GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions

Lifeng Nai[†], Yinglong Xia[*], Ilie G. Tanase[*], Hyesoon Kim[†], and Ching-Yung Lin[*]
[†]Georgia Institute of Technology, Atlanta GA
[*]IBM Thomas J. Watson Research Center, Yorktown Heights NY
[†]{lnai3,hyesoon.kim}@gatech.edu
[*]{yxia,igtanase,chingyung}@us.ibm.com

ABSTRACT

With the emergence of data science, graph computing is becoming a crucial tool for processing big connected data. Although efficient implementations of specific graph applications exist, the behavior of full-spectrum graph computing remains unknown. To understand graph computing, we must consider multiple graph computation types, graph frameworks, data representations, and various data sources in a holistic way.

In this paper, we present GraphBIG, a benchmark suite inspired by IBM System G project. To cover major graph computation types and data sources, GraphBIG selects representative datastructures, workloads and data sets from 21 real-world use cases of multiple application domains. We characterized GraphBIG on real machines and observed extremely irregular memory patterns and significant diverse behavior across different computations. GraphBIG helps users understand the impact of modern graph computing on the hardware architecture and enables future architecture and system research.

1. INTRODUCTION

In the big data era, information is often linked to form large-scale graphs. Processing connected big data has been a major challenge. With the emergence of data and network science, graph computing is becoming one of the most important techniques for processing, analyzing, and visualizing connected data [24] [29].

Graph computing is comprised of multiple research areas, from low level architecture design to high level data mining and visualization algorithms. Enormous research efforts across multiple communities have been invested in this discipline [19]. However, researchers focus more on analyzing and mining graph data, while paying relatively less attention to the performance of graph computing [43] [41]. Although high performance implementations of specific graph applications and systems exist in prior literature, a comprehensive study on the full spectrum of graph computing is still missing [27] [44]. Unlike prior work focusing on graph traversals and assuming simplified data structures, graph computing today has a much broader scope. In today's graph applications, not only has the structure of graphs analyzed grown in size, but the data associated with vertices and edges has become richer and more dynamic, enabling new hybrid content and graph analysis [8]. Besides, the computing platforms are becoming heterogeneous. More than just parallel graph computing on CPUs, there is a growing field of graph computing on Graphic Processing Units (GPUs).

The challenges in graph computing come from multiple key issues like frameworks, data representations, computation types, and data sources [8]. First, most of the industrial solutions deployed by clients today are in the form of an integrated framework [1] [40] [37]. In this context, elementary graph operations, such as find-vertex and add-edge are part of a rich interface supported by graph datastructures and they account for a large portion of the total execution time, significantly impacting the performance. Second, the interaction of data representations with memory subsystems greatly impacts performance. Third, although graph traversals are considered to be representative graph applications, in practice, graph computing has a much broader scope. Typical graph applications can be grouped into three computation types: (1) computation on graph structure, (2) computation on rich properties, and (3) computation on dynamic graphs. Finally, as a data-centric computing tool, graph computing is sensitive to the structure of input data. Several graph processing frameworks have been proposed lately by both academia and industry [1] [20] [2] [37] [30] [16]. Despite the variety of these frameworks, benchmarking efforts have focused mainly on simplified static memory representations and graph traversals, leaving a large area of graph computing unexplored [25] [7]. Little is known, for example, about the behavior of full-spectrum graph computing with dynamic data representations. Likewise, graph traversal is only one computation type. What is the behavior of other algorithms that build graphs or modify complex properties on vertices and edges? How is the behavior of graph computing influenced by the structure of the input data? To answer these questions, we have to analyze graph workloads across a broader spectrum of computation types and build our benchmarks with extended data representations.

To understand the full-spectrum of graph computing, we propose a benchmark suite, *GraphBIG*[1], and analyze it on contemporary hardware. GraphBIG is inspired by IBM's

[1]GraphBIG is open-sourced under BSD license. The source

SC '15, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807626

System G framework, which is a comprehensive set of industrial graph computing toolkits used by many commercial clients [37]. Based on our experience with a large set of real world user problems, we selected representative graph data representations, interfaces, and graph workloads to design our GraphBIG benchmark suite. GraphBIG utilizes a dynamic, vertex-centric data representation, which is widely utilized in real-world graph systems, and selects workloads and datasets inspired by use cases from a comprehensive selection of application domains. By ensuring the representativeness of data representations and graph workloads, GraphBIG is able to address the shortcomings of previous benchmarking efforts and achieve a generic benchmarking solution. The characterization of performance on GraphBIG workloads can help researchers understand not only the architectural behaviors of specific graph workloads, but also the trend and correlations of full-spectrum graph computing.

The main contributions of this paper are as follows:

- We present the first comprehensive architectural study of full-spectrum graph computing within modern graph frameworks.
- We propose *GraphBIG*, a suite of CPU/GPU benchmarks. GraphBIG utilizes modern graph frameworks and covers all major graph computation types and data sources.
- We analyze the memory behavior of graph computing. Our results indicate high L2/L3 cache miss rates on CPUs as well as high branch/memory divergence on GPUs. However, L1D cache and ICache both show a low miss rate because of the locality of non-graph data and the flat hierarchy of the underlying framework respectively.
- We investigate various workloads and observe diverse architectural behavior across various graph computation types.
- We explore several data sources and observe that graph workloads consistently exhibit a high degree of data sensitivity.

The rest of this paper is organized as follows. In Section 2, we discuss and summarize the key factors of graph computing. Section 3 introduces previous related work. Section 4 illustrates the methodology and workloads of our proposed GraphBIG. In Section 5, we characterize the workloads from multiple perspectives on CPU/GPU hardware. Finally, in Section 6, we conclude our work.

2. GRAPH COMPUTING: KEY FACTORS

Although graph traversals, such as Breadth-first Search and Shortest-path, are usually considered as representative graph applications, real-world graph computing also performs various other comprehensive computations. In real-world practices, graph computing contains a broad scope of use cases, from cognitive analytics to data exploration. The wide range of use cases introduces not only unique, but also diverse features of graph computing. The uniqueness and diversity are reflected in multiple key factors, including frameworks, data representations, computation types, and data sources. To understand graph computing in a holistic way, we first analyze these key factors of graph computing in this section.

codes, datasets, and documents are released in our github repository (http://github.com/graphbig/graphBIG).

Framework: Unlike standalone prototypes of graph algorithms, graph computing systems largely rely on specific frameworks to achieve various functionalities. By hiding the details of managing both graph data and requests, the graph frameworks provide users primitives for elementary graph operations. The examples of graph computing frameworks include GraphLab [20], Pregel [22], Apache Giraph [2], and IBM System G [37]. They all share significant similarity in their graph models and user primitives. First, unlike simplified algorithm prototypes, graph systems represent graph data as a *property graph*, which associates user-defined properties with each vertex and edge. The properties can include meta-data (e.g., user profiles), program states (e.g., vertex status in BFS or graph coloring), and even complex probability tables (e.g., Bayesian inference). Second, instead of directly operating on graph data, the user defined applications achieve their algorithms via framework-defined primitives, which usually include find/delete/add vertices/edges, traverse neighbours, and update properties.

