2015 IEEE International Parallel and Distributed Processing Symposium Workshop

Graph Coloring on the GPU and Some Techniques to Improve Load Imbalance

Shuai Che, Gregory Rodgers, Brad Beckmann and Steve Reinhardt
{Shuai.Che, Gregory.Rodgers, Brad.Beckmann, Steve.Reinhardt}@amd.com
AMD Research

***Abstract*—Graphics processing units (GPUs) have been increasingly used to accelerate irregular applications such as graph and sparse-matrix computation. Graph coloring is a key building block for many graph applications. The first step of many graph applications is graph coloring/partitioning to obtain sets of independent vertices for subsequent parallel computations. However, parallelization and optimization of coloring for GPUs have been a challenge for programmers.**

This paper studies approaches to implementing graph coloring on a GPU and characterizes their program behaviors with different graph structures. We also investigate load imbalance, which can be the main cause for performance bottlenecks. We evaluate the effectiveness of different optimization techniques, including the use of work stealing and the design of a hybrid algorithm. We are able to improve graph coloring performance by approximately 25% compared to a baseline GPU implementation on an AMD Radeon™HD 7950 GPU. We also analyze some important factors affecting performance.

I. Introduction

Graph processing is a critical workload in data analytics, and its use has gained popularity in the domains of business analytic, social network, knowledge discovery, engineering simulation and so on. Prior studies [4], [7] have shown that certain graph algorithms can achieve performance speedup on graphics processing units (GPUs) against CPUs.

Among different graph algorithms, graph coloring is an important building block in many larger graph applications, and has been widely used for graph partitioning and computation scheduling. In this paper, we use graph coloring as a case study to demonstrate important, common issues useful in improving performance of graph algorithms on GPUs. We choose graph coloring for several reasons. First, it presents load imbalance across GPU threads, a characteristic common in many GPU graph algorithms. For example, different GPU threads may be assigned to process different regions of a graph whose structure can be highly irregular. Imbalanced execution times across GPU threads limit the overall application performance. Second, program behavior keeps changing over time across different iterations, because the working set of coloring keeps changing. The resultant load distributions also vary over time. Therefore, static work-partitioning and preprocessing techniques (e.g., sorting) may not work well. The actual work distribution is hard to predict accurately before runtime due to the program's input-dependent feature.

We evaluate the effectiveness of using two software optimization techniques (i.e., work stealing and a hybrid algorithm) to optimize graph coloring. The observations and techniques discussed in this paper may apply to other irregular problems too (e.g., independent set problems). OpenCL™is used in this work, but our algorithms can be implemented with other programming models (e.g., OpenMP and C++ AMP).

This paper makes the following contributions:

- We implement a work-stealing version of graph coloring. It uses a work-queue model in which GPU workgroups attempt to balance loads dynamically by stealing vertices and their tasks among each other. We show that work stealing only provides limited performance improvement.
- We study a hybrid algorithm which takes advantage of intrinsic features of graphs. The new algorithm labels vertices with colors using a two-phase approach, by examining the local maximum with the degree value first and then with the randomized vertex value.
- We show the performance benefit compared to a baseline implementation. We conduct a detailed analysis of program behaviors from several perspectives (e.g., work distribution, the point to switch phases, and characteristics of using different types of graphs).

II. Background, Platform and Environment

In this section, we describe AMD's GPU architecture, OpenCL, and load imbalance present in GPGPU graph algorithms.

A. AMD GPUs and OpenCL

AMD Radeon HD 7000 series and later GPUs use the AMD Graphics Core Next (GCN) architecture [11]. The AMD GPU includes multiple SIMD compute units (CUs). Each CU contains one scalar unit and four vector units [2]. 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, a data cache for the scalar unit, a L1 data cache and a local data share (LDS) (i.e., software-managed scratchpad). All CUs share a single unified L2 cache. The GPU supports multiple DRAM channels.

OpenCL is a programming model to develop applications for GPUs and other accelerators [18]. In OpenCL, a host program launches a kernel with workitems over an index space (NDRange). Workitems are grouped into workgroups. OpenCL 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).

978-1-4673-7684-6/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPSW.2015.74

In addition, there are constant and texture memory spaces for read-only data structures. The original OpenCL uses two types of barrier synchronizations in different scopes: local barriers for all the threads (i.e., workitems) within a workgroup and global barriers for all the threads launched in a kernel. The new OpenCL 2.0 defines an enhanced execution model and a subset of the C/C++11 memory modes. It supports workitem atomics and synchronizations visible to other workitems in a workgroup, across workgroups executing on a device, or for sharing data between the OpenCL device and host [18]. Our work currently only uses OpenCL 1.0 features.

