double delta =(1 - damping_factor) / vertex_count;
3   parallel for vertex in partition.vertices {
4     double sum = 0;
5     for (neighbour in partition.incoming_neighbours) {
6       sum = sum + neighbour.rank;
7     }
8     vertex.rank = delta + damping_factor * sum;
9   }
10 }
```

Figure 19: Pseudocode of PageRank's compute kernel. *vertex_count* is the total number of vertices in the graph, while *damping_factor* is the damping factor, a constant defined by the PageRank algorithm. The kernel is invoked in each BSP round for each partition. The algorithm terminates after executing the kernel a predefined number of times.

Figure 19 presents the compute kernel of the PageRank algorithm. Note that the kernel is pull-based: each vertex *pulls* the ranks of its neighbors via the incoming edges to compute a new rank. This is faster than a push-based approach, where each vertex *pushes* its rank to its neighbors via the outgoing edges. The latter approach requires atomic operations, and hence is less efficient [Nguyen et al. 2013].

Compared to BFS, PageRank has a higher compute-to-memory access ratio, and does not employ summary data structures, therefore the cache has a lower effect on performance.

Figure 20 shows PageRank's processing rate. While a single GPU offers narrow improvement due to limitations on the size of the offloaded partition, adding a second GPU significantly improves the performance for such a large workload: up to 2.3x speedup compared to processing the whole graph on the CPU only.

Compared to the other two strategies, LOW partitioning allows offloading a larger portion of the edges to the GPU. This happens because PageRank requires a larger per-vertex state than BFS; hence, the number of vertices assigned to a partition has a larger effect on a partition's memory footprint. Since LOW places the high degree vertices on the GPU, the number of vertices assigned to the GPU

partition by LOW is significantly lower than that assigned by HIGH and RAND strategies for the same number of edges.

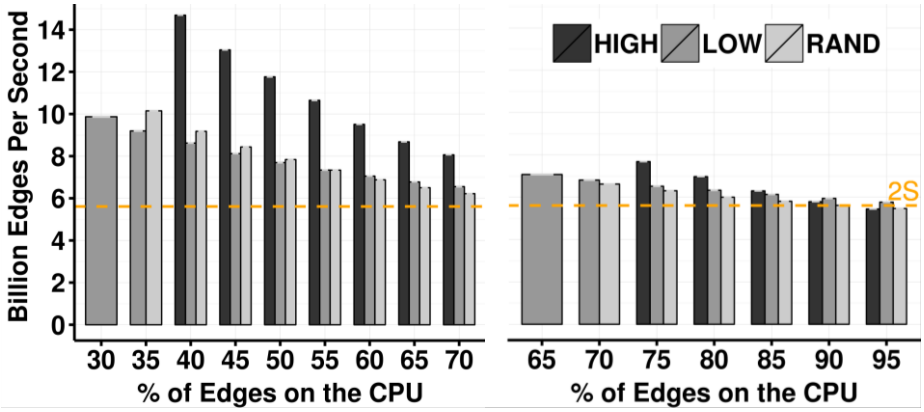

Figure 20: PageRank traversal rate for the UK-WEB graph. *Left*: using two GPUs. *Right*: using one GPU. Missing bars represent cases where the GPU memory space is not enough to fit the GPU partition. The performance of processing the whole graph on two CPU sockets (labelled as 2S) is shown as a straight line.

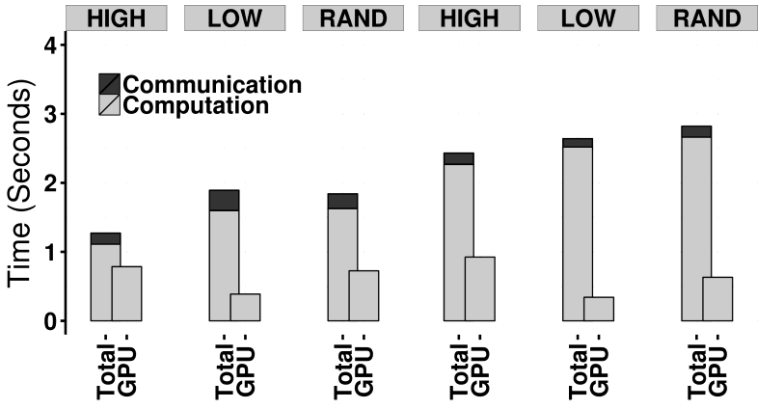

Figure 21: Breakdown of PageRank execution time (five iterations) for the UK-WEB graph when offloading the maximum size partition to two (left three bars) and one GPU (right three bars). The "Computation" bar refers to the compute time of the bottleneck processor (the CPU in this case).

Note that HIGH performs the best among all partitioning strategies. To explain this result, Figure 21 shows the breakdown of execution time. Similar to BFS, the

communication overhead is negligible; the CPU is the bottleneck processor in all partitioning strategies; and that HIGH is the most efficient partitioning strategy due to faster CPU processing.

Two interrelated factors lead to this result. First, from the pseudocode in Figure 19, notice that the number of memory read operations is proportional to the number of edges in the graph (line 6), while the number of write operations is proportional to the number of vertices (line 8). Second, as discussed in the previous section, for the same number of edges, the different partitioning strategies produce partitions with drastically different number of vertices (see Figure 14). Particularly, HIGH produces a CPU partition with significantly fewer vertices.

As a result of these observations, HIGH is expected to result in a CPU partition that performs significantly fewer write operations compared to the other two strategies, while the number of read operations will be similar for all partitioning strategies.

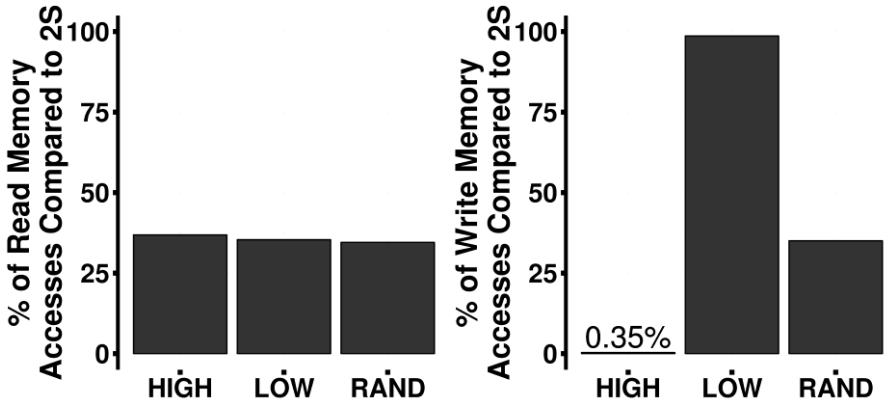

Figure 22: Host memory accesses statistics gathered when running PageRank on UK-WEB graph while when offloading the maximum size partition to two GPUs (2S2G). The performance counter used to collect these statistics is "*mem_uops_retired*". *Left*: read accesses; r*ight:* write accesses compared to processing the graph on the host only.

Figure 22 confirms this analysis: it shows the percentage of write and read memory accesses on the CPU (compared to processing the whole graph on the host)

when offloading the largest possible partition to two GPUs (i.e., the percentage of edges on the CPU is 30%, 35% and 40% for LOW, RAND and HIGH, respectively). The figure demonstrates that the percentage of read accesses (Figure 22 left) is similar for all partitioning strategies, with HIGH performing slightly more reads than the other two as it allows offloading fewer edges, while the percentage of write accesses (Figure 22 right) significantly differ.

One may expect that the overhead of reads will be dominant as the number of edges is much larger than the number of vertices. However, two reasons lead to the visible impact of writes. First, the performance analysis tool LMbench [McVoy and Staelin 1996] shows that the host memory write throughput is lower, almost half, than its read throughput. Second, the reduction in the number of write accesses is significant: HIGH generates two orders of magnitude fewer write operations compared to LOW and RAND. Note that this reduction is compensated by a major increase in write memory operations in the GPU partitions, which is reflected in the increase of the GPU compute time for HIGH and RAND compared to LOW in Figure 21. Still, the GPU's high memory bandwidth allows processing this part of the workload faster than the CPU and, hence, it leads to an overall gain in performance.

Finally, similar behavior is obtained for other graphs. Additionally, one of the publications this thesis is based on [Gharaibeh et al. 2013a] shows that the analysis in this section also hold on a different, older generation, hardware platform.

2.10.2 Finding the Main Actors in a Social Network

A key measure of importance for vertices in social networks is Betweenness Centrality (BC). This section presents an evaluation of BC on a snapshot of the Twitter follower network [Cha et al. 2010]. The workload includes over 52 million users and 1.9 billion directed follower links.

```
1  forwardPropagation(Partition partition, int level){
2   bool finished = true;
3   parallel for vertex in partition.vertices {
4    if (partition.distance[vertex] == level){
5     int numShortestPaths = partition.numShortestPaths[vertex];
6     for (neighbour in vertex.neighbors){
7      if (partition.distance[neighbour] == INF){
8       partition.distance[neighbour] = level + 1;
9       finished = false;
10     } // if
11     if (partition.distance[neighbour] == level + 1){
12      atomicAdd(partition.numShortestPaths[neighbour],
                  numShortestPaths);
13     } // if
14    } // for
15   }
16  }
17  return finished;
18 }

19 backwardPropagation(Partition partition, int level){
20  parallel for vertex in  partition.vertices {
21   if (partition.distance[vertex] == level) {
22    double delta = 0;
23    int numShortestPaths = partition.numShortestPaths[vertex];
24    for (neighbour in vertex.neighbors) {
25     if (partition.distance[neighbour] == (level + 1)) {
26      delta +=
         (numShortestPaths / partition.numShortestPaths [neighbour]) *
         partition.delta[neighbour];
27     } // if
28    } // for
29    partition.delta[vertex] = delta;
30    partition.betweenness[vertex] += delta;
31   } // if
32  } // for
33  return ((level - 1) == 0);
34 }
```

Figure 23: Pseudocode of BC's compute kernels. The algorithm is executed in two BSP cycles. A first BSP cycle is run using the forward propagation kernel. Once the first cycle terminates, a second cycle is run using the backward propagation kernel.

This section presents an evaluation of Brande's BC algorithm [Brandes 2001], which is based on forward and backward BFS traversals. Figure 23 lists the pseudocode of the forward and backward propagation kernels. Overall, the algorithm has different characteristics and is more complex than PageRank and the

basic BFS algorithm presented previously. Compared to basic BFS, BC traversal does not benefit from summary data structures targeted for improving cache efficiency. Compared to PageRank, BC is a traversal-based algorithm, where the set of "active" vertices changes across iterations, and it uses atomic operations.

Figure 24 (left) shows BC processing rate while offloading part of the graph to be processed on one GPU (i.e., 2S1G configuration). The figure demonstrates that for a specific percentage of edges offloaded to the GPU, HIGH offers the best performance. Moreover, similar to PageRank, LOW partitioning allows offloading a larger percentage of the edges to the GPU than HIGH and RAND. In fact, since BC requires relatively large per-vertex state, LOW allows offloading 20% more edges to the GPU compared to HIGH. Unlike PageRank, however, offloading more edges to the GPU via LOW partitioning has a significant impact on improving the overall performance of the hybrid system.

To understand this behavior, Figure 24 (right) shows the breakdown of overheads when offloading the maximum size partition to one GPU (i.e., the percentage of edges offloaded is 50%, 30% and 40% for HIGH, LOW and RAND, respectively). Notice that communication has minimal impact on performance, and that the CPU is again the bottleneck processor. Therefore, in the following, the major operations in the compute kernel are quantified by examining the pseudocode in Figure 23.

The major operations in the algorithm are: 5×|E| scattered reads (lines 7, 11, 12 and 26), 1×|E| atomic additions with scattered writes (line 12), 3×|E| floating point operations, 2×|V| writes (lines 29 and 30) and 1×|V| additions (line 30).

This analysis reveals that, similar to PageRank, BC performs expensive operations proportional to both the number of edges and vertices. Therefore, for a specific percentage of edges offloaded to the GPU, HIGH performs better than LOW and RAND as it results in significantly fewer vertices assigned to the bottleneck processor, the CPU. However, unlike PageRank, BC performs larger and

more expensive operations per edge than per vertex. Therefore, the ability of LOW partitioning scheme to offload more edges to the GPU results in notably better performance than HIGH and RAND partitioning schemes.

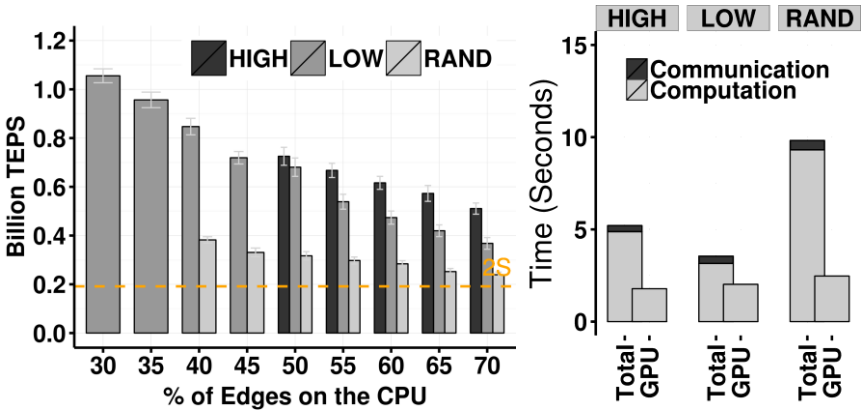

Figure 24: BC performance on the Twitter network for the 2S1G system. *Left*: traversal rate (in Billion TEPS) using one GPU. The horizontal line indicates the performance of a two socket system (2S). *Right*: Breakdown of execution time when offloading the maximum size partition to one GPU (i.e., the percentage of edges offloaded is 50%, 30% and 40% for HIGH, LOW and RAND, respectively).

Next, the performance of the hybrid system is compared with the CPU only (2S) performance (the dotted line in Figure 24 (left)). First, note that this work's implementation of BC applies several CPU-specific optimizations, and that its performance is proportional to the best reported runtimes. In particular, Nguygen et al. [Nguyen et al. 2013] report a runtime of 12 seconds (i.e., 0.32 Billion TEPS) when processing the same Twitter workload on a quad socket platform. This is only 40% faster than the performance reported here on a dual-socket testbed with lower-end processors (Section 2.13 presents a more detailed comparison).

Finally, the hybrid system (2S1G) delivers significant improvement compared to both symmetric platforms discussed above: adding a GPU boosts the performance by 5x compared to the dual socket (2S) configuration. Moreover, the

hybrid 2S1G platform (with lower-end CPU models) offers over 3x speedup compared to the quad-socket system, yet at a much lower energy and cost budget.

2.10.3 Finding Point-to-Point Shortest Paths in a Network

The Single-Source Shortest Path (SSSP) aims to find the shortest path from a given source to all vertices in a network. SSSP algorithms are used in a wide spectrum of application domains such as network routing, VLSI design, transportation network modeling and social network analysis. In this section, we present an evaluation of SSSP on the Twitter workload (Table 2).

