Graph Processing on GPUs: Where are the Bottlenecks?

Qiumin Xu, Hyeran Jeon, Murali Annavaram
Ming Hsieh Department of Electrical Engineering, University of Southern California
Los Angeles, CA
{qiumin, hyeranje, annavara} @usc.edu

***Abstract*—Large graph processing is now a critical component of many data analytics. Graph processing is used from social networking web sites that provide context-aware services from user connectivity data to medical informatics that diagnose a disease from a given set of symptoms. Graph processing has several inherently parallel computation steps interspersed with synchronization needs. Graphics processing units (GPUs) are being proposed as a power-efficient choice for exploiting the inherent parallelism. There have been several efforts to efficiently map graph applications to GPUs. However, there have not been many characterization studies that provide an in-depth understanding of the interaction between the GPGPU hardware components and graph applications that are mapped to execute on GPUs. In this study, we compiled 12 graph applications and collected the performance and utilization statistics of the core components of GPU while running the applications on both a cycle accurate simulator and a real GPU card. We present detailed application execution characteristics on GPUs. Then, we discuss and suggest several approaches to optimize GPU hardware for enhancing the graph application performance.**

I. INTRODUCTION

Large graph processing is now a critical component of many data analytics. Graphs have traditionally been used to represent the relationship between different entities and have been the representation of choice in diverse domains, such as web page ranking, social networks, tracking drug interactions with cells, genetic interactions, and communicable disease spreading. As computing is widely available at very low cost, processing large scale graphs to study the vertex interactions is becoming an extremely critical task in computing [22, 29, 32, 33]. Many graph processing approaches [2, 33] use the Bulk Synchronous Parallel model [37]. In this execution model, graphs are processed in synchronous iterations, called *supersteps*. At every superstep, each vertex will execute a user function that can send/receive messages, modify its value or modify values of its edges. There is no defined order in which the vertices are handled within each superstep, but at the end of each superstep all vertex computations are guaranteed to be completed.

Graph processing has been parallelized to run on large compute clusters using well-known cluster compute paradigms such as Hadoop [3] and MapReduce [19] that have been re-targeted to run graph applications. More recently, graph processing-specific computing frameworks, such as Pregel [33] and Giraph [2] have also been proposed. Pregel, for instance, relies on vertex-centric computing. An application developer defines a $vertex.compute()$ function which specifies the computation that will be performed at each vertex. The computation can be as simple as finding the minimum value of all the adjacent vertex values, or could be a more complex function that simulates how a protein may fold when interacting with amino acids [11].

Within each superstep, or within each MapReduce iteration, there is significant amount of parallelism as the same computation is performed across all vertices in the graph. Today this parallelism is exploited primarily through cluster-based computing where multiple compute nodes concurrently process different subgraphs. Given the loose synchronization demand and repeated computations on vertices, the Single Instruction Multiple Threads (SIMT) parallel execution model supported by graphics processing units (GPUs) provides new venues to significantly increase the power efficiency of graph processing. By mapping a vertex processing to a SIMT lane or a set of SIMT lanes called warp or wavefront, large graphs can be efficiently processed. Several studies showed how to efficiently map graph applications to GPUs [7, 23, 24, 26, 35, 38]. However, there have not been many characterization studies conducted to understand how graph applications interact with GPU-specific microarchitectural components, such as SIMT lanes and warp schedulers. To optimize GPUs for graph processing, understanding the the graph application's characteristics and the corresponding hardware behavior is important.

Che et al. [16] recently characterized graph applications while running on AMD Radeon HD 7950 GPGPU. They implemented eight graph applications in OpenCL and analyzed the hardware behaviors such as cache hit ratio, execution time breakdown, speedup over executing on CPU. In this paper, we collected 12 graph applications written in CUDA from various sources and execute them on NVIDIA GPU as well as a similarly configured cycle-accurate simulator. We then provide in-depth application characterization and analyze the interaction of the application with the underlying microarchitectural blocks through a combination of hardware monitoring with performance counters and software simulators. In order to highlight graph applications' unique execution behavior, we also ran a set of non-graph applications on GPUs and compared the behavior of the two sets of applications.

The followings are the contributions of this work:

- We compiled 12 graph applications written in CUDA. Not to be biased by a certain programming style, we acquired the applications from a broad range of sources. We measured various GPU architectural behaviors while running graph applications on real hardware. As the real machine profiler provides only

a limited set of hardware monitoring capabilities, we also used cycle accurate GPU simulator to understand the impact of warp schedulers, performance bottlenecks and load imbalance across SMs, CTAs and warps.

- To differentiate graph application's unique characteristics, we also executed a set of non-graph applications on the same platforms. We then compare and contrast the various performance and resource utilization measures.
- We discuss several design aspects that need to be considered in the GPU hardware design for more efficient graph processing.

The remainder of this paper is organized as follows. Section II explains the baseline GPU architecture. Section III describes our evaluation methodology and the graph and non-graph applications used in the experiment. Then, we characterize and analyze the GPU hardware behaviors by using the evaluation results in Section IV. We discuss possible hardware optimizations in Section V. Section VI describes the related work and we conclude in Section VII.

II. BACKGROUND

GPU: CPU-GPU heterogeneous computing has recently attracted attention. GPUs provide massive parallel processing power cooperating with CPU. As the host for the GPU device, CPU organizes and invokes application kernel functions that execute on a GPU. Communication between the CPU and the GPU is performed via PCI-Express bus. GPUs consist of a number of streaming multiprocessors, each comprising of simple processing engines, called CUDA cores in the NVIDIA terminology. For instance, NVIDIA Tesla M2050 consists of 14 streaming multiprocessors (SMs), each comprising 32 stream processor (SP) cores and 64 KB shared memory shared among the SPs in a SM. In M2050, up to 1536 hardware threads are supported by each SM and 21,504 threads are supported on the entire GPGPU.

The kernel function called by host CPU is divided into several independent thread blocks or cooperative thread array (CTA). Inside of a thread block, a set of threads (32 in M2050), referred to as a warp, is scheduled on the SM to run concurrently. A warp is a collection of threads that all run the same sequence of instructions but using different data operands. This execution model is referred to as single instruction multiple thread (SIMT) model and each thread within a warp has a dedicated set of execution resources which are referred to as SIMT lanes. GPUs provide a large, but slow off-chip global memory that can be accessed by all thread blocks and the CPU, while providing small but fast on-chip shared memories that are individually shared among threads in the same CTA.