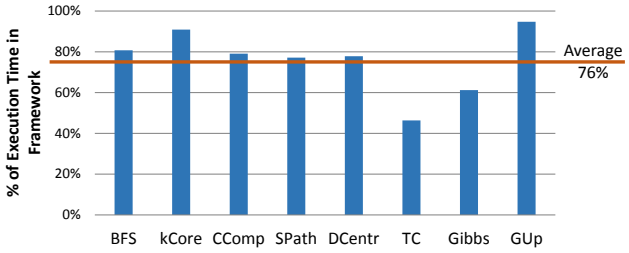

Figure 1: Execution Time of Framework

To estimate the framework's impact on the graph system performance, we performed profiling experiments on a series of typical graph workloads with IBM System G framework. As shown in Figure 1, a significant portion of time is contributed by the framework for most workloads, especially for graph traversal based ones. On average, the in-framework time is as high as 76%. It clearly shows that the heavy reliance on the framework indeed results in a large portion of in-framework execution time. It can bring significant impacts on the architecture behaviors of the upper layer graph workloads. Therefore, to understand graph computing, it is not enough to study only simple standalone prototypes. Workload analysis should be performed with representative frameworks, in which multiple other factors, such as flexibility and complexity, are considered, leading to design choices different from academic prototypes.

Data representation: Within the graph frameworks, various data representations can be incorporated for organizing in-memory graph data. The differences between in-memory data representations can significantly affect the architectural behaviors, especially memory sub-system related features, and eventually impact the overall performance.

One of the most popular data representation structure is Compressed Sparse Row (CSR). As illustrated in Figure 2(a)(b), CSR organizes vertices, edges, and properties of graph G in separate compact arrays. (Variants of CSR also

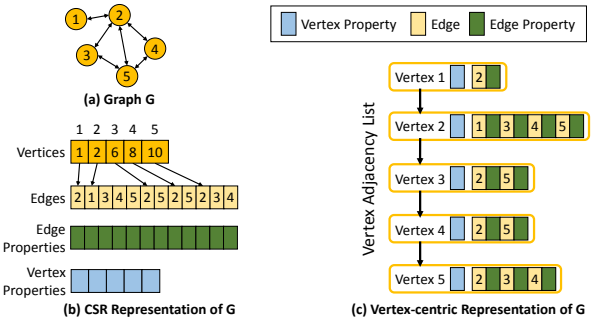

Figure 2: Illustration of data representations. (a) graph G, (b) its CSR representation, and (c) its vertex-centric representation.

exist. For example, Coordinate List (COO) format replaces the vertex array in CSR with an array of source vertices of each edge.) The compact format of CSR saves memory size and simplifies graph build/copy/transfer complexity. Because of its simplicity, CSR is widely used in the literature. However, its drawback is also obvious. CSR is only suitable for static data with no structural updates. This is the case for most graph algorithm prototypes. Nevertheless, real-world graph systems usually are highly dynamic in both topologies and properties. Thus, more flexible data representations are incorporated in graph systems. For example, IBM System G, as well as multiple other frameworks, is using a vertex-centric structure, in which a vertex is the basic unit of a graph. As shown in Figure 2(c), the vertex property and the outgoing edges stay within the same vertex structure. Meanwhile, all vertices form up an adjacency list with indices. Although the compact format of CSR may bring better locality and lead to better cache performance, graph computing systems usually utilize vertex-centric structures because of the flexibility requirement of real-world use cases [20] [37].

Computation types: Numerous graph applications exist in previous literature and real-world practices. Despite the variance of implementation details, generally, graph computing applications can be classified into a few computation types [42]. As shown in Table 1, we summarize the applications into three categories according to their different computation targets: graph structure, graph properties, and dynamic graphs. They have different features in terms of read/write/numeric intensity. (1) Computation on the graph structure incorporates a large number of memory accesses and limited numeric operations. Their irregular memory access pattern leads to extremely poor spatial locality. (2) On the contrary, computation on graphs with rich properties introduces lots of numeric computations on properties, which leads to hybrid workload behaviors. (3) For computation on dynamic graphs, it also shows an irregular pattern as the first computation type. However, the updates of graph structure lead to high write intensity and dynamic memory footprint.

Graph data sources: As a data-centric computing tool, graph computing heavily relies on data inputs. As shown in Table 2, we summarize graph data into four sources [42]. The social network represents the interactions between individuals/organizations. The key features of social networks include high degree variances, small shortest path lengths, and large connected components [26]. On the contrary, an information network is a structure, in which the dominant interaction is the dissemination of information along edges. It usually shows large vertex degrees, and large two-hop neighbourhoods. The nature network is a graph of biological/cognitive objects. Examples include gene network [31], deep belief network (DBN) [6] and biological network [10].They typically incorporate structured topologies and rich properties addressing different targets. Man-made technology networks are formed by specific man-made technologies. A typical example is a road network, which usually maintains small vertex degrees and a regular topology.

3. RELATED WORK

Previous benchmarking efforts of graph computing are summarized in Table 3. Most of existing benchmarks target other evaluation purposes, which are much broader than graph computing. For example, CloudSuite [11] and BigDataBench [39] target cloud computing and big data computing respectively. Graph is only a small portion of their applications. Similarly, PBBS [34] targets evaluations of parallel programming methodologies. Parboil [36] and Rodinia [9] are both for general GPU benchmarking purposes. Lonestar [7] focuses on irregular GPU applications, which include not only graph computing but also several other aspects. As one of the most famous graph benchmarks, Graph 500 [25] was proposed for system performance ranking purposes. Although reference codes exist in Graph 500, because of its special purpose, it provides limited number of workloads.

Besides, graph computing has a broad scope, covering multiple computation types. As shown in Table 3, most of existing benchmarks are highly biased to graph traversal related workloads (CompStruct). The other two graph computation types, computation on dynamic graphs and on rich properties, are less studied. However, as we illustrated in the previous section, both of them are important graph computation types and cannot be overlooked when analyzing the full-scope graph computing.

Moreover, without incorporating realistic frameworks, most prior graph benchmarks assume simplified static graph structures with no complex properties attached to vertices and edges. However, this is not the case for most real-world graph processing systems. The underlying framework plays a crucial role in the graph system performance. Moreover, in real-world systems, graphs are dynamic and both vertices and edges are associated with rich properties.

Multiple system-level benchmarking efforts are also ongoing for evaluating and comparing existing graph systems. Examples include LDBC benchmark, GraphBench, G. Yong's characterization work [13], and A. L. Varbanescu's study [38]. We excluded them in the summary of Table 3 because of their limited usability for architectural research. In these benchmarking efforts, very few ready-to-use open-source benchmarks are provided. Detailed analysis on the architectural behaviors is also lacking.

Examples of existing graph computing frameworks include Pregel [22], Giraph [2], Trinity [33], GraphLab [20], and System G [37]. Multiple academic research efforts also have been proposed, such as GraphChi [17], X-stream [30], Cusha [16], and Mapgraph [12]. They incorporate various techniques to achieve different optimization targets on specific platforms. For example, GraphChi utilizes the Parallel Sliding Window

Graph Computation Type	Feature	Example
Computation on graph structure (CompStruct)	Irregular access pattern, heavy read accesses	BFS traversal
Computation on graphs with rich properties (CompProp)	Heavy numeric operations on properties	Belief propagation
Computation on dynamic graphs (CompDyn)	Dynamic graph, dynamic memory footprint	Streaming graph

Table 1: Graph Computation Type Summary

No.	Graph Data Source	Example	Feature
1	Social(/economic/political) network	Twitter graph	Large connected components, small shortest path lengths
2	Information(/knowledge) network	Knowledge graph	Large vertex degrees, large small hop neighbourhoods
3	Nature(/bio/cognitive) network	Gene network	Complex properties, structured topology
4	Man-made technology network	Road network	Regular topology, small vertex degrees