```
1  SSSP(Partition partition) {
2   finished = true;
3   parallel for v in partition.vertices {
4    if (partition.active[v] == false) { continue; }
5    partition.active[v] = false;
6    for (neighbour in v.neighbours) {
7     new = partition.distance[v] + v.weights[neighbour];
8     old = partition.distance[neighbour];
9     if (new < old) {
10     if (old == atomicMin(partition.distance[neighbour], new)) {
11      partition.active[neighbour] = true;
12      finished = false;
13     }
14    }
15   } //for
16  } //for
17  return finished;
18 }
```

Figure 25: Pseudocode of SSSP's compute kernel based on Bellman-Ford algorithm. The array *distance* contains the computed distances of all the vertices in the partition. Each entry in the array *active* indicates the current state of a vertex. Every time a vertex's distance is updated, it becomes "active" and it may traverse its edge list in the same or the next BSP round. The algorithm terminates when there are no active vertices left. Note that atomicMin atomically updates a memory location with the new value if it is less than the current one, and returns the value stored in the location before the atomic operation gets applied.

Shortest path computation involves weighted graphs, where each edge is associated with a weight. For example, in the Twitter follower network, where vertices represent users, an edge weight can be a measure of common followers between two users or their geographic proximity. Weighted graphs increase

memory footprint, which poses a challenge with respect to offloading a larger fraction of the graph to the GPU. The additional memory required is proportional with the number of edges.

The Bellman-Ford algorithm [Ford 1956; Bellman 1958] is a common parallel SSSP algorithm, it is a traversal-based algorithm, but unlike BFS, the set of "active" vertices changes during an iteration and it also uses atomic operations for consistency. Figure 25 lists the algorithm. One improvement we have made to the algorithm is reducing the number of iterations (BSP rounds) by allowing a vertex to be set to "active" and perform "relax" operations in the same iteration if it has not been processed yet.

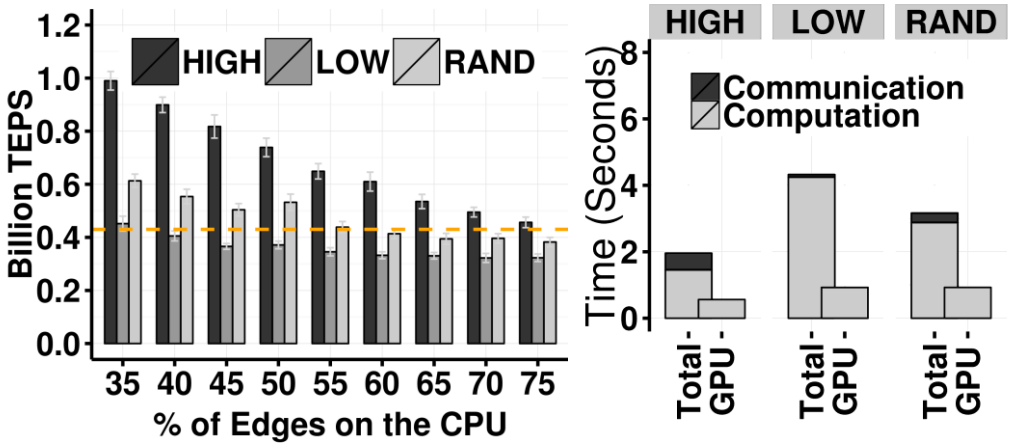

Figure 26: SSSP performance on the Twitter network for 2S2G system. Left: traversal rate (in Billion TEPS) using two GPUs. The horizontal line indicates the performance of a two socket system. Right: breakdown of execution time of the 35% data point.

Figure 29 (left) shows the performance of the SSSP algorithm. As shown, HIGH partitioning offers superior performance compared to the other two portioning strategies. Figure 29 (right) shows the breakdown of execution time. Similar to BFS, PageRank and BC, communication overhead is negligible compared to that of computation. CPU is always the bottleneck processing element and the best CPU performance is achieved when HIGH partitioning is used.

The most critical operation in the SSSP algorithm is when a vertex atomically updates the distance of a neighbor (lines 10 to 12 in Figure 25). Having a significantly lower number of vertices in the CPU partition as a result of using a HIGH partitioning strategy contributes to reducing the contention on the atomic updates, and hence improving the overall performance of the CPU partition.

To better illustrate this analysis, Figure 27 shows host memory access statistics of the three partitioning strategies. The figure demonstrates that while all strategies lead to reduction in read memory accesses, the HIGH partitioning strategy results in a significant reduction in the number of write operations (which, as we discussed before, more expensive than read operations).

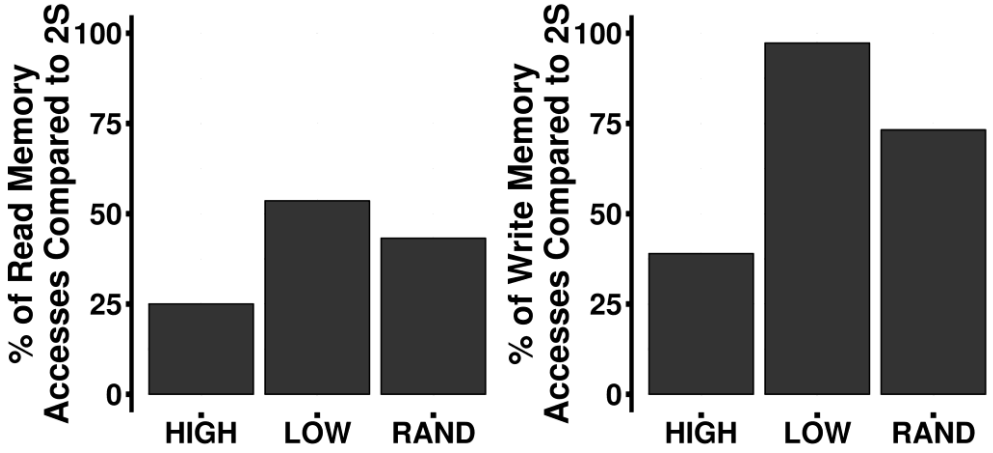

Figure 27: Host memory access statistics when running SSSP on the Twitter workload (2S2G configuration). The y-axis presents the percentage of host memory accesses of the CPU partition in a hybrid configuration compared to the number of accesses performed when running the whole graph on CPU only (i.e., 100*MEM_READ$_{2S2G}$/MEM_READ$_{2S}$ for the left figure and 100*MEM_WRITE$_{2S2G}$/MEM_WRITE$_{2S}$ for the right figure). The x-axis presents the three partitioning algorithms while offloading the maximum size partition to two GPUs.

2.11 Evaluating Scalability Using Synthetic Graphs

This section focuses on the following questions: *How does the hybrid system scale when increasing the graph size and with various hardware configurations? What is more beneficial, adding more CPUs or GPUs?*

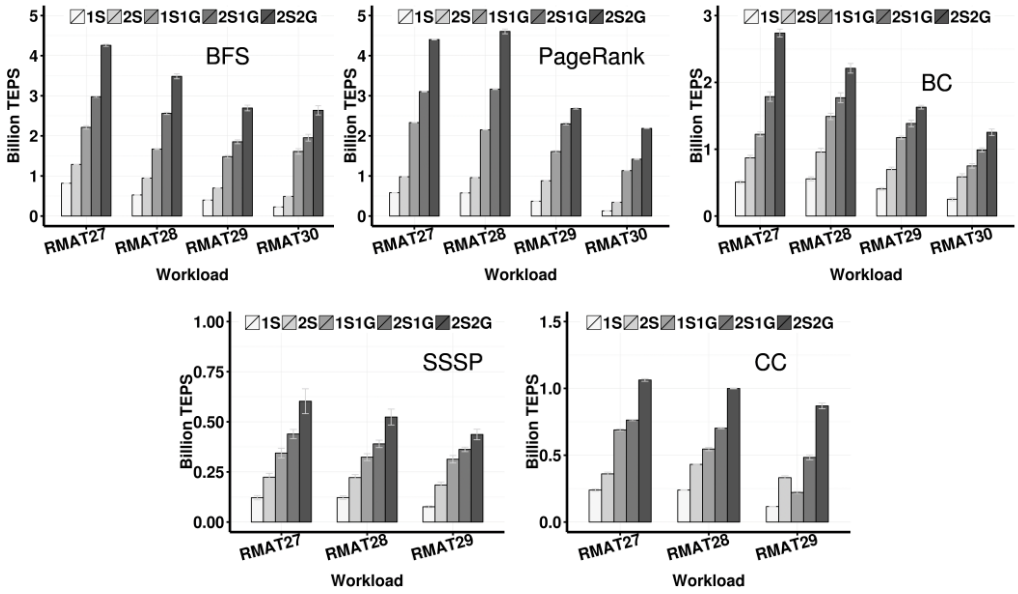

Figure 28: Processing rates for the different algorithms, hardware configurations and RMAT graph sizes. When GPUs are used, the graph is partitioned to obtain best performance. Experiments on configurations with a single socket (i.e., *1S* and *1S1G*) were performed by binding the CPU threads to the cores of a single socket. The results for an RMAT30 graph are missing for SSSP and CC because of memory space constraints (SSSP requires additional memory space to store the edge-weights while CC doubles the number of edges as it operates on undirected graphs).

Figure 28 presents traversal rates for the different algorithms, hardware configurations (up to two sockets and two GPUs) and graph sizes (1 to 16 billion edges).

First, the discussion focuses on the analysis of configurations with two processing units. The figures show that, for all algorithms, the hybrid system (1S1G) performs better than the dual-socket system (2S). On the one hand, adding

a second socket doubles the amount of last level cache and the number of memory channels, which are critical resources for graph processing performance, hence leading to close to double the performance compared to 1S configuration. On the other hand, the performance gain of 1S1G, brought by matching the heterogeneous graph workload with the hybrid system, outperforms that of the dual-socket symmetric system: between 30% to 60% improvement compared to the dual socket system (2S).

Second, the figure also demonstrates the ability of the hybrid system to harness extra processing elements. For example, in the case of BFS, the system achieves up to 3 Billion TEPS for the smallest graph (i.e., |E|= 2B), and, more importantly, it achieves as high as 1.68 Billion TEPS for an RMAT30 graph (i.e., |E|= 16B). It is worth pointing out that such performance is competitive with the performance results of the latest Graph500 competition for graphs of the same size. Also note that TOTEM is a generic graph-processing engine, as opposed to the dedicated BFS implementations for most submissions in Graph500; moreover the BFS implementation evaluated here is the standard top-down algorithm compared with the direction-optimized implementations [Beamer et al. 2013] that top the Graph500 competition.

Finally, the figures also demonstrate that the GPU can provide significant improvements for the large graphs, RMAT29 and RMAT30. This is made possible by employing *mapped memory* to increase the size of the offloaded partition. Particularly, for such large graphs, the GPU's limited memory space significantly constrains the size of the offloaded partition. For example, the GPUs on the testbed used in this study support 6GB of memory, and can host at most 0.625 Billion edges considering 64-bit edge identifiers (not including the space needed for the vertices' state, hence this limit is even lower especially for PageRank and BC); therefore, the GPU's device memory can store less than 5% of graph's edges. To enable offloading a larger partition to the GPU, part of the state is allocated on host

memory and mapped into the GPU's address space. The tradeoff is extra communication overhead over the high latency PCI Express bus.

This overhead has been reduced by taking the following measures: First, the impact of the high latency of the bus is reduced by restricting the use of mapped memory to allocate the part of the state that is (i) read-only, and (ii) can be accessed sequentially in batches; particularly, *mapped memory* is used to allocate the edges array since the focus is on static graphs. Second, transfer throughput is maximized by ensuring that the edges of a vertex are read in a coalesced manner when the vertex iterates over its neighbors. Finally, a side-effect of using mapped memory is that it naturally supports overlapped communication of a vertex's edge list with the computation of another vertex.

In summary, mapped memory affects performance in the following way: for small scale graphs (RMAT28 and below), the benefit from offloading a larger partition to the GPU via mapped memory is masked by the extra overhead of reading the graph data structure via the high latency PCI-E bus (even though mapped memory by design overlaps communication and computation). For large-scale graphs (RMAT29 and above) using mapped memory was beneficial. These points are summarized in a recent poster publication [Sallinen et al. 2014].

2.12 Evaluating Energy Consumption

This section investigates the power and energy characteristics of large-scale graph processing on hybrid (i.e., CPU and GPU) single-node systems. Although current GPUs have limited memory, previous sections demonstrated that large-scale graphs can still benefit from GPU acceleration by partitioning the graph to be processed concurrently on the CPU and the GPU.

On the one hand, GPUs are known to have higher FLOP/watt rate than CPUs [Huang et al. 2009], specifically for workloads that fit their computational model. Moreover, GPU acceleration allows a faster 'race-to-idle', enabling power savings

that are sizeable for newer GPU models which are power-efficient in idle state (as low as 25W [NVIDIA 2013]). In fact, as of writing this thesis, all top ten supercomputers in the Green500 list are GPU-accelerated. The reader is referred to Mittal et al. [Mittal and Vetter 2014] for a comprehensive survey on related works for analyzing and improving the energy efficiency of GPUs.

On the other hand, graph processing workloads are memory bound and have irregular processing patterns that can lead to underutilizing the computational capabilities of the GPU, and hence making the GPU less energy efficient. Moreover, GPUs have high thermal design power (TDP), typically double that of CPUs which may render an accelerated solution efficient in terms of time-to-solution but not in terms of energy. Therefore, it is unclear how using GPUs affects power consumption in the context of this work.

Concretely, this section focuses on the following high-level research questions:

- *Is it energy-efficient to partition the graph to be processed concurrently on a GPU and a CPU?*
- *Given a graph/algorithm workload and a fixed-power or energy budget, what is the (empirically determined) optimal balance between traditional and massively-parallel processors?*
- *What is the impact of increasing the graph scale on energy consumption and efficiency?*

To answer these questions, the rest of this section describes the experiment setup and the testbed's power characteristics (Section 2.12.1), evaluates the power consumption (Section 2.12.2), performance per watt (Section 2.12.3), and the energy-delay product (Section 2.12.4) for different hardware configurations, algorithms and workloads.

2.12.1 Experiment Setup

Measuring Power. Power is measured at the outlet using a WattsUP[7] meter which collects samples at one second intervals. Figure 29 (left) demonstrates the evaluation setup. To get a representative measurement of the energy consumption, each experiment is run for 5 minutes (e.g., repeating BFS searches). For accuracy, CPU-only experiments are conducted after removing the GPUs from the machine (as the GPU draws power from the PCI Express bus as well).

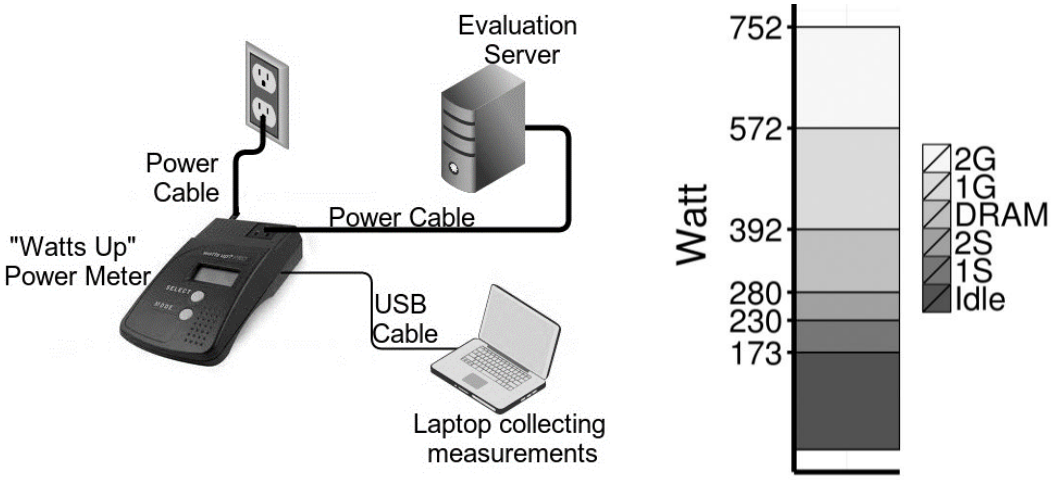

Figure 29: Testbed setup (left) and power characterization (right). The characterization of the evaluation server is obtained by incrementally stressing the different components of the system. Note that the GPUs are removed from the system when characterizing only the host components. Finally, "Idle" measures the idle power of the system without the GPUs.

Testbed Characteristics. Table 1 describes the evaluation platform. Simple compute and memory intensive kernels have been used to characterize the power consumption of the machine. Figure 29 shows the power consumption at idle, then when stressing one and both CPUs (listed as 1S and 2S in the plot), then the memory, and then each of the two GPUs. The high idle power consumption, which

[7] http://www.wattsupmeters.com

includes idle power of CPUs and RAM only, is mainly caused by the sizeable amount of available RAM (256GB). Note that the two CPUs consume less power than the DRAM, and less than one GPU. Two points are worth highlighting: (*i*) at peak load a significant share of the power is consumed by DRAM, and (*ii*) when loaded, GPUs consume significant power compared to other system components.

Metrics. Three energy metrics used are: *ii)* power consumption in Watts (Section 2.12.2), *iii)* power-normalized processing rate in TEPS/Watt (Section 2.12.3), and, *iv)* the energy-delay product, a metric biased for low-time-to-solution while taking into account the energy cost (Section 2.12.4).

2.12.2 Power Consumption

Power consumption is evaluated in this section with two key goals in mind: firstly, to understand the degree to which additional processing elements lead to additional power consumption (and how this relates to their TDP rating), and secondly, to characterize the variability in power drawn during processing.

Figure 30 shows the system power consumption under different workload/hardware combinations for all algorithms. To better illustrate the variation in power consumption during execution, the data is presented as boxplots.

The main differentiating factor in terms of power consumption is the hardware configuration (i.e., the number and type of processing elements used). Note that there is no major power difference across algorithms and workloads for the same hardware configuration.

Although the hybrid 1S1G configuration has a 155W higher TDP rating than a 2S configuration, it draws on average (across all configurations) only 50W more power than the symmetric configuration 2S, which has the same number of processing elements.

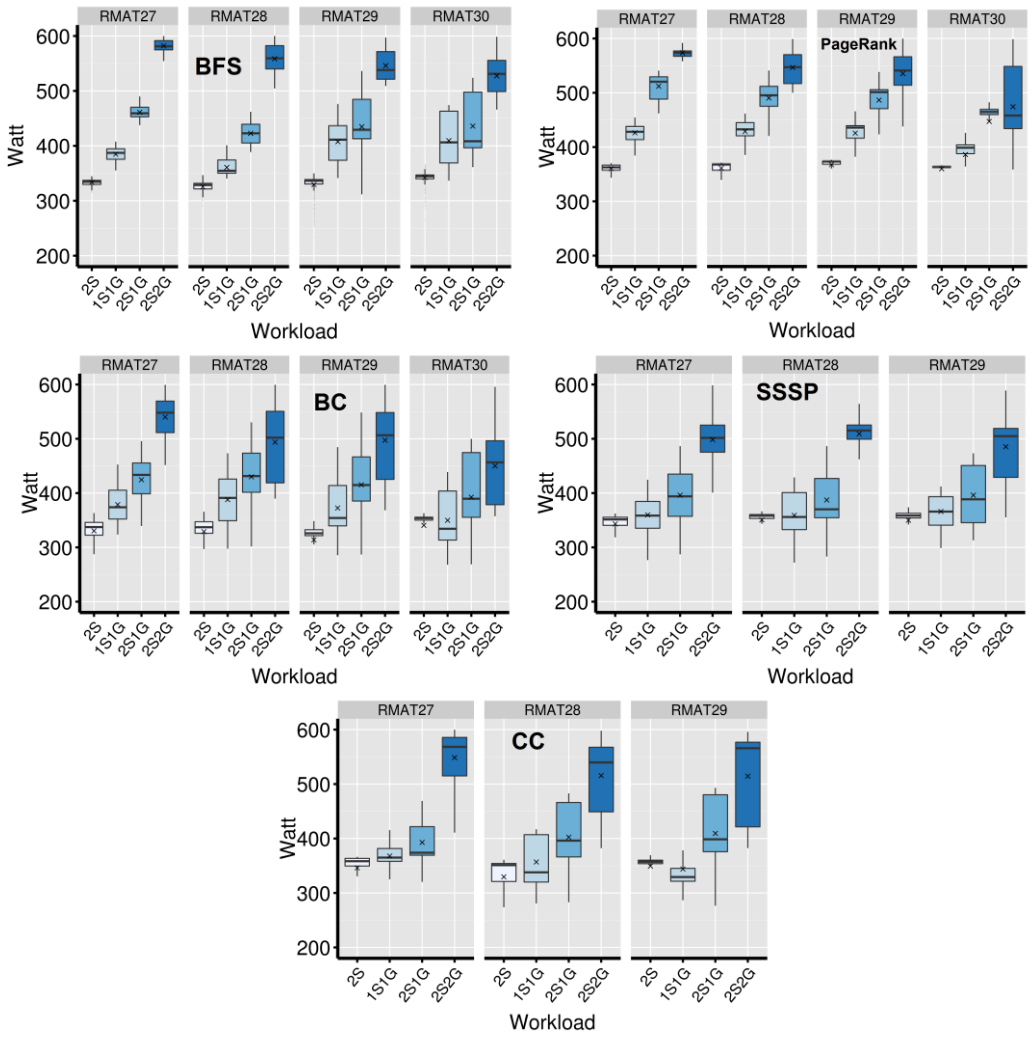

Figure 30: Power consumption (the lower the better). The upper and lower "hinges" of the boxplot correspond to the first and third quartiles. The middle line corresponds to the median. The whiskers extend from the lowest data point within 1.5 IQR of the lower quartile, to the highest data point within 1.5 IQR of the upper quartile (IQR is the Interquartile Range, which is the distance between the first and third quartiles). The mean is shown as a cross. Note the y-axis starts at 200W.

Also, while adding GPUs to the 2S configuration increases power drawn, the increase is below the TDP of the GPU. Adding a GPU increases power by ~100W, which is ~40% of the GPU's TDP. The reason is that the GPUs finish processing

their partition first and go in an energy-efficient idle state that consumes only a fraction of their peak power (25W) (see Figure 31).

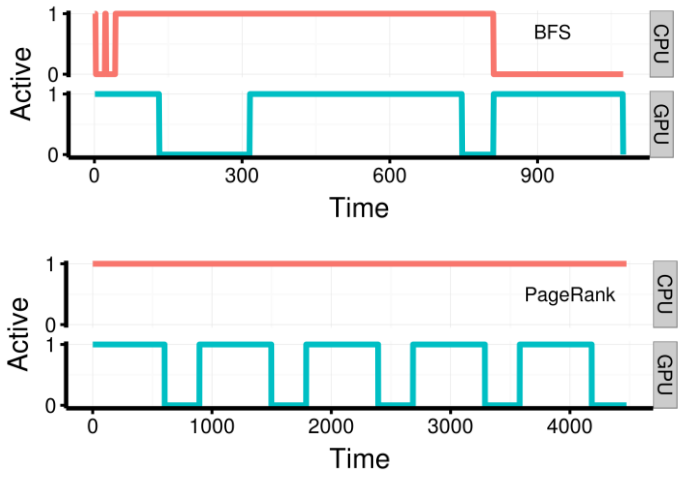

Figure 31: CPU/GPU active/idle state while processing an RMAT27 graph on a 2S1G setup (time is in milliseconds) for BFS (*top*) and PageRank (*bottom*). For BFS, the 'frontier' evolves in unpredictable ways, which results in having a processing element active in specific rounds and not in others. This behavior applies to Betweenness Centrality as well. For PageRank, the GPU finishes execution before the CPU in each execution round.

The hybrid configurations generate more variation in power consumption than processing on the CPU only. This is because, for some workloads, the computation is unbalanced between the CPU and the GPU(s). There are two reasons for this unbalance: First, GPUs do not have enough memory to hold a large enough partition that would balance the work for some workloads. Second, for BFS and BC, SSSP and CC the load varies across iterations. The time-series that presents the active/idle states for BFS and PageRank shed more light on this effect (Figure 31).

2.12.3 Power-normalized Processing Rates

To estimate the energy efficiency of different configurations, Figure 32 shows the power-normalized performance for all benchmarks (i.e., raw performance reported in Figure 28 divided by average drawn power). Note that, for each workload, the

plots can also be viewed as a comparison of raw energy consumed to process the graph.

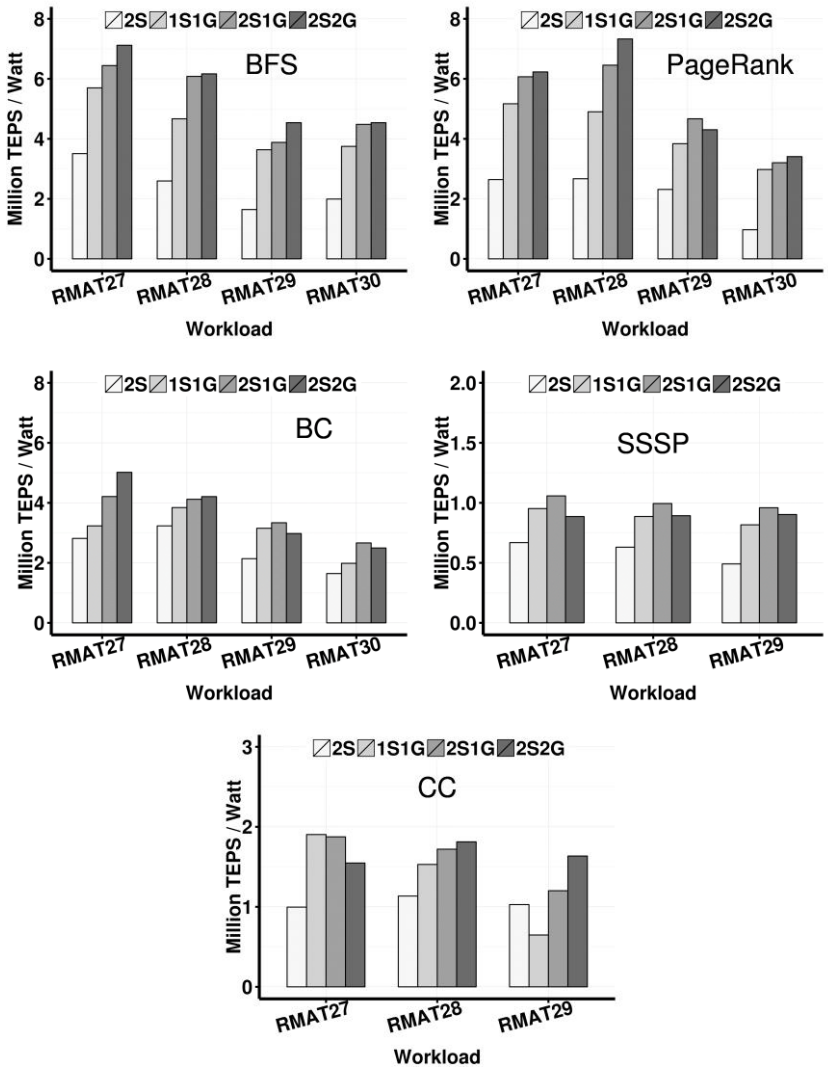

Figure 32: BFS, PageRank and BC power-normalized processing rate (the higher the better).

First, compare the power-normalized performance of configurations with two processing elements. A hybrid 1S1G system improves *both* raw performance and power-normalized performance compared to the symmetric 2S system. In the best

case, *the hybrid system achieves 1.9x higher efficiency* for the power-normalized performance metric.

Second, for most cases, adding more GPUs improves power-normalized performance as the gain in raw performance is higher than the increase in power consumption.

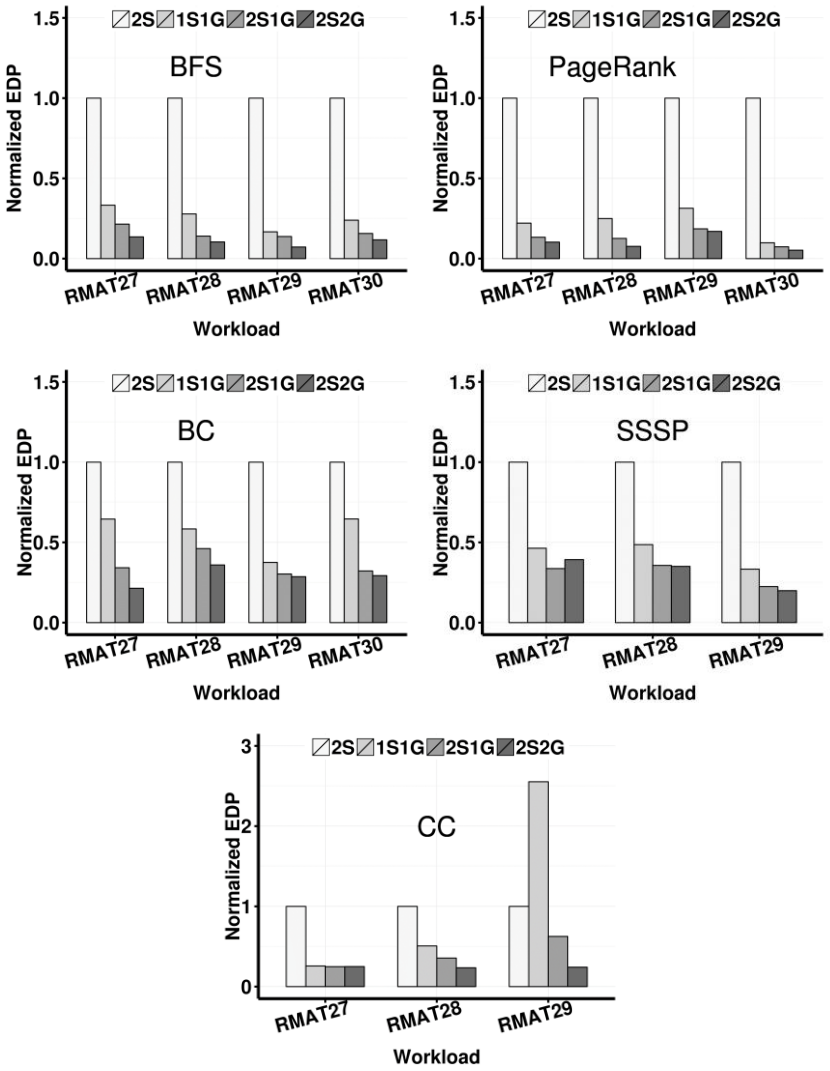

Figure 33: Normalized energy-delay product (the lower the better). The baseline is the CPU-only configuration with two processors (2S).

2.12.4 Energy-delay Product

Using a different energy-oriented metric, the energy-delay product (EDP), would not only support the same qualitative observations, but the relative advantage of the hybrid solution is higher. This is because EDP is biased more towards performance. The EDP is calculated as follows: $T^2 \times W$, where T is the processing time and W is the average power drawn.

Figure 33 presents the results of this experiment *normalized* to the 2S configuration to make the plot readable. *Importantly, these gains are preserved when processing larger graphs and for most executions.*

2.13 Comparing TOTEM's Performance with Other Frameworks

This section focuses on the following questions: *How does TOTEM's performance compare with other parallel graph processing frameworks? How does the hybrid system compare with a high-end symmetric one?*

These questions are motivated by the fact that new commodity single-node machines can be provisioned with as many as four CPU processors, where each processor can support more than 20 hardware threads. To make it easy to utilize these shared-memory machines for parallel graph processing, a number of frameworks have been proposed. The most notable are Ligra [Shun and Blelloch 2013] and the Galois [Nguyen et al. 2013] projects (these frameworks, and others, are discussed in more detail in the related work section in Section 2.14.3).

In particular, Nguyen et al. [Nguyen et al. 2013] proposed a lightweight graph processing framework for single-node shared memory systems named Galois. The work compared Galois with a number of other graph processing frameworks (including Ligra [Shun and Blelloch 2013] and PowerGraph [Gonzalez et al. 2012]) on a quad-socket system, and demonstrated that Galois compares favorably. The

largest workload that Nguyen et al. used was the Twitter network described in Table 2.

In this section, TOTEM's performance is compared with that of Galois, Table 4 shows the performance of Galois when executed on the evaluation machine used in this work (labeled 2S-Galois[8]), and the best performance reported by Nguyen et al. in their paper (labeled 4S-Galois in the table to indicate a quad-socket configuration). The table compares the four algorithms detailed in this work as well as a fifth one, namely connected components, when processing the same Twitter graph.

Table 4: Processing times in seconds for different algorithms and hardware configurations for the Twitter workload. The 2S-Galois column reports the performance of Galois on our evaluation machine. The performance of the four socket platform (labeled 4S-Galois) is the best performance reported by [Nguyen et al. 2013] when processing the same workload for various frameworks that include Galois, Ligra, and PowerGraph. The characteristics of the 4S platform are: Four Intel E7-4860 processors, each with 10 cores (20 hardware threads) @ 2.27GHz and 24MB of LLC per processor, hence a total of 80 hardware threads and 96MB of LLC – significantly better than our platform. Note that the processing time for PageRank is for a single round, while for BC it is for a single source.

Algorithm/Configuration	2S Galois	2S TOTEM	4S Galois	1S1G TOTEM	2S1G TOTEM	2S2G TOTEM
BFS	5.0	4.0	2.3	1.1	0.85	0.4
PageRank	24.3	8.1	10.7	1.5	1.12	0.5
BC	29.7	20.8	12.0	4.8	3.7	2.5
SSSP	13.2	4.6	8.6	3.3	3.1	1.9
Connected Components	41.1	42.0	31.9	38.7	25.8	13.5

First, the table demonstrates that TOTEM's performance on a 2S configuration is not only better than that of Galois on our evaluation machine, but also competitive with the best reported numbers on the 4S one, even surpassing it in the

[8] The Galois experiments presented in this section where executed on the by my colleague Tahsin Reza.

cases of PageRank and SSSP. This increases my confidence that the speedup results reported throughout this work use a meaningful baseline.

Second, the hybrid configurations offer significant speedups compared to both symmetric systems (2S and 4S). In the case of BFS, while the 4S system delivers 60% better performance than 2S, a modest 1S1G hybrid configuration speeds up the performance by 3.5x compared to 2S, and 2.1x compared to 4S at a much lower cost in terms of both acquisition and energy. Moreover, the hybrid configuration 2S2G offers over 5.5x speedup compared to 4S, the symmetric system with the same number of processing elements.

In the case of PageRank, a 1S1G hybrid configuration offers close to one order of magnitude better performance than the 4S system, while a 2S2G hybrid configuration delivers an impressive, 20x improvement. Two reasons behind these impressive speedups: First, the ability of the hybrid system to reshape the workload to run much faster on the CPU; second, a significant portion of the Twitter workload fits in the GPUs (up to 70% of the workload), which are able to more efficiently process the floating point operations performed by the PageRank algorithm.

Finally, the table shows similar significant performance improvements for Betweenness Centrality, SSSP and Connected Components algorithms.

In summary, the comparison in this section demonstrates two important points: First, the CPU-only implementation used to evaluate the performance of the symmetric system throughout this work is comparable to the best reported numbers in the literature. Second, this comparison reaffirms the results observed in previous sections in that a hybrid GPU-accelerated platform offers tangible speedups compared a symmetric one: a modest one CPU processor and one GPU (1S1G) hybrid configuration performs significantly faster than a symmetric system with as many as four high-end CPU processors.

2.14 Related Work

This section discusses related work from several aspects. First, Section 2.14.1 reviews efforts on optimizing graph algorithms for multi- and many-core platforms. Next, Section 2.14.2 reviews work related to graph partitioning. Finally, Section 2.14.3 reviews abstractions similar to TOTEM that aim to hide the complexity of implementing graph algorithms on parallel platforms.

2.14.1 Optimizing Graph Algorithms

While I am unaware of previous works on optimizing graph processing on hybrid systems, many efforts exist on optimizing graph algorithms on homogeneous systems: either on multicore CPUs or on GPUs alone. For example, several studies focus on optimizing BFS on multi-core CPUs [Agarwal et al. 2010; Hong et al. 2011b; Chhugani et al. 2012]. For example, Chhugani et al. [Chhugani et al. 2012] apply a set of sophisticated techniques to improve the cache hit rate of the "visited" bit-vector, reduce inter-socket communication, and eliminate the overhead of atomic operations by using probabilistic bitmaps. My approach to partition the graph goes in the same direction in terms of improving the cache hit rate on the CPU using a hybrid system.

Past projects have also explored GPU-only solutions. These projects either assume that the graph fits the memory of one [Hong et al. 2011a; Katz and Kider Jr 2008], or multiple GPUs [Merrill et al. 2012]. In both cases, due to the limited memory space available, the scale of the graphs that can be processed is significantly smaller than the graphs presented in this paper.

Hong et al. [Hong et al. 2011b] work is, perhaps, the closest in spirit to this work as it attempts to harness platform heterogeneity: the authors propose to divide BFS processing into a first phase done on the CPU (as, at the beginning, only limited parallelism is available), and a second phase on the GPU once enough parallelism is exposed, having the whole graph transferred to the GPU to accelerate

processing. However, this technique still assumes that the whole graph fits the GPU memory; moreover, the work is focused on BFS only.

In summary, techniques that aim to optimize graph processing for either the CPU or the GPU are complementary to the approach proposed in this work in that they can be applied to the compute kernels to improve the overall performance of the hybrid system. In fact, this work uses some of these techniques in the hybrid implementations, such as using pull-based approach in PageRank and optimizing thread allocation on the GPU [Li and Becchi 2013; Hong et al. 2011a].

2.14.2 Graph Partitioning

There is no shortage of work on graph partitioning for parallel processing. Traditionally, the problem is defined as to partition a graph in a balanced way, while minimizing the edge cut. It has been shown that this problem is NP-hard [Garey et al. 1974], therefore several heuristics were proposed to provide approximate solutions. Some heuristics, such as Kernighan–Lin [Kernighan 1970], have quadratic $O(n^2 \log n)$ time complexity, which is prohibitively expensive for the scale of the graphs targeted by this work. Multilevel partitioning techniques, such as METIS by Karypis et al. [Karypis and Kumar 1998], offer an attractive moderate time complexity.

I believe that classical solutions do not properly address the requirements for graph partitioning on hybrid platforms. Such techniques are mainly optimized to minimize communication, which is not the bottleneck in the platform this work targets. Moreover, such solutions target homogeneous parallel platforms as they focus on producing balanced partitions, which is not sufficient for a hybrid system that has processing units with largely different characteristics.

2.14.3 Graph Processing Frameworks

A number of frameworks have been proposed to simplify the task of implementing graph algorithms at scale, which can be divided into two categories depending on

the target platform. On the one hand, frameworks for shared-nothing clusters, such as Pregel [Malewicz et al. 2010] and PowerGraph [Gonzalez et al. 2012], partition the graph across the cluster nodes, and provide abstractions to implement algorithms as vertex programs run in parallel. Cluster-based solutions offer the flexibility to scale with the size of the workload by adding more nodes. However, this flexibility comes at performance and complexity costs. Particularly, performance suffers from the high cross-node communication overhead: over one order of magnitude slower compared to single-node systems [Nguyen et al. 2013]. Moreover, the fact that the system is distributed introduces new problems such as network partition, partial failures, high latency and jitter, which must be addressed when designing the framework and when implementing algorithms on top of it, hence greatly increasing the complexity of the solution.

On the other hand, single-node platforms are becoming increasingly popular for large-scale graph processing. Recent advances in memory technology make it feasible to assemble single-node platforms with significant memory space that is enough to load and process large-scale graphs for a variety of applications. Such platforms are more efficient in terms of both performance and energy, and potentially less complex to program compared to shared-nothing clusters. Examples of frameworks that capitalize on this opportunity include Ligra [Shun and Blelloch 2013], Galois [Nguyen et al. 2013] and STINGER [Ediger et al. 2012]. However, I am not aware of any frameworks that harness GPUs in a hybrid setup for large-scale graph processing.

2.15 Lessons and Discussion

The results presented in this work allows putting forward a number of guidelines on the opportunity and the supporting techniques required to harness hybrid systems for graph processing problems. These guidelines are phrased as answers to a number of questions.

- ***Q1:*** *Is it beneficial to use a hybrid system for large-scale graph processing?*

  A1: Yes. One concern when considering using a hybrid system is the limited GPU memory that may render using a GPU ineffective when processing large graphs. This work shows, however, that it is possible to offload a relatively small portion of the graph to the GPU and obtain benefits that are higher than the proportion of the graph offloaded for GPU processing. This is made possible by exploiting the heterogeneity of the graph workload and the characteristics of the hybrid system to reshape the workload to execute faster on the bottleneck processor.

- ***Q2:*** *Is it possible to design a graph processing engine that is both generic and efficient?*

  A2: Yes. A range of graph algorithms can be implemented on top of TOTEM, which exposes similar BSP-based computational model and functionality to that offered by a number of other widely accepted generic graph processing engines designed for cluster environments (e.g., Pregel). My experiments show that being generic – that is, being able to support multiple algorithms and not only the popular Graph500 BFS benchmark, did not hinder TOTEM's ability to efficiently harness hybrid systems, and scale when increasing the number of processing elements. We have also implemented on top of TOTEM the direction-optimized BFS algorithm [Beamer et al. 2013]. The results support the main takeaways presented here. Based on this implementation, TOTEM's performance on a hybrid system with dual-socket and dual-GPU is capable of 10.31 Billion breadth-first search traversed undirected edges per second on a graph with one Billion vertices and 16 Billion undirected edges. We have

submitted this result to the Green Graph500[9] competition, and ranked 6th in the 'Big Data' category.

- ***Q3:*** *Is the partitioning strategy key for achieving high performance?*
  A3: Yes. The low-cost partitioning strategies this work explores – which are informed by vertex connectivity – provide in all cases better performance than blind, random partitioning.

- ***Q4:*** *Which partitioning strategies work best?*
  A4: The answer is nuanced and the choice of the best partitioning strategy depends on the graph size and on the specific characteristics of the algorithm (particularly on how much state is maintained and on the read/write characteristics). If the graph is large, then the CPU will likely be the bottleneck as it is assigned the larger portion of the graph, while only a small fraction can be offloaded to the GPU. Thus, the goal of partitioning is to improve the CPU performance by producing and assigning to it the friendliest workload to its architecture. The evaluation in this work shows that placing the high degree vertices on the CPU offers the best overall performance: it improves the cache hit rate for algorithms that use summary data structures, and, for the ones that do not use them, it offloads most of the expensive per-vertex work to the accelerator. However, for algorithms with large state per vertex, placing the few high degree nodes on the GPU allows for offloading significantly more edges (20% more in the case of Betweenness Centrality when processing the Twitter network), and hence better balances the load between the CPU and the GPU.

- ***Q5:*** *Should one search for partitioning strategies that reduce communication overheads in order to improve overall performance?*

[9] green. graph500.org

A5: No. This work shows that, in the case of scale-free graphs, the communication overhead can be significantly reduced – to the point that it becomes negligible relative to the processing time – by simple aggregation techniques. Aggregation works well for four reasons. First, many real-world graphs have skewed connectivity distribution. Second, the number of partitions the graph is split into is relatively low (only two for a hybrid system with one GPU). Third, aggregation can be applied to many practical graph algorithms, such as BFS, PageRank, Single-source Shortest Path and Betweenness Centrality to mention only a few. Fourth, there is practically no visible cost for aggregation: conceptually, aggregation moves the computation to where the data is, which must happen anyway. In contrast, partitioning algorithms that aim to reduce communication have typically high computational or space complexity and may be themselves 'harder' than the graph processing required [Feldmann 2012].

- ***Q6:*** *Is there an energy cost to the time-to-solution gains provided by the hybrid platform?*