B. Load Imbalance in Graph Processing

Graphs consist of vertices connected with edges (see Fig. 1). In a parallel graph algorithm, a graph is partitioned into smaller subsets with each assigned to a thread to process. Because graphs present irregular structures, one thread may process a subgraph which is different from that of another; this causes runtime differences across threads (i.e., inter-thread load imbalance). In other words, when individual threads process different vertices and their neighborlists, some threads may complete their tasks earlier than the others. Therefore, the short-running threads will need to wait for the long-running threads to finish, wasting computation resources and power. In addition, it is a challenge to make an accurate estimate on load distribution before actually running the application; this behavior is also highly dependent on graph input. Load imbalance may be present in different levels (e.g., across OpenCL workitems, wavefronts or workgroups).

C. Experimental Setup

In this paper, the experimental results are measured on real hardware using an AMD Radeon HD 7950 discrete GPU. The AMD Radeon HD 7950 features 28 GCN CUs with 1792 processing elements with 3 GB of device memory. We use AMD APP SDK 2.8 with OpenCL support. This study is restricted to cases when the working sets of applications do not exceed the capacity of the GPU device memory. We leave graph partitioning for multi-node processing for future work.

III. Graph Coloring

In this paper, we study vertex coloring, which assigns colors to vertices of a graph such that no two adjacent vertices share the same color. Fig. 1 shows a simple graph labelled with five different colors. This paper does not discuss edge coloring; however, our technique can be applied to edge coloring too. There are different types of parallel graph coloring algorithms, each presenting their own unique advantages. The algorithm we developed uses on the well-known Jones-Plassmann algorithm [14] as the GPU baseline. This algorithm is also applied in a state-of-art GPU graph coloring implementation [13]. This algorithm is inherently parallel and is also an ideal example to study the load balancing issue.

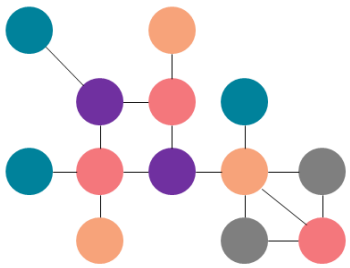

Fig. 1. A graphical representation of a graph labelled with different colors

Algorithm 1 The baseline randomization-based algorithm (device-side kernel)

```
procedure BaseColoring
    for all i = 0 to n do
        if color[i] = uncolored then
            max ← −1
            for j = rows[i], rows[i + 1] − 1 do
                if color[col[j]] = uncolored then
                    cont ← true
                    if node_val[col[j]] > max then
                        max ← node_val[col[j]]
                    end if
                end if
            end for
            max_array[i] ← max
        end if
    end for
    for all i = 0 to n do
        if color[i] = uncolored then
            if node_val[i] > max_array[i] then
                color[i] ← cur_color
            end if
        end if
    end for
end procedure
```

A. Baseline Algorithm

The core of the baseline algorithm [14] is based on inter-vertex value comparisons in local neighborhood. In the initialization step, each vertex is numbered with a random value. We call this approach a **randomization-based labelling** in this paper. The algorithm takes multiple iterations to color all the vertices. Each iteration labels a set of vertices with one color. Each GPU thread (i.e., an OpenCL workitem) is assigned to process a vertex and its edge list. In each iteration, for each vertex, a GPU thread compares the assigned random value of a vertex with that of its neighbors. If the value of a given vertex is the maximum (or minimum) among its neighbors, the thread labels its associated vertex with the color of the current iteration. The algorithm converges when all vertices are colored. The pseudocode (see Algorithm 1) shows the baseline.

Algorithm 1 consists of two *for-all* loops. Each loop index

i is a GPU thread ID (e.g., a workitem) which is obtained through get_global_id(). The graph can be stored in an adjacency matrix where an element (v1, v2) represents that vertex v1 connects to vertex v2. Each row represents the neighborlist for a vertex. We use a compressed sparse row (CSR) format to store a graph in a sparse matrix. Future work will evaluate other formats. In Algorithm 1, the $node\_val[]$ array is initialized with randomized values (one random number for each vertex). The $color[]$ array stores the coloring result for all the vertices and is initialized with negative numbers ($uncolored$). Once a vertex is colored, its corresponding position will be set to a positive integer associated with the color. For each vertex i which is not colored, one thread is assigned to navigate its neighborlist. j represents the index to the element in the neighborlist array ($col[]$), and is used to get the actual vertex id (i.e., $col[j]$) for each neighbor associated with i. The thread evaluates the randomly-assigned values for all the uncolored neighbors (i.e., $node\_val[col[i]]$) associated with the vertex i, calculates the maximum and stores it in the max_array array. The second loop compares the randomized value of each vertex i to the maximum of its neighborlist. If the value of the vertex is larger, then it will be labelled with the color of the current iteration. The $cont$ variable is used by the while loop on the host to determine if all the vertices are colored (set $false$ before the kernel launch in each iteration).