III. METHODOLOGY

A. Graph applications

In this section, we briefly describe the benchmark suite of graph algorithms we evaluated in this study. Many of the benchmarks are collected from recent research papers which implemented state of art algorithms for a variety of graph processing demands.

- **Approximate Graph Matching (AGM)** is used to find maximal independent edge set in a graph. It has applications in minimizing power consumption in wireless networks, solving traveling sales person problem, organ donation matching programs, and graph coarsening in computer vision. The version of AGM used in this study is a fine-grained shared-memory parallel algorithm for greedy graph matching [20].
- **All Pairs Shortest Path (APSP)** is used to find the shortest path between each pair of vertices in a weighted graph. The standard algorithm for solving the APSP problem is the Floyd-Warshall algorithm [21]. Buluç *et al.,* showed that APSP problem is computationally equivalent to computing the product of two matrices on a semiring [13]. The APSP algorithm we selected uses this more efficient implementation.
- **Breadth First Search (BFS)** is a well known graph traversal algorithm. The parallel implementation of BFS is widely available. We use the version in Rodinia benchmark [17].
- **Graph Clustering (GCL)** is concerned with partitioning the vertices of a given graph into sets consisting of vertices related to each other. It is a ubiquitous subtask in many applications, such as social networks, image processing and gene engineering. GCL used in this paper refers to the implementation in [7] using a greedy agglomerative clustering heuristic algorithm.
- **Connected Component Labeling (CCL)** involves identifying which nodes in a graph belong to the same connected cluster or component [24, 27, 34]. It is widely used in simulation problems and computer vision. CCL here refers to implementation of prior work using label equivalence method [24].
- **Graph Coloring (GCO)** partitions the vertices of a graph such that no two adjacent matrices share the same color. There are several known applications of graph coloring such as assigning frequencies to wireless access points, register allocation at compile time, and aircraft scheduling. Graph coloring is NP-hard, therefore, a number of heuristics have been developed to assign colors to vertices, such as first fit, largest degree order and saturation degree order. GCO selected in this study uses a parallelized version of first fit algorithm presented in [23].
- **Graph Cuts (GCU)** partitions the vertices of a graph into two disjoint subsets that are joined by at least one edge. It can be employed to efficiently solve a wide variety of low level computer vision problems, such as image segmentation, stereo vision, image restoration. The maxflow / mincut algorithm to compute graph cuts is computationally expensive. The authors in [38, 39] proposed a parallel implementation of the push-relabel algorithm for graph cuts which achieves higher performance, which is used in this study.

- **Maximal Independent Set (MIS)** finds a maximal collection of vertices in a graph such that no pair of vertices is adjacent. It is another basic building block for many graph algorithms. MIS here refers to the standard cusp implementation [4] using Luby's algorithm [31].
- **Minimum Spanning Tree (MST)** finds a tree that connects all the vertices together with minimum weight. It has applications in computer networks, telecommunication networks, transportation networks, water supply networks and smart electrical grid management. This benchmark computes a minimum spanning tree in a weighted undirected graph using Boruvka's algorithm [14].
- **Page Rank (PR)** is an algorithm used by Google to rank websites. PageRank is a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page. The implementation uses Mars MapReduce framework on GPU [25, 35].
- **Survey Propagation (SP)** The satisfiability (SAT) problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. Survey propagation [12] is heuristic SAT-solver based on Belief Propagation(BP), which is a generic algorithm in probability graph. It is implemented in LonestarGPU [14].
- **Single Source Shortest Path (SSSP)** computes the shortest path from a source node to all nodes in a directed graph with non-negative edge weights by using a modified Bellman-Ford algorithm [10, 14].

B. Non-graph applications

In addition to the above described graph applications, we evaluated 9 benchmarks from non-graph application domains from the Rodinia benchmark [17] and NVIDIA SDK [5] suites. The benchmarks are selected to cover a wide range of application domains: LU decomposition (LUD), matrix multiplication (MUL) are dense liner algebra applications, which manipulate dense matrices. Discrete Cosine transform (DCT) and Heartwall (HW) are image processing applications. Hotspot (HS) is a physical simulation application which is used to do thermal simulation to plot the temperature map of processors. We also included statistics and financial applications such as Histogram (HIST) and Binomial options (BIN). Finally, we included widely used parallel computing primitives scan (SCAN) and reduction (RDC) to represent a large range of applications from parallel application developers.

C. Experiment environment

Each of the selected applications were written in CUDA. We only modified the makefiles to change the compilation flags for collecting specific data from GPU hardware that we will describe shortly. We ran these applications both on the native hardware as well as on a cycle accurate GPU simulator. Hardware measurements are performed on Tesla M2050 GPGPU, which has 14 CUDA SMs running at 1.15GHz. Meanwhile, simulation studies are performed on GPGPU-Sim [9] to collect detailed runtime statistics that were not possible in hardware measurements. As shown in Table I, GPGPU-Sim is configured with Tesla C2050 parameters, which has the same architecture as M2050. The only difference between the two different GPGPUs is that they have different heat sinks. We run multiple experiments with different input sets for each application as shown in Table II. The input sets differ in size and categories; some of the input sets are obtained from Dimacs [8], some are obtained from Florida Matrix Collection [18] and others come with original application.

GPU			
Model	Tesla M2050 [30]		
Core	14 CUDA SMs@1.15GHz		
Memory	2.6GB, GDDR5@1.5GHz		
Comm.	PCI-E GEN 2.0		
Simulator			
Version	GPGPU-Sim v3.2.2 [9]		
Configs	Tesla C2050		
Core	14 CUDA SMs@1.15GHz		
Memory	GDDR5@1.5GHz		
L1D	16KB	RF	128KB
Const	8KB	Shared	48KB
L2D	786KB		