  A6: No. One concern is that the GPU's high peak power consumption may make an accelerated solution inefficient in terms of energy. The experiments in Section 2.12 rejects this concern: GPU-acceleration allows a faster 'race-to-idle', enabling energy savings that are sizeable for newer GPU models which are power-efficient in idle state (as low as 25W [NVIDIA 2013]). Additionally, as demonstrated in the various profiling figures in this paper (Figure 9, Figure 11, Figure 21, and Figure 24), the GPU finishes much faster than the CPU, and that allows it to go to the idle state even sooner. The experiments show that a hybrid system is not only efficient in terms of time-to-solution, but also in terms of energy and energy-delay product.

- ***Q7:*** *Why not use DVFS to lower energy footprint?*

A7: On the CPU side, a recent analysis [Schöne et al. 2012] shows that, for the new Intel processors (e.g., Intel's Sandy Bridge), both memory latency and bandwidth strongly depend on processor frequency (this result is confirmed on the platform used in this study). This limits the opportunity to use dynamic voltage and frequency scaling (DVFS) to save energy on the CPU side. On the GPU side, however, recent GPU models support setting different frequencies for the memory and the compute cores. Previous work [Jiao et al. 2010; Abe et al. 2012] shows that energy consumption can be reduced by lowering the core frequency for memory-intensive kernels, an opportunity that could improve the energy efficiency of the hybrid system.

- ***Q8:*** *Is it possible that the results presented in this dissertation are dependent on the hardware platform used for experimentation?*

  A8: The evaluation presented here was performed on a new machine with state-of-the-art CPU and GPU models as of writing this dissertation. Previous publications [Gharaibeh et al. 2012; Gharaibeh et al. 2013a; Gharaibeh et al. 2013b], which this chapter is based on, each used a different evaluation machine (state-of-the-art at that time). More importantly, the results obtained on the older hardware generations are consistent with the latest results published here on a newer CPU and GPU hardware models. Practically, the ideas presented in this dissertation have been evaluated on three hardware generations.

- ***Q9: Why not stream the whole graph into the GPU?***

  A9: GPUs have limited memory space and hence they cannot host a large scale graph. TOTEM addresses this problem by partitioning the graph between the CPU and the GPU(s). Another possible approach is to stream the graph into the GPU(s). In particular, the graph can be split into smaller subgraphs that can fit into the GPU memory and processed one after the other on the GPU. Using

double buffering, the copying of a subgraph can overlap with the processing of another, and hence the communication overhead can be partially hidden.

Streaming has two main limitations. First, it will be bottlenecked by the communication channel. Specifically, the processing rate of a streaming solution can be represented using the following simple model:

$$\text{Processing Rate} = min\{T_r, G_r\},$$

where T_r is CPU-to-GPU transfer rate in Edges Per Second (EPS), and G_r is the GPU processing rate. If we assume that the GPU processing rate is significantly higher than that of the communication channel, the overall processing rate can be at most equal to T_r.

The measured transfer rate of the PCI Express 3.0 bus is 10GB/Sec, therefore, in the optimistic case where an edge is represented by 4bytes, T_r can be at most 2.5 Billion TEPS. Considering undirected edges, where an edge requires at least 8bytes, the transfer rate will be half of that; moreover, if a weight is associated with each edge, then T_r will be even lower. As the evaluation of TOTEM shows, a CPU-GPU partitioned approach can achieve better results (see Figure 20 and Figure 28). For example, as mentioned before, TOTEM was able to achieve 10.3Billion TEPS (undirected edges) using two GPUs, more than four times better than what a streaming approach can achieve in the best case.

A second limitation that streaming the whole graph approach has is that it does not take advantage of the opportunity of specialization offered by the hybrid system. Figure 15 demonstrates the traversal rate of an RMAT25 graph, which is small enough to fit the GPU memory space. The figure shows that partitioning the graph such that keeping on the CPU 25% of the edges of the low degree vertices double the performance compared to processing the whole graph on the GPU.

- ***Q10: How does this work apply to integrated GPUs?***

  A10: Although this work focuses on discrete GPUs, the proposed techniques also apply to integrated ones. The goal of integrated GPUs is to remove the PCI Express bus by placing the main processor and the accelerator on the same die and share the same memory space. AMD's APU (Accelerated Processing Unit) is an example of such a hybrid setup.

  While current APU models do place the main processor and the accelerator on the same die, they still employ distinct memory partitions, and hence the techniques related to reducing communication overhead still apply for current APU generations. More importantly, the techniques related to partitioning the graph to achieve specialization apply irrespective whether the GPU is integrated or discrete. This is because the partitioning strategies proposed here are geared towards the processing characteristics of the CPUs and accelerators rather than how they are connected.

- ***Q11:*** *What other factors that could improve the performance of single-node hybrid platforms?*

  A11: I believe that having more memory on the GPU would significantly improve performance as a larger partition can be offloaded. Also, using low-voltage DRAM, could reduce the power drawn by the large memory space. Finally, high-bandwidth, low-power SSDs are now available (e.g., Intel's 900 family, supports 1GB/s sequential read and draws as little as 25W); such storage can be used to offload part of the read-mostly graph state (e.g., the graph data structure), and hence reduce power drawn by memory.

- ***Q12:*** *What platform offers the best tradeoff between acquisition cost, energy and performance?*

  A12: The experience collected throughout this work supports recommending the following simple decision process: If the graph is small and fits the GPU

memory, the recommendation is to process it on GPU only (a single GPU draws power comparable to a dual-socket CPU, but it is at least 2x faster). For larger graphs, the recommendation is to boost the host's memory, adding GPUs and using TOTEM to implement algorithms on such a hybrid setup. Finally, for massive many-billion vertices graphs, if energy is the main concern, I speculate that a single-node solution along the lines of GraphChi [Kyrola et al. 2012], which processes the graph from SSDs, will be most advantageous. If time-to-solution is the primary concern then I conjecture that a cluster composed of as few *fat* nodes as possible, where each node is provisioned with as much memory and GPUs as possible, will be the most efficient setup (compared to a cluster of many low-end commodity nodes as used today).

Chapter 3

Efficient Large-Scale Sequence Alignment on Hybrid Platforms

GPUs offer drastically different performance characteristics compared to traditional multicore architectures. To explore the tradeoffs exposed by this difference, this project refactors MUMmer [Kurtz et al. 2004], a widely-used, highly-engineered bioinformatics application which has both CPU- and GPU-based implementations.

The experience from this project is synthesized as three high-level guidelines to design efficient applications for hybrid GPU-accelerated platforms. First, minimizing the communication overheads is as important as optimizing the computation. Second, trading-off higher computational complexity for a more compact in-memory representation is a valuable technique to increase overall performance (by enabling higher parallelism levels and reducing transfer overheads). Finally, ensuring that the chosen solution entails low pre- and post-processing overheads is essential to maximize the overall performance gains.

Based on these insights, I designed and developed MUMmerGPU++, a new GPU-based design of the MUMmer sequence alignment tool. MUMmerGPU++

The research presented in this chapter resulted in the following publications:

(i) **Abdullah Gharaibeh** and Matei Ripeanu, Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance, IEEE/ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), New Orleans, Louisiana, November 2010 *(20% acceptance rate)*.

(ii) **Abdullah Gharaibeh** and Matei Ripeanu, Accelerating Sequence Alignment on Hybrid Architectures, Scientific Computing Magazine, February 2011.

achieves, on realistic workloads, significant speedups compared to a previous, highly optimized GPU port.

3.1 Context

This work advocates the need for a careful space/time tradeoff analysis when designing applications for (or porting applications to) hybrid GPU-accelerated platforms. In particular, this project analyzes and evaluates these tradeoffs in the context of a well-engineered, widely-used bioinformatics application [Delcher et al. 1999; Delcher 2002; Kurtz et al. 2004] which performs exact sequence alignment: a memory-intensive operation involving exact string matching for a large number of strings. The tool has both CPU- and GPU-based implementations named MUMmer [Delcher et al. 1999; Delcher 2002; Kurtz et al. 2004] and, MUMmerGPU [Schatz et al. 2007; Trapnell and Schatz 2009], respectively.

Using a GPU to accelerate sequence alignment is appealing for two reasons. First, GPUs support massive hardware multithreading that is able to hide memory access latency, a main bottleneck for this application. Second, parallelizing this operation is straightforward since queries can be processed independently and the problem space can be easily partitioned.

Profiling the latest version of MUMmerGPU, however, reveals that only a relatively low share of the total application runtime is spent on computing. Figure 34 shows that more than 50% of the time is spent on data transfers and post-processing results produced by the GPU kernels.

My hypothesis is that the culprit for this arguably low use of the GPU is the core data structure (namely the *suffix tree*) that is used for performance-efficient string matching by both the original CPU-based tool, MUMmer, and its GPU port, MUMmerGPU. I contend that this data structure is not a good match for GPU implementations: it offers fast matching at the cost of large memory footprint

(which translates to large data transfers and limited parallelism) and relatively complex post-processing.

Thus, the goal of this study is to: *(i)* explore the feasibility of using a different data structure that offers different space/time tradeoffs, *(ii)* evaluate the effect of this choice on the overall application performance, and, *(iii)* to build a solution that makes best use of both types of processing units of the hybrid system.

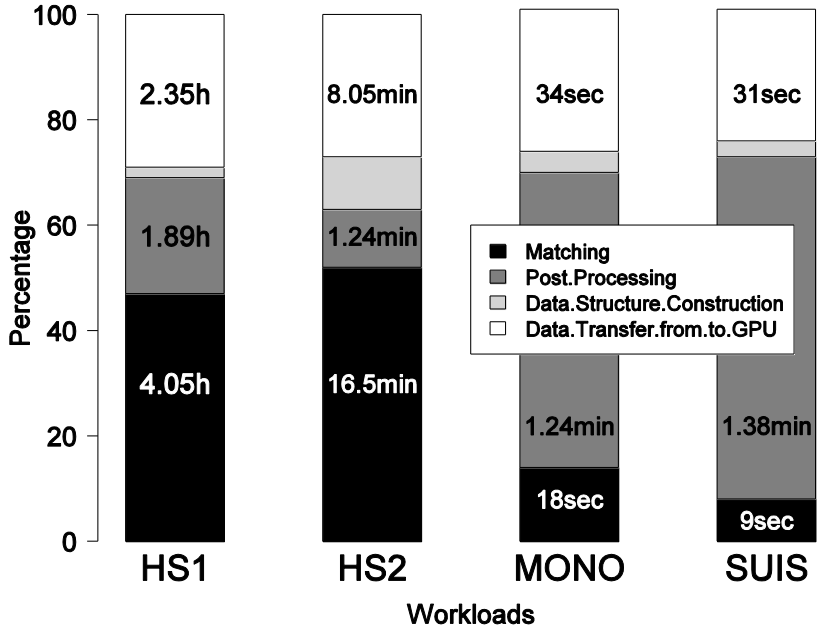

Figure 34: Percentage of time spent in each processing stage using MUMmerGPU for the workloads presented in Table 1, for config2 (discussed in Section 3.5.1).

Note that, to highlight the effect of the choice of the data structure, the study focuses on the high-level application design, and, throughout the design and implementation effort, little attention was paid to low-level performance optimizations.

3.2 Research Questions

This work investigates techniques to improve the performance of sequence alignment on hybrid GPU-accelerated platforms. The following research questions guide this investigation:

Q1. Is it feasible to efficiently process large-scale sequence alignment workloads on GPUs? While previous works [Trapnell and Schatz 2009; Schatz et al. 2007] demonstrated that exact sequence alignment can be accelerated using GPUs, it is not clear whether the proposed solution scale to large workloads (such as the human genome).

Q2. Given the difference in performance characteristics between GPUs and CPUs, how does the choice of data structures, which offer different space/time tradeoffs, affect the overall performance? While GPUs offer an order of magnitude higher peak memory access bandwidth and peak computational power, current GPUs have limited, often an order of magnitude lower, internal memory space, hence it is not clear if the data structure that offers the lowest time complexity is the best solution.

Q3. Consider a system with one CPU processor, what is more performance advantageous, in terms of both time to solution and energy, adding a GPU or a second CPU processor to the system? The question is motivated by the requirement of current GPU models to operate within a host machine, therefore a fair CPU vs GPU comparison should take this requirement into account.

Making progress on answering these questions is important in the context of current hardware trends: future computing systems will host processing elements with different performance characteristics. These differences make reconsidering the choice of the data structures used a necessary step (for efficient time or energy execution) when porting applications from one processing element to another.

At the same time, answering these questions in the context of natural sciences is equally important: commoditized DNA sequencing technologies have unleashed immense data volumes, and hence extracting the best possible performance from the different processing elements is essential to efficiently transform this data into new knowledge.

3.3 Chapter Structure

A fair amount of background material is presented in this chapter to make it self-contained. If the reader is familiar with the sequence alignment problem and the data structures to accelerate string matching and their space/time tradeoffs (Section 3.5), then (s)he can skip directly to Section 3.6, which discusses in detail the effect of space/time tradeoffs. Section 3.7 discusses the effort to offload sequence alignment computation to the GPU using a different data structure. Section 3.8 presents a performance analysis of the proposed solution to assess its value. Section 3.9 presents a detailed evaluation over multiple directions: a comparison with the past approach, ability to harness high-end GPUs and energy efficiency. Finally, Section 3.10 summarizes the lessons learned from this work and discusses a number of interrelated questions.

3.4 Contributions

The processing elements in a hybrid GPU-accelerated system have drastically different performance characteristics. The GPU has up to two orders of magnitude higher peak memory access bandwidth, one order of magnitude higher peak computational power per Byte of memory, yet one order of magnitude lower internal memory space.

This work argues that these differences make reconsidering the choice of the data structures used a necessary step when porting applications to hybrid, GPU-supported platforms. In more detail, the contributions are:

First, *the study confirms the feasibility of harnessing GPUs to accelerate an important irregular application, sequence alignment*. This result is contrary to the common believe that GPUs can only accelerate applications that expose regular computations and have predictable memory access patterns. For example, for the experimental setup used in this study, a hybrid one CPU and one GPU configuration offers over 2x speedup compared to the performance achieved by two CPU processors.

Second, *the study demonstrates the importance of a careful choice of the data structure used to support GPU applications*. A data structure that matches well the space/time tradeoffs specific to the GPU can unlock dramatic performance gains. The direct implication of this observation is that, when porting applications to a GPU-supported platform, designers should not only focus on extracting the application parallelism usable in a SIMT model; but, in order to maximize the performance gains, they may need to reconsider the choice of the data structures used.

Third, *the study contrasts, in the context of the sequence alignment application, the energy consumption of traditional and hybrid systems*. The study shows that, although the energy consumption *rate* (i.e. power) of a hybrid system is higher, the *total* energy consumed to complete a full sequence alignment workload is lower due to its higher performance. For the experimental setup in this work, which compares a hybrid platform (one GPU and one CPU processor) with a symmetric traditional one (two CPU processors), the hybrid platform requires a performance gain of at least 65% to become more energy efficient than the traditional one. The hybrid GPU-accelerated platform achieves significantly higher performance, which enables it to consume 40% less energy (i.e., 40% less Joules consumed to process a workload), and up to 2.8x more energy efficient when considering an energy metric that is biased for time-to-solution (i.e., energy-delay product (EDP)).

3.5 Background

Genome sequencing is the biochemical process of determining the order of nucleotides in a DNA molecule. This is an essential process to gain important information needed for biological and medical studies. New high-throughput sequencing technologies, such as 454 life sciences[10] and Illumina[11], enabled dramatic increase in sequencing rates, while significantly reducing the overall sequencing costs. This advancement enables producing an enormous volume of data (generated at the rate of terabytes per day) which needs to be processed and analyzed, leading, as a result, to increased demand for high-performance sequence analysis tools.

Table 5: Sample sequence alignment workloads. For experimental purposes, three different minimum-match length values are used.

Workload / Species	Reference Length	Number of Queries	Sequencing Technology (Read Length)	Minimum-match Length
HS1 / Homo sapiens chromosome 2	238,202,930	78,310,972	454 (~200)	Config1: 25, Config2: 50, Config3: 100
HS2 / Homo sapiens chromosome 3	100,537,107	2,622,728	Sanger (~700)	Config1: 50, Config2: 100, Config3: 200
MONO / L. monocytogenes	2,944,528	6,620,471	454 (~120)	Config1: 20, Config2: 40, Config3: 80
SUIS / S. suis	2,007,491	26,592,500	Illumina (~36)	Config1: 15, Config2: 20, Config3: 30

This project focuses on sequence alignment: the operation on genomic data which aims to find all occurrences of a sequence in another longer one, where a sequence is a string composed of some alphabet Σ (e.g., the alphabet set {A,C,G,T}

[10] http://454.com
[11] http://www.illumina.com

in case of genome sequences). Sequence alignment is widely used in computational biology studies such as gene finding, comparative genomics and genome assembly [Li and Homer 2010]. In particular, this project focuses on a specific, yet important, use case in sequence alignment, called genome sequence alignment.

3.5.1 The Sequence Alignment Problem

In *sequence alignment*, a large number of short sequences, (called 'reads') and referred hereafter as the query set, are aligned to a longer genome *reference sequence*. This process is an essential time-intensive operation in comparative genome assembly [Pop 2004; Trapnell and Salzberg 2009; Nagarajan and Pop 2013].

Formal Problem Definition

The exact sequence alignment problem can be formally defined as follows: For each query q in the query set Q, find all *maximal matches* of minimum length l in the reference string S. A *maximal match* is defined as a match of a suffix q_i of query q starting at position i (and referred hereafter as a subquery) to a suffix S_j of the reference string S that is at position j. The match is assumed to be as long as possible, and not contained in any suffix q_k, with $k < i$.

For example, for a query string "ACACT" and a match length of at least three, the following three subqueries must be searched in the reference string: ACACT, CACT, and ACT. For each subquery, all match occurrences that are at least three characters long must be reported. Figure 35 shows a snapshot of a reference sequence, query set and alignment result.

Workload Characteristics

Depending on the species, the length of the genome reference sequence ranges from a few million nucleotides (e.g., for *Streptococcus Suis*), to a few billion nucleotides

(e.g., for *Homo Sapiens*), to hundreds of billions nucleotides (e.g., for *Amoeba Dubia*). A nucleotide is represented as a character from the alphabet set {A,C,G,T}.

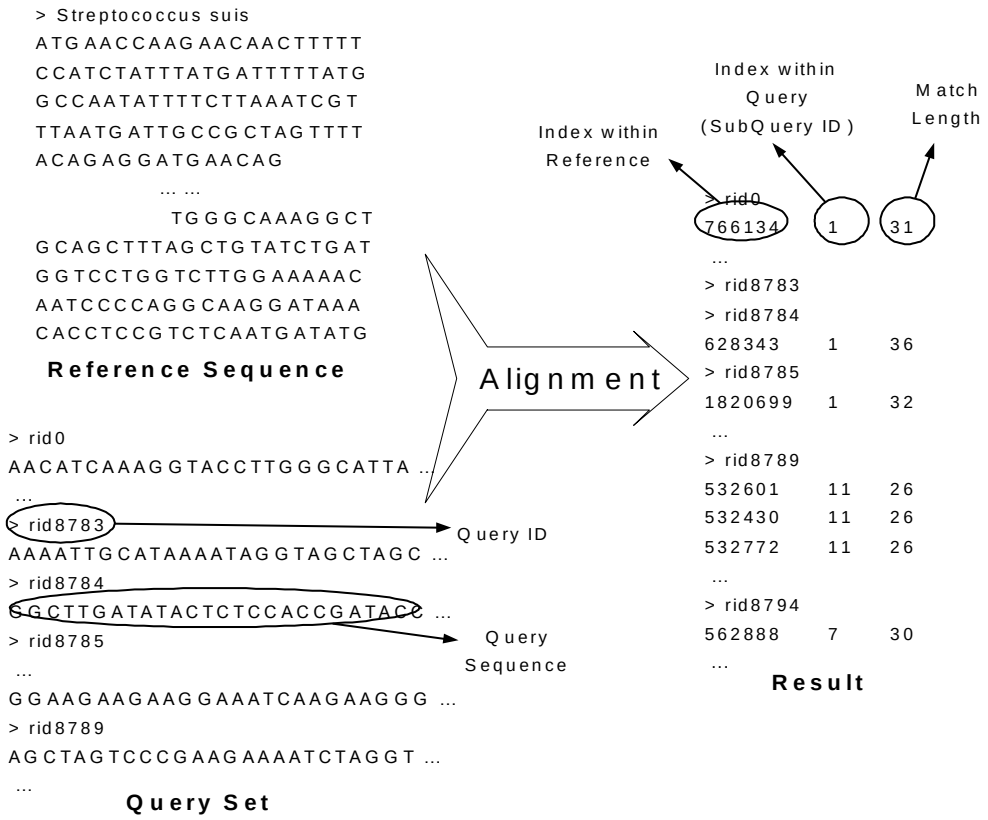

Figure 35: Genome sequence alignment example.

The number of queries ranges from few thousand to hundreds of millions, and the query length ranges from tens to several hundred nucleotides depending on the sequencing technology used. In particular, current high-throughput sequencing technologies, such as Illumina and 454, produce significantly shorter queries (30–200 nucleotides) compared to previous sequencing generations such as sanger (~700 nucleotides).

Table 5 presents a sample of sequence alignment workloads fetched from the National Center for Biotechnology Information (NCBI) archive [NCBI 2014], and

used to drive the experiments in this project. The workloads include sequencing data that cover a range of usage scenarios. For example, **HS1** is a relatively large scale workload for a *Homo Sapiens* that aligns about 78M queries of average length 200 to the genome sequence of the human chromosome #2 which is about 238M nucleotides long. **MONO** is a smaller scale workload for a *Listeria Monocytogenes* species which aligns ~6M queries to a reference genome sequence of ~2M nucleotides long.

Finally, the minimum match length is a user-specified parameter. A short minimum-match length implies a relaxed assumption on what is considered a match, and vice versa. On the one hand, since all the suffixes of each query need to be aligned, a short minimum-match length increases the number of subqueries to be aligned per query, and, at the same time, increases the chance to find matches; therefore the workload becomes larger, and requires more processing time. On the other hand, a longer minimum-match results in reducing the workload demands.

For each workload, three minimum-match length values were chosen that represent relaxed (config1), moderate (config2) and conservative (config3) configurations with respect to typical values used in practice [Schatz et al. 2007; Trapnell and Schatz 2009] (see Table 5).

3.5.2 Substring Matching

The core of the sequence alignment problem is a basic substring matching operation: find a string of length m in another reference string of length n, where $n >> m$. A naïve approach to this problem is to exhaustively search the reference string. This approach has linear space complexity, $O(n)$; in fact, if a nucleotide is represented using one Byte, the space requirement of this approach is exactly n Bytes. However, the time complexity is daunting: $O(mn)$, especially when considering that matching needs to be done on a large number of queries.

A more time-efficient approach to solve this problem is to pre-process the long reference string into a data structure that allows for efficient search. The rest of this section discusses the two main data structures that have been proposed in the literature: suffix trees [Weiner 1973] and suffix arrays [Manber and Myers 1993].

3.5.2.1 Suffix Tree

A suffix tree (Figure 36) is a trie-like data structure that stores all the suffixes of a given string *S* (the reference string in the case of sequence alignment). Each suffix has exactly one path from the root of the tree to a leaf. The tree has *n* leaf nodes, corresponding to the *n* suffixes in *S*. Moreover, each edge in the tree is labeled with a substring of *S* such that the concatenation of the edge-labels from the root to a leaf represents a suffix S_j of *S*.

Search procedure and its complexity. Searching the suffix tree is done by navigating the tree starting from the root node, matching the characters of the query string with the edge-labels. The search complexity is O(*m*), where *m* is the length of the query string. This is an attractive linear-time search solution which does not depend on *n*, the length of the reference. Also, suffix trees can be augmented with additional pointers, called *suffix links* (shown as dashed arrows between internal nodes in Figure 36), which enable time-efficient maximal-matching (discussed below). Conceptually, a suffix link is an internal pointer from a node with path αw (i.e., the concatenation of edge-labels from the root to the node) to another node with path *w*, where α is a single character and *w* is a substring.

Processing the maximal-matches of a query *q* of length *m* requires searching the suffix tree for all subqueries q_0 to q_{m-l} (where *l* is the minimum-match length). This can be done by treating each subquery as a separate query, and performing a separate search operation for each one. However, this approach fails to take advantage of the fact that the problem searches for a group of related suffixes. To this end, suffix links allow for exploiting this opportunity: instead of traversing the

suffix tree from the root node for each subquery, the matching can be resumed for subquery q_i by following the suffix link of the last matching node of the previous subquery q_{i-1}, hence saving i-1 comparisons for each suffix, and rendering the complexity of matching *all* the subqueries of a query to be O(m).

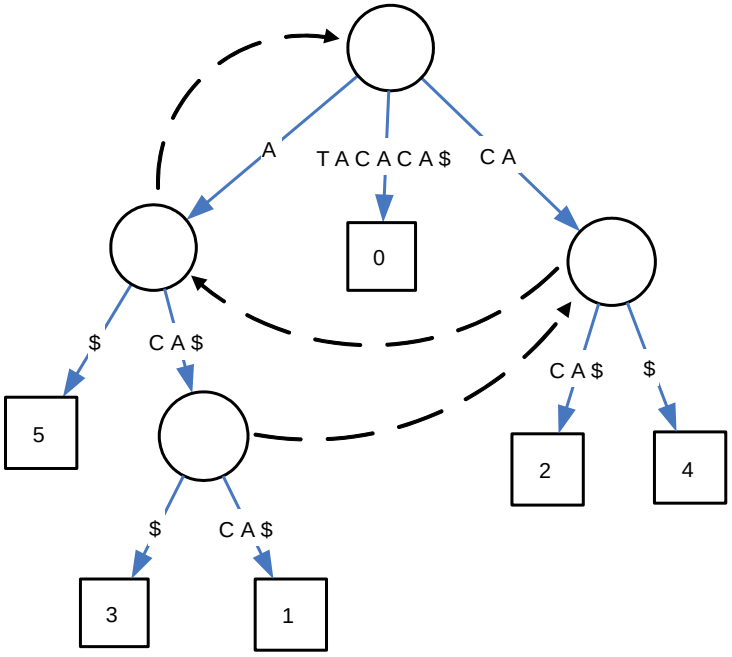

Figure 36: The suffix tree for the string TACACA. Dashed arrows represent suffix links.

Space complexity. The time efficiency of the suffix tree comes at the cost of additional computational and space overheads to build and store the suffix tree. Although the space complexity grows linearly with the reference sequence length as the tree requires only O(n) nodes, in practice the constant factors are high and suffix trees occupy a significant amount of space: between 22.4n and 32.7n Bytes for DNA sequences [Manber and Myers 1993; Abouelhoda et al. 2004; Kurtz 1999], where n is the sequence length. Storing the suffix links will require 4i additional Bytes, where i is the number of internal nodes. As a result, efforts have been made to reduce the space requirements of the tree, which resulted in reducing

the space requirement to 20*n* Bytes in the worst case [Kurtz 1999], without considering the suffix links.

Construction. The tree can be constructed in O(*n*) time [Weiner 1973], which in practice becomes negligible when matching a large number of queries. Further, suffix links are a by-product of suffix tree construction, hence no extra pre-processing time is required to produce them, yet they still consume additional space to store.

3.5.2.2 Suffix Array

To address the large space requirements of suffix trees Manber et al. [Manber and Myers 1993] proposed the suffix array, a data structure that enables similar string matching operations yet consuming less space in practice. A suffix array is a sorted array of all the suffixes of *S* in lexicographical order (presented in Table 6 for the same reference string as in Figure 36). The data structure is represented as an array of integers which correspond to the indices of the suffixes in order (column labeled 'suffix array' in Table 6).

Search procedure and its complexity. A naïve search in the suffix array takes O(*m*log *n*) time when supported by a classic binary search: O(*log n*) string comparisons demanded by the binary search, and each string-comparison requires O(*m*) character comparisons. In practice, however, a smart binary search implementation that takes advantage of the fact that the problem searches for related suffixes significantly improves the search time. Manber et al. [Manber and Myers 1993] proved that the *worst case* time complexity can be improved to O(*m* + log *n*) at the expense of increased space usage by associating the suffix array with an extra array of information, namely the longest common prefix (LCP) array: an array that stores the length of the longest common prefix between the suffix stored in the current entry and that stored in the previous array entry. Using the LCP array allows 'priming' the binary search: that is, the search does not start from scratch

for each string-comparison. In a nutshell, the results of earlier string-comparison iterations along with the LCP information are used to skip unnecessary comparisons in subsequent iterations.

Table 6: Suffix array for the string TACACA. The suffix and index columns are shown for illustration only (i.e., they do not present in the actual data structure). The *LCP array* represents the longest common prefix between the suffixes in the current and the previous array entry. The *rank array* represents the reverse index of the suffix array and has the same role as the suffix links in suffix trees: it is used to efficiently calculate maximal matches as discussed in Section 3.7.

Index	Suffix	Suffix Array	LCP Array	Rank Array (Suffix Array $^{-1}$)
0 (smallest)	A	5	0	5
1	ACA	3	1	2
2	ACACA	1	3	4
3	CA	4	0	1
4	CACA	2	2	3
5 (largest)	TACACA	0	0	0

Space complexity. The suffix array has O(*n*) entries, the same asymptotic space complexity as the suffix tree; in practice, however, it consumes three to five times less space than suffix trees [Manber and Myers 1993; Abouelhoda et al. 2004]. In particular, if an integer is represented by four Bytes, the array requires exactly *4n* Bytes. The LCP and the rank array (discussed in Section 3.7) add another 8*n* Bytes.

Construction. The suffix array can be constructed in linear time [Kärkkäinen et al. 2006; KIM et al. 2003; KO and ALURU 2003]. As with the suffix tree, construction overheads are amortized even for a relatively small number of queries.

3.6 Offloading Sequence Alignment

This section discusses the challenges to offload sequence alignment to the GPU (Section 3.6.1) and presents MUMmerGPU's approach to the problem based on suffix trees (Section 3.6.2).

3.6.1 Challenges

The efficient use of GPUs to speedup sequence alignment faces two main challenges:

- *Limited onboard GPU memory.* Current GPU models have one order of magnitude less memory compared to the host's main memory. This limitation may constrain applications to partition the problem space and perform computations in several rounds, hence adding significant data transfer overheads especially for data-intensive applications.

  The space requirement of the sequence alignment problem is fairly large, especially when considering long sequences such as those of mammalian genomes [Trapnell and Salzberg 2009]. For example, the human reference genome spans more than 3 billion DNA nucleotides (i.e., more than 3GB string) which, when processed into a suffix tree or suffix array, would require significantly more space (20x more, i.e., 60GB when using a suffix tree). Moreover, current sequencing projects typically produce more than 10x oversampling of the genome (i.e., the total length of all queries is 10x the length of the reference sequence) which needs to be aligned against the entire reference genome [Pop 2009]. As a result, the space requirements of the problem are at least one order of magnitude larger than the size of the onboard memory in current and near-future GPU models (for example, current high-end GPU models have up to 12GB of onboard memory).
- *Limited access to other I/O devices (e.g., disk).* As mentioned before, the GPU has access only to its onboard memory; hence results have to be stored internally then transferred to the host's main memory. As a result, GPU applications with a large output size must divide the limited onboard memory efficiently between the input and output buffers. This becomes a challenge when the result size cannot be determined in advance for a specific input size,

or the maximum result size is too large to be allocated. Addressing this limitation requires a compressed, deterministic representation of the results, which needs to be decompressed on the CPU (or possibly by another round on the GPU), consequently introducing extra overheads.

In the case of sequence alignment problem, the output size cannot be determined in advance as the number of alignments for each subquery is not known beforehand. Moreover, the maximum result size is O($mn|Q|$), which is infeasible to allocate.

3.6.2 A Previous Effort: MUMmerGPU

Delcher et al. [Delcher et al. 1999; Delcher 2002] implemented MUMmer, a widely used tool that performs sequence alignments on the CPU using suffix trees. The tool has also been significantly improved in terms of performance and space efficiency by Kurtz et al. [Kurtz et al. 2004]. Schatz et al. developed [Schatz et al. 2007] then optimized [Trapnell and Schatz 2009] a GPU version of the program, called MUMmerGPU, which also uses suffix trees. To address the space challenges of the problem (i.e., the long reference sequence, the large number of queries, the unpredictable result size, and the limited GPU memory), MUMmerGPU divides the computation into smaller-sized sub-computations that fit the GPU's memory. This is done by (*i*) dividing the long reference string into shorter overlapping *segments*, (*ii*) dividing the query set into smaller sized subsets, and (*iii*) reporting a "compressed" representation of the results to the host's memory. Figure 37 presents the high-level GPU offloading algorithm employed by MUMmerGPU.