Table 2: Graph Data Source Summary

Benchmark	Graph Workloads	Framework	Data Representation	Computation Type	Data Support
SPEC int	mcf, astar	NA	Arrays	CompStruct	Data type 4
CloudSuite [11]	TunkRank	GraphLab [20]	Vertex-centric	CompStruct	Data type 1
Graph 500 [25]	Reference code	NA	CSR	CompStruct	Synthetic data
BigDataBench [39]	4 workloads	Hadoop	Tables	CompStruct	Data type 1
SSCA [4]	4 kernels	NA	CSR	CompStruct	Synthetic data
PBBS [34]	5 workloads	NA	CSR	CompStruct	Synthetic data
Parboil [36]	GPU-BFS	NA	CSR	CompStruct	Synthetic data
Rodinia [9]	3 GPU kernels	NA	CSR	CompStruct	Synthetic data
Lonestar [7]	3 GPU kernels	NA	CSR	CompStruct	Synthetic data
GraphBIG	12 CPU workloads 8 GPU workloads	IBM System G [37]	Vertex-centric /CSR	CompStruct/CompProp /CompDyn	All types & synthetic data

Table 3: Comparison between GraphBIG and Prior Graph Benchmarks. Computation and Data Types are Summarized in Table 1 and Table 2.

(PSW) technique to optimize disk IO performance. Cusha extends the similar technique on GPU platforms to improve data access locality.

System G and other related projects like Boost Graph Library [1], Neo4j [40], Giraph [2], GraphLab [20], are comprehensive graph processing toolkits that are used to build real world applications deployed by clients in both industry and academia. We refer to these projects as industrial solutions and we oppose them to simple graph infrastructures that only aim to study the property of various datastructures and algorithms [25]. The design of an industrial framework is concerned with not only performance, but also usability, complexity, and extensibility. In GraphBIG, instead of directly using a specific industrial framework, we abstract one based on our experience with System G and a large number of interactions with our clients. The workloads are all from representative use cases and cover all major computation types. Meanwhile, the framework and data representation design are both following generic techniques widely used by multiple graph systems.We anticipate that GraphBIG and its comprehensive analysis we include in this paper can better illustrate the trend of full-spectrum graph computing and help the future graph architecture/system design.

4. OVERVIEW OF GRAPHBIG

4.1 Methodology

To understand the graph computing, we propose GraphBIG, a benchmark suite inspired by IBM System G, which is a comprehensive set of graph computing tools, cloud, and solutions for Big Data [37]. GraphBIG includes representative benchmarks from both CPU and GPU sides to achieve a holistic view of general graph computing.

Framework: To represent real-world scenarios, GraphBIG utilizes the framework design and data representation inspired by IBM System G, which is a comprehensive graph computing toolsets used by several real-world scenarios. Like many other industrial solutions, the major concerns of SystemG design include not only performance, but also flexibility, complexity, and usability. System G framework enables us to utilize its rich use case and dataset support and summarize workloads from one of the representative industrial graph solutions. By ensuring the representativeness of workloads and data representations, GraphBIG help users understand the full-spectrum graph computing.

Like several other graph systems, GraphBIG follows the vertex-centric data representation, in which a vertex is the basic unit of a graph. The vertex property and the outgoing edges stay within the same vertex structure. All vertices' structures form an adjacency list and the outgoing edges inside the vertex structure also form an adjacency list of edges. The graph computing workloads are implemented via framework primitives, such as find/add/delete vertex/edge and property update.By linking the same core code with various CUDA kernels, the GPU benchmarks are also utilizing the same framework. In addition, because of the nature of GPU computing, the graph data in GPU memory is organized as CSR/COO structure. In the graph populating step,

the dynamic vertex-centric graph data in CPU main memory is converted and transferred to GPU side. Moreover, by replacing System G's commercial components with rewritten source codes, we are able to open source GraphBIG for public usage under BSD license.

Workload Selection: GraphBIG follows the workflow shown in Figure 3. By analyzing real-world use cases from IBM System G customers, we summarize computation types and graph data types. Meanwhile, we select workloads and datasets according to their popularity (use frequency). To ensure the coverage, we then reselect the workloads and datasets to cover all computation and data types. After that, we finalize the workloads and datasets to form our GraphBIG benchmark suite. In this way, the representativeness and coverage are addressed at the same time.

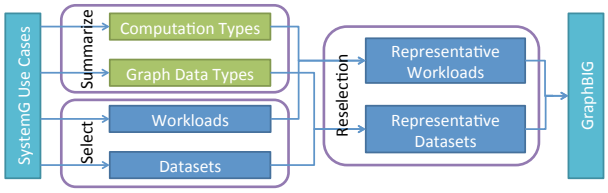

Figure 3: GraphBIG Workload Selection Flow

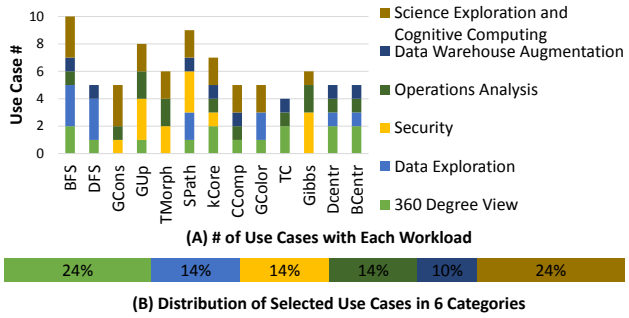

Figure 4: Real-world use case analysis

To understand real-world graph computing, we analyzed 21 key use cases of graph computing from multiple application domains. The use cases are all from real-world practices of IBM System G customers [37] [42]. As shown in Figure 4, the use cases come from six different categories, from cognitive computing to data exploration. Their percentage in each category is shown in Figure 4(B), varying from 24% to 10%. Each use case involves multiple graph computing algorithms. As explained previously, we then select representative workloads from the use cases according to the number of used times. Figure 4(A) shows the number of use cases of each chosen workload with the breakdown by categories. The most popular workload, BFS, is used by 10 different use cases, while the least popular one, TC, is also used by 4 use cases. From Figure 4(A), we can see that the chosen workloads are all widely used in multiple real-world use cases. After the summarize step, a reselection is performed in merge step to cover all computation types.

4.2 Benchmark Description

As explained in Figure 3 and Figure 4, we analyze real-world use cases and then select workloads by considering the key factors together. The workloads in our proposed GraphBIG are summarized in Table 4. For explanation purpose, we group the workloads into four categories according to their high level usages. The details are further explained below.

Graph traversal: Graph traversal is the most fundamental operation of graph computing. Two workloads – Breadth-first Search (BFS) and Depth-first Search (DFS) are selected. Both are widely-used graph traversal operations.

Graph construction/update: Graph update workloads are performing computations on dynamic graphs. Three workloads are included as following. (1) Graph construction (GCons) constructs a directed graph with a given number of vertices and edges. (2) Graph update (GUp) deletes a given list of vertices and related edges from a existing graph. (3) Topology morphing (TMorph) generates an undirected moral graph from a directed-acyclic graph (DAG). It involves graph construction, graph traversal, and graph update operations.

Graph analytics: There are three groups of graph analytics, including topological analysis, graph search/match, and graph path/flow. Since basic graph traversal workloads already cover graph search behaviors, here we focus on topological analysis and graph path/flow. As shown in Table 4, five chosen workloads cover the two major graph analytic types and two computation types. The shortest path is a tool for graph path/flow analytics, while the others are all topological analysis. In their implementations, the shortest path is following Dijkstra's algorithm. The k-core decomposition is using Matula & Beck's algorithm [23]. The connected component is implemented with BFS traversals on the CPU side and with Soman's algorithm [35] on the GPU side. The triangle count is based on Schank's algorithm [32] and the graph coloring is following Luby-Jones' proposal [14]. Besides, the Gibbs inference is performing Gibbs sampling for approximate inference in bayesian networks.