IV. Work Stealing

We reimplemented graph coloring, and modified the OpenCL codes to use work-stealing. The work-stealing implementation uses a state-of-art algorithm similar to the work described by Tsigas and Cedermann [21]. Fig. 2 shows a graphical representation of how work queues are organized and how work stealing works.

We allocate and associate each work queue to a workgroup (with one or more wavefronts). Each element in the queue represents the work consumed by all the workitems (threads) in a workgroup. Important operations include:

- **Pop** dequeues an element from the tail of the local queue.
- **Steal** dequeues an element from the head of a remote queue.

Pushing to the tail of the local queue can be supported but we find it is not necessary for this particular workload. Initially each queue is assigned with some work statically (e.g., no. of vertices / no. of queues). For each workgroup, one thread on behalf of the entire workgroup *pops* a queue element from its local queue, and application-specific information (e.g., parameters and pointers/offsets to the actual work) specified in the element is passed to all the threads in the workgroup to perform the actual computation. When the local queue is empty, a workgroup will attempt to *steal* an element from a remote non-empty queue. The kernel terminates when all the queues are empty. During the execution, contentions may happen, for example, when *pop* and *steal* collide and try to dequeue the same element concurrently. The algorithm uses a lock-free solution [21] to ensure these cases are correctly handled using Compare-and-Swap (CAS) applied on the queue

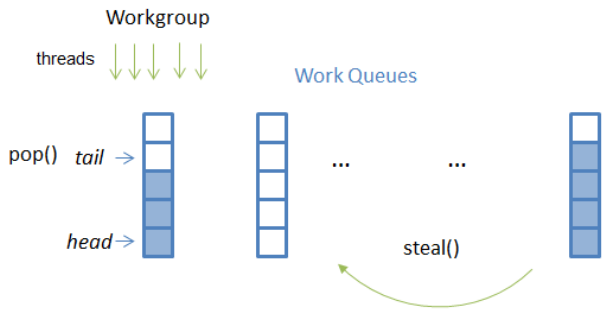

Fig. 2. A graphical representation of work queues and work stealing. A workgroup will pop an element from the tail of its own queue for execution. When its own queue is empty, it will steal a queue element from the head of another queue.

pointers (e.g., heads). For example, with stealing, when the queue is not empty, the steal operation will read the task that head is pointing to and try to move the head to point to the next element in the queue. Multiple workgroups might try to steal the task at the same time and the owner might try to pop it. CAS is used to ensure that the head pointer is not moved while the stealing is happening.

We use compressed sparse row (CSR) format to store the graph. Different rows (i.e., the neighborlists of vertices) have variable lengths (i.e., degrees). This incurs load imbalance across different threads. In our implementation, each workgroup is responsible for processing multiple neighborlists. Individual queue elements in the work queue represent the amount of work for a workgroup. Each queue element encodes some application specific information which is set up by the programmer on the host side. In the case of coloring, each queue element (e.g., a struct) stores the offset to the first row of the associated portion (chunk) of neighborlists the workgroup should process in the CSR sparse matrix. Once the queue element is dequeued by a workgroup (i.e., by a delegate thread), each thread will use its local ID (i.e., ID within a workgroup) and dequeued base offset to calculate its global offset to locate the neighborlist it should process.

Algorithm 2 shows the main computation kernel of coloring with work stealing. One thread in the workgroup, on behalf of the entire workgroup, pops or steals a queue element from its own queue or another queue (*PopSteal()*). The queue element will be stored in the local scratchpad memory and broadcasted to the rest of the workitems in the workgroup which further launches the task (*LaunchTask()*).