```
refIndex = PreprocessReference(reference)
subsets = DivideQuerys(queries)
foreach subset in subsets do {
   results = NULL
   CopyIn(subset)
   foreach index in refIndex do {
      CopyIn(index)
      LaunchMatchKernel(subset, index)
      CopyOut(results) /* append result */
   }
   Postprocess(results)
}
```

Figure 37: High-level GPU offloading algorithm

MUMMERGPU constructs a suffix tree for each segment (a partition of the reference string), and aligns each query subset to all trees in rounds. Conceptually, a "round" is a four-stage process:

- *Copy in.* The query subset and the suffix tree of the segment are transferred to the GPU.
- *Matching*. The queries of a query subset are aligned to the tree in parallel on the GPU. All subqueries of a query are processed by a single GPU thread in order to take advantage of suffix links. To make the result size predictable, the match kernel does not report all the matches of each subquery (as discussed previously, a subquery could have one or more matches; however, the number of matches is not known in advance). Instead, the match kernel reports only the longest match of each subquery (node **Q** in Figure 38). This is done by matching the characters of the subquery string with the edge-labels until a mismatch or the end of the subquery is reached.
- *Copy out*. The results are transferred back to main memory.
- *Post-processing.* The results of the match kernel are "decompressed" to find the other matches of each subquery. This is done as follows (Figure 38 presents an example). First, starting from node **Q** that corresponds to the longest match for

a subquey, the algorithm traverses back to the node at which the match length equals the minimum-match length *l* (labeled **P** in Figure 38). Intuitively, **P** is the lowest common ancestor of the leaves that represent all subquery matches. Second, the algorithm performs a depth-first traversal to report all the leaves of the subtree rooted at **P** as the final result (i.e., the indices in the reference string where the subquery occurs).

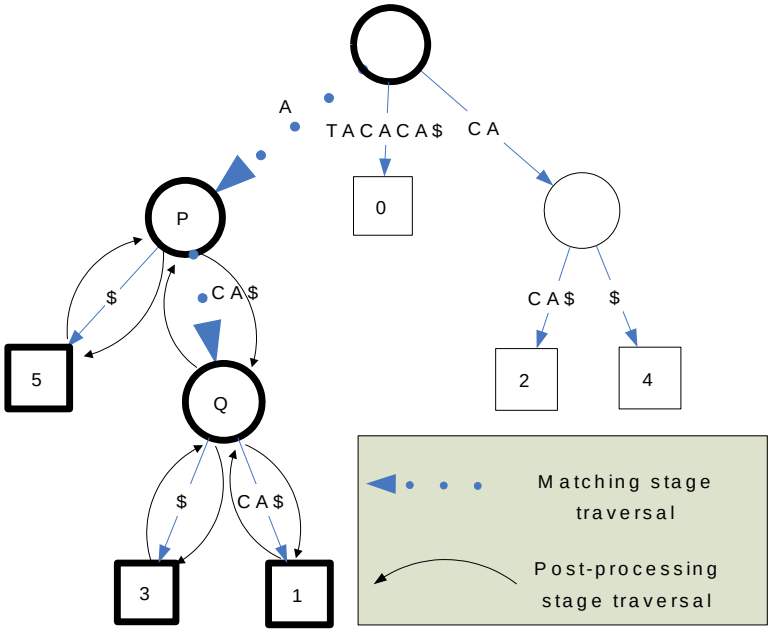

Figure 38: Alignment of query ACACT to reference TACACA for a minimum-match length of one. The figure demonstrates the alignment for only the first subquery (i.e., the string ACACT, itself). The dotted path is traversed in the matching stage. Node Q, and the corresponding maximum match length of 4, are reported as the result of the traversal in the matching stage. The post-processing stage produces the final output through a depth-first traversal starting from node P. The output includes three alignments: at position 5 with length 1, at position 3 with length 3 and at position 1 with length 4.

3.7 MUMmerGPU++

Schatz et al. report that MUMmerGPU achieves significant speedups compared to the original CPU-based MUMmer program [Schatz et al. 2007]. A closer look at the match between suffix tree-based search and the GPU characteristics prompted me to investigate whether a suffix array implementation can enable better utilization of the GPU. This section presents the suffix array-based algorithms used by MUMmerGPU++ while the following section estimates analytically the potential performance gains brought by this data structure.

At the high level, MUMmerGPU++ follows the same structure as MUMmerGPU (described in Figure 37). However, the core of MUMmerGPU++ is significantly different as it replaces the core data structure, the suffix tree, with a suffix array. This change entails completely different matching and post-processing algorithms, which the rest of this section describes.

Matching. Similar to MUMmerGPU, queries are searched in the suffix array in parallel, and all subqueries of a query are processed sequentially by a single GPU thread. For each subquery, the match kernel reports the index in the suffix array corresponding to the longest match in the reference. The matching algorithm processes a query q as follows: the first subquery q_0 is matched via a binary search on the suffix array, which, as discussed in Section 3.5.2.2, has $O(m + \log n)$ worst case complexity, where m is the query length and n is the reference string length. To process the next subquery and avoid processing the characters already processed by the previous subquery, a two-phase procedure is used (pseudocode presented in Figure 39):