TABLE I: Experiment environments

IV. RESULTS AND ANALYSIS

A. Kernel execution pattern

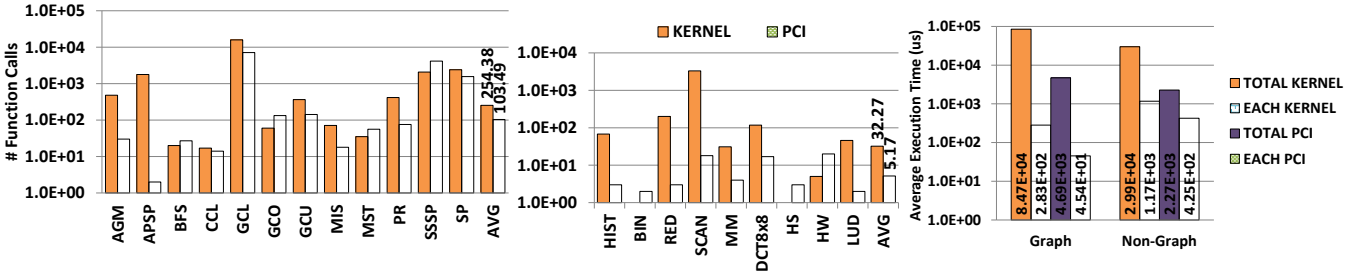
First of all, we measured the number of kernel function calls that were invoked by the CPU on the GPU when executing on native hardware. In Figure 1(a) and (b), the bar charts named *KERNEL* indicate the total number of kernel functions invoked during the execution of each graph and non-graph application. The average number of kernel invocations is nearly an order magnitude higher in graph applications (about 300 invocations) compared to non-graph applications (about 25 invocations). Graph applications require frequent synchronizations: for instance, after each superstep in BSP model all vertex computations must return to CPU for synchronization before starting the next superstep. Thus graph applications require frequent CPU interventions to provide synchronization capability for both BSP and MapReduce-style graph computations.

The amount of computation done per each kernel invocation is significantly smaller in graph applications than non-graph applications. The first two bars in Figure 1(c) show the total execution time spent while executing all the kernel invocations (labeled *TOTAL KERNEL*) and the average amount of time spent per kernel invocation (labeled *EACH KERNEL*). In graph applications the per kernel execution time is only 24% of the per kernel time spent in non-graph applications. Thus non-graph applications, at least the applications that we evaluated, execute relatively large functions on each kernel invocation from CPU and require fewer CPU interventions.

One of the negative side effect of communicating frequently with CPU is that in current systems CPU and GPU communicate via PCI interface. Thus even short messages require long latencies to communicate over PCI. In Figure 1(a) and (b), the bar charts named *PCI* show the number of cudaMemcpy function calls that uses PCI to transfer between CPU and GPU. Graph applications interact with CPU nearly

Name	Input Set	Size	Name	Input Set	Description	Name	Input Set	Size
AGM	coAuthor	300K*300K	APSP	1K	1K-V 6K-E	BFS	4096	4K
	In2004	13M*13M		4K	4K-V 24K-E		65536	64K
GCL	email	1K*5K	CCL	logo	2.7MB		1MW	1M
	coAuthor	300K*300K		Trojan	5.5MB	GCU	flower	4.7MB
	belgium	1.4M*1.4M	MST	rmat12 [8]	4K-V 165K-E		person	4.7MB
GCO	hood [18]	215K-V 5.2M-E		fla [8]	1M-V 2.6M-E	PR	wiki	7K-V 104K-E
	pwtk [18]	212K-V 5.7M-E	SP	17K	4K-V 17K-E		Edge14	16K-V 256K-E
MIS	128	128*128		42K	10K-V 42K-E	SSSP	fla [8]	1M-V 2.6M-E
	512	512*512					rmat20 [8]	1.1M-V 45M-E

TABLE II: 12 Graph applications collected from various sources(V: Vertices, E: Edges)



(a) (b) (c)

Fig. 1: Kernel function invocation count for (a) graph and (b) non-graph applications. (c) Average function execution times.

20X more frequently than non-graph applications. Graph applications tend to transfer data almost once per two kernel invocations while the non-graph applications execute average of ten kernels without extra data transfer when multiple kernels are executed. The primary reason for large communication overhead is that graph applications use kernel invocation as a global synchronization. Whenever an SM finishes its processing on the assigned vertices, the next set of vertices to process is determined only when all the other SMs finish their work assigned in the kernel because there can be dependencies among the vertices processed in multiple SMs. However, as GPUs do not support any global synchronization mechanism across the SMs, the graph applications are typically implemented to call a kernel function multiple times to use the kernel invocation as a global synchronization.

In addition to synchronization overheads, once a kernel function is complete the output data needs to be properly redeployed in the GPU memory so that the vertices that are processed in the next kernel can read their data appropriately. At the end of each kernel execution some vertices may simply stop further computations since they reach a termination condition. However, these termination condition checks are done at the end of each superstep by making sure no other vertex has sent a message to the given vertex. Thus the vertex computation can only be terminated by CPU after it has processed the results from the current superstep from all SMs. The net result of all these frequent CPU-GPU interactions is that the total time spent on PCI transfers is higher in graph applications, as can be seen in Figure 1(c) bar labeled *TOTAL PCI*. Since graph applications invoke many more PCI transfers but each call only transfers smaller amount of data, the time per each PCI transfer is 10X smaller than non-graph applications as can be seen in Figure 1(c) (bar labelled Each PCI).

B. *Performance bottlenecks*

In this section we focus on where the performance bottlenecks are while executing the selected applications on GPUs.

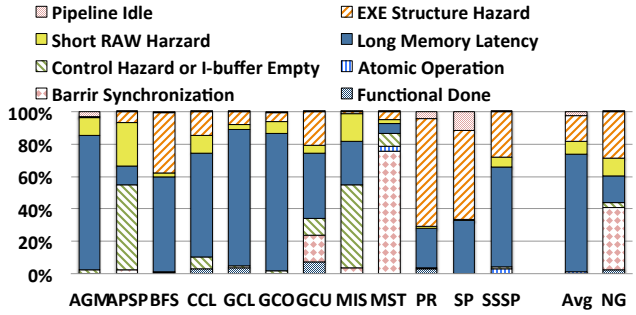
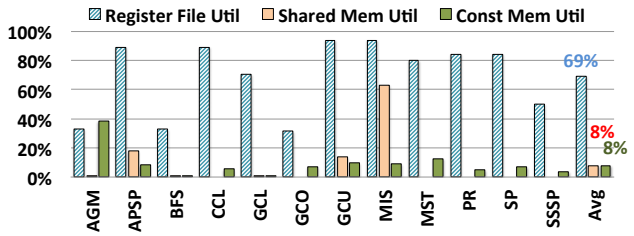


