Pannotia: Understanding Irregular GPGPU Graph Applications

Shuai Che[†], Bradford M. Beckmann[†], Steven K. Reinhardt[†] and Kevin Skadron[‡]
{Shuai.Che, Brad.Beckmann, Steve.Reinhardt}@amd.com, skadron@cs.virginia.edu

AMD Research[†] and Computer Science, University of Virginia[‡]

***Abstract*—GPUs have become popular recently to accelerate general-purpose data-parallel applications. However, most existing work has focused on GPU-friendly applications with regular data structures and access patterns. While a few prior studies have shown that some irregular workloads can also achieve speedups on GPUs, this domain has not been investigated thoroughly.**

Graph applications are one such set of irregular workloads, used in many commercial and scientific domains. In particular, graph mining –as well as web and social network analysis– are promising applications that GPUs could accelerate. However, implementing and optimizing these graph algorithms on SIMD architectures is challenging because their data-dependent behavior results in significant branch and memory divergence.

To address these concerns and facilitate research in this area, this paper presents and characterizes a suite of GPGPU graph applications, *Pannotia*, which is implemented in OpenCL and contains problems from diverse and important graph application domains. We perform a first-step characterization and analysis of these benchmarks and study their behavior on real hardware. We also use clustering analysis to illustrate the similarities and differences of the applications in the suite. Finally, we make architectural and scheduling suggestions that will improve their execution efficiency on GPUs.

I. Introduction

There is a growing trend of using graphics processing units (GPUs) for high-performance parallel computing. The GPU's high compute throughput and memory bandwidth make it a desirable platform for accelerating applications with massive data parallelism. Prior work [12], [15], [41] showed that diverse applications benefit from a GPU's high parallelism, achieving higher performance and increased energy efficiency than conventional CPUs. These applications come from a variety of domains, including multimedia, data mining, bioinformatics, and other HPC numerical algorithms. However, most of these applications are "GPU friendly" in that they use regular data structures and present regular parallelism and accesses.

Graph algorithms are fundamental to many application domains, yet perform poorly on today's GPUs. Large graph structures with millions of vertices and edges are common in many HPC and commercial applications, including networks, electronic design automation, graph mining and social network analysis. Accelerating graph algorithms is a challenge for GPUs and other SIMD architectures. The high performance of GPU applications relies on high SIMD lane occupancy and efficient memory coalescing for inter-thread data locality. The former requires minimal divergent branching for threads in a SIMD group, while the latter requires regular memory access patterns and data structure layouts. Unfortunately, graph applications tend to present both significant branch and memory divergence on GPUs. Furthermore, memory accesses are input-dependent and hard to predict. Load imbalance among threads is also a common challenge.

To improve the efficiency of graph applications on GPUs, designers must first understand their characteristics and performance bottlenecks. To this end, we present a suite of graph applications, *Pannotia*, and evaluate them on contemporary GPUs. *Pannotia*'s applications present diverse parallelism and access patterns and varying GPU resource utilization characteristics. Prior work has studied GPU acceleration of individual graph applications [11], [19], [32], [43], or included some graph applications in a larger study of irregular workloads [10]. However, none of them provide an overall picture of the range of characteristics, similarities, and differences among a broad set of graph applications. Prior research has also neglected the important domains of web and social network analysis.

In this work, we make the following contributions:

- We present *Pannotia*, a suite of applications with the focus of studying graph algorithms on GPUs and other emerging SIMD architectures.
- We conduct a preliminary characterization of these graph applications on real GPU hardware and analyze their challenges.
- We perform a hierarchical clustering analysis to characterize the range of benchmark behaviors, and study their sensitivity to diverse graph structures. We show that different program-input pairs may show vastly different characteristics.
- Finally, we discuss some architectural design features that allow more efficient execution of these graph workloads on SIMD hardware.

The set of applications we evaluate show diversity on the GPU platform. They differ in the performance benefits achieved through GPU acceleration, and demonstrate different levels of SIMD lane occupancy and memory access efficiency. In addition, workload behaviors can be quite different varying graph inputs and across different phases of a single application.

II. Background

In this section, we briefly introduce the architecture of AMD Radeon™ HD 7000 series GPUs and the OpenCL programming model. Though this study is limited to characterizations on an AMD GPU, the methodology and insight can be applied to other SIMD architectures.

A. A Short Primer on AMD GPUs

In this paper, we use AMD GPUs to report our measurement results. The AMD Radeon HD 7000 series GPUs use the Graphics Core Next (GCN) Architecture [5], which is a radically new approach compared to prior AMD designs based on very long instruction word (VLIW).

The AMD Radeon HD 7950 GPU includes 28 SIMD compute units (CUs). Each CU has one scalar unit and four vector units [4]. Each vector unit contains an array of 16 processing elements (PEs). Each PE consists of one ALU. The four vector units use SIMD execution of a scalar instruction. Each CU contains a single instruction cache, an SC cache (the data cache for the scalar Unit), a 16-KB L1 data cache and a 64-KB local data share (LDS) (i.e., software managed scratchpad). All CUs share a single 768-KB L2 cache. The AMD Radeon HD 7950 GPU supports GDDR5 with twelve memory channels and 3 GB DRAM.

B. OpenCL

The OpenCL programming model is a domain-based model for programming GPUs and other accelerators [4]. In OpenCL, a host program launches a kernel with work-items over an index space (called an NDRange). Work-items are grouped into work-groups. OpenCL currently supports multiple memory spaces (e.g., the global memory space shared by all workgroups, the per-workgroup local memory space, the per-workitem private memory space, etc.). In addition, there are also constant and texture memory (i.e., image) spaces for read-only data structures. OpenCL uses a relaxed consistency model. Two types of barrier synchronizations are supported in different scopes: local barriers for all the threads (i.e., work-items) in a workgroup and global barriers for all the threads launched in a kernel. On the host side, OpenCL has a variety of options for buffer and queue management (e.g., specifying inter-kernel dependencies).

III. Challenges of Graph Algorithms on GPUs

In this section, we discuss some common workload behaviors and challenges shared by many graph applications on the GPU platform. In Section VI, we will focus our evaluation of graph workloads from these perspectives. Understanding these issues is helped by visualizing the two sample graph structures in Figure 1. It shows a jazz musician network and a dolphin social network [2] we generated using the Gephi [6] framework.