```
/* Assumes SA, LCP and l global variables */
procedure Match(q, qlen) {
   i = 0
   while i ≤ qlen − l do {
      (si, ml) = BinarySearch(qi)
      RecordResult(qi, si, ml)
      i = i + 1
      while si != NULL and i ≤ qlen − l do {
       /* phase 1: cut the search space */
         i = i + 1
         s = ml − 1
         si = Rank[SA[si] + 1]
         j = SA[si] + s
         (r, ml) = Comp(Sj, qi+s)
       /* phase 2: find the longest */
         if r > 0 then {
            (si, ml) = ScanUp(s+ml, qi)
         } else {
            (si, ml) = ScanDown(s+ml, si, qi)
         }
         RecordResult(qi, si, ml)
         i = i + 1
      }
   }
}
procedure ScanUp(s, si, qi) {
   r = 1
   while LCP[si] > s and r > 0 do {
      si = si − 1
      j = SA[si] + s
      (r, ml) = Comp(Sj, qi+s)
      s = s + ml
   }
   return (si, s)
}
```

Figure 39: Pseudo-code of the core matching algorithm of MUMmerGPU++. The procedure "Match" is executed for each query by a dedicated GPU thread. The following is a summary of the variables names used: *i*=subquery index, *l*=minimum match length, *ml*=match length, *s*=skip (processed characters), *si*=suffix index. The procedure "Comp" evaluates which string is greater lexicographically and returns the maximum match length. Finally, the procedure "ScanDown" is similar to "ScanUP" but examines the entries in the other direction by incrementing the suffix index *si*.

- The first phase uses the result of the previous subquery to reduce the search space in the suffix array. This is done by combining the suffix array with

another one called the *rank array*: the reverse index of the suffix array (see Table 6).

For example, let S_j be the reference suffix that matched x characters of subquery q_i, where $x >= l$, also let k be the rank of S_j in the suffix array (i.e., SuffixArray[k] = j and Rank[j] = k); then the subquery q_{i+1} matches x - 1 characters of the reference suffix S_{j+1}, and Rank[j+1] is the corresponding suffix array index. Conceptually, the Rank array has the same role as the suffix links in suffix trees.

- The second phase searches for the longest match by sequentially comparing the subquery with the suffixes adjacent to the one produced by the first phase. The LCP array is used to avoid comparing a character more than once.

Note that if a subquery does not have a match in the reference string, the search for the next subquery falls back to the binary search procedure. Hence, the efficiency of this approach is related to the characteristics of the workload: the larger the number of matching subqueries, the lower the number of times the algorithm searches the whole array.

I anticipate that this approach is efficient for the sequence alignment problem since generally the queries are aligned to a reference genome of the same species; hence the percentage of positive matches is relatively high.

Post-processing. The result reported by the match stage represents the longest match occurrence for each subquery. Since the suffix array is ordered lexicographically, the other occurrences are adjacent: above and under the result reported by the match phase. Getting the other occurrences, and their maximum match length, is done via a simple sequential scan on the LCP array. The algorithm is presented in Figure 40.