Fig. 2: Breakdown of reasons of pipeline stalls

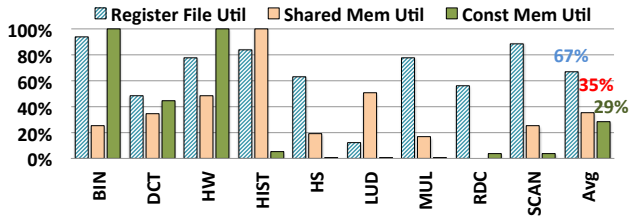
For this purpose we monitored pipeline stalls in our GPGPU-Sim simulator and analyzed for each pipeline stall what was the primary reason for the stall. Figure 2 shows the breakdown of various pipeline stall reasons. Note that the warp scheduling policies on GPUs allow multiple warps to be concurrently alive in various stages of the pipeline. Hence, in a given cycle, each warp could stall in a pipeline stage for different reasons: several of the warps could wait on long memory latency operations, while others could wait on barrier synchronization. We weight individual contribution of each pipeline stall reason by the number of warps stalled due to that reason. Due to space constraint, in this figure we plot all the graph applications individually but for non-graph application we simply plot the average of all the 9 non-graph applications in the last bar in the figure (labeled as *NG* on X-axis). On average, long memory latency is the biggest bottleneck that causes over 70% of all pipeline stalls in graph applications. The execution structure hazard and short RAW data dependency are the second and third most prominent bottlenecks. On the other hand, the non-graph applications exhibit different distribution of pipeline stall bottlenecks. The execution structure hazard is the biggest performance bottleneck. Graph applications by nature work with large datasets, and hence they are likely to experience

higher memory related stalls. Apart from the larger dataset size, CPU has to transfer data to GPU frequently and re-deploy vertex data, potentially at different locations, between two successive supersteps (or kernel calls). Hence, the data cached from one superstep is unlikely to be useful in the second superstep. As we will show later with cache miss statistics, in fact, graph applications suffer from higher miss rates. Due to the ineffective cache usage, the impact of long memory latency is higher in the graph applications than in the non-graph applications.

C. SRAM resource sensitivity



(a) Graph application



(b) Non-graph application

Fig. 3: Utilization of SRAM structures (register file, shared memory, and constant memory)

We collected the demand on SRAM resources by graph and non-graph applications from our simulation infrastructure. The three largest SRAM resources on GPUs are register file, shared memory and constant memory. As listed in Table I, in the simulated GPU configuration, the size of the register file, shared memory, and constant memory are 128KB, 48KB, and 8KB, respectively. Figure 3 shows the utilization of register file, shared memory, and constant memory per SM as a percentage of the total size. We used a compiler option -*Xptxas="-v"* to collect the usage of the three SRAM structures per each thread in an application. Then, the total usage per SM is calculated by using the number of concurrent CTAs per SM and the total number of threads within a CTA.

Across all the graph applications, the register file is the most effectively leveraged SRAM structure among the three. Graph applications use only 8% of the shared memory and constant memory. When we compare with non-graph applications that are plotted in Figure 3(b), it is clear that the graph applications tend to use the shared memory and the constant memory ineffectively.

It is important to provide a brief overview of how CUDA applications use global and shared memory. Shared memory is smaller and faster memory that can be shared by all the threads that are running within a CTA, while global memory is the large but slow memory that can be accessed by all SMs. Unlike traditional CPUs, GPU put the burden on applications to explicitly manage the usage of shared and global memory. If an application wants to use the shared memory then it must first execute a special move instruction to move the data from global memory to shared memory. Once the data is in the shared memory then the application can issue a second load instruction to bring the data into the execution unit. If there is not enough reuse of data then moving data from global memory to shared memory and then to the execution lanes actually consumes more time than simply loading data directly from global memory. Thus in the absence of sufficient data reuse, shared memory access only increases the memory access time as well as the instruction count as the data in the global memory needs to be loaded to the shared memory first by a load instruction and then another load instruction should be executed to get the data from the shared memory to the register. Therefore, in the relatively short kernel functions that are used in graph applications, it is hard to effectively leverage the shared memory. Thus graph applications do not try to exploit the shorter latency shared memory and instead simply load data from global memory. This memory usage statistic also explains why the main performance bottleneck of graph applications is the long memory latency as discussed in the previous section.

Constant memory is a region of global memory that is cached to the read-only constant cache. Given that GPU's L1 cache size is relatively small, maintaining some repeatedly accessed read-only data in constant cache helps to conserve memory bandwidth. However, constant cache is typically the smallest among the SRAM structures embedded in the GPU die as specified in Table I. Therefore, to gain performance benefits from using constant cache, programmers should carefully decide which data to store in the constant memory. If the data structure is too big, it may not benefit from using constant memory due to frequent constant cache replacements. Such data maybe better accessed directly from global memory. In the large graph processing applications where giga bytes or tera bytes of data are processed, it is hard to fit the data structures in the small constant cache. Thus, graph application developers is less inclined to use constant memory.

We also measured the performance impact of L1 and L2 caches. The default per SM L1 and per device L2 cache sizes are 16KB and 768KB, respectively. We also evaluated the impact of using different cache sizes, no L1 + 768KB L2, 32KB L1 + 1.5MB L2, 64KB L1 + 3MB L2, and finally an extreme data point of 4MB L1+196MB L2. Obviously the last design option was explored as a near limit study of the importance of very large cache. As the performance impact of caches can depend on the input size, we measured the performance impacts by varying the inputs for each application. For example, *AGM_coAuthor* and *AGM_in2004* are the executions of AGM but using different inputs; the input size is listed in Table II. Figure 4 shows the normalized IPC under various cache configurations that were listed above. The Y-axis is normalized IPC over the IPC of default cache configuration. Figure 5 shows the cache miss rate during the kernel execution under the default cache configuration but with varying input size.

As can be seen in Figure 5, almost all the graph applications have fairly high L1 cache miss rates (average of