A. Performance of Work Stealing

Fig. 3 shows the performance of the work stealing version of coloring compared with a baseline implementation with queuing but without the stealing enabled. The performance improvement with work stealing is not significant (less than 10%). The reason can be clearly explained with Fig. 4. It shows an example of two GPU compute units with each processing multiple workgroups. For each workgroup, there are a small number of threads with extremely long running

Algorithm 2 Work stealing (device-side kernel)

```
procedure WORKSTEALING
    global_id = get_global_id()
    local_id = get_local_id()
    queue_id = global_id/workgroup_size
    if local_id = 0 then
        INITIALIZE_QUEUES(QMeta, QData)
        while true do
            if local_id = 0 then
                IsEmpty =
                POPSTEAL(Elem, QMeta, QData, queue_id)
            end if
            if IsEmpty ≠ 0 then
                break
            end if
            LAUNCHTASK(Elem, local_id, BaseColoring)
        end while
    end if
end procedure
```

times which limit the progress of the entire workgroup. However, the work stealing implementation can only steal tasks in the granularity of workgroups. Load imbalance can only be partially resolved by rebalancing the number of workgroups processed by different GPU compute units. It is not effective to deal with significant load imbalance within each workgroup. This issue is especially critical for processing many real-world unstructured graphs which share a common pattern; that is, the majority of vertices have small degrees while a minority of vertices have very large degrees. Such graphs, including webs and social networks, follow a *Power Law* (see Section VI-A).

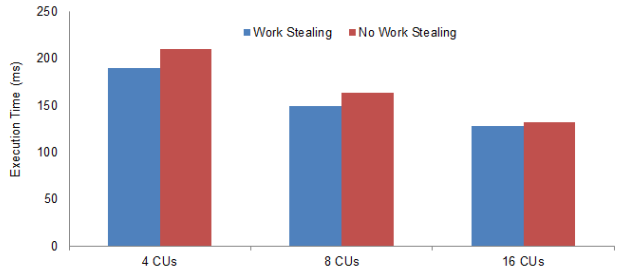

Fig. 3. Work stealing achieves only a small performance benefit to improve load imbalance. The graph uses dip20090126_MAX as an example. Y-axis is execution time in ms.

Table I shows the statistics of queue elements eventually consumed by different CUs after applying work-stealing. Each element represents the task of a workgroup. Before coloring runs, we assign each queue with equal number of queue elements statically. For example, in the 4-CU case, we use 4 queues. The table shows the maximum number of elements consumed by a CU is 81 and the minimum is 75 by another CU with an average of 78 elements across 4 CUs. The standard deviation is 2.45. It suggests that the CU which

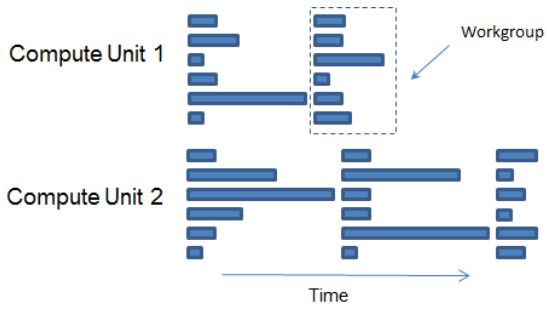

Fig. 4. Work stealing in the workgroup granularity only partially resolves the overall load imbalance. This is due to the significant imbalance within a workgroup/wavefront

TABLE I
NUM OF CUS AND MAX, MIN, AVG AND STDEV OF NUMBER OF ELEMENTS ACROSS ALL THE QUEUES (THE DIP DATASET)

Num. of CUs	max	min	avg (stdev)
4 CUs	81	75	78 (2.45)
8 CUs	41	37	39 (1.28)
16 CUs	22	18	19.5 (0.92)

consumes 75 queue elements has long running tasks and some of its originally-assigned elements are stolen by other threads. Table I also shows that for this input, using larger CU counts, workgroup-level load imbalance lessens, which leaves work stealing only a small room for performance improvement.

V. HYBRID ALGORITHM

This section describes a new hybrid algorithm to improve load imbalance within a workgroup/wavefront. As discussed, our algorithm is designed based on two major observations:

1) A few threads in a SIMD group (e.g., wavefront) may limit the performance of the entire workgroup
2) Once vertices are colored, they will not be evaluated in later iterations.

If the task for a vertex and its neighborlist is assigned to a thread, a larger degree usually suggests longer running time (especially power-law graphs). Therefore, only a few vertices with large degrees may be the actual bottleneck. Other threads, which complete sooner, will be masked off and idle. Therefore, vertex degree can be a hint for the length of a task. We use this property to propose **degree-based labelling** where the local maximum is calculated using the degree value of each vertex. This is similar to the concept proposed by the Welsh-Powell algorithm [9].

In Section III-A, we describe the baseline algorithm with **randomization-based labelling**, which relies on comparison among the vertex and its neighbors using random numbers. How well colors are labelled depends on the randomization quality, which is not the focus of our work. However, load imbalance would be significant especially for irregular graphs, because randomization is performed without any notion of a graph structure.