Branch divergence occurs when threads in the same wavefront take different execution paths. GPUs execute instructions in SIMD lockstep and can execute only one path of the branch at a time for a given wavefront, with some threads masked off if they took the branch in a different direction [4], [18], [31]. Many graph algorithms visit a sub-set of "active" vertices or edges in each iteration (see Section IV-C). There is a high probability that only part of the SIMD cores are active when processing different threads in a wavefront. This leads to low SIMD throughput due to underutilization of compute resources and wastes of power.

Memory divergence occurs when threads from a single wavefront experience different memory access latencies caused by cache misses or accessing different memory banks. In current organizations, the entire wavefront must wait until the last thread finishes memory access [4], [18], [31]. This is a common issue for graph algorithms. For instance, the concurrently visited adjacency lists may be distributed in different regions of memory. Supplying these data for multiple threads may take multiple memory transactions. In addition, the access patterns of these applications are hard to predict to improve data locality. The efficiency of memory references would be poor with unoptimized data layouts and access patterns. Furthermore, there is not much data reuse (i.e. computation per memory access) in certain graph applications.

Load imbalance is due to uneven work distribution across different threads inside a kernel call. This imbalance is often related to the structure of the graph being processed. In most graphs, some vertices have higher degree (i.e., the number of connected neighboring vertices) than others. Kernels are often structured such that each GPU thread is assigned to process one vertex, iterating over its edge list. The total running time of a wavefront will be determined by the thread processing the vertex with the largest number of edges, even though many other threads in the wavefront will have completed their tasks much earlier.

Parallelism may vary during the entire program execution (e.g., across iterations) depending on the traversal patterns of specific algorithms as well as the graph structures processed. In contrast to many regular applications (e.g., stencil applications) in which each iteration processes the same amount of work, graph applications have unpredictable computation loads over time. It is common that only a few vertices or edges are being processed in some iterations while hundreds of thousands of vertices or edges are being processed in other iterations. This raises a challenge of scheduling and partitioning work across SIMD units –or even across the CPU and the GPU– to save power consumed by the idling processing elements, and to avoid being bottlenecked by the GPU's low single-thread performance when it is not achieving sufficient parallelism to compensate.

IV. Overview of Pannotia

Pannotia is an OpenCL application library, including a set of graph applications and kernels, common graph utility routines, and datasets. The chosen applications are widely used in many scientific and enterprise applications. The suite includes basic graph algorithms such as graph traversal, graph partitioning, shortest paths, etc., and also includes emerging applications used for analyzing webs and social networks (e.g., Facebook,

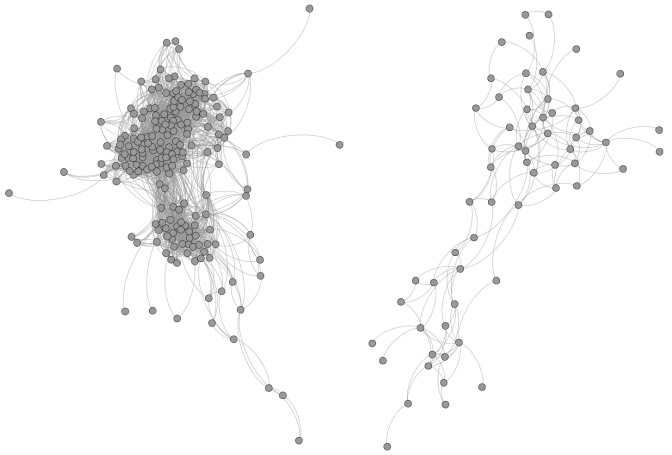

Fig. 1. A graph illustration of jazz musician (left) and dolphin social (right) networks

Twitter). *Pannotia* currently includes eight diverse applications, supporting both multi-core CPU and GPU execution. We have developed some of the *Pannotia* applications ourselves, while others are based on prior work [19], [21], [35], [40].

Pannotia also includes library routines to help users generate and parse graphs. We develop a set of running scripts capable of being configured to work with hardware profilers to gather application characteristics. The graph output formats are also compatible with the Gephi framework for easy visualization of graph structures.

A. Graph Input Formats and Data Structures

Pannotia currently supports two types of graph formats: *COO* and *METIS* [33]. The library contains a set of functions to parse graph input files stored in these formats. For the coordinate format, *Pannotia* supports both the 9th DIMACS Implementation Challenge [3] and the Matrix Market formats [29].

Graphs are represented in different data structures internally in different applications. For instance, some applications use the compressed sparse row (CSR) representation, storing the vertices, their corresponding edge lists and weights in three separate compact vectors (e.g., in *graph coloring* and *maximum independent set*). *Betweenness centrality* uses a representation similar to the concept of *COO*. The ith position (i.e., edge i) of two vectors are used to store IDs of the two connecting vertices, while the third vector is used to store the corresponding edge weight. Additionally, *Floyd-Warshall* uses 2D adjacency matrices to store the weights and shortest paths for each pair of vertices. *Connected component labeling* uses a 2D array to store the images to be labeled.

B. Graph Data Sets

Pannotia uses many real-world graphs from different domains (e.g., co-author and citation graphs, road networks, numerical simulation meshes, clustering instances, etc.). We evaluate graphs from the 9th DIMACS Implementation Challenges [3] for shortest-path related problems (e.g., US-road-NW and CA) and the 10th DIMACS Implementation Challenges for graph partitioning and clustering [2] problems (e.g., ecology, shell and G3_circuit). Also, some other graphs (e.g., flickr) are chosen from the University of Florida Sparse Matrix Collection [42]. *Pannotia* also uses graph generators to produce random graphs for experiments. We use Georgia Tech's *GTgraph* random-graph generator [34] to generate synthetic graphs.

TABLE I
PANNOTIA APPLICATIONS AND KERNELS.

Applications/Kernels	Graph Domains	Abbreviation
Connected components labeling	Graph clustering	CCL
Dijkstra	Shortest path	DJK
Graph coloring	Graph partitioning	CLR
Maximal independent set	Graph partitioning	MIS
Floyd-Warshall	Shortest path	FW
Friend recommendation	Social network	FRD
Betweenness centrality	Social network	BC
Page rank	Web algorithm	PRK

C. Benchmark Description

In this section, we present a brief introduction of the current benchmarks included in the *Pannotia* suite.

Connected components labeling (CCL) is used in computer vision and image processing to detect connected regions in images. Given an image or graph that needs to be labeled, an auxiliary structure L, with the same size, stores labels for all the corresponding nodes in the data structure. A label is a value in L that points to a pixel or node. *CCL* labels all the nodes so if node a and b are in the same region, they will have the same label in L [35]. A typical task of *CCL* is to find the parent of a node. *CCL* in *Pannotia* is an OpenCL implementation of a prior work [35].