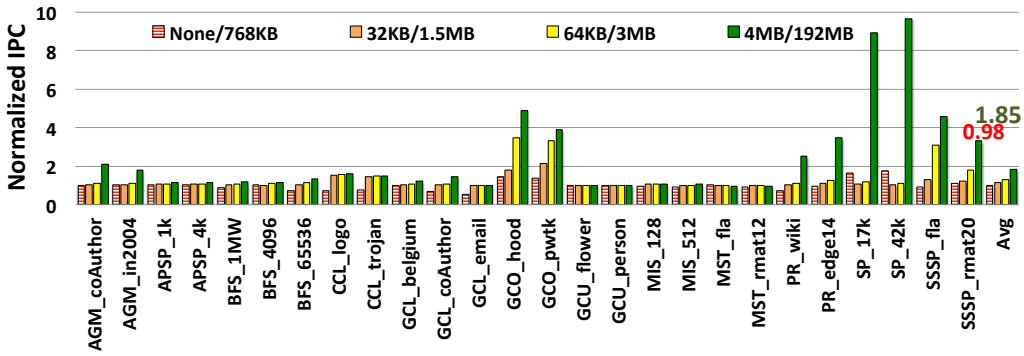


Fig. 4: Normalized IPC w.r.t. cache size

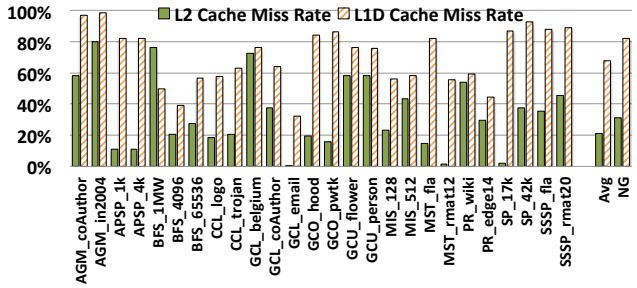
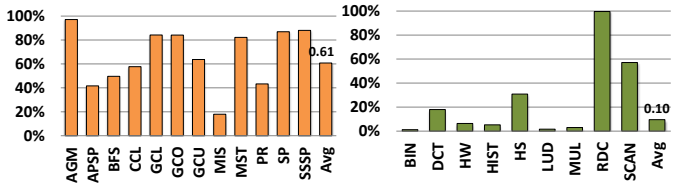


Fig. 5: Cache miss rate



(a) Graph application (b) Non-graph application

Fig. 6: L1 cache misses as a fraction of total accesses to L1 cache and shared memory

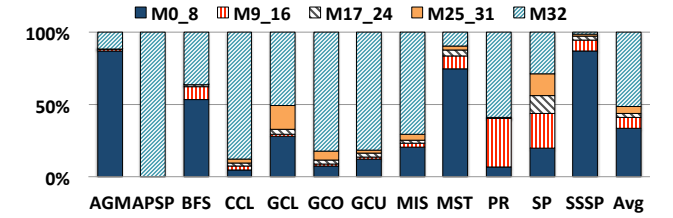
70% L1 accesses encounter misses). Given such an extremely high cache miss rate, we conducted a study without any L1 cache while keeping the L2 cache size at 768KB per device; we measured the performance without using L1 cache as plotted in the first bar charts named *None/768KB* in Figure 4. Interestingly, the IPC difference between the default and zero L1 cache configurations is only 2%. This means that L1 cache is entirely ineffective for graph processing.

We then measured the performance by doubling the size of both caches with L1 cache size reaching up to 4MB. The IPC improvement of larger cache size is quite small on most applications, except for applications such as SP and GCO that do take advantage of larger L1 cache, as can be seen in Figure 4. Even with a 4MB L1, that is plotted in the last bar named *4MB/192MB*, most of the applications derive small IPC increase. The reason for this ineffectiveness of caches can be inferred from the fact that between two kernel invocations, the CPU has to do memory transfer on GPU memory. Hence, each kernel invocation essentially looses any cache locality that was present at the end of prior kernel invocation.

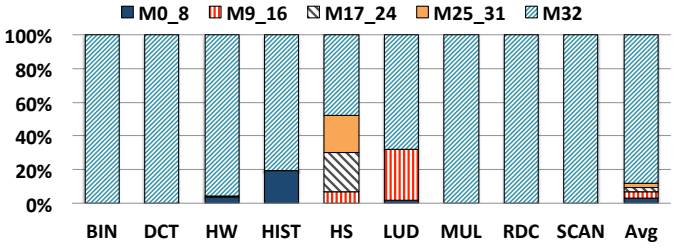
Interestingly, graph applications derive lower cache miss rate than non-graph applications as plotted in the last two bar charts in Figure 5, which is inconsistent with the findings of this paper. However, we found that non-graph applications' active use of shared memory decreases the total accesses to L1 cache, thereby the cache miss rate becomes relatively high. Recall that once a data is loaded from global memory to shared memory, the application reads the data from shared memory and never accesses global memory for the data. As a result, L1 cache encounters a cold miss while loading the data from global memory to shared memory at the first read on the data and never experience hit on the corresponding cache line. Note that loading a data from global memory to shared memory is not treated specially by the GPU hardware but is implemented by using a load instruction that reads global memory and writes the loaded data back to a register and a store instruction that stores the register value to the shared memory address. Therefore, L1 cache is accessed while executing the load instruction. However, once a data is stored to the shared memory, L1 cache is no more accessed for the data. This fact becomes obvious once we plot the cache miss rate as a fraction of the total accesses to shared memory and L1 cache combined. Figure 6 plots this data for both graph and non-graph applications. It is clear that this metric shows a vastly lower cache miss rate for non-graph applications due to their overwhelming number of shared memory accesses.

D. SIMT lane utilization

We also compare the SIMT lane utilization of graph and non-graph applications. Figure 7 shows the breakdown of SIMT lane utilization while running the applications. The bar labeled *M0_8* plots the fraction of instructions that are executed on at most 8 SIMT lanes. Similarly, *M9_16* shows the fraction of instructions that are executed on 9 to 16 SIMT lanes and so on. *M32* denote the portion of instructions that fully utilize the available 32 SIMT lanes. In all the graph applications except APSP, as shown in Figure 7(a), the SIMT lane utilization varies considerably. For example, in SSSP, over 87% of instructions are executed by only 0 to 8 SIMT lanes, which means the remaining 24 SIMT lanes are idle during the 87% of the execution time. On the other hand, non-graph applications plotted in Figure 7(b) tend to use SIMT lanes more effectively. Five among nine non-graph applications used in the experiment used all the 32 available SIMT lanes 100% of the warp execution time.



(a) Graph application



(b) Non-graph application

Fig. 7: SIMT lane utilization

The nature of graph applications leads them to have variable amounts of parallelism during their execution. The vertices in a graph have different degrees (i.e. the number of edges). One typical way of graph application implementation is to run a loop for each vertex to process one edge per iteration. If each vertex is processed by one SIMT lane so that multiple vertices are processed by a warp in parallel, then the number of iterations executed by each SIMT lane varies as the degree of each vertex varies. This leads to a significant diverged control flow. Thus the SIMT lane utilization varies significantly in graph applications.

E. Execution frequency of instruction types

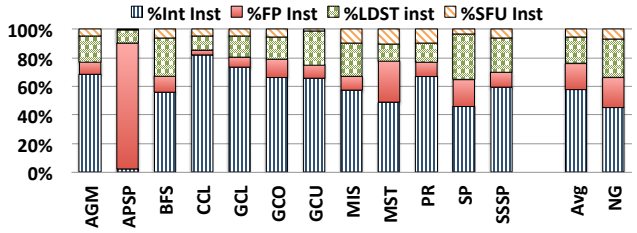


Fig. 8: Execution frequency of instruction types

Figure 8 shows the instruction type breakdown among the executed instructions. In almost all the graph applications, the dominant instruction type is integer instruction. The memory instruction is the second most frequently executed instruction type. Similar pattern is also found from non-graph applications that is shown in the last bar chart named *NG*. Thus, the execution time differences between graph and non-graph applications are not influenced by the instruction mix, rather the memory subsystem plays a significant role in these differences.