Social analysis: Due to its importance, social analysis is listed as a separate category in our work, although generally social analysis can be considered as a special case of generic graph analytics. We select graph centrality to represent social analysis workloads. Since closeness centrality shares significant similarity with shortest path, we include the betweenness centrality with Brandes' algorithm [21] and degree centrality [15].

4.3 Graph data support

To address both representativeness and coverage of graph data sets, we consider two types of graph data, real-world data and synthetic data. Both are equally important, as explained in Section 2. The real-world data sets can illustrate real graph data features, while the synthetic data can help in analyzing workloads because of its flexible data size and relatively short execution time. Meanwhile, since the focus of our work is the architectural impact of graph computing, the dataset selection should not bring obstacles for architectural characterizations on various hardware platforms. Large datasets are infeasible for small-memory platforms because of their huge memory footprint sizes. Therefore, we only include one large graph data in our dataset selection.As shown in Table 5, we collect four real-world data sets and a synthetic data set to cover the requirements of both sides.

Category	Workload	Computation Type	CPU	GPU	Use Case Example
Graph traversal	BFS	CompStruct	√	√	Recommendation for Commerce
	DFS	CompStruct	√		Visualization for Exploration
Graph update	Graph construction (GCons)	CompDyn	√		Graph Analysis for Image Processing
	Graph update (GUp)	CompDyn	√		Fraud Detection for Bank
	Topology morphing (TMorph)	CompDyn	√		Anomaly Detection at Multiple Scales
Graph analytics	Shortest path (SPath)	CompStruct	√	√	Smart Navigation
	K-core decomposition (kCore)	CompStruct	√	√	Large Cloud Monitoring
	Connected component (CComp)	CompStruct	√	√	Social Media Monitoring
	Graph coloring (GColor)	CompStruct		√	Graph matching for genomic medicine
	Triangle count (TC)	CompProp	√	√	Data Curation for Enterprise
	Gibbs inference (GI)	CompProp	√		Detecting Cyber Attacks
Social analysis	Degree centrality (DCentr)	CompStruct	√	√	Social Media Monitoring
	Betweenness centrality (BCentr)	CompStruct	√	√	Social Network Analysis in Enterprise

Table 4: GraphBIG Workload Summary

The details of the chosen data sets are explained below. All data sets are publicly available in our github wiki.

Data Set	Type	Vertex#	Edge#
Twitter Graph	Type 1	120M	1.9B
IBM Knowledge Repo	Type 2	154K	1.72M
IBM Watson Gene Graph	Type 3	2M	12.2M
CA Road Network	Type 4	1.9M	2.8M
LDBC Graph	Synthetic	Any	Any

Table 5: Graph Data Set Summary

(1) Real-world data: Four real-world graph data sets are provided, including twitter graph, IBM knowledge Repo, IBM Watson Gene Graph, and CA road network. The vertex/edge numbers of each data set are shown in Table 5. The twitter graph is a preprocessed data set of twitter transactions. In this graph, twitter users are the vertices and twit/retwit communications form the edges. In IBM Knowledge Repo, two types of vertices, users and documents, form up a bipartite graph. An edge represents a particular document is accessed by a user. It is from a document recommendation system used by IBM internally. As an example of bio networks, the IBM Watson Gene graph is a data set used for bioinformatic research. It is representing the relationships between gene, chemical, and drug. The CA road network is a network of roads in California [18]. Intersections and endpoints are represented by nodes and the roads connecting these intersections or road endpoints are represented by undirected edges.

(2) Synthetic data: The LDBC graph is a synthetic data set generated by LDBC data generator and represents social network features [28]. The generated LDBC data set can have arbitrary data set sizes while keeping the same features as a facebook-like social network. The LDBC graph enables the possibility to perform detailed characterizations of graph workloads and compare the impact of data set size.

5. CHARACTERIZATIONS

5.1 Characterization Methodology

Hardware configurations: We perform our experiments on an Intel Xeon machine with Nvidia Tesla K40 GPU. The hardware and OS details are shown in Table 6. To avoid the uncertainty introduced by OS thread scheduling, we schedule and pin threads to different hardware cores.

Processor	Type	Xeon E5-2670
	Frequency	2.6 GHz
	Core #	2 sockets x 8 cores x 2 threads
	Cache	32 KB L1, 256 KB L2, 20 MB L3
	MemoryBW	51.2 GB/s (DDR3)
GPU	Type	Nvidia Tesla K40
	CUDA Core	2880
	Memory	12 GB
	MemoryBW	288 GB/s (GDDR5)
	Frequency	Core-745 MHz Memory-3 GHz
Host System	Memory	192 GB
	Disk	2 TB HDD
	OS	Red Hat Enterprise Linux 6

Table 6: Test Machine configurations

Experiment Data Set	Vertex #	Edge #
Twitter Graph (sampled)	11M	85M
IBM Knowledge Repo	154K	1.72M
IBM Watson Gene Graph	2M	12.2M
CA Road Network	1.9M	2.8M
LDBC Graph	1M	28.82M

Table 7: Graph Data in the Experiments

Datasets: In the characterization experiments, we first use synthetic graph data to enable in-depth analysis for multiple architectural features of both CPU and GPU sides. As shown in Table 7, the LDBC graph with 1 million vertices is selected. Four real-world data sets are then included for data sensitivity studies. The Twitter graph is sampled in our experiments because of the extremely large size of the original graph. In our experiments, although the test platform incorporates a large memory capacity on CPU side, the GPU side has only 12GB memory, which limits the dataset size. Thus, huge size datasets are infeasible in the experiments. Moreover, we intentionally select datasets from diverse sources to

cover different graph types. With the combination of different graph sizes and types, the evaluation can illustrate the data impact comprehensively. In addition, because of the special computation requirement of Gibbs Inference workload, the bayesian network MUNIN [3] is used. It includes 1041 vertices, 1397 edges, and 80592 parameters.

Profiling method: In our experiments, the hardware performance counters are used for measuring detailed hardware statistics. In total, around 30 hardware counters of the CPU side and 25 hardware metrics of the GPU side are collected. For the profiling of CPU benchmarks, we designed our own profiling tool embedded within the benchmarks. It is utilizing the perf_event interface of Linux kernel for accessing hardware counters and the libpfm library for encoding hardware counters from event names. For GPU benchmarks, the nvprof tool from Nvidia CUDA SDK is used.

Metrics for CPUs: In the experiments, we are following a hierarchical profiling strategy. Multiple metrics are utilized to analyze the architectural behaviors.

For the CPU benchmarks, *Execution cycle breakdown* is first analyzed to figure out the bottleneck of workloads. The breakdown categories include frontend stall, backend stall, retiring, and bad speculation. In modern processors, frontend includes instruction fetching, decoding, and allocation. After allocated, backend is responsible for instruction renaming, scheduling, execution, and commit. It also involves memory sub-systems. *Cache MPKI* is then analyzed to understand memory sub-system behaviors. We estimated the MPKI values of L1D, L2, and LLC. In addition, we also measured multiple other metrics, including *IPC, branch miss rate, ICache miss rate, and DTLB penalty.* These metrics cover major architectural factors of modern processors.