Dijkstra (DJK) solves the single-source shortest path (SSSP) problem for a graph with non-negative edge path costs, producing a shortest-path tree [14]. This algorithm is often used as a subroutine in various graph algorithms. Given a user-specified source vertex in the graph, the algorithm searches the path with lowest cost (i.e., the shortest path) between the source node and all the other nodes in the graph. *DJK* keeps track of a distance array, saving the shortest distances of all the vertices evaluated so far. Given a new visited vertex, for each of its neighbors, if the calculated distance via passing through the vertex is smaller than the old distance, a new value of distance will be updated. In a multithreaded implementation, this must be done with atomics, since thread contention may occur if two threads converge on the same vertex from different paths.

Graph coloring (CLR) partitions the vertices of a graph such that no two adjacent vertices share the same color. The *CLR* implementation in *Pannotia* is an OpenCL implementation based on the algorithm described in the work [21]. This work does not attempt to achieve optimal coloring. The goal is to divide a graph into independent set of vertices for parallel computation; vertices with the same color are in the same

set. Doing such coloring is among the first steps in many parallel graph algorithms. In the initialization step, each vertex is labeled with a random integer value. The algorithm then launches multiple iterations, each responsible for labeling one color. For each vertex, the algorithm compares its vertex value with that of its neighboring vertices. If the vertex value of a given node happens to be the largest (or smallest) among its neighbors it marks itself with the current iteration colors (one each for the largest and smallest in each set). The algorithm terminates when all vertices are colored.

Maximal independent set (MIS) finds a maximal subset of vertices in a graph such that no two are adjacent. MIS is another basic building block for many graph algorithms. The first step of *MIS* is similar to that of *CLR*. Each vertex is labeled with a random integer value and each vertex judges whether it can be included in the set. If so, the vertex is added to the array which stores the current set. For the vertices added to the set in the current iteration, the algorithm expands their neighbor lists and marks all neighbors inactive; they will be removed from the candidate list and not participate in the evaluation of the next iteration of *MIS*. The algorithm will terminate when all nodes are visited and evaluated. MIS is a GPU implementation of Luby's algorithm [27].

Floyd-Warshall (FW) is a classical dynamic programming algorithm, solving the *all-pairs shortest paths (APSP)* problem. Given a graph *G(V, E)*, a function *shortestPath(i, j, k)* returns the shortest possible path from i to j using vertices only from the set *1, 2 , ..., k* as intermediate vertices. One important step of the algorithm attempts to find the shortest path from each i to each j using only vertices 1 to $k+1$. For each pair of vertices, the shortest path can be either a path that uses vertices in the set *1, 2,..., k* or a path that goes from i to $k+1$ and then from $k+1$ to j. The core of the algorithm is:

```
shortestPath(i,j,k+1) = min(shortestPath(i,j,k),
                            shortestPath(i,k+1,k)
                          + shortestPath(k+1,j,k))
```

This *Floyd-Warshall* implementation is an OpenCL version based on the algorithm of prior work [19].

Friend recommendation (FRD) One common functionality in social websites, such as Facebook and LinkedIn, is to recommend people or friend connections. *FRD* algorithms recommend people who do not know each other but have common friends. Friend relationships can be maintained with an adjacency list. For instance, for each person i, we keep track of a list of persons who are friends of i.

```
"Andy" — ["Mark", "Dave", "Bob", ...]
```

A simple algorithm finds the top n persons with whom a person has common friends. For each person and their contact list, the algorithm populates a set of $n/(n-1)$ triples, where n is the number of friends a person has. For instance, *Andy* will populate the triples including *(Bob, Mark, Andy)*, *(Bob, Dave, Andy)*. For *Bob*, the algorithm recommends *Mark* and *Dave* through *Andy*. After the algorithm generates all the triples, a filtering step eliminates the pairs who are already friends.

Betweenness centrality (BC) measures vertices' centrality in a network [9]. It is a widely-used algorithm to identify a set of popular vertices in a network (e.g., a social network). For a graph $G(V, E)$ with n vertices and m edges, the *BC* of a vertex $v \in V$ is defined as:

$$\sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{1}$$

where σ_{st} represents the number of shortest paths between s and t, and $\sigma_{st}(v)$ represents the number of shortest paths that pass through a specified vertex v. The main part of the algorithm contains a series of kernels including doing *APSP* across vertices, and performing backtracking and reduction to update the σ value for each vertex to calculate the *BC* values. The implementation currently includes the use of atomic operations to handle thread contentions when doing reductions. Our *BC* OpenCL implementation is based on the algorithm of a prior work [40].

Page rank (PRK) is an algorithm used by Google to calculate probability distributions representing the likelihood that a person randomly clicking on links arrives at any particular page [36]. In the first step, the value of each vertex is initialized to $\frac{1}{num\_vertices()}$. In each step of the main computation loop, each vertex sends along its outgoing edge its current PageRank divided by the number of outgoing edges [28]. Each vertex then sums up the values arriving at it and calculates the PageRank value. The algorithm terminates until convergence determined by an aggregator or after running a user-specified number of iterations. There are be multiple possible implementations of PageRank; ours is based on the Pregel description [28].

V. Methodology

In this section, we discuss the experiment setup and methodology to characterize and analyze *Pannotia* applications.

A. Experiment Setup

The experiment results are measured on real hardware using an AMD Radeon HD 7950 (Tahiti) discrete GPU. The AMD Radeon HD 7950 features 28 GCN CUs with 1792 processing elements running at 800 MHz with 3 GB of device memory. We compare the GPU results with those obtained from four CPU cores on an AMD A8-5500 accelerated processing unit (APU) with a 1.4-GHz clock rate and 2 MB L2 cache. We use AMD APP SDK 2.8 with OpenCL 2.1 support. AMD APP Profiler v2.5 is used to collect profiling results. In addition, this study is restricted to cases when the working sets of applications do not exceed the capacity of the GPU device memory. For much larger graphs, research is needed to design algorithms for graph processing and partitioning; simple overlaying and chunking will not work efficiently for certain graph applications, because their data-dependent memory accesses tend to be difficult to predict, and this risks incurring too much GPU-CPU interaction..

TABLE II
THE PROFILER COUNTERS USED IN THIS STUDY [1].