F. Coarse and fine-grain load balancing

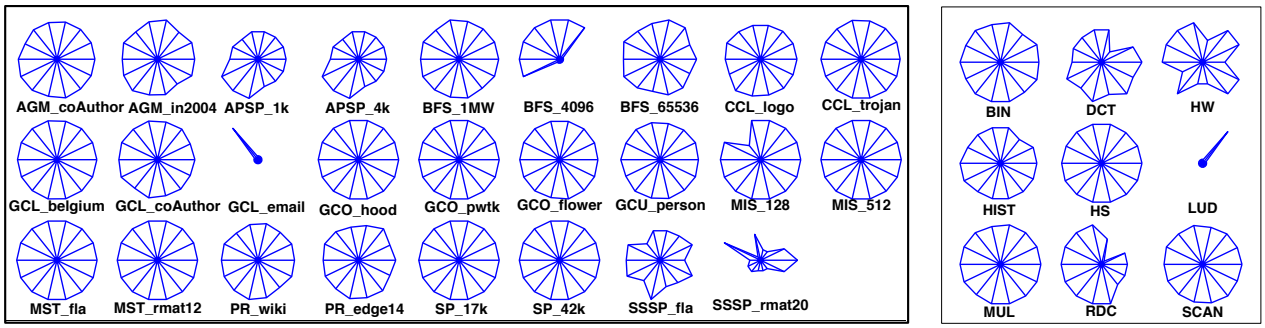
We measured load balancing in two ways: coarse and fine-grain. We measured coarse-grain load balancing as the number of CTAs assigned to each SM. For the fine-grain load balancing, we collected two metrics. The first metric is the execution time difference across CTAs. Since each CTA has different amounts of computation based on the number of vertices and edges processed, the execution time of CTAs can vary. The second fine-grain metric measures the execution time variance across warps within a CTA. The execution time variations of warps and CTAs can have significant negative impact on performance due to GPGPU execution model constraints. A kernel can be terminated only when all the assigned CTAs finish their execution. Likely, a CTA's execution can only be finished when all the warps within the CTA finish their work. Therefore, the performance is highly dependent on few warps or CTAs that have long execution time. Hence, coarse and fine-grain load balancing is critical for performance in GPGPUs.

Figure 9 shows the degree of balancing in the CTA distribution across the SMs. From the center of each circle, a line stretches to one of 14 directions to indicate there is at least one CTA in one of the 14 SMs in the GPGPU. The length of the line indicates the number of CTAs assigned to the SM. If all 14 SMs are assigned the same number of CTAs (well balanced), the chart forms a perfect circle. Otherwise (unbalanced load distribution), the circle has a distorted shape. Due to space limitation, we only present the load balancing graphs of the last kernel executed.

The SM level imbalance shown in Figure 9 depends on input size and program characteristics. Let's assume there are m SMs, maximum n CTAs can be assigned to one SM. The default CTA scheduling policy is round robin. If a kernel to be scheduled fits in exactly $m * n$ CTAs, there would be exactly n CTAs assigned per SM at the start of kernel execution. Such a perfect balance is seen in GCO, MST and SP benchmarks which have perfect circles. On the other hand, if a kernel to be scheduled has less than $m * n$ CTAs, CTAs would be assigned unevenly across SMs. For example, RDC has 64 CTAs in total and can schedule maximum 6 CTAs per SM: it assigns 5 CTAs on 8 SMs and 4 CTAs on the remaining 6 SMs.

The last case is when the number of CTAs in a kernel far exceeds the $m * n$ CTAs that can be scheduled at the start of a kernel execution. For such large kernel, CTAs would be initiated continuously onto the SMs after a previously scheduled CTA finishes execution. In such a scenario, there are two reasons that play opposing roles in balancing CTA assignments. On one hand, as the number of CTAs increase, there is a higher likelihood to assign similar number of CTAs per SM. Typically the number of CTAs created per kernel is a function of the input size. Large inputs lead to more CTAs and hence the likelihood of balancing CTA assignments per SM also increase. For example, BFS shows more balanced circles as the input size increases from 4K (BFS_4096) to 64K (BF_65536) and to 1M (BFS_1MW). Similarly, GCL also has more balanced circles when the input size is bigger (email is the smallest input set and belgium is the largest).

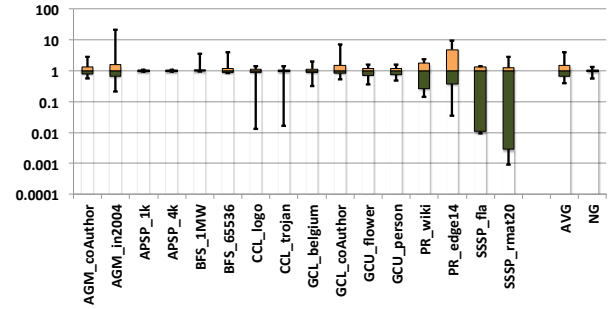
There is an opposing force to achieving balance as the number of CTAs increase. If different SMs complete their assigned CTAs after different amounts of time, the scheduler will assign more CTAs to an SM that executes faster. Thus the imbalance in CTA assignment increases as the execution time imbalances increase between CTAs. Figure 10(a) shows the execution time variance across CTAs in box plot. The lowest



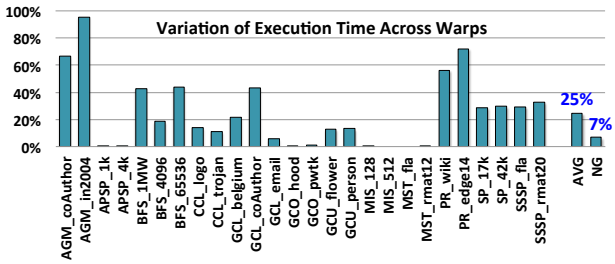
(a) Graph application

(b) Non-graph application

Fig. 9: Coarse grained load distribution: # assigned CTAs across SMs



(a)



(b)

Fig. 10: Fine grained load distribution: (a) execution time variance across CTAs and (b)coefficient of execution time variation across warps within a CTA

point in the error bar indicates the minimum CTA execution time while the highest error bar indicates the maximum CTA execution time. The box part goes from the first quartiles to the median and then to the third quartiles. All the execution times are normalized to the respective median execution time. We only included benchmarks which executed more than 4 CTAs in one kernel invocation, to calculate the variance across CTAs. In general larger input size increases the execution time variation. For instance, in SSSP more nodes need to be searched in the longest path as we increase the input size. Therefore, SSSP_rmat20 has more distorted shape than SSSP_fla in Figure 9. Furthermore, applications that exhibit more warp divergence also have higher execution time variance at the CTA level. For instance, SSSP, AGM, and PR have the highest CTA execution time variation, while APSP has the smallest variation. We found this corresponds to the fact that in Figure 7(a) that SSSP, AGM and PR are more divergent, while APSP is well parallelized with full warp utilization.