Metrics for GPUs: For the GPU side experiments, we first analyzed the divergence of given benchmarks. Two metrics are measured, one is *branch divergence rate* (BDR) and another is *memory divergence rate* (MDR). We use the following equations to express the degree of branch and memory divergence.

$$\text{branch divergence rate (BDR)} = \frac{\text{inactive threads per warp}}{\text{warp size}}$$

$$\text{memory divergence rate (MDR)} = \frac{\text{replayed instructions}}{\text{issued instructions}}$$

BDR is the average ratio of inactive threads per warp, which is typical caused by divergent branches. MDR is the fraction of issued instructions that are replayed. In modern GPUs, a load or store instruction would be replayed if there is a bank conflict or the warp accesses more than one 128-byte block in memory. The replay may happen multiple times until all the requested data have been read or written. Thus, we estimate the memory divergence by measuring the number of replayed instructions. Both BDR and MDR range from 0 to 1 with higher value representing higher divergence.

5.2 CPU Characterization Results

5.2.1 Workload Characterization

In this section, we characterize GraphBIG CPU workloads with a top-down characterization strategy. The results are explained as following.

Execution time breakdown: The execution time breakdown is shown in Figure 5 and grouped by computation types. The Frontend and Backend represent the frontend

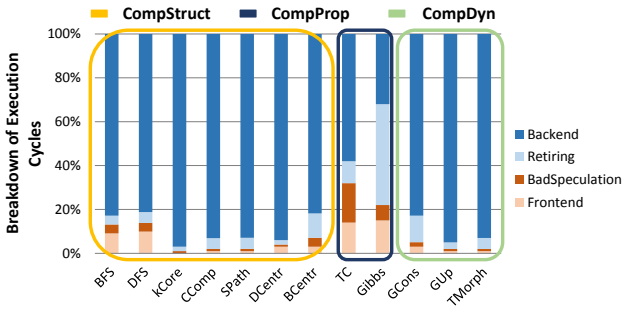

Figure 5: Execution Time Breakdown of GraphBIG CPU Workloads

bound and backend bound stall cycles respectively. The BadSpeculation is the cycles spent on wrong speculations, while the Retiring is the cycles of successfully retired instructions. It is a common intuition that irregular data accesses are the major source of inefficiencies of graph computing. The breakdown of execution time also supports such intuition. It is shown that the backend indeed takes dominant time for most workloads. In extreme cases, such as kCore and GUp, the backend stall percentage can be even higher than 90%. However, different from the simple intuition, the outliers also exist. For example, the workloads of computation on rich properties (CompProp) category shows only around 50% cycles on backend stalls. The variances between computation types further demonstrates the necessity of covering different computation types.

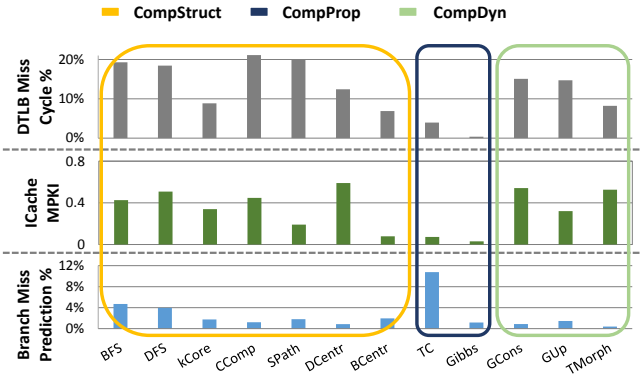

Figure 6: DTLB Penalty, ICache MPKI, and Branch Miss Rate of GraphBIG CPU Workloads

Core analysis: Although execution stall can be triggered by multiple components in the core architecture, instruction fetch and branch prediction are usually the key inefficiency sources. Generally, a large number of ICache misses or branch miss predictions can significantly affect architectural performance, because modern processors usually don't incorporate efficient techniques to hide ICache/branch related penalties. In previous literatures, it was reported that many big data workloads, including graph applications, suffer from high ICache miss rate [11]. However, in our experiments, we observe different outcomes.As shown in Figure 6, the ICache MPKI of each workload all show below 0.7 values, though small variances still exist. The differ-

ent ICache performance values are resulted from the design differences of the underlying frameworks. Open-source big data frameworks typically incorporate many other existing libraries and tools. Meanwhile, the included libraries may further utilize other libraries. Thus, it eventually results in deep software stacks, which lead to complex code structures and high ICache MPKI. However, in GraphBIG, very few external libraries are included and a flat software hierarchy is incorporated. Because of its low code structure complexity, GraphBIG shows a much lower ICache MPKI.

The branch prediction also shows low miss prediction rate in most workloads except for TC, which reaches as high as 10.7%. The workloads from other computation types show a miss prediction rate below 5%. The difference comes from the special intersection operations in TC workload. It is also in accordance with the above breakdown result, in which TC consumes a significant amount of cycles in BadSpeculation.

The DTLB miss penalty is shown in Figure 6. The cycles wasted on DTLB misses is more than 15% of totaly execution cycles for most workloads. On average, it still takes 12.4%. The high penalty is caused by two sources. One is the large memory footprint of graph computing applications, which cover a large number of memory pages. Another is the irregular access pattern, which incorporates extremely low page locality. Diversity among workloads also exists. The DTLB miss penalty reaches as high as 21.1% for Connected Component and as low as 3.9% for TC and 1% for Gibbs. This is because for computation on properties, the memory accesses are centralized within the vertices. Thus, low DTLB-miss penalty time is observed.

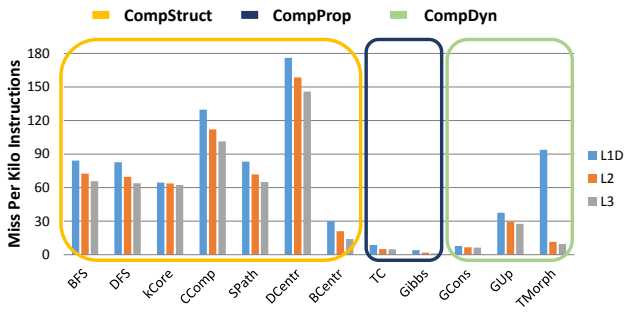

Figure 7: Cache MPKI of GraphBIG CPU Workloads

Cache performance: As shown in previous sections, cache plays a crucial role in graph computing performance. In Figure 7, the MPKI of different levels of caches are shown. On average, a high L3 MPKI is shown, reaching as high as 48.77. Degree Centrality and Connected Component show even higher MPKI, which are 145.9 and 101.3 respectively. For computations on the graph structure (CompStruct), a generally high MPKI is observed. On the contrary, CompProp shows an extremely small MPKI value compared with other workloads. This is in accordance with its computation features, in which memory accesses happen mostly inside properties with a regular pattern. The workloads of computation on dynamic graphs (CompDyn) introduce diverse results, ranging from 6.3 to 27.5 in L3 MPKI. This is because of the diverse operations of each workload. GCons adds new vertices/edges and sets their properties one by one, while GUp mostly deletes them in a random manner. In GCons, significantly better locality is observed because each new vertex/edge will be immediately reused after insertion. The TMorph involves graph traversal, insertion, and deletion operations. Meanwhile, unlike other workloads, TMorph includes no small size local queues/stacks, leading to a high MPKI in L1D cache. However, its graph traversal pattern results in relatively good locality in L2 and L3.