Counters	Descriptions	Types of Metrics
ALUInsts	The average number of ALU instructions executed per thread	ALU instructions
FetchInsts	The average number of fetch instructions from the memory executed per thread	Memory instructions
WriteInsts	The average number of write instructions to the memory executed per thread	Memory instructions
ALUUtilization	The percentage of active vector ALU threads in a wavefront	SIMD utilization
ALUBusy	The percentage of GPUTime ALU instructions processed	Compute intensity
CacheHit	The percentage of fetch, write, atomic, and other instructions that hit the L2 cache	Memory locality
MemUnitBusy	The percentage of GPUTime the memory unit is active	Memory BW util
MemUnitStalled	The percentage of GPUTime the memory unit is stalled	Memory BW util
WriteUnitStalled	The percentage of GPUTime the write unit is stalled	Memory BW util
FetchSize	The total data in kB fetched from the memory	Memory traffic
WriteSize	The total data in kB written to the memory	Memory traffic

B. Metrics and Hierarchical Clustering

We evaluate and characterize these graph applications from several different perspectives, including the relative performance of GPU and CPU implementations and breaking down execution time among CPU execution, GPU kernel execution, and CPU-GPU data communications. We also study the sensitivity of application behaviors with different graph input structures.

Table II lists a set of hardware counters we measure on AMD GPUs. The counters include metrics related to arithmetic and memory instructions, SIMD lane utilization, cache locality, and memory bandwidth utilization.

To demonstrate the similarity or dissimilarity of the benchmarks we evaluate, we use an approach similar to that of prior GPU program analysis efforts [13], [17]. We apply principal component analysis (PCA) and hierarchical clustering to the characteristics we collect, then use a dendrogram to show a clustering tree to demonstrate similarity among benchmarks. These techniques have been widely applied for benchmark comparison in similar contexts [7], [20], [22], [37]. However, the question of how to perform more fair and accurate evaluation and comparison of benchmarks and how to determine what metrics are the most important remain open problems that are beyond the scope of this paper.

VI. CHARACTERIZATIONS AND RESULTS

In this section, we report experimental results characterizing *Pannotia*'s parallelism and load imbalance, performance speedups and execution-time breakdown, SIMD utilization and cache efficiency. Finally, we examine the similarity/dissimilarity of these workloads.

A. Parallelism and Load Imbalance

Depending on the traversal patterns and particular graph inputs processed, certain graph algorithms demonstrate varying degrees of parallelism running through different execution phases. We use two examples to show this behavior. Figure 2 shows the number of active vertices processed during the entire execution of *DJK*. The number of vertices that can be processed in parallel first increases, then reaches a peak, and then decreases. Similarly, Figure 3 shows the number of active vertices colored during the entire execution of *CLR*. In contrast, the application begins with a large number of vertices to label, which gradually decreases over time. GPUs are good at computing problems with massive parallelism and big data sets; therefore, such phase behavior must be understood by developers and system designers to utilize the available computation resources efficiently.

Load imbalance across threads is undesirable for SIMD execution, especially for threads in a SIMD wavefront, leading to underutilization of SIMD lanes (See Section VI-C). Figure 4 shows the distribution of the length of edge lists expanded for all active vertices for *DJK* over time. Most of the vertices have a degree of one to four. Though the vertices with degrees of five to eight are a minority in the whole distribution, they cause the long tails of execution making the rest of the threads in the same wavefront idle, doing nothing and wasting power. As another example, Figure 5 shows the edge list degree distribution for *CLR*. Most of the vertices have a degree ranging from two to five. Furthermore, in both cases, degree distributions change over time.

B. Performance Speedup and Execution Time Breakdown

We measure the performance of these graph applications by running OpenCL programs on an AMD Radeon HD 7950 GPU and compare their execution times against four CPU cores of an AMD A8-5500 APU. The speedups are calculated on the main computation parts excluding I/O and initial setup. When calculating GPU execution times, we include the PCIe transfer overhead.

Figure 6 shows the performance results across all the benchmarks. We observe performance improvements up to $10.6\times$ on the GPU compared to the CPU. *FW* achieves the highest speedups. It uses a 2D matrix to keep track of the path distances for each pair of vertices in the graph. In each iteration, all points in the array can be processed in parallel for a given k value (see Section IV-C). Its data structures and access patterns both map well to the GPU architecture. *DJK* and *CLR* achieve speedups ranging from $4\times$ to $8\times$. There is abundant parallelism in these applications, especially for large graphs, which can leverage GPU parallel compute resources. *FRD* and *CCL* achieve modest speedups of only 1–2$\times$ compared to multicore CPU executions. Even though the threads in *FRD* are independent of each other, threads can be waiting for other long-running threads in the same wavefront to finish. Each thread generates $n(n-1)$ pairs,

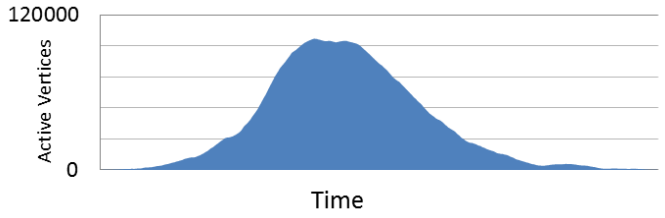

Fig. 2. The number of active vertices processed during the entire runtime of *Dijkstra*

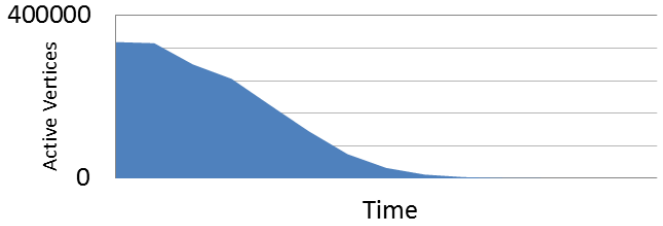

Fig. 3. The number of active vertices are colored during the entire runtime of *Graph Coloring*

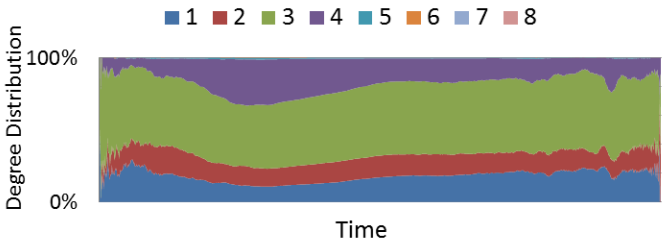

Fig. 4. The distribution of the length of edge lists expanded for all active vertices during the entire runtime of *Dijkstra*

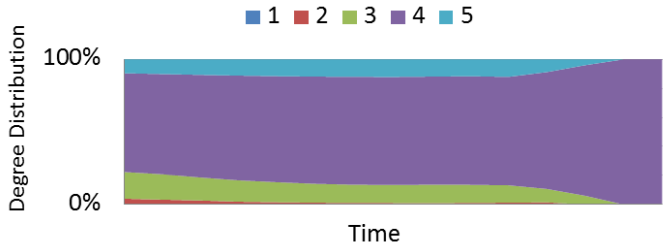

Fig. 5. The distribution of the length of edge lists expanded for all active vertices during the entire runtime of *Graph Coloring*

where n is the degree of a vertex, exacerbating the imbalance caused by variations in vertex degrees. For instance, for the coauthor dataset we use, some authors have hundreds of co-authors, while others have only a few or even one. Similarly, in *CCL*, certain threads spend more time searching for their parent node, doing indirect accesses, than other threads in a wavefront.