Figure 10(b) shows the variation of execution time across warps within a CTA. We present its coefficient of variation $\frac{\sigma}{\mu}$, where σ is the standard deviation and μ is the average execution time (averaged over all warps). The standard deviation is normalized over the averaged execution time to compare the variation across different benchmarks. AGM and PR show more than 50% of variation, which indicates that their standard deviation is more than half of the averaged execution time. On the other hand, some other benchmarks, like APSP, GCO, MIS and MST do not show large variations. GCO, MIS and MST only have 1, 2 and 4 warps in a CTA respectively. Therefore, the variation should not be high. On the other hand, as we mentioned previously, APSP is implemented using an optimized matrix multiply operation. Therefore, execution time variation for warps within CTAs is not high. Compared with non-graph applications, graph applications show higher variation across the warps, 25% versus 7%.

G. Scheduler Sensitivity

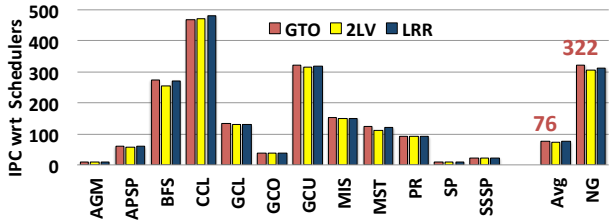


Fig. 11: Performance w.r.t. scheduler

Finally, we explored the scheduler impact on the graph application performance. Figure 11 shows the instructions per cycle (IPC) when three different warp scheduling algorithms are used. As we cannot change the warp scheduler in the real machine, we used GPGPU-Sim for this experiment. GTO is a greedy algorithm that issues instructions from one warp until the warp runs out of ready instructions. 2LV uses two level scheduler in which the warps in the ready queue issue instructions until they encounter the long latency memory instructions. Once a warp encounters a long latency memory instruction, then the warp is scheduled out to the pending queue. LRR is a simple round robin algorithm. Among the three algorithms, GTO derived slightly better performance but the performance difference is not significant. Due to poor

memory performance and divergence issues, as explained in earlier sections, graph applications have significantly lower IPC than non-graph applications.

V. DISCUSSION

In this section, we discuss some potential hardware optimizations for efficient graph processing.

A. Performance bottleneck

Based on the quantitative data shown, graph applications tend to execute kernel and data transfer functions more frequently than non-graph applications. The frequent kernel invocations lead to ineffective use of caches as well. Therefore, the performance overhead due to PCI calls as well as long latency memory operations is higher in the graph applications than in the non-graph applications.

There are two possible solutions to resolve this issue. First of all, the unified system memory that can be accessed by both CPU and GPU will be very helpful to reduce the performance overhead due to frequent data transfers between CPU and GPU. Recently, AMD announced support for unified memory access [1, 6]. AMD's proposed heterogeneous uniform memory access (hUMA) allows heterogeneous processor cores such as CPU and GPU to use the same physical memory. If the unified main memory is used, CPU and GPU can communicate with each other by using a simple memory copy operation rather than using PCI-E bus transmission. The memory copy can be conducted within the system memory and hence, the performance can be significantly improved.

The second solution is to actively leverage the underutilized SRAM structures such as cache and shared memory for reducing the overhead of long latency memory operation. As our evaluation showed, cache and shared memory are not effectively leveraged in graph processing. Over 60% of the L1 cache accesses encounter misses. The miss rate is not significantly reduced even when larger cache is used. Also, only 8% of shared memory is used in graph applications. These results imply that the data reuse in graph applications is rare. Therefore, we believe the cache and shared memory need to be used efficiently to handle data with limited reuse. One possible way is to use the two SRAM structures as large buffers for data prefetching. Recent studies [28] showed a memory prefetching approach for graph applications in GPGPU. They reused spare registers to store the prefetched data. However, for graph applications it would be easier to store the prefetched data in the cache or shared memory than in register files. By applying similar prediction algorithm used in the study [28], but using the cache and the shared memory, the overhead due to long latency memory operation can be reduced.

B. Load imbalance

According to our evaluation, coarse-grain load distribution in many graph application is well balanced once the input data is large enough. However, the fine grained load distribution that is measured across the CTAs, warps, and SIMT lanes exhibits higher levels of imbalance. As briefly explained earlier, vertices in a graph have different degrees. Therefore, the amount of tasks that needs to be processed by each vertex is different and hence the load is imbalanced in graph applications. Given that the CTA execution time is determined by the longest warp execution time and kernel execution time is only determined by the longest CTA execution time, such load imbalance can significantly degrade the overall performance.

This problem can be statically resolved by the programmer's effort. For example, if the programmer collects the vertices that have similar degrees and assigns them to the same CTA, the warp level load imbalance can be resolved. Due to the dynamic nature of graph processing it may be hard to find vertices that have similar degrees at every kernel invocation. Therefore, some sort of hardware support is necessary. The dynamic load monitoring and migration methods that are used in operating system domain [36] might be a solution. For example, if there is a CTA that processes high degree vertices and the kernel's termination is delayed because of the CTA's long execution, then migrating some of the warps assigned to the CTA to the other idle SMs might be helpful. Once the migration penalty is low enough, as the migrated warps can use the resources of the idle SM, the execution time can be balanced.

VI. RELATED WORK

Che et al. [16] recently showed a preliminary characterization of graph applications on a real GPGPU machine. They implemented eight graph applications in OpenCL and analyzed the hardware behaviors such as cache hit ratio, execution time breakdown, speedup over CPU version execution, and SIMT lane utilization while running those applications on AMD Radeon HD 7950. In this paper we not only run applications on hardware but we also use cycle accurate simulation infrastructure to provide deeper insight into the hardware behavior. For example, we can measure the impact of having no L1 cache to show that L1 cache is entirely ineffective in graph applications. Using detailed simulations we can also measure load imbalance metric at the warp, CTA and SM level. The load balancing statistics provide useful insights to optimize the CTA distribution across SMs.