```
/* Assumes SA, LCP and l global variables */
procedure PrintSubQueryAlignments(i, si, ml){
   /* print the longest one */
   PRINT(SA[si], i, ml)
   /* Scan up */
   v = si
   m = ml
   while v > 0 and m ≥ l do {
   /* the lcp could be longer than the
      match length, hence the minimum */
      m = MIN(m, LCP[v])
      v = v - 1
      PRINT(SA[v], i, m)
   }
   /* Scan down */
   v = si + 1
   m = MIN(ml, LCP[v])
   while v < reflen and m ≥ l do {
      PRINT(SA[si], i, ml)
      v = v + 1
      m = MIN(m, LCP[si])
   }
}
```

Figure 40: Pseudo-code of the core post-processing procedure. This procedure is invoked for each subquery in each query to decompress the result of the matching stage.

3.8 A Detailed Analysis of Space/Time Tradeoffs

This section uses simple complexity analysis to shed light on the effect of using suffix arrays instead of suffix trees on the running time of each of the matching, data transfer, and post-processing stages. In brief this section argues that even though suffix arrays may not enable a faster matching stage, they will enable significantly lower data transfer volumes and faster post-processing. These gains can be significant as these two stages consume a large share of the processing time (between 50% and 93% depending on the workload as seen in Figure 34). The evaluation using real workloads presented in Section 3.9 supports these conclusions.

3.8.1 The Matching Stage

Suffix trees and suffix arrays provide different trade-offs in search and space complexity which can be summarized as follows: on the one hand, suffix trees support O(m) search complexity, while suffix arrays support O(m + log n) to align a string of length m to a reference string of length n; on the other hand, although their asymptotic space complexity is similar, in practice suffix arrays are 3-5x more space efficient than suffix trees.

As mentioned before, tackling the constraints imposed by limited memory requires dividing the large query set into smaller subsets, and the long reference sequence into shorter segments.

The following notations are used to compare the time complexity of the matching stage for suffix trees and suffix arrays: let k be the number of query subsets and c_d be the number of segments the reference is divided into when using data structure d. Also, let t_d be the time complexity of matching a single query on the GPU. Finally, let α be the ratio between the number of queries in a query subset and the number of SIMD processors in the GPU.

Assuming that α does not depend on the data structure used leads to the implicit assumption that the size of a query subset is the same for both the array- and tree-based solutions, and that the space savings achieved in the suffix array-based solution will be used to increase the segment size (i.e., reduce the number of segments c_d).

Since processing all queries requires matching all query subsets to all reference segments, the time complexity of matching all queries using data structure d can be expressed as:

$$T_d = kc_d t_d \alpha$$

Suffix tree-based tool. As discussed in Section 3.5.2.1, suffix links enable O(m) search time for all subqueries (suffixes) of a single query. As a result, the time

complexity to search a query on the GPU using suffix trees can be expressed as: $t_{tree} = O(m)$. Thus, the time to process the query on all segments using suffix trees can be expressed as:

$$T_{tree} = kc_{tree} \alpha O(m)$$

Suffix array-based tool. In the case of suffix arrays, $t_{array} = O((m + \log(n / c_{array})) / r_{array})$, where r_{array} is the efficiency of calculating the subqueries of a query. Note that r_{array} is less than or equal to one: the value is close to one for workloads with high similarity with the reference, and lower values for workloads with lower similarity. Therefore, the overall time complexity when using suffix arrays:

$$T_{array} = kc_{array} \alpha O((m + \log(n / c_{array})) / r_{array})$$

Speedup. Based on the previous two equations, the speedup for the matching stage can be calculated as:

$$Speedup = \frac{T_{tree}}{T_{array}} = \frac{c_{tree}}{c_{array}} \times \frac{O(m)}{O((m + \log(n / c_{array})) / r_{array})}$$

Since the search procedures for both suffix array and suffix tree exhibit similar behavior: excessive memory accesses and byte-to-byte comparisons, the constants in the asymptotic bound of the search complexity for the suffix array and the suffix tree can be assumed to be close, hence the speedup ratio becomes:

$$Speedup = \frac{c_{tree}}{c_{array}} \times r_{array} \times \frac{m}{m + \log(n / c_{array})}$$

Next, the three terms that influence the speedup in the formula above are analyzed. First, from a practical view point, the query length *m* ranges from 35 to 700 depending on the sequencing technology used; while a reference segment length is up to hundreds of millions of nucleotides (leading to sizes in the order of

gigabytes limited by the available memory on the GPU), hence the term $\log(\ n / c_{array}\ )$ ranges from 20 to up to 30. As a result, the ratio $\frac{m}{m + \log(\ n / c_{array}\ )}$ is practically between 0.5, for short queries (small values of m), and 1.0 for long queries. Second, suffix arrays are more space efficient than suffix trees, with a space ratio c_{tree}/c_{array} greater than one, typically three. Finally, as mentioned before, the value r_{array} is less than one, and depends on the workload characteristics.

Summary. The main factors that affect the speedup ratio are: (*i*) the space ratio, which is typically three (*ii*) the query to segment length ratio, which is typically between 0.5 and 1.0, and (*iii*) the efficiency of calculating maximal matches in suffix arrays, which depends on the workload. In conclusion, a value of r_{array} larger than 50% makes the running time of the matching phase of a suffix array-based tool comparable with that of a suffix tree-based one, which I anticipate to be the case in realistic workloads as the query-set is aligned to a related reference sequence.

3.8.2 The Post Processing Stage

The post-processing stage decompresses the result of the matching stage, and writes the final results to the output file.

Suffix tree-based tool. In the MUMmerGPU case, the matching stage produces a single match for each subquery. Decompressing this into the final result is done via a depth-first traversal for each subquery as discussed in Section 3.6.2. This is an expensive pointer chasing procedure, especially when considering typical workloads with millions of queries.

To accelerate this stage, MUMmerGPU performs the decompression on the GPU using a second kernel. Therefore, the post-processing stage is executed as a three-stage GPU offloading process itself: (*i*) copy-in the information required to facilitate post-processing, (*ii*) launch the post-processing kernel which determines the matches for each subquery in parallel and (*iii*) copy-out the final results from

the GPU. Note that, due to the same reasons related to GPU memory limitations and the massive output size, offloading the post-processing stage is also done in rounds on the GPU. Finally, once transferred to main memory from the GPU, the results are written to the output file.

Two issues related to the above described GPU offloading process are worth mentioning. First, as mentioned before, it is essential to know the result size of a GPU kernel launch. Hence, in this case, the algorithm needs to know the number of matches for each subquery. MUMMERGPU addresses this is by storing additional information in the suffix tree: each node in the tree stores the number of leaves of the subtree rooted at that node. The post-processing stage is then performed in two phases: the first phase is processed on the CPU wherein, for each subquery, the algorithm traverses back to the node at which the match length equals the minimum-match length (labeled node P in Figure 38). The number of leaves stored in node P is in fact the number of matches for that subquery, and is used to allocate the required result space on the GPU. The second phase is performed on the GPU where the algorithm determines the matches through a depth-first traversal for each subqeury.

Second, MUMMERGPU designers adopted a stackless depth-first traversal algorithm as, on the GPU, a stackless tree traversal has been shown to be significantly more efficient than an approach that maintains a stack [Popov et al. 2007]. However, this improvement comes at the cost of, again, storing additional information in the tree: each node in the tree has to store a pointer to its parent node to facilitate this approach.

MUMMERGPU implementers report that offloading the post-processing stage to the GPU enabled a 4x speedup of this stage compared to performing it on the CPU [Trapnell and Schatz 2009]. However, as demonstrated in Figure 34, this stage is still time consuming: it occupies more than 20% of the total processing time. Note that this percentage represents only the post-processing GPU kernel time (i.e.,

copy-in and copy-out are considered as part of the data transfer overhead discussed in the next section) and writing the final result to the output file.

Suffix array-based tool. MUMMERGPU++ design places the entire post-processing stage on the CPU. As described in Section 3.7, the matching stage produces a suffix array entry index for each matching subquery. The *LCP* array is then used to determine all other alignments by directly scanning (practically just writing the results to the output file) the entries above and below the reported index with a minimum longest common prefix of *l*.

Summary. On the one hand, a suffix tree-based alignment tool requires costly additional traversal steps in the post-processing stage. MUMMERGPU offloads this stage to the GPU as a second processing round which, by itself, requires a post-processing phase that writes the final results to the output file. On the other hand, a suffix array based tool requires only a simple sequential scan to post-process the results. Hence, I expect the later approach to enable significant time savings for the post-processing stage.

3.8.3 The Data Transfer Stage

The GPU is connected to the host via an I/O bus. For a data-intensive application, data transfers represent a significant overhead. As Figure 34 shows, in the case of MUMMERGPU, this stage can take more than 20% of the total execution time.

The main advantage of suffix arrays over suffix trees is space efficiency. A suffix array typically enables three times better space efficiency compared to its suffix tree counterpart. As discussed previously, this space saving enables a suffix array-based alignment tool to divide the long reference sequence into a smaller number of segments, thus reducing the number of GPU execution rounds and the data transfer overhead associated with moving the query set to the GPU.

Additionally, note that offloading the post-processing stage to the GPU in the suffix tree-based approach entails extra data transfers, which I anticipate to be relatively significant especially when the number of positive matches is large.

3.9 Evaluation

This section presents an evaluation on two different hardware generations and workload sets. Section 3.9.1 details the experimental setup.

The initial evaluation was performed at the beginning of the project back in 2010: it was conducted on workloads and state-of-the-art hardware at that time. In particular, Section 3.9.2 presents an evaluation of the speedup delivered by MUMMERGPU++ compared to the most recent version of MUMMERGPU, while Section 3.9.3 investigates the factors that influence the observed performance and the effect of each processing stage on the total execution time.

Since significant changes happened in hardware and workloads since the initial evaluation, Section 3.9.4 revaluates the performance of MUMMERGPU++ compared MUMMERGPU on recent and larger workloads, and state-of-the-art hardware. The goal is to verify that the performance observed initially is maintained as workloads and hardware evolved.

Section 3.9.5 extends the evaluation to compare the performance of a hybrid system with a symmetric, CPU-only one. In particular, the section presents an evaluation of the idea of dividing the workload to be processed concurrently on both the GPU and the CPU while using on each processor the data structure that matches best its characteristics. Finally, Section 3.9.6 evaluates the energy footprint of processing sequence alignment on hybrid GPU-accelerated platforms.

3.9.1 Experimental Setup

The machine used to conduct the initial evaluation (Sections 3.9.2 and 3.9.3) has the following characteristics: Intel Core 2 Quad CPU (Q6700) clocked at 2.66 GHz

per core, 8GB of host memory, an NVIDIA GeForce 9800GX2 GPU: a dual-GPU card with 128 hardware threads clocked at 1.5 GHz and 1GB of memory. The GPU is connected to the host via a PCI Express 2.0 bus.

The evaluation was done under the real sequencing workloads introduced in Table 5. Unless otherwise mentioned, *config2* (see Table 5) is used as the default configuration for the minimum-match length in the experiments.

The extended evaluation (Sections 3.9.4, 3.9.5 and 3.9.6) was conducted on the machine described in Table 1. The new machine offers significant improvements in all aspects compared to the one used in the initial evaluation and described above: it includes dual socket Intel Sandy Bridge (Xeon 2650) with 16 hardware threads per socket clocked at 2.00 GHz, 256GB of host memory, and two NVIDIA GeForce GTX Titan GPUs each has 2688 hardware threads clocked at 800MHz and 6GB of memory. The GPUs are connected to the host via a PCI Express 3.0 bus.

It is important to note that MUMMERGPU++ implementation focuses on achieving a good match between the core data structure used and the GPU characteristics. To this end, the implementation is a 'common sense' one that does not aggressively optimize for caching, optimal use of shared memories, or coalesced data access – to enumerate only a few of the optimizations often used.

As a baseline for comparison the latest optimized version (v2.0) of the suffix tree-based MUMMERGPU is used. This version allows for seven data layout alternatives which determine: first, on which GPU memory type (i.e., global, texture, and constant memory) different parts of the input data (i.e., the reference string, suffix tree, and queries) are placed; and, second, how the suffix tree is stored in memory to enable maximum data access locality to improve cache hit rate when placed in texture memory. For MUMMERGPU, these choices resulted in a total of 128 different configuration combinations which impact the performance of the matching and post-processing stages. In their extensive analysis, Trapnell et al. [Trapnell and Schatz 2009] illustrated that the performance of different

configuration combinations is sensitive to the workload. However, they concluded that a single configuration provides reasonably good performance across all workloads. This configuration uses “a reordered one-dimensional texture for the suffix tree, global linear memory for the queries and reference”, and it is used to configure MUMMERGPU in all the experiments presented here. For a detailed discussion on these configurations, the reader is referred to [Trapnell and Schatz 2009].

As discussed before, due to the limited GPU onboard memory space, the workload is divided into smaller chunks by dividing the reference string into segments, and the query set into subsets processed in rounds. This raises the question on how to divide the onboard memory space between the queries and the reference in each round. Both MUMMERGPU and MUMMERGPU++ follow the same policy: maximize the segment size, while leaving space to accommodate enough queries to feed all cores on the device and extract maximum parallelism. Maximizing the segment size results in reducing the number of segments; this proportionally reduces the matching time as each query is processed fewer times.

For all experiments the time spent reading queries from the disk is excluded as this overhead is the same regardless of the used data structure and lies outside the optimization space of this work. Note that the disk I/O overhead represents 10% to 15% of the total MUMMERGPU execution time for the workloads used in the experiments.

Each experiment was run several times, and the execution time was stable in all experiments; hence, the plots show only averages (the variations in performance were too small to be visible on the graphs as 95% confidence intervals). Finally, the correctness of MUMMERGPU++ implementation was validated by comparing its output with the one produced by MUMMERGPU.

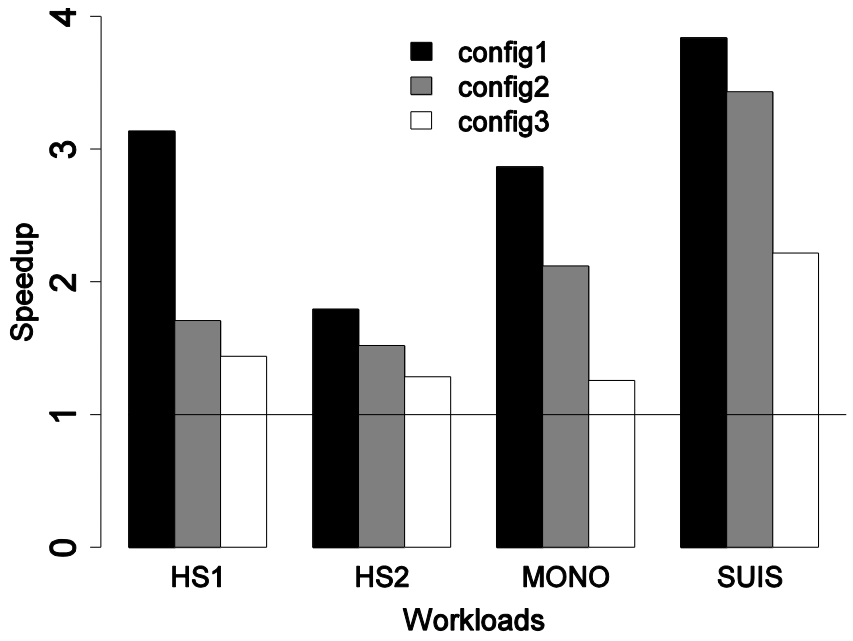

Figure 41: MUMMERGPU++ speedup compared to MUMMERGPU.

3.9.2 Overall Speedup

Figure 41 presents the speedup achieved by MUMMERGPU++ compared to MUMMERGPU for all configurations and workloads presented in Table 5.

While the speedup varies with the workload, MUMMERGPU++ performs better for all workloads: it delivers between 1.25x and 3.83x speedup compared to MUMMERGPU**.** This significant performance gain is achieved by a better match between the data structure used and the GPU's characteristics. MUMMERGPU++ achieves between 1.52x to 3.43x speedup for what is estimated to be the most frequently used configuration (*config2)*. The speedup is lower (between 1.25x and 2.21x) for configurations with a longer minimum-match length (*config3)*. This is because increasing the minimum-match length decreases the probability of finding matches, hence, as discussed previously, decreases the efficiency of subquery processing when using suffix arrays (represented by r_{array} in Section 3.8.1), and hurts the performance of the matching stage in MUMMERGPU++. Finally, as

expected for a short minimum-match length (*config1*), MUMMERGPU++ offers the best speedup: from 1.7x up to 3.83x.

3.9.3 Dissecting the Overheads

To validate our analysis in Section 3.7, better understand the source of the performance gains observed, and explore the opportunity for further performance tuning, this section explores the absolute and relative time spent in each processing stage.

Figure 42 compares the absolute time spent in each of the processing stages by both MUMMERGPU++ and MUMMERGPU for the largest workload: **HS1**. Note the following:

First, as discussed in Section 3.8.1, although the suffix tree-based tool, MUMMERGPU, has better asymptotic time complexity per query; MUMMERGPU++, the suffix array-based tool, achieves almost equal overall performance because it is more memory efficient and, as a result, requires a fewer matching rounds on the GPU when all queries are considered.

Second, although the post-processing stage in MUMMERGPU is performed on the GPU, the time spent in this stage is reduced by more than a factor of three by MUMMERGPU++, where it is performed on a single CPU core. This translates to 17% overall speedup improvement, hence supporting our argument in Section 3.8.2.

Third, the experiment validates our insight in Section 3.8.3 that a suffix array-based tool, like MUMMERGPU++, significantly reduces the data transfer overhead from/to the GPU: the total time spent transferring data is reduced by a factor of seven or more, which translates to more than 31% overall speedup improvement.

Finally, for both tools, the time spent in the construction stage is almost negligible compared to other stages.

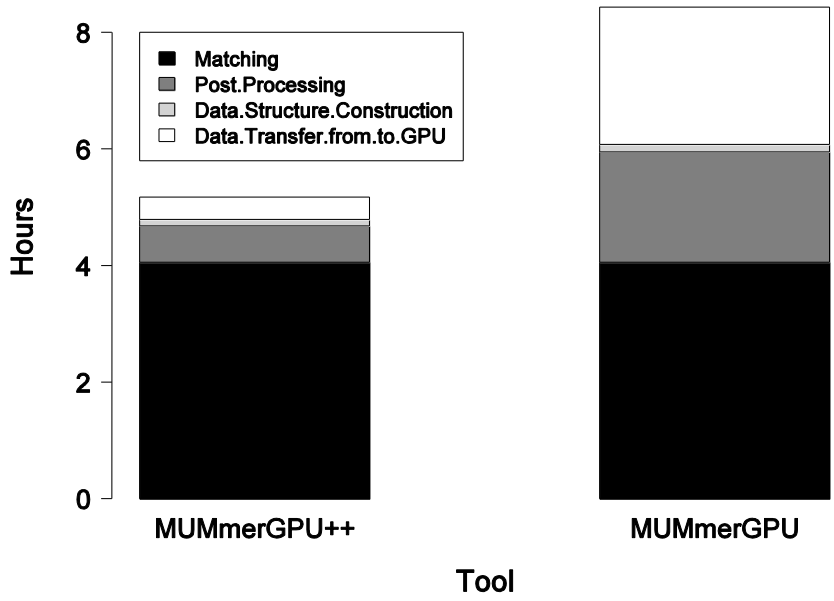

Figure 42: Absolute time spent in each processing stage for workload HS1 for both MUMMERGPU++ and MUMMERGPU (for the default configuration *config2*).

Figure 43 demonstrates the proportion of total processing time that corresponds to each processing stage for MUMMERGPU++ for all workloads. Compared to Figure 34, which presents similar data for MUMMERGPU, MUMMERGPU++ significantly changes the distribution of processing effort across stages. It significantly reduces the share of post-processing and data-transfer stages, and increases the share of the matching stage.

This is important from two perspectives. First, these tools are expected to run on multi-GPU systems. From this perspective, the intense data-transfers employed by MUMMERGPU make the PCI Express bus a bottleneck and limit the feasibility of using multiple GPUs on the same host. MUMMERGPU++ reduces the I/O overhead (by a factor of 6x-12x in our experiments) and thus eliminates the shared communication (PCI Express bus) as a potential scalability bottleneck. Second, from a performance optimization perspective, the fact that the compute (matching)

stage now takes 75%-80% of the time for the large workloads, including the human genome, allow focusing the performance optimizations on this stage only.

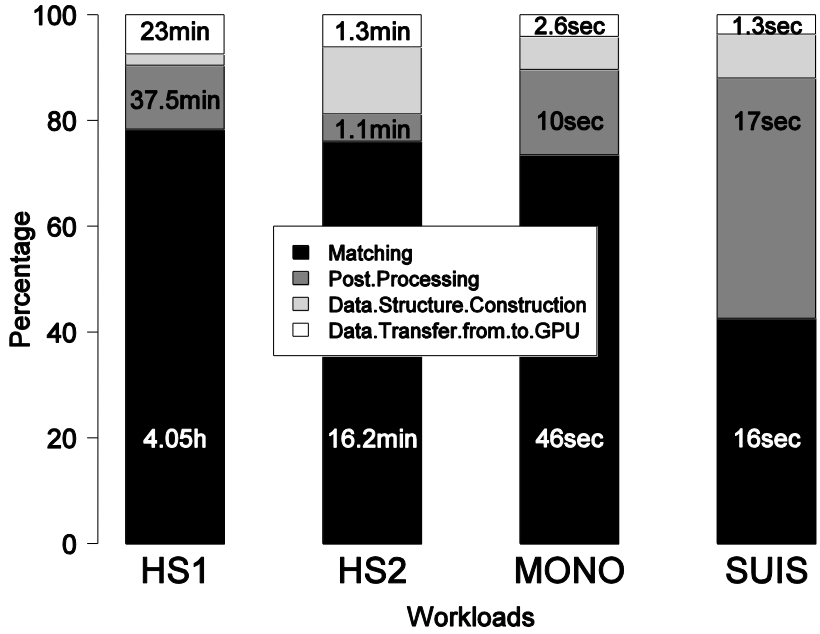

Figure 43: Percentage of total execution time spent in each processing stage for MUMMERGPU++. The numbers on the bars show the absolute time spent in each stage.

3.9.4 Evaluation on Newer Hardware Platform and Workloads

This section aims to answer the following question: *Is the performance observed initially preserved as workloads and hardware evolve?* To answer this question, this section presents a comparison of MUMMERGPU++ (the suffix array-based tool) and MUMMERGPU (the suffix tree-based tool) on a newer hardware platform (Table 1 lists its characteristics).

Particularly, compared to the platform used in the initial experiments, the GPUs in the new platform has six times more onboard memory space, four times more memory bandwidth and the GPUs are connected via a PCI Express 3.0 bus, which offers double the CPU-GPU transfer bandwidth. Note that the new GPUs bring important improvements to the core bottlenecks of the suffix tree-based tool.

Table 7: A newer set of sequence alignment workloads used in the extended evaluation study. Compared to the previous set of workloads, the focus here is on longer reference sequences. The workloads were obtained from the NCBI archive [NCBI 2014].

Workload / Species	Reference Sequence Length	Number of Queries	Sequencing Technology (Read Length)
DMEL / Drosophila Melanogaster (Fruit Fly)	122,696,361	69,580,156	Illumina (151)
HS - Homo Sapiens chromosome 2 (Human)	238,202,930	205,904,051	Illumina (74)
ORYZA - Oryza Sativa (Rice)	361,636,301	27,280,835	Illumina (76)

Moreover, the updated evaluation presented in this section (and the following sections) uses more recent workloads (Table 7). Compared to the first set of workloads, the new set is focused on larger workloads: longer reference sequences and larger number of queries.

Figure 44 (left) compares the performance of the two tools on the newer hardware platform and the updated workloads. The figure confirms the ability of the suffix array-based tool (MUMMERGPU++) to deliver better performance than the suffix tree-based one (MUMMERGPU) when run on the GPU. Even though the new hardware offers more memory space on the GPU and higher data transfer bandwidth, the suffix tree based tool still suffers from high communication overhead, especially as the reference sequence becomes longer: when run on the GPU, the suffix array delivers 1.33x speedup on the shortest sequence (**DMEL**), 1.79x on the medium sequence (**HS**), and 2.15x on the longest one (**ORYZA**).

Figure 44 (right) also compares the processing rate of two CPU-only solutions: a suffix tree and a suffix array-based ones. The suffix tree implementation is the original MUMMER tool [Kurtz et al. 2004] (modified to process queries in parallel using OpenMP), while the CPU-based suffix array tool is a modified version of MUMMERGPU++ that runs on the CPU in parallel using OpenMP.

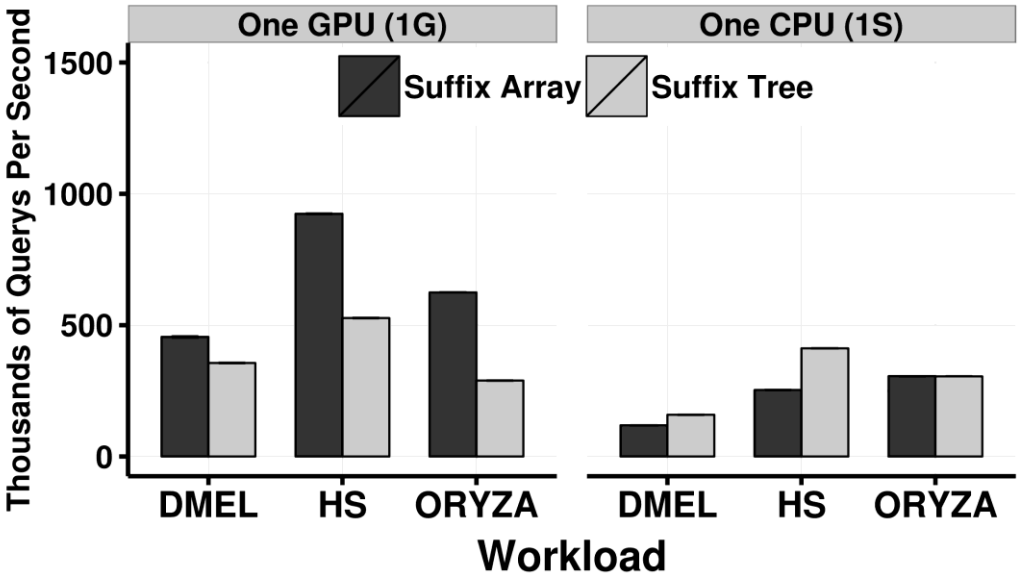

Figure 44: Comparison of the processing rates of the two data structures. *Left:* when processing is offloaded to a single GPU. *Right:* when processing is not offloaded to the GPU and is performed entirely on a single CPU socket. The processing rate is calculated as the number of queries divided by the processing time. The minimum match length is fixed at 40 for all experiments. The CPU and the GPU versions of the suffix tree tools are the original MUMMER (which I modified to process queries in parallel using OpenMP) and its GPU port, MUMMERGPU, respectively. The GPU version of the suffix array tool is the MUMMERGPU++ presented in Section 3.7, while the parallel CPU-based one is a modified version of MUMMERGPU++ that runs on the CPU and parallelized using OpenMP. The notation 1G refers to processing the workload on a single GPU, while 1S refers to processing the workload only on the CPU, and on one of the two CPU sockets (i.e., using the 16 hardware threads of one of the two CPU processors available on the machine).

Unlike when offloading to the GPU, the suffix tree offers better performance than the suffix array. This is because, when processing on the CPU, there are no communication overheads, and the processing time is exclusively spent on the matching phase. Since the suffix tree has better time complexity in the matching phase than its suffix array counterpart, the former solution outperforms the latter.

3.9.5 Hybrid Processing of Sequence Alignment

One way to maximize the utilization of a hybrid platform's processors for sequence alignment is to partition the workload to be processed concurrently on both the CPU and the GPU. This section presents the performance of such a solution: dividing the queries between the CPU and the GPU while using for each processor a data structure that matches best its characteristics: a suffix array for the GPU partition and a suffix tree for the CPU one.

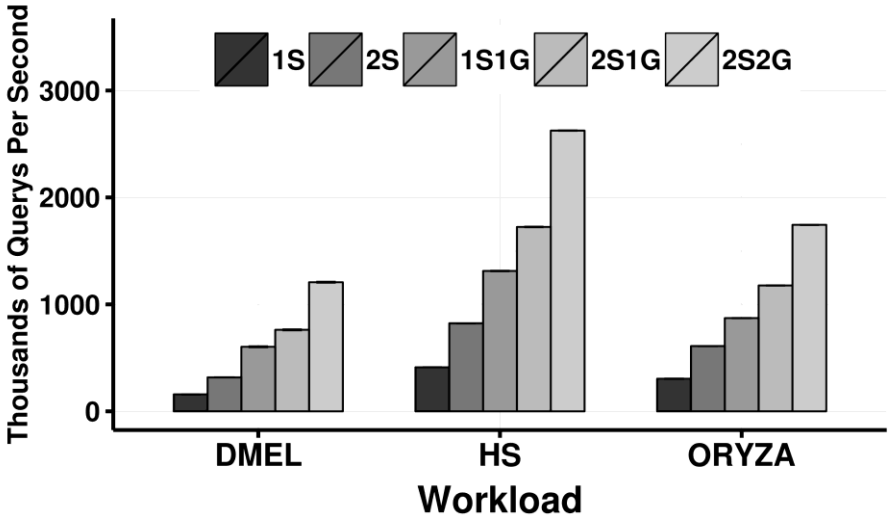

Figure 45: Performance of different hybrid configurations and workloads.

Figure 45 shows the performance of different hybrid configurations. First, the discussion focuses on the analysis of configurations with two processing units. The figures show that, for all algorithms, the hybrid system (1S1G) performs better than the dual-socket system (2S). On the one hand, adding a second socket doubles the the number of memory channels, which are critical resources for sequence alignment performance, hence leading to close to double the performance compared to 1S configuration. On the other hand, the performance gain of 1S1G, brought by matching the core data structure with the hybrid system, outperforms that of the

dual-socket symmetric system: between 1.40x to 2x speedup compared to the dual socket system (2S).

Second, the figure also demonstrates the ability of the hybrid system to harness extra processing elements. The sequence alignment workload is easy to partition, by dividing the queries among the processing elements, and hence the performance scales seamlessly as more processors are added.

3.9.6 Power and Energy Evaluation

This section evaluates the power and energy characteristics of sequence alignment on hybrid (i.e., CPU and GPU) single-node systems. Section 2.12.1 presents the evaluation setup and the power characteristics of the machine.

While section 2.12.1 discusses in more detail the evaluation setup, two main points are worth stressing: (*i*) a significant share of the power is consumed by DRAM, and (*ii*) when loaded, GPUs consume significant power compared to other system components.

The rest of this section evaluates energy efficiency via three metrics: power consumption in Watts, power-normalized processing rate and the energy-delay product.

Power Consumption

Figure 46 shows the system power consumption under different workload and hardware combinations. To better illustrate the variation in power consumption during execution, the data is presented as boxplots. The main differentiating factor in terms of power consumption is the hardware configuration (i.e., the number and type of processing elements used). Note that there is no major power difference across workloads for the same hardware configuration.

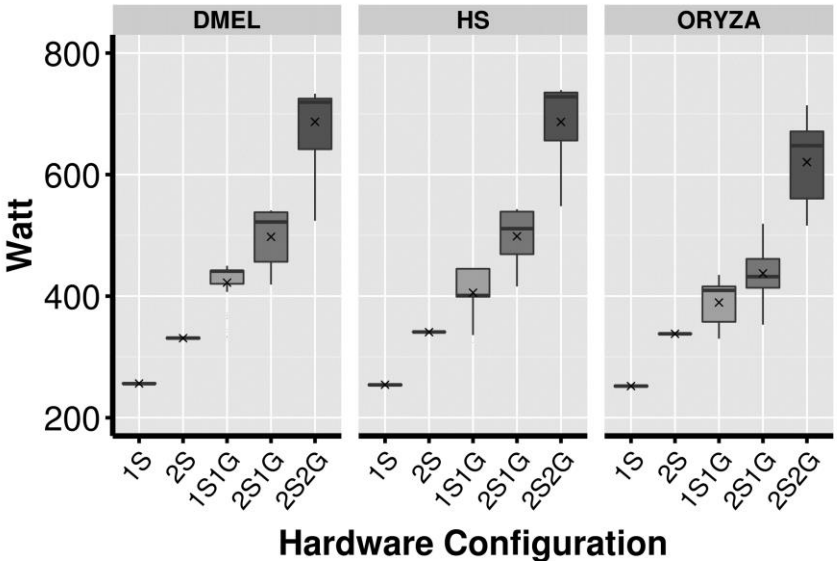

Figure 46: Power consumption (the lower the better) for different hardware configurations and workloads. The upper and lower "hinges" of the boxplot correspond to the first and third quartiles. The middle line corresponds to the median. The whiskers extend from the lowest data point within 1.5 IQR of the lower quartile, to the highest data point within 1.5 IQR of the upper quartile (IQR is the Interquartile Range, which is the distance between the first and third quartiles). The mean is shown as a cross. Note the y-axis starts at 200W.

Note that the increase in power consumption when processing on two CPU processors (2S) compared to one (1S) is about 75W, which is 78% of the CPU's TDP. Similarly, the increase in power drawn when adding GPUs is close to the TDP of the GPU: adding a GPU increases power consumption by about 200W, which is 80% of the GPU's TDP (see Table 1).

Since the workload is balanced between the processing elements, adding a processing element (a CPU or a GPU) entails an increase in power consumption that is close to the processor's TDP.

Power-normalized Processing Rate

To estimate the energy efficiency of different configurations, Figure 47 shows the power-normalized performance for all benchmarks (i.e., raw performance reported in Figure 45 divided by average drawn power). Note that, for each workload, the plot can also be viewed as a comparison of raw energy consumed to process the graph.

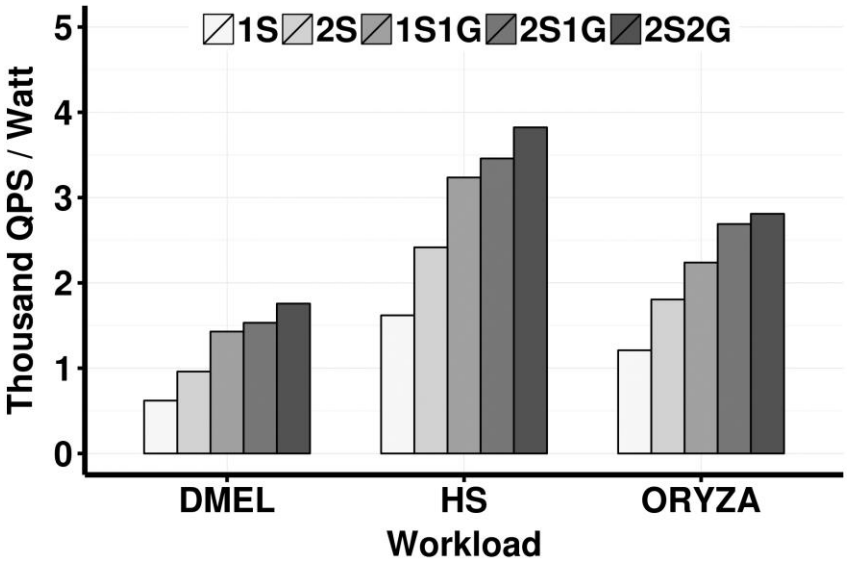

Figure 47: Power-normalized processing rate (the higher the better). QPS refers to queries per second.

First, compare the configurations with two processing elements. The performance gains that the hybrid 1S1G system achieves compared to a 2S system do not come at the expense of energy inefficiency, in fact the hybrid system is more energy efficient than its symmetric counterpart. In the best case, *the hybrid system achieves 40% higher efficiency* for the power-normalized performance metric.

Second, in all cases, adding more GPUs improves power-normalized performance as the gain in raw performance is higher than the increase in power consumption.

Energy-delay Product

Since the energy-delay product (EDP) is an energy metric that is biased for low-time-to-solution, the relative advantage of the hybrid solution is higher. Figure 48 presents the results of this experiment *normalized* to the 1S configuration to make the plot readable.

The figure shows that, in the best case, a hybrid 1S1G can be 2.8x better than a symmetric configuration with the same number of processors (2S). Equally important, EDP continues to improve as more are added, especially when a second GPU is added.

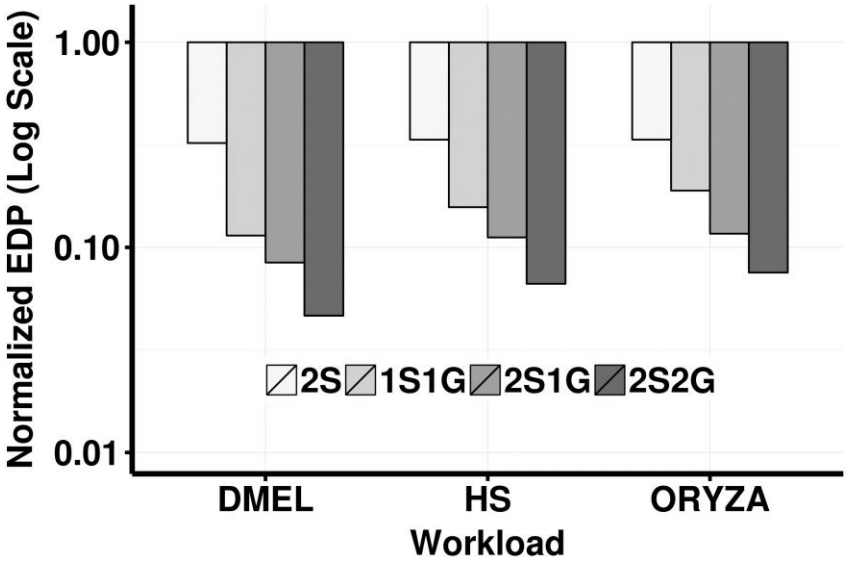

Figure 48: Normalized energy-delay product (the lower the better). Note that the y-axis is log-scale. The baseline is the CPU-only configuration with one processor (1S).

In summary, compared to a symmetric configuration, a hybrid GPU-accelerated platform offers not only better time-to-solution, but also saves in the amount of energy consumed. Although a GPU draws almost double the power of a CPU, energy savings were possible because of the significant performance improvement brought by the hybrid system through better data structure-processor matching.

3.10 Lessons and Discussion

The results presented in this chapter allow putting forward a number of guidelines on the opportunity and the supporting techniques required to efficiently harness hybrid systems for sequence alignment. These guidelines are phrased as answers to a number of questions.

- ***Q1:*** *Is it beneficial to use a hybrid system to accelerate sequence alignment?*

  A1: Yes. One concern when considering using a hybrid system is the limited GPU memory that may render using a GPU ineffective when processing large sequence alignment workloads. This work shows, however, that trading-off higher computational complexity for a more compact in-memory representation significantly improves the accelerator's performance (by enabling higher parallelism levels and reducing transfer overheads).

- ***Q2:*** *Can the data transfer overheads be hidden by overlapping the transfers with the GPU kernel execution?*

  A2: No, especially for large-scale workloads. The reason is that the computation on the GPU requires a set of input/output buffers. Facilitating communication-computation overlap requires double buffering for the input and output (such that the GPU computes on one set of buffers while the transfers are concurrently performed to/from the others). This entails allocating two sets of input/output buffers on a scarce resource: GPU's onboard memory.

  To further investigate this opportunity, I ran an experiment (for both MUMmerGPU and MUMmerGPU++) in which the tool assumed half of the memory available on the device to simulate a double buffering condition. The results demonstrated that the increase in the time spent in the matching stage was larger than the total time spent transferring data from/to the GPU (and could potentially be hidden by the overlapping technique mentioned). Hence, for this application, overlapping would actually hurt performance.

- ***Q3:*** *Is it fair to use MUMMERGPU as a baseline to evaluate the advantages of the suffix array-based approach? Otherwise said, is it possible that the speedup offered by MUMMERGPU++ is simply due to a better optimized GPU implementation and not to the choice of a data-structure that inherently offers a better fit to for the computing platform at hand?*

  A3: I have three arguments to support the choice of MUMMERGPU as the reference for a suffix tree-based tool. First, the analysis of the opportunities a suffix array-based implementation offers (Section 3.8.1) is solely based on the characteristics of the core data structure, and is agnostic to the detailed GPU implementation of the tool. Second, MUMMERGPU is a well optimized GPU-based tool. The tool's authors exhaustively examined 128 data layout configurations to select the configuration which delivers the best overall performance. The results were presented in two previous publications [Schatz et al. 2007; Trapnell and Schatz 2009]. Finally, as mentioned in (Section 3.9.1) I have not specifically optimized MUMMERGPU++: apart from placing the reference string in texture memory, the kernel places all input and output data in global memory, it does not employ the shared memory available on each multiprocessor and does not try to improve memory throughput by coalescing memory accesses.