Even for a single application, different inputs show different performance benefits. For instance, the speedups achieved by *MIS* on the GPU range from $2.3\times$ (shell) to $4\times$ (ecology). Additionally, the speedups achieved by *CLR* range from $4.6\times$ (ecology) to $7.6\times$ (shell). Section VI-E2 will analyze further the diverse application behaviors exhibited by different inputs.

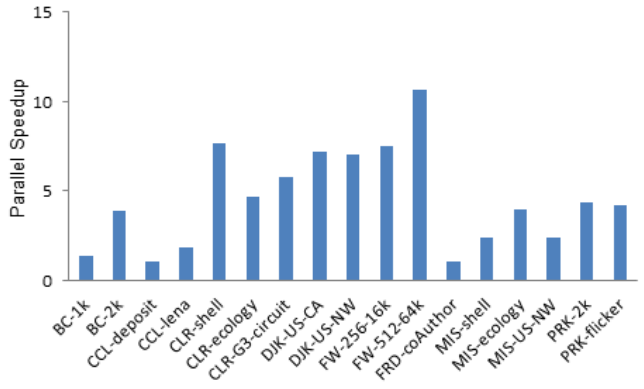

Fig. 6. The speedup of running applications on the GPU compared to multi-core CPUs. The execution time for calculating the speedup is measured on the CPU and GPU for the core part of the computation, excluding the I/O and initial setup.

Figure 7 shows the execution time breakdown due to CPU computation, GPU kernel execution, and CPU-GPU data communication (PCIe). The fraction of time spent on the GPU computation ranges from 8% to 99%. Many applications share a common pattern of looping through several consecutive GPU kernels and spend a majority of time on GPU execution. For instance, *BC* takes multiple iterations, searching the shortest paths for each vertex, updating the score matrix and performing backward computation to calculate the *BC* value for each vertex. Each iteration of *CLR* and *MIS* calculates a subset towards an overall solution and prunes a set of vertices for the next iteration. *DJK* expands the active edges in each iteration.

There are also a few applications which contain a large fraction of CPU computation. For instance, *FRD* sets up the positions where tuples are stored in the final array on the CPU. *MIS* needs randomization of vertex values for each iteration, which is done on the CPU. We anticipate moving some or all of these operations to the GPU as future work.

C. SIMD Utilization

Graph workloads pose significant challenges for efficient GPU SIMD utilization due to their irregular data structures and parallelism. This causes workload imbalance for threads in a SIMD group. Prior research used software techniques to achieve better data and work packing [32] for *breadth-first search*. However, less effort has been applied to other applications.

Figure 8 captures the average percentages of active vector ALU threads in a wavefront across all benchmarks as reported by the AMD APP Profiler. In other words, when a wavefront instruction is issued, this counter provides the average number of threads active as specified by its execution mask. *Pannotia* shows diverse control divergence across benchmarks, with SIMD utilization ranging from less than 10% to more than

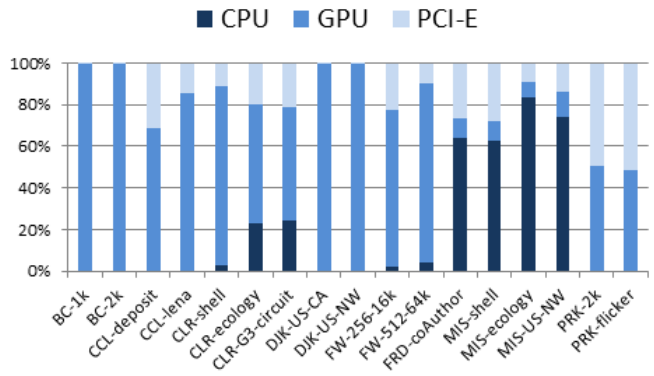

Fig. 7. The fraction of each GPU implementations runtime due to the core part of computation (GPU execution, CPU-GPU communication and CPU execution)

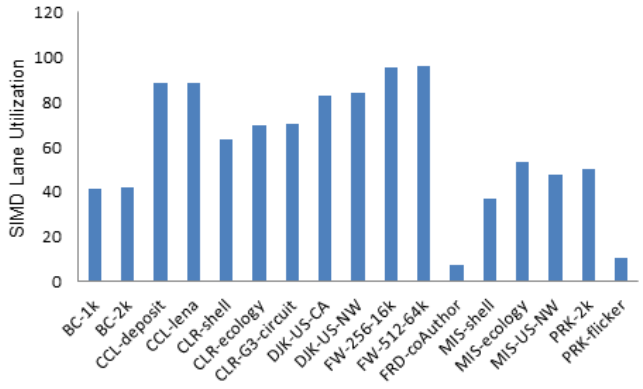

Fig. 8. The percentage of active vector ALU threads in a wavefront reported by AMD APP Profiler (100% is ideal with full utilization)

90%. As discussed in Section VI-B, for the co-authors data set used in *FRD*, certain authors have much longer coauthor lists than others, resulting in significant load imbalance within wavefronts. *CLR* shows an average of 67% threads active. In addition, this application also exhibits an interesting phase behavior: for *CLR* with G3-circuit, some phases show high utilization (around 90%) while other phases show low utilization (around 20%). This of course depends on the input graph structures. Because the inputs we used for *DJK* are road networks, the degrees of most vertices range from one to four. Thus, they show relatively better utilization than applications such as *FRD* with a degree range of one to several hundreds.