Burtscher et al. [15] investigated performance impact of irregular GPU programs on NVIDIA Quadro 6000. They compiled eight irregular programs and compared the performance in several aspects with a set of regular programs. They basically measured two runtime-independent metrics, the control-flow irregularity and the memory-access irregularity at the warp level. The metrics are measured while varying the input size and optimizing the code itself. Most of the analysis provided by Burtscher is useful to optimize the application code and hence the purpose of the work is orthogonal to the focus of our research.

Che et al. [17] also conducted CUDA application characterization on NVIDIA GeForce GTX 280. The evaluation is similar to [16] but the domain of the evaluated applications is more general than [16]. The target application domain of Che's work [17] and our study is different and the focus of their characterization is software improvement, while this study focuses on understanding existing hardware bottlenecks in GPGPUs.

VII. CONCLUSION

Graph processing is a key component of many data analytics. There have been several studies to optimize graph

applications on GPU platform. However, there has not been a study that focuses on how graph applications interact with GPU microarchitectural features. To provide insights to the GPU hardware designers for more efficient graph processing, we measured several (micro)architectural behaviors while running a set of graph applications. To understand the graph application's unique characteristics, we also ran a set of non-graph applications and then compared the evaluation results. Based on the measurements, we also discuss the hardware level optimization points that can help enhance the performance of graph processing on future GPGPUs.

VIII. Acknowledgments

We thank anonymous reviewers for their valuable comments on this work. This work was supported by DARPA-PERFECT-HR0011-12-2-0020 and NSF-CAREER-0954211.

References

[1] "AMD Unveils its Heterogeneous Uniform Memory Access (hUMA) Technology," http://www.tomshardware.com/news/AMD-HSA-hUMA-APU,22324.html.

[2] "Apache giraph," https://giraph.apache.org/.

[3] "Apache hadoop," http://hadoop.apache.org/.

[4] "Cusp : A c++ templated sparse matrix library," http://cusplibrary.github.com.

[5] "Nvidia gpu computing sdk," https://developer.nvidia.com/gpu-computing-sdk.

[6] "The programmer's guide to the apu galaxy," http://developer.amd.com/wordpress/media/2013/06/Phil-Rogers-Keynote-FINAL.pdf.

[7] B. F. Auer, "Gpu acceleration of graph matching, clustering, and partitioning," in *Contemporary Mathematics*, vol. 588, 2013, pp. 223–240.

[8] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, "10th dimacs implementation challenge on graph partitioning and graph clustering," vol. 588, 2013.

[9] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*, April 2009.

[10] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, pp. 87–90, 1958.

[11] B. Berger, R. Singht, and J. Xu, "Graph algorithms for biological systems analysis," in *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2008, pp. 142–151.

[12] A. Braunstein, M. Mzard, and R. Zecchina, "Survey propagation: An algorithm for satisfiability," vol. 27, no. 2, 2005, pp. 201–226.

[13] A. Buluç, J. R. Gilbert, and C. Budak, "Solving path problems on the gpu," vol. 36, no. 5-6, June 2010, pp. 241–253.

[14] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *Proceedings of the 2012 IEEE International Symposium on Workload Characterization*, 2012, pp. 141–151.

[15] Burtscher, Martin and Nasre, Rupesh and Pingali, Keshav, "A quantitative study of irregular programs on gpus," in *Proceedings of the 2012 IEEE International Symposium on Workload Characterization*, 2012, pp. 141–151.

[16] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *Proceedings of the 2013 IEEE International Symposium on Workload Characterization*.

[17] 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 *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*.

[18] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[19] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, June 2008.

[20] B. Fagginger Auer and R. Bisseling, "A gpu algorithm for greedy graph matching," in *Facing the Multicore - Challenge II*, vol. 7174, 2012, pp. 108–119.

[21] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, pp. 345–, Jun. 1962.

[22] B. Gao, T. Wang, and T.-Y. Liu, "Ranking on large-scale graphs with rich metadata," in *Proceedings of the 20th International Conference Companion on World Wide Web*, ser. WWW '11, 2011.

[23] A. V. P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, and M. Hall, "Evaluating graph coloring on gpus," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, 2011, pp. 297–298.

[24] K. Hawick, A. Leist, and D. Playne, "Parallel graph component labelling with gpus and cuda," vol. 36, no. 12, 2010, pp. 655 – 678.

[25] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A mapreduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008, pp. 260–269.

[26] H. Jeon, Y. Xia, and V. K. Prasanna, "Parallel exact inference on a cpu-gpgpu heterogenous system," in *Proceedings of the 2010 39th International Conference on Parallel Processing*, 2010, pp. 61–70.

[27] O. Kalentev, A. Rai, S. Kemnitz, and R. Schneider, "Connected component labeling on a 2d grid using cuda," vol. 71, no. 4, 2011, pp. 615 – 620.

[28] N. B. Lakshminarayana and H. Kim, "Spare register aware prefetching for graph algorithms on gpus," in *Proceedings of the 2014 IEEE International Symposium on High Performance Computer Architecture*, 2014.

[29] K. Lee and L. Liu, "Efficient data partitioning model for heterogeneous graphs in the cloud," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 46:1–46:12.

[30] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, 2008.

[31] M. Luby, "A simple parallel algorithm for the maximal independent set problem," in *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, 1985, pp. 1–10.

[32] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 1, pp. 5–20, 2007.

[33] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010.

[34] V. M. A. Oliveira and R. A. Lotufo, "A study on connected components labeling algorithms using gpus," in *SIBGRAPI*, 2010.

[35] K. Shirahata, H. Sato, T. Suzumura, and S. Matsuoka, "A scalable implementation of a mapreduce-based graph processing algorithm for large-scale heterogeneous supercomputers," in *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing,*, May 2013, pp. 277–284.

[36] B. A. Shirazi, K. M. Kavi, and A. R. Hurson, Eds., *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.

[37] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, pp. 103–111, Aug. 1990.

[38] Vineet , V. and Narayanan, P. J., "Cuda cuts: Fast graph cuts on the gpu," in *Computer Vision and Pattern Recognition Workshops, IEEE Computer Society Conference on*, June 2008, pp. 1–8.

[39] Vineet, V. and Narayanan, P. J., "Cuda cuts: Fast graph cuts on the gpu," in *Technical Report, International Institute of Information Technology, Hyderabd.*