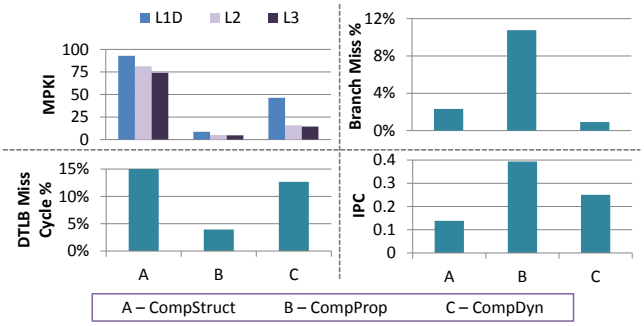

Figure 8: Average Behaviors of GraphBIG CPU Workloads by Computation Types

Computation type behaviors: The average behaviors of each computation type are shown in Figure 8. Although variances exist within each computation type, the average results demonstrate their diverse features. The CompStruct shows significantly higher MPKI and DTLB miss penalty values because of its irregular access pattern when traversing through graph structure. Low and medium MPKI and DTLB values are shown in CompProp and CompDyn respectively. Similarly, the CompProp suffers from a high branch miss rate while other two types do not. In the IPC results, CompStruct achieves the lowest IPC value due to the penalty from cache misses. On the contrary, CompProp shows the highest IPC value. The IPC value of CompDyn stays between them. Such feature is in accordance with their access patterns and computation types.

Data sensitivity: To study the impact of input data sets, we performed experiments on four real-world data sets from different types of sources and the LDBC synthetic data (We excluded the workloads that cannot take all input datasets).

Despite the extremely low L2/L3 hit rates, Figure 9 shows relatively higher L1D hit rates for almost all workloads and data sets. This is because graph computing applications all incorporate multiple small size structures, such as task queues and temporal local variables. The frequently accessed meta data introduces a large amount of L1D cache hits except for DCentr, in which there is a only limited amount of meta data accesses. From the results in Figure 9, we can also see that twitter data shows highest DTLB miss penalty in most workloads. Such behavior eventually turns into lowest IPC values in most workloads except SPath, in which higher L1D cache hit rate of the twitter graph helps performance significantly. Triangle Count (TC) achieves highest IPC with the knowledge data set, because of its high L2/L3 hit rate and low TLB penalty. The high L3 hit rate of the watson data also results in a high IPC value. However, the twitter graph's high L3 hit rate is offsetted by its extremely high DTLB miss cycles, leading to the lowest IPC value. The diversity is caused by the different topological and property features of the real-world data sets. It is clearly shown that significant impacts are introduced by the

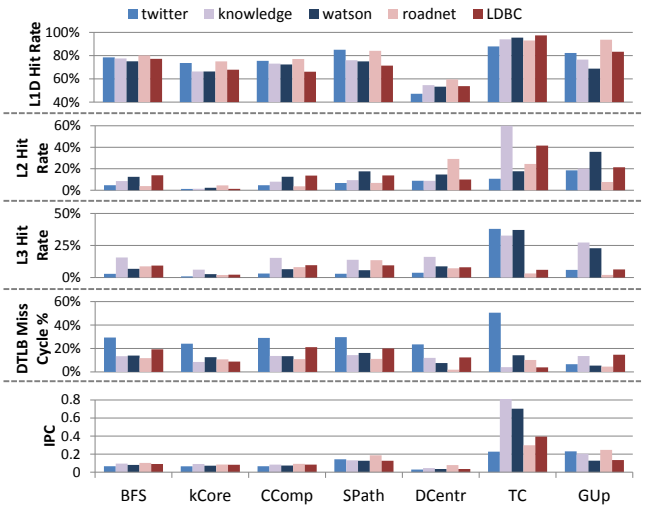

Figure 9: Cache Hit Rate, DTLB Penalty, and IPC of GraphBIG CPU Workloads with Different Data Sets

graph data on overall performance and other architectural features.

5.2.2 *Observations*

In the characterization experiments, by measuring several architectural factors, we observed multiple graph computing features. The key observations are summarized as following.

- Backend is the major bottleneck for most graph computing workloads, especially for CompStruct category. However, such behavior is much less significant in CompProp category.
- The ICache miss rate of GraphBIG is as low as conventional applications, unlike many other big data applications, which are known to have high ICache miss rate. This is because of the flat code hierarchy of the underlying framework.
- Graph computing is usually considered to be cache-unfriendly. L2 and L3 caches indeed show extremely low hit rates in GraphBIG. However, L1D cache shows significantly higher hit rates. This is because of the locality of non-graph data, such as temporal local variables and task queues.
- Although typically DTLB is not an issue for conventional applications, it is a significant source of inefficiencies for graph computing. In GraphBIG, a high DTLB miss penalty is observed because of the large memory footprint and low page locality.
- Graph workloads from different computation types show significant diversity in multiple architectural features. The study on graph computing should consider not only graph traversals, but also the other computation types.
- Input graph data has significant impact on memory sub-systems and the overall performance. The impact is from both the data volume and the graph topology.

The major inefficiency of graph workloads comes from memory subsystem. Their extremely low cache hit rate introduces challenges as well as opportunities for future graph architecture/system research. Moreover, the low ICache miss rate of GraphBIG demonstrates the importance of proper software stack design.

5.3 GPU Characterization Results

We characterize the GPU workloads of GraphBIG in this section. The experiments are performed via *nvprof* on real machines. The results are summarized below.

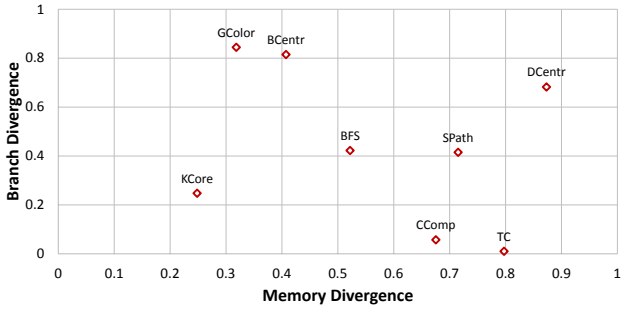

Figure 10: Branch and Memory Divergence of GraphBIG GPU workloads

Irregularity analysis: To estimate the irregularity of graph computing on GPU, we measured the degree of both branch and memory divergence. As explained in Section 5.1, the metrics we used in the experiments are *branch divergence rate* (BDR) and *memory divergence rate* (MDR). With a higher BDR value, more threads in the same warp take different execution paths, leading to a lower warp utilization. Similarly, a higher MDR value indicates that more memory requests contain bank conflicts or are accessing more than 128-byte blocks. In this case, instruction replays are triggered to fulfill all memory requests.

Figure 10 shows a scatter plot of the workloads where the x-axis represents MDR and the y-axis represents BDR. Each dot in the figure corresponds to a GraphBIG workload. From Figure 10, we can observe that most workloads cannot be simply classified as branch-bound or memory-bound. A generally high divergence degree from both sides are shown. In addition, the workloads show a quite scatter distribution across the whole space. For example, kCore stays at the lower-left corner, showing relatively lower divergence in both branch and memory. On the contrary, DCentr is showing extremely high divergence in both sides. Meanwhile, branch divergence is the key issue of GColor and BCentr, while for CComp and TC, the issue is only from memory side.

The high branch divergence for graph computing comes from the thread-centric design, in which each thread processes one vertex. However, the working set size of each vertex is corresponding its degree, which can vary greatly. The unbalanced per-thread workload introduces divergent branches, especially for the loop condition checks, leading to branch divergence behaviors.In Figure 10, a relatively higher BDR is observed in GColor and BCentr because of the heavier per-edge computation. On the contrary, CComp and TC show small BDR values because they are both following an edge-centric model, in which each thread processes one edge.