D. Cache Access

High GPU performance requires efficient memory bandwidth utilization. In general, multiple GPU thread accesses in a wavefront will be coalesced into a minimum number of memory transactions. Ideally, threads should touch contiguous data elements in memory (e.g., in the same cache line or memory block). However, memory-coalescing optimization is a challenge for graph algorithms due to their hard-to-predict and close-to-random access patterns. The GPU cache is designed primarily to optimize inter-thread spatial locality.

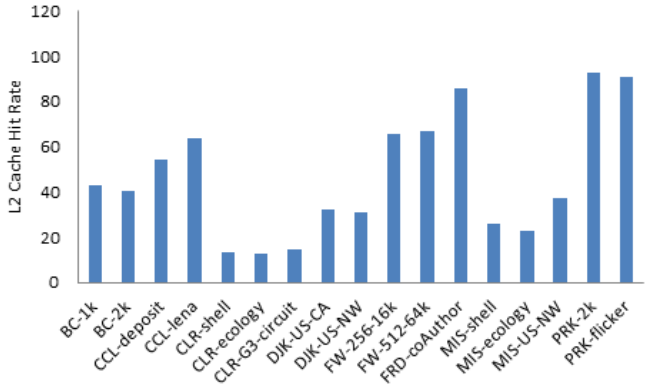

Fig. 9. The percentage of memory accesses that hit the L2 data cache of the GPU, including read, write, atomic operations.

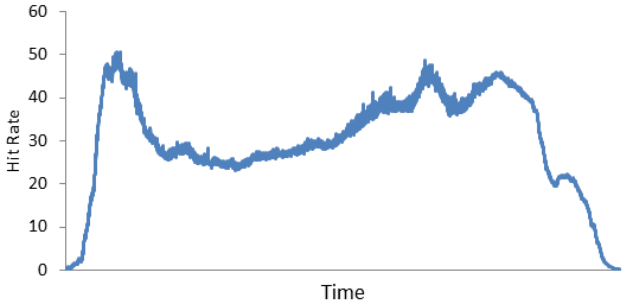

Fig. 10. Hit-rate variation of *DJK* during the entire execution run.

It is interesting to see whether the L2 cache is effective for these workloads.

Figure 9 shows the L2 cache hit rates with different inputs of all the *Pannotia* applications. Again, the workloads show diverse behaviors. The cache hit rates range from about 10% to more than 90%. *FW*, *PRK* and *FRD* demonstrate better cache hit rates. *CLR* shows the worst hit rates (12-14%). Even for a single application, hit rates vary across different inputs (e.g., 26–37% for *MIS*). A detailed analysis requires us to understand the working set and memory footprints of each iteration during the entire program execution, which we leave for future work.

Cache hit rates also show very interesting phase behaviors for some applications. Figure 10 is an example of the hit-rate statistics over time for the main single-source shortest path kernel in *DJK*. The cache hit rate first improves, then degrades, improves again and finally degrades with some fluctuations in the middle. This phenomenon correlates with the phase behavior we studied in Figure 2. We hypothesize that the increasing number of active threads initially brings data into the cache that is likely to be used by other threads. However, as more and more vertices are processed, cache contention and conflicts in turn degrade the cache hit rate.

E. Hierarchical Clustering

In this section, we use clustering analysis to explore similarities and differences across workloads and among different

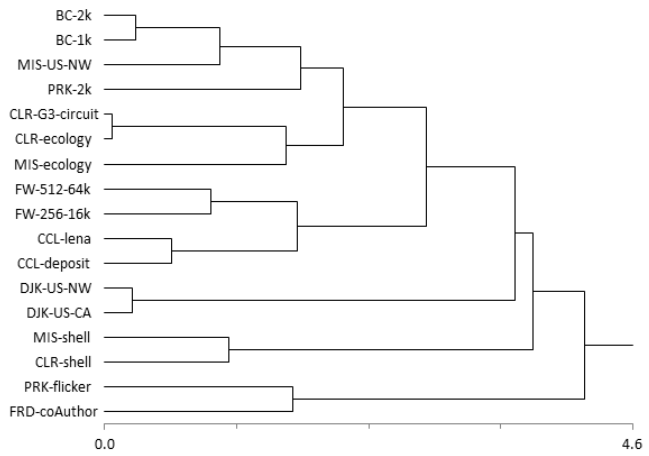

Fig. 11. Dendrogram showing the similarity between different graph workloads(x-axis represents the linkage distance in the PCA coverage space)

inputs for each workload.

1) Workload Similarity: We first conduct a principal component analysis (PCA) with the metrics mentioned in Table II. We perform certain preprocessing for the statistics including calculating ratios such as ALU/mem and fetches/writes with the reported instructions, and normalize the values for each metric across all the benchmarks. Figure 11 is a dendrogram obtained after conducting PCA and hierarchical analysis. It shows the similarity among these graph applications. The applications classified in a cluster are more similar than those classified in another cluster. In the figure, the magnitude of the link distance (x-axis) quantifies the measure of dissimilarity. For instance, *BC* and *MIS* (US-NW data set) show similar behaviors. The next most similar benchmark to them is *PRK* (2k data set). Figure 11 shows that *FW* and *CCL* are similar for the counters that we evaluate and are classified into the same cluster. Both of these two benchmarks show high SIMD utilization and L2 cache hit rates from Figures 8 and 9. *PRK* (flickr data set) and *FRD* are significantly different from others; they demonstrate significant SIMD underutilization compared to the other benchmarks (Figure 8).

2) Diverse Behaviors for Different Program-Input Pairs: The dendrogram also illustrates an important issue: choosing appropriate workloads for research. First, target benchmarks need to represent real-world workloads. Second, benchmarks need to process representative inputs. In other words, designers should consider representative *program-input pairs* for evaluation so they can make appropriate design decisions. Different program-input pairs may show vastly different characteristics on a given platform, which is clearly shown in Figure 11. For instance, benchmark instances with three different inputs (i.e., ecology, shell, US-road-NW) of *MIS* are classified into different clusters. They present very different fetch-versus-write ratios to the global memory as well as percentages of GPU time when the write unit is stalled. *CLR* with G3-circuit and ecology present similar behaviors; however when using the shell input, it becomes more similar to *MIS* with the same input. This implies that researchers using different program-input pairs to optimize their hardware or compilers may reach quite different designs. Similar observations have been made when evaluating single and multi-threaded CPU workloads [8], [16]. But this issue has been overlooked by many researchers when doing computer architecture evaluation and benchmarking. We show that input sensitivity is also an issue for data-parallel applications on SIMD architectures. Choosing the appropriate program-input pairs is especially critical and a challenge for graph applications given diverse and large number of existing graph structures. Research is essential for a high-level abstraction and definition of graph structures and associated analysis approaches for easy understanding of program behaviors.