- ***Q4:*** *Is there an energy cost to the time-to-solution gains provided by the hybrid platform?*

  A4: No. One concern is that the GPU's high peak power consumption may render an accelerated solution inefficient in terms of energy. The experiments in section 3.9.6 reject this concern: GPU-acceleration allows a faster 'race-to-idle', enabling energy savings that are sizeable for newer GPU models which are power-efficient in idle state (as low as 25W [NVIDIA 2013]). The

experiments show that a hybrid system is not only efficient in terms of time-to-solution, but also in terms of energy and energy-delay product.

- ***Q5:*** *Since sequence-alignment is a memory-bound operation, why not lower the processor's frequency to lower the energy footprint?*

  A5: On the CPU side, a recent analysis [Schöne et al. 2012] shows that, for recent Intel processors (e.g., Intel's Sandy Bridge), both memory latency and bandwidth strongly depend on the processor's frequency (this result is confirmed on the platform used in this study). This limits the opportunity to use dynamic voltage and frequency scaling (DVFS) to save energy on the CPU side. On the GPU side, however, recent GPU models support setting different frequencies for the memory and the compute cores. Previous work [Jiao et al. 2010; Abe et al. 2012] shows that energy consumption can be reduced by lowering the core frequency for memory-intensive kernels, an opportunity that will further improve the energy efficiency of the hybrid system.

- ***Q6:*** *Is it possible that the results presented in this dissertation are dependent on the hardware platform and workloads used for experimentation?*

  A6: Unlikely. The evaluation presented here was performed on two generations of hardware and workloads. The older machine dates back to 2010 when this work was first conducted. The new machine includes state-of-the-art CPU and GPU models as of writing this dissertation. Interestingly, the assumptions regarding the differences in characteristics between the two types of processing elements of the hybrid system stayed the same: GPUs still have significantly higher peak memory access bandwidth and higher peak computational power per Byte of memory compared to CPUs, yet one order of magnitude lower internal memory space. More importantly, the results obtained on the older generation of hardware and workloads (sections 3.9.2 and 3.9.3) are consistent

with the latest results obtained on the newer set of workloads and hardware generation (section 3.9.4 and 3.9.5).

Chapter 4

Summary and Impact

Current computing platforms for processing large-scale irregular datasets (such as processing graphs with billions of edges), have in common that they store and process data in their aggregate memory. These platforms, however, differ over a number of key choices and can succinctly be characterized by describing the extremes of a price/time-to-solution spectrum. At the one end of this spectrum stand platforms that emphasize a low time-to-solution, e.g., supercomputers such as Cray XMT. Hardware-supported shared memory and fast interconnects offer low latency for non-local memory accesses, while the massive parallelism provided by such platforms helps further hide memory access latency. These platforms, however, are costly to build and operate as they use non-commodity components.

At the other end of the spectrum, shared-nothing architectures (e.g., commodity clusters) are commonly used as low-cost alternatives to support large data-intensive processing. Compared to supercomputers, clusters are cheaper and arguably easier to extend incrementally. However, due to higher interconnect latency, lack of hardware support for shared memory, and inability to mask memory access latency (e.g., by using a massive number of threads), these platforms are typically less efficient than their supercomputer counterparts.

This dissertation starts from the observation that today more efficient solutions are affordable: it is feasible to assemble single-node platforms that aggregate 100s of gigabytes to terabytes of memory and immense processing power (using massively-parallel accelerators such as GPUs) [Gupta et al. 2013; Rowstron et al. 2012; Shun and Blelloch 2013] all from commodity components. Compared to the two extreme platforms discussed above (i.e., supercomputers and clusters), a hybrid

GPU-accelerated platform with significant amount of host memory can be obtained for a relatively low budget, and has the potential to offer significant performance and energy efficiency for a large class of applications.

However, realizing these opportunities in the context of irregular applications is not a trivial task. While pervious works demonstrated that significant gains can be obtained for regular applications (such as linear algebra), little experience has been accumulated to date related to using hybrid GPU-accelerated platforms to improve the performance of irregular applications, particularly the ones that process massive datasets. Indeed, the GPU's strict parallel model and limited onboard memory, among other challenging characteristics, makes it unclear if it is beneficial to offload part of the massive workload of an irregular application to be processed concurrently on the GPU.

This dissertation attempts to bridge this gap by exploring the opportunities, design methodologies and middleware to improve the efficiency and, at the same time, reduce the complexity of harnessing hybrid GPU-accelerated platforms to improve the performance of large-scale irregular applications.

Using two high-impact applications, this work provides evidence that hybrid GPU-accelerated platforms improve the performance, in terms of both time-to-solution and energy, of large-scale irregular applications. To reach this point, this work offers performance models, low-cost workload assignment strategies, design techniques and optimizations customized for processing on hybrid platforms. Equally important, the ideas presented in this work have been extensively evaluated on large-scale synthetic and real-world workloads. The evaluation examines two important metrics, time-to-solution and energy, and carefully explains the reasons for obtained performance.

The rest of this section summarizes the impact and possible extensions of the two main lines of research large-scale graph processing (Section 4.1) and large-scale sequence alignment (Section 4.2).

4.1 Large-Scale Graph Processing on Hybrid Platforms

This work demonstrates the ability to harness hybrid platforms for graph processing and proposes techniques that allows it to perform efficiently. Specifically, this work shows that GPU-acceleration improves both time-to-solution as well as energy efficiency, and that this improvement scales when increasing the graph size and adding more GPUs. Further, although current GPU models have one order of magnitude less memory, and are connected to high-latency communication bus, this work shows that a hybrid, one CPU and one GPU, system can be more efficient in terms of time-of-solution and energy efficiency than a dual-CPU symmetric one.

4.1.1 Impact

In addition to the research contributions detailed in Section 2.2, this project had the following impact:

- First, this project highlights the significant change in computing systems design, which enables building commodity single-node machines with significant amounts of memory and computer power, and allows for processing large-scale workloads that until recently were only processed on clusters or supercomputers. To the best of my knowledge, this is the first work to process graphs as large as one billion vertices and 16 billion edges on a single node machine. Based on this, system designers have a wider design spectrum to consider when designing infrastructure for large-scale graph processing.
- Second, contrary to the belief that GPU acceleration is only viable for regular computations, this project confirms the viability of using GPUs to accelerate a challenging irregular problem: graph processing.
- Third, to the best of my knowledge, this is the first work to explore the idea of using both CPUs and GPUs to concurrently process large-scale graphs: graphs that are an order of magnitude larger than what a GPU memory can host. A number of techniques made this possible. Some of these techniques were

proposed by this work, such as partitioning strategies that aim to reshape the workload to run faster on the bottleneck processor. Other techniques were imported from works on distributed systems design, and this work showed their applicability in the context of single-node hybrid platforms, such as the bulk synchronous parallel (BSP) computation model and message aggregation [Malewicz et al. 2010].

- Finally, this work resulted in an open source software artifact, TOTEM: a programming framework that makes it easier to implement graph algorithms for hybrid GPU-accelerated platforms.

4.1.2 Possible Extensions

In the following, I summarize two possible directions to extend this work.

A more detailed performance model

This work presented a simple, yet effective performance model that helps estimating the benefits of offloading part of the graph workload. Given the current characteristics of hybrid platforms, the model shows that it is beneficial to partition the graph workload and process it on a hybrid platform, and highlights the importance of minimizing the communication overhead to improve the overall performance.

Notwithstanding the model's simplicity and its demonstrated usefulness, it can be improved (at the cost of making it more complex). For example, the model assumes that the CPU's processing rate is constant, determined by a benchmark independent of the graph characteristics of the actual workload. A more accurate modeling would take into account the characteristics of the partition. To address this issue, one could analyze the effect of workload characteristics (e.g., degree distribution and graph structure) on obtained performance. For example, one could perform controlled experiments on diverse hardware while varying graph

characteristics, and feed the results to a machine learning approach to better predict the processing rate for partitions with specific characteristics.

Most importantly, considering the hardware characteristics as parameters in this machine learning methodology has the potential to predict what is more beneficial, adding more CPU sockets or accelerators, given a workload pattern and energy or dollar budget; hence providing valuable information needed for efficient system provisioning.

Another possible extension to the model is to take into consideration that part of the graph may reside on non-volatile memory such as SSDs, which have higher access latency than DRAM, but higher storage capacity and are more energy efficient.

A graph processing engine for distributed hybrid platforms

This work presented the design and implementation of TOTEM, a graph processing engine for hybrid single-node platforms. TOTEM's importance, however, comes not only from enabling harnessing single-nodes, but also as a building block to harness GPU-accelerated clusters which have become common in the HPC space. For instance, four of the first five supercomputers in the latest (June, 2014) Top500[12] supercomputer list host accelerators and heterogeneous architectures are increasingly popular [TITAN 2013].

In this context, one possible extension to TOTEM is to harness GPU-accelerated clusters. Shared-nothing architectures that aggregate heterogeneous nodes, that is, clusters of GPU-accelerated nodes, can offer a cost-efficient, yet high performance graph processing platform. The fact that new commodity nodes can support multi-hundred gigabytes of memory space, offers the opportunity to aggregate large memory space using smaller number of components; therefore, reducing inter-node

[12] www.top500.org

communication cost. At the same time, adding GPUs to each node offsets the loss in parallelism resulted from reducing the number of nodes.

Furthermore, TOTEM can be used as a back-end module of a domain specific language (DSL) for graph processing. For example, it can be used to extend the DSL developed by Hong et al. for graph analysis which currently targets only symmetric shared-memory platforms [Hong et al. 2012].

4.2 Large-Scale Sequence Alignment on Hybrid Platforms

GPUs have drastically different performance characteristics compared to traditional multicore architectures: up to one order of magnitude higher peak memory access bandwidth, one order of magnitude higher peak computational power per Byte of memory, yet one order of magnitude lower internal memory space.

This work argues that these differences make reconsidering the choice of the data structures used a necessary step when porting applications to hybrid, GPU-accelerated platforms. In particular, the experience from this project is synthesized as three guidelines. First, a solution that supports minimum computational overhead does not necessarily enable maximum overall performance: a better optimization point is one that maintains a balance between communication and computation overheads. Second, GPUs' high computational power per Byte of memory compared to traditional multiprocessor architectures, makes trading-off additional per thread processing time for a more compact in-memory data representation an attractive technique to increase overall performance (by enabling higher parallelism levels and reducing data transfer overheads). Finally, ensuring that the chosen GPU-offloaded part of the application entails low pre- and post-processing overheads is essential to maximize the overall performance gains.

4.2.1 Impact

In addition to the research contributions detailed in section 3.4, this project had the following impact:

- First, this project highlights the significant difference in the characteristics of two commodity processors: GPUs and traditional CPUs. More importantly, it stresses the value of space-time tradeoffs to improve the performance of GPGPU applications. These ideas inspired and used by other related works, such as [Drozd et al. 2012] which proposes to build a GPU-accelerated sequence alignment solution that is based on pre-processing the reference string into an index based on Burrows-Wheeler transform, which has even lower memory footprint than the suffix array, but higher computational complexity.
- Second, similar to the impact that the TOTEM project had (section 4.1.1), this project confirms the viability of using GPUs to accelerate a challenging irregular problem, sequence alignment. This is contrary to the belief that GPU acceleration is only viable for regular computations.
- Third, this work resulted in an open-source software artifact, MUMMERGPU++, which has been used as a benchmark by several studies related to improving GPU architecture and design [Fung and Aamodt 2011; Rhu and Erez 2012; Lashgar et al. 2012; Blem et al. 2011; ElTantawy et al. 2014]; moreover, as of writing this thesis, MuMmerGPU++ is part of the NVIDIA bioinformatics benchmark.

4.2.2 Possible Extensions

While this work is focused on discrete GPUs, an interesting extension is to explore the effect of the techniques proposed here on performance when using integrated GPUs.

The goal of integrated GPUs is to remove the PCI Express bus by placing the main processor and the accelerator on the same die and share the same memory

space. AMD's APU (Accelerated Processing Unit) with its Fusion architecture [Branover et al. 2012] is an example of such hybrid setup.

While current APU models do place the main processor and the accelerator on the same die, they still employ distinct memory partitions, and hence the techniques proposed in this work still apply for the current APU generations.

Moreover, even though current APU models are almost an order of magnitude less compute powerful and have a lower memory bandwidth than discrete GPUs, recent works show that for communication-intensive applications, APUs can be competitive with their discrete counterparts [Hetherington et al. 2012; Calandra et al. 2013].

4.3 Limitations

The limitations of this work can be summarized as follows.

First, this work focuses on irregular workloads. For example, in the graph processing project, the work targets graphs with power-law degree distribution. This is because efficiently utilizing hybrid platforms requires heterogeneity in the workload that allows for using the different types of processing elements for different parts of the workload. Therefore, regular workloads, such as grid grids, may not benefit from the ideas proposed in this work that are related to hardware specialization. However, such workloads may still obtain improvement that is linear with the size of workload offloaded to the GPU (i.e., similar to random partitioning for irregular workloads).

Second, this work focuses on single-node platforms. While this is clearly an advantage for a wide range of workloads, massive-scale workloads (e.g., Google-scale web crawling workload), do not benefit. A multi-node setup has different overheads compared to single-node one. For example, inter-node communication can be a major overhead that may influence the way the graph is partitioned

between nodes, and it is not clear how that affects partitioning between the CPU and the GPU within the node.

Third, hybrid platforms increase the complexity of software development. While this work offers frameworks to maximize the utilization of hybrid platforms while hiding some of the development complexity, it is still more complex to develop for such platforms compared to shared memory ones. For example, using TOTEM, the developer still needs to implement two kernels, one for the CPU and one for the GPU; while this can be viewed as an opportunity to optimize them differently based on the characteristics of each processor, it is an extra effort that the developer needs to put. Emerging technologies, such as OpenACC which enables having a single kernel implementation and integrated GPUs which makes it easier to manage CPU-GPU communication, may help reduce development complexity; however, a software system that manages data placement (i.e., define and assign partitions to different processing elements) will still be needed, and hence it will always be more complex to develop for a hybrid platform compared to a symmetric one.

Bibliography

ABE, Y., SASAKI, H., PERES, M., INOUE, K., MURAKAMI, K., AND KATO, S. 2012. Power and performance analysis of GPU-accelerated systems. *HotPower*.

ABECASIS, G.R., AUTON, A., BROOKS, L.D., ET AL. 2012. An Integrated Map of Genetic Variation from 1,092 Human Genomes. *Nature 491*, 7422, 56–65.

ABOUELHODA, M.I., KURTZ, S., AND OHLEBUSCH, E. 2004. Replacing Suffix Trees With Enhanced Suffix Arrays. *Journal of Discrete Algorithms 2*, 1, 53–86.

AGARWAL, V., PETRINI, F., PASETTO, D., AND BADER, D.A. 2010. Scalable Graph Exploration on Multicore Processors. *The International Conference for High Performance Computing, Networking, Storage, and Analysis*.

AHN, Y.-Y., HAN, S., KWAK, H., MOON, S., AND JEONG, H. 2007. Analysis of topological characteristics of huge online social networking services. *Proceedings of the 16th international conference on World Wide Web - WWW '07*, ACM Press, 835.

BARABÁSI, A.-L. 2003. *Linked: How Everything Is Connected to Everything Else and What It Means.* Plume.

BARABÁSI, A.-L., ALBERT, R., AND JEONG, H. 2000. Scale-Free Characteristics of Random Networks: the Topology of the World-Wide Web. *Physica A: Statistical Mechanics and its Applications 281*, 1-4, 69–77.

BARRETT, R., BERRY, M., CHAN, T.F., ET AL. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM.

BARROSO, L.A., DEAN, J., AND HOLZLE, U. 2003. Web Search for a Planet: the Google Cluster Architecture. *IEEE Micro 23*, 2, 22–28.

BEAMER, S., ASANOVIĆ, K., AND PATTERSON, D. 2013. Direction-optimizing breadth-first search. *Scientific Programming 21*, 3, 137–148.

BELLMAN, R. 1958. On a Routing Problem. *Quarterly of Applied Mathematics 16*, 87–90.

BLEM, E., SINCLAIR, M., AND SANKARALINGAM, K. 2011. Challenge Benchmarks that must be Conquered to Sustain the GPU Revolution. *Proceedings of the 4th Workshop on Emerging Applications for Manycore Architecture (EAMA)*.

BOLDI, P., SANTINI, M., AND VIGNA, S. 2008. A Large Time-Aware Web Graph. *ACM SIGIR Forum 42*, 2, 33–38.

BRANDES, U. 2001. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology 25*, 2, 163–177.

BRANOVER, A., FOLEY, D., AND STEINMAN, M. 2012. AMD Fusion APU: Llano. *IEEE Micro 32*, 2, 28–37.

CALANDRA, H., DOLBEAU, R., FORTIN, P., LAMOTTE, J.-L., AND SAID, I. 2013. Evaluation of Successive CPUs/APUs/GPUs Based on an OpenCL Finite

Difference Stencil. *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 405–409.

CATANZARO, B., FOX, A., KEUTZER, K., ET AL. 2010. Ubiquitous Parallel Computing from Berkeley, Illinois, and Stanford. *IEEE Micro 30*, 2, 41–55.

CHA, M., HADDADI, H., BENEVENUTO, F., AND GUMMADI, P.K. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy. *International AAAI Conference on Weblogs and Social Media* .

CHAKRABARTI, D., ZHAN, Y., AND FALOUTSOS, C. 2004. R-MAT: A Recursive Model for Graph Mining. *SIAM International Conference on Data Mining*.

CHAMBERLAIN, B.L. 1998. *Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations.* .

CHEN, T., RAGHAVAN, R., DALE, J.N., AND IWATA, E. 2007. Cell Broadband Engine Architecture and its First Implementation - A Performance View. *IBM Journal of Research and Development 51*, 5, 559–572.

CHHUGANI, J., SATISH, N., KIM, C., SEWALL, J., AND DUBEY, P. 2012. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. *International Parallel and Distributed Processing Symposium*.

CURTISS, M., BECKER, I., BOSMAN, T., ET AL. 2013. Unicorn: A System for Searching the Social Graph. *Proceedings of the VLDB Endowment 6*, 11, 1150–1161.

DELCHER, A.L. 2002. Fast Algorithms for Large-Scale Genome Alignment and Comparison. *Nucleic Acids Research 30*, 11, 2478–2483.

DELCHER, A.L., KASIF, S., FLEISCHMANN, R.D., PETERSON, J., WHITE, O., AND SALZBERG, S.L. 1999. Alignment of Whole Genomes. *Nucleic Acids Research 27*, 11, 2369–2376.

DROZD, A., MARUYAMA, N., AND MATSUOKA, S. 2012. Sequence Alignment on Massively Parallel Heterogeneous Systems. *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IEEE, 2498–2501.

EDIGER, D., MCCOLL, R., RIEDY, J., AND BADER, D.A. 2012. STINGER: High Performance Data Structure for Streaming Graphs. *High Performance Extreme Computing*.

ELTANTAWY, A., MA, J.W., O'CONNOR, M., AND AAMODT, T.M. 2014. A Scalable Multi-Path Microarchitecture for Efficient GPU Control Flow. *The International Symposium on High-Performance Computer Architecture (HPCA)*.

ERDŐS, P. AND RÉNYI, A. 1960. On the Evolution of Random Graphs. *Publications of tke Matkemafical Insfifufe of the Hungarian Academy of Sciences 5*.

FALOUTSOS, M., FALOUTSOS, P., AND FALOUTSOS, C. 1999. On Power-Law Relationships of the Internet Topology. *ACM SIGCOMM Computer Communication Review 29*, 4, 251–262.

FELDMANN, A. 2012. Fast Balanced Partitioning Is Hard Even on Grids and Trees. In: B. Rovan, V. Sassone and P. Widmayer, eds., *Mathematical Foundations of Computer Science 2012*. Springer Berlin / Heidelberg, 372–382.

FORD, L.A. 1956. Network Flow Theory. *Report P-923*.

FUNG, W.W.L. AND AAMODT, T.M. 2011. Thread Block Compaction for Efficient SIMT Control Flow. *International Symposium on High Performance Computer Architecture (HPCA)*.

GAREY, M.R., JOHNSON, D.S., AND STOCKMEYER, L. 1974. Some Simplified NP-Complete Problems. *Symposium on the Theory of Computing*.

GELADO, I., CABEZAS, J., NAVARRO, N., STONE, J.E., PATEL, S., AND HWU, W.W. 2010. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. *ACM SIGPLAN Notices 45*, 3, 347.

GHARAIBEH, A., BELTRÃO COSTA, L., SANTOS-NETO, E., AND RIPEANU, M. 2012. A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing. *International Conference on Parallel Architectures and Compilation Techniques*.

GHARAIBEH, A., COSTA, L.B., SANTOS-NETO, E., AND RIPEANU, M. 2013a. On Graphs, GPUs, and Blind Dating: A Workload to Processor Matchmaking Quest. *International Parallel and Distributed Processing Symposium*.

GHARAIBEH, A., SANTOS-NETO, E., BELTRÃO COSTA, L., AND RIPEANU, M. 2013b. The Energy Case for Graph Processing on Hybrid CPU and GPU Systems. *Workshop on Irregular Applications: Architectures and Algorithm*.

GONZALEZ, J.E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. *Symposium on Operating Systems Design and Implementation*.

GUPTA, P., GOEL, A., LIN, J., SHARMA, A., WANG, D., AND ZADEH, R. 2013. WTF: The Who to Follow Service at Twitter. *International World Wide Web Conference* .

HARISH, P., NARAYANAN, P., ALURU, S., PARASHAR, M., BADRINATH, R., AND PRASANNA, V. 2007. Accelerating Large Graph Algorithms on the GPU Using CUDA. *HiPC*.

HETHERINGTON, T.H., ROGERS, T.G., HSU, L., O'CONNOR, M., AND AAMODT, T.M. 2012. Characterizing and Evaluating a Key-Value Store Application on Heterogeneous CPU-GPU Systems. *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, IEEE, 88–98.

HILL, M.D. AND MARTY, M.R. 2008. Amdahl's Law in the Multicore Era. *Computer 41*, 7, 33–38.

HONG, S., CHAFI, H., SEDLAR, E., AND OLUKOTUN, K. 2012. Green-Marl: A DSL for Easy and Efficient Graph Analysis. *ASPLOS*.

HONG, S., KIM, S.K., OGUNTEBI, T., AND OLUKOTUN, K. 2011a. Accelerating CUDA Graph Algorithms at Maximum Warp. *Symposium on Principles and Practice of Parallel Programming*.

HONG, S., OGUNTEBI, T., AND OLUKOTUN, K. 2011b. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. *International Conference on Parallel Architectures and Compilation Techniques*.

HUANG, S., XIAO, S., AND FENG, W. 2009. On the Energy Efficiency of Graphics Processing Units for Scientific Computing. *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*.

HWU, W.W. 2011. *GPU Computing Gems Jade Edition.* Morgan Kaufmann Publishers Inc.

IORI, G., DE MASI, G., PRECUP, O.V., GABBI, G., AND CALDARELLI, G. 2008. A Network Analysis of the Italian Overnight Money Market. *Journal of Economic Dynamics and Control 32*, 1, 259–278.

JEONG, H., MASON, S.P., BARABÁSI, A.L., AND OLTVAI, Z.N. 2001. Lethality and Centrality in Protein Networks. *Nature 411*, 6833, 41–2.

JIAO, Y., LIN, H., BALAJI, P., AND FENG, W. 2010. Power and Performance Characterization of Computational Kernels on the GPU. *GREENCOM*.

JOHNSON, D., JOHNSON, M., KELM, J., TUOHY, W., LUMETTA, S., AND PATEL, S. 2011. Rigel: A 1,024-Core Single-Chip Accelerator Architecture. *IEEE Micro 31*, 4, 30–41.

KAISER, J. 2008. A Plan to Capture Human Diversity in 1000 Genomes. *Science 319*, 5862, 395.

KÄRKKÄINEN, J., SANDERS, P., AND BURKHARDT, S. 2006. Linear Work Suffix Array Construction. *Journal of the ACM 53*, 6, 918–936.

KARYPIS, G. AND KUMAR, V. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing 20*, 1.

KATZ, G.J. AND KIDER JR, J.T. 2008. All-Pairs Shortest-Paths for Large Graphs on the GPU. *Symposium on Graphics Hardware*.

KERNIGHAN, B. 1970. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal 49*, 1, 291 – 307.

KIM, D.K., SIM, J.S., PARK, H., AND PARK, K. 2003. Linear-Time Construction of Suffix Arrays. *Lecture notes in computer science 2676*, 186–199.

KIRK, D.B. AND HWU, W.W. 2010. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann Publishers Inc.

KISS, C. AND BICHLER, M. 2008. Identification of Influencers — Measuring Influence in Customer Networks. *Decision Support Systems 46*, 1, 233–253.

KO, P. AND ALURU, S. 2003. Space Efficient Linear Time Construction of Suffix Arrays. *Lecture notes in computer science 2676*, 200–210.

KURTZ, S. 1999. Reducing the Space Requirement of Suffix Trees. *Software: Practice and Experience 29*, 13, 1149–1171.

KURTZ, S., PHILLIPPY, A., DELCHER, A.L., ET AL. 2004. Versatile and Open Software for Comparing Large Genomes. *Genome biology 5*, 2, R12.

KWAK, H., LEE, C., PARK, H., AND MOON, S. 2010. What is Twitter, a Social Network or a News Media? *International World Wide Web Conference*.

KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. *OSDI*.

LASHGAR, A., BANIASADI, A., AND KHONSARI, A. 2012. Dynamic Warp Resizing: Analysis and Benefits in High-performance SIMT. *2012 IEEE 30th International Conference on Computer Design (ICCD)*, IEEE, 502–503.

LEE RODGERS, J. AND NICEWANDER, W.A. 1988. Thirteen Ways to Look at the Correlation Coefficient. *The American Statistician 42*, 1, 59–66.

LEE, S. AND VETTER, J.S. 2014. OpenARC: Open Accelerator Research Compiler for Directive-based, Efficient Heterogeneous Computing. *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ACM, 115–120.

LEE, V.W., HAMMARLUND, P., SINGHAL, R., ET AL. 2010. Debunking the 100X GPU vs. CPU myth. *ACM SIGARCH Computer Architecture News 38*, 3, 451.

LI, D. AND BECCHI, M. 2013. Deploying Graph Algorithms on GPUs: An Adaptive Solution. *International Parallel and Distributed Processing Symposium*.

LI, H. AND HOMER, N. 2010. A Survey of Sequence Alignment Algorithms for Next-Generation Sequencing. *Briefings in bioinformatics 11*, 5, 473–83.

MALEWICZ, G., AUSTERN, M.H., BIK, A.J.C., ET AL. 2010. Pregel: A System for Large-Scale Graph Processing. *SIGMOD International Conference on Management of data* .

MANBER, U. AND MYERS, G. 1993. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing 22*, 5, 935–948.

MCPHERSON, J.D. 2009. Next-Generation Gap. *Nature Methods 6*, 11 Suppl, S2–5.

MCVOY, L. AND STAELIN, C. 1996. lmbench: Portable Tools for Performance Analysis. *USENIX Annual Technical Conference*.

MERRILL, D., MICHAEL, G., AND GRIMSHAW, A. 2012. Scalable GPU Graph Traversal. *Symposium on Principles and Practice of Parallel Programming*.

MITTAL, S. AND VETTER, J.S. 2014. A Survey of Methods For Analyzing and Improving GPU Energy Efficiency. *CoRR abs/1404.4*.

NAGARAJAN, N. AND POP, M. 2013. Sequence Assembly Demystified. *Nature reviews. Genetics 14*, 3, 157–67.

NCBI. 2014. http://www.ncbi.nlm.nih.gov/. .

NGUYEN, D., LENHARTH, A., AND PINGALI, K. 2013. A Lightweight Infrastructure for Graph Analytics. *Symposium on Operating Systems Principles*.

NVIDIA. 2012. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110.* .

NVIDIA. 2013. TESLA K20 GPU Active Accelerator Board Specification. .

OPENACC. 2012. OpenACC: Directives for Accelerators. http://www.openacc-standard.org/.

PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* .

PINEDO, M.L. 2012. *Scheduling: Theory, Algorithms, and Systems.* Springer Verlag.

PINGALI, K., KULKARNI, M., NGUYEN, D., ET AL. 2009. Amorphous Data-Parallelism in Irregular Algorithms. *Department of Computer Science, The University of Texas at Austin, Tech. Rep. TR-09-05*.

POP, M. 2004. Comparative Genome Assembly. *Briefings in Bioinformatics 5*, 3, 237–248.

POP, M. 2009. Genome Assembly Reborn: Recent Computational Challenges. *Briefings in bioinformatics 10*, 4, 354–66.

POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum 26*, 3, 415–424.

RHU, M. AND EREZ, M. 2012. CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures. *ACM SIGARCH Computer Architecture News 40*, 3, 61.

ROWSTRON, A., NARAYANAN, D., DONNELLY, A., O'SHEA, G., AND DOUGLAS, A. 2012. Nobody Ever Got Fired for Using Hadoop on a Cluster. *International Workshop on Hot Topics in Cloud Data Processing*.

SALLINEN, S., BORGES, D., GHARAIBEH, A., AND RIPEANU, M. 2014. Exploring Hybrid Hardware and Data Placement Strategies for the Graph 500 Challenge. *SC*.

SCHATZ, M.C., TRAPNELL, C., DELCHER, A.L., AND VARSHNEY, A. 2007. High-Throughput Sequence Alignment Using Graphics Processing Units. *BMC bioinformatics 8*, 474.

SCHÖNE, R., HACKENBERG, D., AND MOLKA, D. 2012. Memory Performance at Reduced CPU Clock Speeds: An Analysis of Current x86 Processors. *HotPower*.

SHUN, J. AND BLELLOCH, G.E. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *Symposium on Principles and Practice of Parallel Programming*.

STEYVERS, M. AND TENENBAUM, J.B. 2005. The large-scale structure of semantic networks: statistical analyses and a model of semantic growth. *Cognitive science 29*, 1, 41–78.

TIAN, Y., BALMIN, A., CORSTEN, S.A., TATIKONDA, S., AND MCPHERSON, J. 2013. From "think like a vertex" to "think like a graph." *Proceedings of the VLDB Endowment 7*, 3.

TITAN. 2013. *TITAN: Paving the Way to Exascale.* .

TRAPNELL, C. AND SALZBERG, S.L. 2009. How to Map Billions of Short Reads onto Genomes. *Nature biotechnology 27*, 5, 455–457.

TRAPNELL, C. AND SCHATZ, M.C. 2009. Optimizing Data Intensive GPGPU Computations for DNA Sequence Alignment. *Parallel Computing 35*, 8, 429–440.

DE VALCK, K., VAN BRUGGEN, G.H., AND WIERENGA, B. 2009. Virtual Communities: A Marketing Perspective. *Decision Support Systems 47*, 3, 185–203.

VALIANT, L.G. 1990. A Bridging Model for Parallel Computation. *Communications of the ACM 33*, 8, 103–111.

VENTER, J.C. 2010. Multiple Personal Genomes Await. *Nature 464*, 7289, 676–7.

VUDUC, R., CHANDRAMOWLISHWARAN, A., CHOI, J., GUNEY, M., AND SHRINGARPURE, A. 2010. On the Limits of GPU Acceleration. *Hot Topics in Parallelism (HotPar)*, USENIX Association.

WANG, R., CONRAD, C., AND SHAH, S. 2013. Using Set Cover to Optimize a Large-Scale Low Latency Distributed Graph. *Workshop on Hot Topics in Cloud Computing*.

WANG, X.F. AND CHEN, G. 2003. Complex networks: Small-World, Scale-Free and Beyond. *IEEE Circuits and Systems Magazine 3*, 1, 6–20.

WARD, R.M., SCHMIEDER, R., HIGHNAM, G., AND MITTELMAN, D. 2013. Big Data Challenges and Opportunities in High-Throughput Sequencing. *Systems Biomedicine 1*, 1, 29–34.

WEINER, P. 1973. Linear Pattern Matching Algorithms. *14th Annual Symposium on Switching and Automata Theory (SWAT)*, IEEE.

Appendices

Appendix A: A BFS Implementation on Top of TOTEM

This appendix details a simplified implementation of the BFS algorithm using TOTEM. The code is thoroughly commented, and hence relatively long. The best way to read the code is to start from the **main** function, which can be found at the end of the appendix.