In typical graph workloads, because of the sparse dis-

tributed edges, accesses to both the neighbor list and vertex property are spreading across the whole graph. Hence, the traversal of each edge involves cache misses and divergent memory accesses. Depending on the reliance of graph properties and warp utilization, the degree of memory divergence may vary. As shown in Figure 10, the MDR value can be as low as 0.25 in kCore and as high as 0.87 in DCentr.

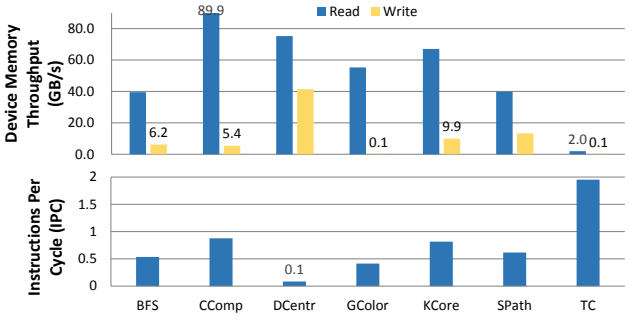

Figure 11: Memory Throughput and IPC of GraphBIG GPU Workloads

Memory throughput and IPC: The GPU device memory throughput results are shown in Figure 11. Although the Tesla K40 GPU supports up to 288 GB/s memory bandwidth, in our experiments, the highest read throughput is only 89.9 GB/s in CComp. The inefficient bandwidth utilization is cause by divergence in both branch and memory. The memory throughput results shown in Figure 11 are determined by multiple factors, including data access intensity, memory divergence, and branch divergence. For example, CComp incorporates intensive data accesses and low branch divergence. It shows the highest throughput value. DCentr has extremely intensive data accesses. Hence, even though DCentr has high branch and memory divergence, its throughput is still as high as 75.2 GB/s. A special happens at TC, which shows only 2.0 GB/s read throughput. This is because TC is mostly performing intersection operations between neighbor vertices with quite low data intensity. Since data accesses typically are the bottleneck, the memory throughput outcomes also reflect application performance in most workloads except for DCentr and TC. In DCentr, despite the high memory throughput, intensive atomic instrcutions significantly reduce performance. Meanwhile, unlike other workloads, TC involves lots of parallel arithmetic compare operations. Hence, TC shows the highest IPC value.

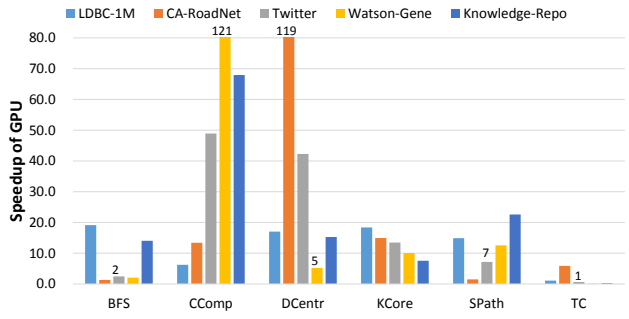

Figure 12: Speedup of GPU over 16-core CPU

Speedup over CPU: Although GPU is usually considered to be suitable for regular applications, irregular graph computing applications still receive performance benefits by utilizing GPUs. In Figure 12, the speedup of GPU over 16-core CPU is shown. In the experiments, we utilized the GraphBIG workloads that are shared between GPU and CPU sides. In our comparison, the major concern is in-core computation time, not end-to-end time. The data loading/transfer/initialize time are not included. Besides, as explained in Section 4, the dynamic vertex-centric data layout is utilized at CPU side, while GPU side uses CSR graph format.

From the results in Figure 12, we can see that GPU provides significant speedup in most workloads and datasets. The speedup can reach as high as 121 in CComp and around 20x in many other cases. In general, the significant speedup achieved by GPU is because of two major factors, thread-level parallelism (TLP) and data locality. It is difficult to benefit from instruction-level parallelism in graph applications. However, the rich TLP resources in GPUs can still provide significant performance potentials. Meanwhile, the CSR format in GPUs brings better data locality than the dynamic data layout in CPUs. Specifically, the DCentr shows high speedup number with CA-RoadNet because of the low branch divergence and static working set size. Likewise, CComp also shows similar behaviors. On the contrary, BFS and SPath show significant lower speedup values because of the low efficiency introduced by varying working set size. The speedup of TC is even lower. This is because of its special computation type. In TC, each thread incorporates heavy per-vertex computation, which is inefficient for GPU cores.

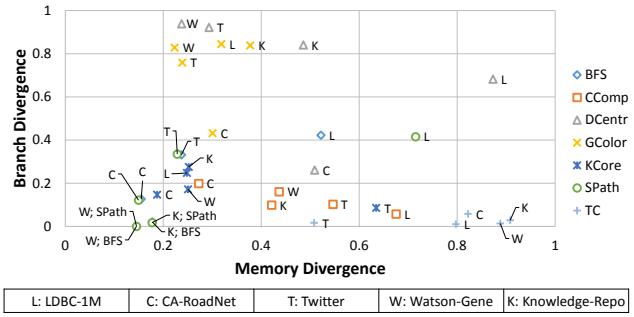

Figure 13: Branch and Memory Divergence of GraphBIG GPU Workloads with Different Datasets

Dataset sensitivity: To estimate the sensitivity of input datasets, we performed divergence analysis on the four real-world datasets and LDBC-1M synthetic graph. In Figure 13, the results of different workloads are shown with different symbols in the same space, meanwhile different datasets are marked with corresponding initial letters.

From Figure 13, we can see that in many workloads, the divergence changes significantly for different datasets. As data-centric computing, graph workloads' behaviors are data dependent. However, the results of several datasets also tend to cluster in the same region. For CComp and TC, the branch divergence rate does not change much between different input graphs. This is expected behavior for them. They both incorporate an edge-centric implementation, in which workload is partitioned by edges, ensuring balanced

workset size between threads. Because of its low branch divergence feature, kCore also shows quite low variability in branch divergence. Both BFS and SPath show similarly low BDR values for CA-RoadNet, Watson-gene, and Knowledge-repo. This is in accordance with the graph features. Both Watson and Knowledge graphs contains small-size local subgraphs, while CA-RoadNet includes much smaller vertex degree. Nevertheless, in Twitter and LDBC graphs, their social network features brings high BDR values. Meanwhile, unlike Twitter has a few vertices with extremely higher degree, the unbalanced degree distribution in LDBC involves more vertices. It leads to even higher warp divergence. Similar diversity happens in GColor and DCentr, which show much lower BDR values for CA-RoadNet graph because of its quite low vertex degrees.

Unlike branch divergence, MDR generally shows much higher variability for most workloads. It demonstrates the data sensitivity of memory divergence. Meanwhile, exceptions also exist. For example, BFS and SPath both show similar MDR values for CA-RoadNet, Watson-gene, and Knowledge-repo. As explained above, their special graph structures lead to a small number of traversed edges in each iteration. Thus, the impact of input graph is reduced. Moreover, the higher irregularity in edge distribution of LDBC leads to significantly higher MDR values in most workloads.

5.3.1 Observations

Unlike the conventional applications, the irregularity of graph computing brings unique behaviors on GPUs. We summarize the key observations as follows.