VII. Discussion

While developing and characterizing these graph benchmarks, we made the following observations, which may allow more efficient execution of these graph applications on the GPU.

A. Architecture and Scheduling

Some graph applications evaluated in this paper share a common parallel access pattern. As we discussed in Section IV-A, graphs in CSR format are typically stored in three compact vectors for vertices, edges, and weights. Each element in the vertex array is an index pointing to the corresponding edge and weight lists.

With such an organization, a typical pattern can be summarized as the following pseudo-code, assuming each thread is assigned to one vertex:

```
//get the vertex id. Each thread processes
//the task for one vertex
int tid = get_thread_id()
//get the start and ending pointers of the edge list
int start = vertex[tid];
int end   = vertex[tid + 1];
...
//expand the edge list for each vertex
for(i = start; i < end; i++){
    compute(edge[i], weight[i],...);
}
...
```

Listing 1. A common pattern in many graph algorithms

In the current GPU model, the instructions of the code in List 1 will be issued to different SIMD lanes. However, when expanding the edge list, the length of edge lists (i.e., the range of i) may vary across different vertices. Thus, for threads in the same SIMD group, the threads with shorter edge lists have to wait for those with longer edge lists to finish executing the *for* loop.

We propose that in each SIMD unit, some scalar cores working collaboratively with SIMD cores can be beneficial particularly for such graph traversal patterns. For instance, the major flow control for each vertex can be executed on the scalar unit. The *for* loop of expanding the edge list can be scheduled across SIMD lanes for the range of loop indices. In this case, instead of each SIMD lane processing the edge list for one vertex, each lane is now processing one edge in the

edge list for the same vertex. Figure 12 illustrates this idea and compares two different schedules on architectures with and without scalar units.

B. Heterogeneous SIMDs

We have shown that these graph applications show significant SIMD underutilization. It is common that only a few threads are active in a wavefront. We discussed in Section VI-C that low utilization can be caused by vertices with different degrees processed by the threads. One potential solution is that, when only a few threads are active, these threads can be scheduled on the scalar unit for execution. On the other hand, if we use the scheduling policy described in Section VII-A, it is still common that the average degree of the vertices is smaller than the SIMD width (e.g., one to five in Figure 5 and one to eight in Figure 4). Today's GPUs offered by NVIDIA and AMD typically use SIMD widths of 16 and 32. Based on our characterization, it might be useful for vendors to consider including additional CUs with narrower SIMDs (e.g., four to eight). Similarly, long wavefronts can be time-sliced to execute on these narrow SIMD units or wide SIMDs as usual. Prior research has also investigated using a dynamic approach adapting SIMD width [30].

C. Resource Management

Our analysis shows that these applications demonstrate diverse phase behaviors during program execution. Even for a single application, phase behaviors may vary with different inputs. For some applications, some iterations process a small amount of work while the other iterations process a large amount of work. For instance, *DJK* begins computation from a single source vertex and gradually explores the vertex frontier level by level. From both performance and energy efficiency perspectives, GPU offload is beneficial when there is sufficient work to leverage the GPU's parallel resources. Therefore, for these graph applications, it is interesting to consider how to partition work of different phases across the CPU and the GPU. This approach would be more efficient on systems with integrated GPUs (e.g., AMD APUs) in which the CPU and the GPU share memory. This configuration allows more efficient task distribution across devices without costly PCIe data transfers. A good example is a face detection implementation optimized for an APU [38]. Power management techniques for phases that underutilize compute resources are another interesting research direction.

VIII. Related Work

There are several benchmark development efforts for heterogeneous systems, including the Rodinia [12], [13], Parboil [41] and SHOC [15] benchmark suites. A majority of the included benchmarks are regular; only a few are graph or tree-based algorithms, for instance, *breadth-first search* from Parboil and Rodinia and *B+tree* from Rodinia.

Burtscher et al. [10] performed a quantitative study of irregular programs on GPUs. The study also included other

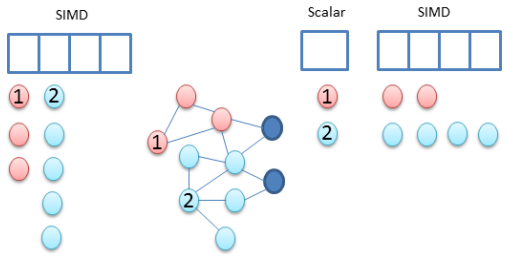

Fig. 12. Two types of scheduling with GPU scalar and SIMD units

irregular benchmarks, e.g., data compression, pointer-to analysis. Unlike their work, *Pannotia* is focused primarily on evaluating and understanding graph algorithms on the GPU platform. We also introduce some representative applications from web and social network analysis domains, which have not been covered previously.

Other works studied how to accelerate graph algorithms efficiently on GPUs. Harish et al. [19] parallelized several graph algorithms on the GPU using NVIDIA's CUDA. Merrill et al. [32] proposed an optimized breadth-first search implementation with *prefixsum* on both single and multiple GPU nodes. Burtscher et al. [11] presented an efficient CUDA implementation of a classical tree-based *Barnes-Hut* n-body algorithms. Several works [23], [35] have studied the approaches to optimize *CCL* problems on the GPU platform. In addition, Vineet et al. [43] implemented a minimum spanning tree (MST) algorithm for GPU acceleration. Sengupta et al. [39] studied the scan primitives useful for programming irregular applications.

In contrast to these individual studies, we introduce new graph algorithms and their OpenCL implementations on the GPU and provide an overall analysis of these applications by providing new insights into different performance bottlenecks, characterizing program behavior across different inputs, and analyzing similarity among these graph workloads with hierarchical clustering.

Prior works also benchmark a variety of applications on the GPU platform. Che et al. [13] characterized the Rodinia benchmark suite on the GPU platform and compared its multithreaded version against the PARSEC benchmark. Kerr et al. evaluate a set of metrics for GPU workloads [24] and use them to analyze the behavior of GPU programs. Goswami et al. [17] compared NVIDIA SDK, Parboil, and Rodinia benchmarks using hierarchical clustering. Only a few targeted benchmarks in these analysis are graph-based applications.

Other works studied graph algorithms on non-SIMD architectures. For instance, Pregel [28] is a system to process large-scale graphs; their focused domain is web graph and various social networks. GraphLab [26] is a MapReduce-like API and framework designed for data mining with optimizations for graph algorithms. GraphChi [25] is a disk-based system for processing large graphs by breaking graphs into small parts and using a parallel-sliding window method.