```
// A structure that encapsulates per-partition
// algorithm-specific state.
typedef struct {
  level_t*   levels;     // One slot per vertex in the partition.
  bool*      finished;   // Refers to Totem's finish flag.
  level_t    cur_level;  // The current level being processed by
                         // the partition.
} bfs_local_state_t;

// A structure that encapsulates algorithm-specific global state,
// which is shared between all partitions.
typedef struct {
  level_t* levels;  // The final output buffer.
  vid_t    source;  // The source vertex id.
} bfs_global_state_t;
static bfs_global_state_t state_g = {0};

// A helper function that is used by the CPU and GPU compute
// functions to process a vertex. The function iterates over the
// vertex's neighbors, and sets their level if it has not been set
// before. The function returns false when at least one neighbor
// has been updated indicating that processing has not finished
// yet, which is eventually translated to an additional BSP round.
// The function returns true when no neighbors have been updated,
// which translates to termination in case the function returns
// true for all processed vertices.
static __device__ __host__
bool bfs_process_vertex(partition_t* par, bfs_state_t* state,
                        vid_t v) {
  bool finished = true;
  if (v >= par->subgraph.vertex_count ||
      state->levels[v] != state->cur_level) { return finished; }
  for (eid_t i = par->subgraph.vertices[v];
       i < par->subgraph.vertices[v + 1]; i++) {
    const vid_t nbr = par->subgraph.edges[i];
```

```
    // The following Totem function returns a reference to the state
    // of the neighbor. If the neighbor is in the same partition,
    // the function returns a reference to the neighbor's state in
    // the local "state->levels" array. If the neighbor is remote,
    // the function returns a reference to its state in the outbox
    // buffer.
    level_t* nbr_level = totem_engine_get_dst_ptr(par, nbr,
                                                  state->levels);

    // Update the neighbor's level if it has not been set before.
    // Note that reduction for remote neighbors happens implicitly
    // here: all vertices in this partition that has an edge to
    // this remote neighbor would test and update the same state
    // which exist as part of the outbox buffer. During the
    // communication phase, a single value will be communicated to
    // the partition that owns the neighbor.
    if (*nbr_level == INF_LEVEL) {
      finished = false;
      *nbr_level = state->cur_level + 1;
    }
  }
  return finished;
}

// The CPU compute kernel which is called by the compute callback
// if the partition is CPU resident.
static void bfs_compute_cpu(partition_t* par,
                            bfs_state_t* state) {
  const graph_t* subgraph = &par->subgraph;
  bool finished = true;
  #pragma omp parallel for schedule(runtime) reduction(&: finished)
  for (vid_t v = 0; v < subgraph->vertex_count; v++) {
    finished &= process_vertex(par, state, v);
  }
  if (!finished) { *(state->finished) = false; }
}

// The GPU compute kernel, which is called by the compute callback
// if the partition is CPU resident.
static __global__
void bfs_gpu_kernel(partition_t par, bfs_state_t state) {
  const vid_t v = THREAD_GLOBAL_INDEX;
  if (!process_vertex(&par, &state, v)) {
    // state.finished is a reference to a flag that is shared
    // between all partitions. Totem sets this flag to true at the
    // beginning of each superstep. A partition sets this flag to
    // false if there are active vertices that needs to be processed
    // in the next round. Totem will launch another BSP round if
    // any partition sets this flag to false.
    *(state.finished) = false;
```

```
  }
}

// A wrapper for the GPU compute kernel, it configures and launches
// the CUDA kernel.
static void bfs_compute_gpu(partition_t* par,
                            bfs_local_state_t* state) {
  dim3 blocks, threads;
  totem_kernel_configure(par->subgraph.vertex_count, &blocks,
                         &threads);
  bfs_gpu_kernel<<<blocks, threads, 0, par->stream>>>(*par,
                                                      *state);
}

// The compute callback function. Totem calls this function for
// each partition as part of the BSP compute phase. Depending on
// the partition's processor, this function calls either the CPU or
// the GPU kernel.
static void bfs_compute(partition_t* par) {
  bfs_local_state_t* state = (bfs_local_state_t*)par->algo_state;
  if (par->processor.type == PROCESSOR_CPU) {
    compute_cpu(par, state);
  } else if (par->processor.type == PROCESSOR_GPU) {
    compute_gpu(par, state);
  }
  state->cur_level++;
}

// The callback to "scatter" the messages received from remote
// partitions to the partition's local state. Totem invokes this
// callback at the end of the communication phase after the data
// has been copied from the outbox buffers of the remote partitions
// to the inbox buffers of this partition.
static void bfs_scatter(partition_t* par) {
  bfs_local_state_t* state = (bfs_local_state_t*)par->algo_state;
  // For each message in the inbox buffer, the following template
  // function computes the minimum of the value in the message and
  // the one the vertex currently have in the local state (i.e.,
  // state->levels). The minimum is then stored in the local state
  // as the vertex's new level.
  totem_engine_scatter_inbox_min(par->id, state->levels);
}

// Callback to collect the final result from the partitions' local
// "levels" array to the final output array that will be returned
// to the user.
static void bfs_collect(partition_t* par) {
  bfs_local_state_t* state    =
     (bfs_local_state_t*)par->algo_state;
  // The following Totem function copies each value in the local
```

```
  // state->levels array to its corresponding entry in the final
  // state_g.levels array. To do this, the function uses a "map"
  // that maps each vertex in the partition from its local id space
  // (the vertex id within the partition which is used to index the
  // local "state->levels" array) to its global id space (the vertex
  // id in the original graph which is used to index the final
  // output array "state_g.levels").
  totem_engine_collect(par->id, state->levels, state_g->levels);
}

// Callback to allocate and initialize a "bfs_local_state_t"
// structure, a per-partition and algorithm-specific state. This is
// called for each partition by Totem at the beginning before the
// first BSP superstep.
static void bfs_init(partition_t* par) {
  // Removed for brevity. In summary, the function allocates a
  // bfs_local_state_t structure for this partition.
  // "par->alg_state" is the reference to the allocated structure.
  // It also initializes the allocated local state, such as setting
  // the level of the source vertex to 0 (if it belongs to this
  // partition).
}

// Callback to free the buffers allocated in initialize. This is
// called by Totem at the end (i.e., after all partitions vote for
// termination).
static void bfs_finalize(partition_t* par) {
  // Removed for brevity.
}

// The hybrid BFS algorithm entry function. Given a graph and a
// source vertex, the algorithm computes the distance (named level)
// of every vertex from the source.
void bfs_simplified_hybrid(graph_t* graph, vid_t source,
                           level_t* levels) {
  // Initialize the global state.
  totem_memset(levels, INF_LEVEL, totem_engine_vertex_count(),
               TOTEM_MEM_HOST);
  state_g.levels = levels;
  state_g.source  = source;

  // Configure and trigger Totem's BSP engine. TOTEM_COMM_PUSH
  // indicates that the communication direction is from the source
  // to the destination vertex of a remote edge, this is in contrast
  // to TOTEM_COMM_PULL which indicates the opposite. The former is
  // used by algorithms in which a vertex pushes a value to update
  // its neighbors (such as BFS), while the latter is used in
  // algorithms where a vertex pulls the state of its neighbors to
  // update its own state (such as PageRank).
  totem_bsp_config_t config = {
```

```
    bfs_compute, bfs_scatter, bfs_init, bfs_finalize, bfs_collect,
    TOTEM_COMM_PUSH
  };
  totem_bsp_config(&config);
  totem_bsp_execute();
}

// The program's main function.
void main() {
  // Load the graph.
  graph_t* graph;
  graph_initialize("/path/to/graph/file", &graph);

  // Initialize Totem. "attr" includes a number of parameters that
  // can be set, the most important of which is the partitioning
  // strategy, which is set to random in TOTEM_DEFAULT_ATTR.
  totem_attr_t attr = TOTEM_DEFAULT_ATTR;
  totem_init(graph, &attr);

  // Allocate the output array and invoke BFS on a random seed.
  level_t* levels =
    (level_t*)malloc(graph->vertex_count * sizeof(level_t));
  vid_t source = rand() % graph->vertex_count;
  bfs_simplified_hybrid(graph, source, levels);
  graph_finalize(graph);
}
```

Appendix B: Other Projects and Publications

Besides the work presented in this dissertation, I collaborated and provided key contributions to a number of other projects during my PhD. This includes work on using GPUs to accelerate distributed storage systems (collaboration with colleagues from UBC, NetSysLab) [*iii*, *vii*, *viii*, *ix*], work on enabling data deduplication for tape-based systems (collaboration with IBM, Almaden) [*i*, *iv*, *v*, *vi*] and designing energy-price aware scheduling algorithms for cloud workloads (collaboration with IBM, Almaden) [*ii*].

(i) **Abdullah Gharaibeh**, Cornel Constantinescu, Maohua Lu, Anu Sharma, Ramani Routray, Prasenjit Sarkar, David Pease and Matei Ripeanu, *DedupT: Deduplication for Tape Systems*, International Conference on Massive Storage Systems and Technology (MSST), Santa Clara, California, June 2014 (*13% acceptance rate*).

(ii) Rini Kaushik, Prasenjit Sarkar, and **Abdullah Gharaibeh**, *Greening the Compute Cloud's Pricing Plans***,** Workshop on Power-Aware Computing and Systems (HotPower), New York, NY, November 2013.

(iii) Samser Al-Kiswany, **Abdullah Gharaibeh** and Matei Ripeanu, *GPUs as Storage System Accelerators*, IEEE Transactions on Parallel and Distributed Systems (TPDS), Volume 24, Issue 8, August 2013.

(iv) **Abdullah Gharaibeh**, Cornel Constantinescu, Maohua Lu, Anu Sharma, Ramani Routray, Prasnejit Sarkar, Matei Ripeanu and David Pease, *CloudDT: Efficient Tape Resource Management using Deduplication in Cloud Backup and Archival Services*, Conference on Network and Service Management (CNSM), Las Vegas, NV, October 2012.

(v) **Abdullah Gharaibeh**, Cornel Constantinescu and Maohua Lu, *Scalable Graph Modeling for Deduplicated Systems*, US patent accepted, 2011.

(vi) Maohua Lu, **Abdullah Gharaibeh**, Cornel Constantinescu, Anu Sharma and David Pease, *A Data Placement Method Optimized for Individual File Accesses on Deduplication-Enabled Tapes*, US patent accepted, 2011.

(vii) **Abdullah Gharaibeh**, Samer Al-Kiswany, Sathish Gopalakrishnan and Matei Ripeanu, *A GPU Accelerated Storage System*, ACM/IEEE International Symposium on High Performance Distributed Computing (HPDC), Chicago, IL, June 2010. *(25% acceptance rate)*.

(viii) **Abdullah Gharaibeh**, Samer Al-Kiswany and Matei Ripeanu, CrystalGPU: Transparent and Efficient Utilization of GPU Power, Technical Report, Networked Systems Lab, The University of British Columbia, NetSysLab-TR-2010-01, January 2010.

(ix) Samer Al-Kiswany, **Abdullah Gharaibeh**, Elizeu Santos-Neto, George Yuan and Matei Ripeanu, StoreGPU: *Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems*, ACM/IEEE International Symposium on High Performance Distributed Computing (HPDC), Boston, MA, June 2008. *(17% acceptance rate)*.