- Although graph computing is usually considered as less suitable for GPUs, with proper designs, GPU graph applications can achieve significant speedup over the corresponding CPU implementations.
- Branch divergence is known to be the top issue for graph computing on GPUs. We observe that besides branch divergence, graph computing suffers from even higher memory divergence, leading to inefficient memory bandwidth utilizations.
- Graph workloads cannot fully utilize the GPU's execution capability. An extremely low IPC value is observed for most GraphBIG workloads.
- The behaviors of graph computing are data dependent. Input graph has comprehensive impacts on both branch and memory divergence. Specifically, memory divergence shows higher data sensitivity.
- Interestingly, the GPU graph workloads have significantly higher data sensitivity than the CPU ones. CPU/GPU sides show different data-application correlations, because of the architecture differences.
- Although traversal based workloads show similar behaviors, significant diverse behaviors across all workloads are observed. It is difficult to summarize general features that can be applied on all graph workloads. Hence, a representative study should cover not only graph traversals, but also the other workloads.

Although suffering from high branch and memory divergence, graph computing on GPUs still show significant performance benefits in most cases. Meanwhile, like CPU workloads, GPU graph computing also incorporate workload diversity and data dependent behaviors. In addition, comparing with CPU workloads, GPU graph workloads have much higher data sensitivity and more complex correlations between input data and application. To improve the performance of GPU graph computing, new architecture/system techniques considering both workload diversity and data sensitivity are needed.

6. CONCLUSION

In this paper, we discussed and summarized the key factors of graph computing, including frameworks, data representations, graph computation types, and graph data sources. We analyzed real-world use cases of IBM System G customers to summarize the computation types, and graph data sources. We also demonstrated the impact of framework and data representation.

To understand the full-spectrum graph computing, we presented *GraphBIG*, a suite of CPU/GPU benchmarks. Our proposed benchmark suite addressed all key factors simultaneously by utilizing System G framework design and following a comprehensive workload selection procedure. With the summary of computation types and graph data sources, we selected representative workloads from key use cases to cover all computation types. In addition, we provided real-world data sets from different source types and synthetic social network data for characterization purposes.

By performing experiments on real machines, we characterized GraphBIG workloads comprehensively. From the experiments, we observed following behaviors. (1) Conventional architectures do not perform well for graph computing. Significant inefficiencies are observed in CPU memory sub-systems and GPU warp/memory bandwidth utilizations. (2) Significant diverse behaviors are shown in different workloads and different computation types. Such diversity exists on both CPU and GPU sides, and involves multiple architectural features. (3) Graph computing on both of CPUs and GPUs are highly data sensitive. Input data has significant and complex impacts on multiple architecture features.

As the first comprehensive architectural study on full-spectrum graph computing, GraphBIG can be served for architecture and system research of graph computing. In the future, we will also extend GraphBIG to other platforms, such as near-data processing (NDP) units [5], IBM BlueGene/Q, and IBM System Z.

Acknowledgment

We gratefully acknowledge the support of National Science Foundation (NSF) XPS 1337177. We would like to thank IBM System G group, other HPArch members, our shepherd, and the anonymous reviewers for their comments and suggestions. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF.

References

[1] *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman, 2002.

[2] Apache Giraph. http://giraph.apache.org/, 2015.

[3] S. Andreassen and et al. MUNIN — an expert EMG assistant. In *Computer-Aided Electromyography and Expert Systems*. 1989.

[4] D. A. Bader and et al. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. HiPC'05.

[5] R. Balasubramonian and et al. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 2014.

[6] Y. Bengio. Learning deep architectures for ai. *Found. Trends Mach. Learn.*, 2(1), Jan. 2009.

[7] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on gpus. IISWC'12.

[8] M. Canim and Y.-C. Chang. System G Data Store: Big, rich graph data analytics in the cloud. IC2E'13.

[9] S. Che and et al. Rodinia: A benchmark suite for heterogeneous computing. IISWC'09, Oct 2009.

[10] E. Chesler and M. Haendel. *Bioinformatics of Behavior:*. Number pt. 2. Elsevier Science, 2012.

[11] M. Ferdman and et al. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. ASPLOS'12, 2012.

[12] Z. Fu, M. Personick, and B. Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. GRADES'14, 2014.

[13] Y. Guo and et al. Benchmarking graph-processing platforms: A vision. ICPE'14, 2014.

[14] M. T. Jones and P. E. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, May 1993.

[15] U. Kang and et al. Centralities in large networks: Algorithms and observations. SDM'11, 2011.

[16] F. Khorasani and et al. Cusha: Vertex-centric graph processing on gpus. HPDC'14, 2014.

[17] A. Kyrola and et al. Graphchi: Large-scale graph computation on just a pc. OSDI'12, 2012.

[18] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014.

[19] C.-Y. Lin and et al. Social network analysis in enterprise. *Proceedings of the IEEE*, 100(9), Sept 2012.

[20] Y. Low and et al. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8), Apr. 2012.

[21] K. Madduri and et al. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. IPDPS'09.

[22] G. Malewicz and et al. Pregel: A system for large-scale graph processing. SIGMOD'10, 2010.

[23] D. Matula and et al. Smallest-last ordering and clustering and graph coloring algorithms. *J. ACM*, 1983.

[24] J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. SIGMOD'12, 2012.

[25] R. C. Murphy and et al. Introducing the graph 500. In *Cray User's Group (CUG)*, 2010.

[26] S. A. Myers and et al. Information network or social network?: The structure of the twitter follow graph. WWW Companion'14.

[27] L. Nai, Y. Xia, C.-Y. Lin, B. Hong, and H.-H. S. Lee. Cache-conscious graph collaborative filtering on multi-socket multicore systems. CF'14, 2014.

[28] M.-D. Pham and et al. S3g2: A scalable structure-correlated social graph generator. TPCTC'12, 2012.

[29] M. J. Quinn and N. Deo. Parallel graph algorithms. *ACM Comput. Surv.*, 16(3), Sept. 1984.

[30] A. Roy and et al. X-stream: Edge-centric graph processing using streaming partitions. SOSP'13, 2013.

[31] W. RW and et al. Genetic and molecular network analysis of behavior. *Int Rev Neurobiol*, 2012.

[32] T. Schank and et al. Finding, counting and listing all triangles in large graphs, an experimental study. WEA'05, 2005.

[33] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. SIGMOD'13, 2013.

[34] J. Shun and et al. Brief announcement: The problem based benchmark suite. SPAA'12, 2012.

[35] J. Soman, K. Kishore, and P. Narayanan. A fast gpu algorithm for graph connectivity. IPDPSW'10.

[36] J. A. Stratton and et al. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, UIUC, 2012.

[37] I. Tanase, Y. Xia, L. Nai, Y. Liu, W. Tan, J. Crawford, and C.-Y. Lin. A highly efficient runtime and graph library for large scale graph analytics. GRADES'14, 2014.

[38] A. L. Varbanescu and et al. Can portability improve performance?: An empirical study of parallel graph analytics. ICPE'15, 2015.

[39] L. Wang and et al. Bigdatabench: A big data benchmark suite from internet services. HPCA'14, 2014.

[40] J. Webber. A programmatic introduction to neo4j. SPLASH '12, 2012.

[41] Z. Wen and C.-Y. Lin. How accurately can one's interests be inferred from friends. WWW'10, 2010.

[42] Y. Xia. System G: Graph analytics, storage and runtimes. In *Tutorial on the 19th ACM PPoPP*, 2014.

[43] Y. Xia, J.-H. Lai, L. Nai, and C.-Y. Lin. Concurrent image query using local random walk with restart on large scale graphs. ICMEW'14, July 2014.

[44] Y. Xia and V. K. Prasanna. Topologically adaptive parallel breadth-first search on multicore processors. PDCS'09.