IX. Conclusion and Future Work

This paper presents and characterizes *Pannotia*, a library of graph applications. The suite of applications is implemented in OpenCL and includes applications from a variety of graph domains. We primarily focus on graph applications because they present significant challenges on SIMD architectures with branch and memory divergence, and input-dependent load imbalance and parallelism.

We then conduct a characterization of these graph applications using hardware counters on an AMD GPU. We observe that this suite of applications demonstrates diversity in terms of parallel speedups, SIMD utilization and cache efficiency both across applications and across different phases within applications. We conduct a workload similarity study among these benchmarks and observe that even for a single application, different input data sets may show vastly different program behaviors. This is an important issue that must be considered by researchers for hardware and software designs.

We also discuss architecture features that may allow more efficient execution of these graph workloads on the GPU, including the use of scalar units for flow control and possible narrower SIMD units for efficient lane utilization.

Directions for future work include:

- Evaluating more diverse graph applications with more pointer-chasing and divide-and-conquer features, and those that update the original graph structure.
- Developing new metrics and visualization methods to improve analysis of workload behaviors, especially as a function of different inputs and application phases.
- Studying general software optimization techniques and primitives for these graph applications for better performance and programmability.
- Evaluating and optimizing *Pannotia* on a variety of platforms, including APUs, GPUs and many-core architectures from other vendors. Integrated GPUs pose unique opportunities for load balancing and work partitioning.

In addition, as we discussed, these graph applications share common characteristics (e.g. low arithmetic intensity, low data reuse, memory and branch divergence, and varying parallelism). It would be interesting to study how to design architectures specifically for this set of workloads and how to improve existing GPUs so they are better suited to graph applications.

Acknowledgments

We would like to thank the reviewers for their constructive comments and suggestions.

References

[1] AMD Radeon HD 7000 series counters. Web resource. http://developer.amd.com/tools/heterogeneous-computing/amd-app-profiler/user-guide/app-profiler-settings/.

[2] The 10th DIMACS Implementation Challenge Graph Partitioning and Graph Clustering. Web resource. http://www.cc.gatech.edu/dimacs10/.

[3] The 9th DIMACS Implementation Challenge Shortest Paths. Web resource. http://www.dis.uniroma1.it/challenge9/.

[4] AMD Accelerated Parallel Processing: OpenCL Programming Guide. Web resource. http://developer.amd.com/download/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf.

[5] AMD Graphics Core Next Architecture. Web resource. http://www.amd.com/us/products/technologies/gcn/Pages/gcn-architecture.aspx.

[6] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An open source software for exploring and manipulating networks. In *ICWSM*, May 2009.

[7] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *IISWC*, Sep 2008.

[8] C. Bienia and K. Li. Fidelity and scaling of the PARSEC benchmark inputs. In *IISWC*, Dec 2010.

[9] Ulrik Brandes. A faster algorithm for betweenness centrality. *J. Math. Sociol.*, 25:163–177, 2001.

[10] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In *IISWC*, Nov 2012.

[11] M. Burtscher and K. Pingali. An efficient CUDA implementation of the tree-based Barnes Hut n-body algorithm. In *GPU Computing Gems*, pages 75–92. Morgan Kaufmann, 2011.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, Oct 2009.

[13] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *IISWC*, Dec 2010.

[14] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2001.

[15] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable HeterOgeneous computing (SHOC) benchmark suite. In *GPGPU*, Mar 2010.

[16] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Workload design: Selecting representative program-input pairs. In *PACT*, Sept 2002.

[17] N. Goswami, R. Shankar, M. Joshi, and Tao Li. Exploring gpgpu workloads: Characterization methodology, analysis and microarchitecture evaluation implication. In *IISWC*, Dec 2010.

[18] NVIDIA CUDA Programming Guide. Web resource. http://developer.nvidia.com/object/gpucomputing.html.

[19] P. Harish and P. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *HiPC*, Dec 2007.

[20] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72, 2007.

[21] J. Cohen and P. Castonguay. Efficient graph matching and coloring on the gpu. http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0332-GTC2012-Graph-Coloring-GPU.pdf.

[22] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John. Measuring benchmark similarity using inherent program characteristics. *IEEE Trans. Comp,*, 55(6):769–782, 2006.

[23] O. Kalentev, A. Rai, S. Kemnitz, and R. Schneider. Connected component labeling on a 2D grid using CUDA. *J. Parallel and Dist. Comp.*, 71(4):615–620, 2011.

[24] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of PTX kernels. In *IISWC*, Oct 2009.

[25] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, Oct 2012.

[26] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, July 2010.

[27] M. Luby. A simple parallel algorithm for the maximal independent set problem. In *STOC*, May 1985.

[28] G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, June 2010.

[29] Matrix Market Format. Web resouce. http://math.nist.gov/MatrixMarket/formats.html.

[30] J. Meng, J. W. Sheaffer, and K. Skadron. Robust SIMD: Dynamically adapted simd width and multi-threading depth. In *IPDPS*, May 2012.

[31] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *ISCA*, June 2010.

[32] D. G. Merrill, M. Garland, and A. S. Grimshaw. Scalable GPU graph traversal. In *PPoPP*, Feb 2012.

[33] METIS File Format. Web resource. http://people.sc.fsu.edu/~jburkardt/data/metis_graph/metis_graph.html.

[34] GTGraph: A Suite of Synthetic Random Graph Generators. Web resource. http://www.cse.psu.edu/~madduri/software/GTgraph/index.html.

[35] V. M. A. Oliveira and R. A. Lotufo. A study on connected components labeling algorithms using GPUs. In *SIBGRAPI*, Aug 2010.

[36] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report SIDL-WP-1999-01204, Stanford Univerisity, 1999.

[37] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *ISCA*, June 2007.

[38] AFDS 2012 Phil Rogers Keynote: The programmer's guide to a universe of possibility. Web resource. http://hsafoundation.com/publications/.

[39] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *GH*, Aug 2007.

[40] Z. Shi and B. Zhang. Fast network centrality analysis using gpus. *BMC Bioinformatics*, 12(140), 2011.

[41] Parboil Benchmark suite. Web resource. http://impact.crhc.illinois.edu/parboil.php.

[42] The University of Florida Sparse Matrix Collection. Web resource. http://www.cise.ufl.edu/research/sparse/matrices/.

[43] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the GPU. In *HPG*, Jul 2009.