Our new **hybrid algorithm** improves inter-thread load imbalance by taking vertex degree into consideration. The

key idea is that when we assign colors to vertices, we color those vertices with the largest degrees first, and then they will be inactive during subsequent iterations and thus will not be the performance bottleneck of a workgroup. After a few iterations, most of the remaining vertices to be colored are those with similar and smaller degrees. Thus, active workitems in a workgroup will have similar amounts of work to do. We call it a two-phase hybrid approach, because the first phase uses **degree-based labelling** running a number of iterations before switching to the second phase which uses the original **randomization-based labelling**. One requirement for the algorithm to work is that a vertex should determine if it is the only local maximum in its neighborhood. This requires that the values of different vertices in neighborhood do not have duplication. Using degree-based labelling, the vertices with the largest degrees will be colored first. Therefore, this leads to a fact that later iterations may observe more and more vertices with the same degrees in local neighborhood. Therefore, we switch to randomization-based labelling at a certain point, because randomization-based labelling does not have this issue of local duplication with a good random-number generator.

In specific, the pseudocode in Algorithm 3 shows our overall algorithm. The algorithm has a preprocessing step to calculate the degree for each vertex. Next, similar to the baseline algorithm, each thread is assigned to process a vertex and its edge list. The difference is that comparisons are conducted on degree values instead of randomized values. By comparing the degree of itself with that of all its neighbors, each vertex determines if it is a vertex with a degree that is the local maximum. If it is true, the maximum degree value is stored in a *max_degree_array* array. Separating the coloring process into two *for-all* loops is to avoid the data races to the *color* array. This is achieved through a global synchronization. In the meantime, we use a *flag* array to keep track if a vertex finds any neighbors with the same degree. For each vertex, if a thread finds that its degree is equal to the degree of any vertex, its flag will be set to indicate a duplicate. We only assign a color to a vertex if and $onlyif$ the degree of the vertex is the local maximum and it is the only vertex with the maximum degree (i.e., there is no second local maximum). If none of the vertices satisfies this condition, the algorithm will switch to the second phase of using randomization-based labelling. Alternatively, the user can set up a threshold or user-defined function (e.g., evaluating colorable vertices vs. parallel resources), specifying when to switch from degree-based labelling to randomization-based labelling.

Algorithm 3 Hybrid algorithm (host + device-side)

```
procedure DEGREECOLORING
    for all i = 0 to n do
        if color[i] = uncolored then
            max_degree ← −1
            for j = rows[i], rows[i + 1] − 1 do
                if color[col[j]] = uncolored then
                    cont ← true
                    if degree[i] = degree[col[j]] then
                        flag[i] = true
                    end if
                    if degree[col[j]] > max_degree then
                        max_degree ← degree[col[j]]
                    end if
                end if
            end for
            max_degree_array[i] ← max_degree
        end if
    end for
    for all i = 0 to n do
        if color[i] = uncolored then
            if flag[i] ≠ true then
                if degree[i] > max_degree_array[i] then
                    color[i] ← cur_color
                end if
            end if
        end if
    end for
end procedure

procedure HYBRID
    PRECOMPUTE(degree[])
    while cont do
        cont ← false
        if color[] ≠ color′[]||cur_color < thr||usrdef() then
            DEGREECOLORING
        else
            BASECOLORING
        end if
        color ← color + 1
    end while
end procedure
```

VI. Result Analysis

In this section, we discuss the performance benefits of using the new hybrid algorithm , and how it affects load distributions and quality of graph coloring.

A. Degree Distribution and Input Characteristics

In this study, we choose three sample inputs representing the characteristics of two typical types of real-world graphs (e.g., structured and unstructured). The inputs are chosen from the DIMACS implementation challenge [1], Florida sparse-matrix collection [20] and other sources [10] (see Table II).

Fig. 5 shows degree distributions of three sample graphs. For G3-Circuit, the degree of vertices ranges from two to five. Such regular graphs cause less inter-thread load imbalance. On the other hand, dip and coAuthor are more irregular. A majority of vertices have small degrees, while there are long tails with small number of vertices of very large degrees. Many real-world graphs (e.g., in social network and web domains) present similar characteristics. Prior works use a hybrid data

TABLE II
GRAPH INPUTS

Graph Input	Description	No. of Vertices	No. of Edges
dip20090126_MAX	Database for interacting proteins [10]	19,928	82,406
G3_circuit	Circuit simulation matrix [20]	1,585,478	7,660,826
coAuthorsDBLP	Citation network [1]	299,067	977,676

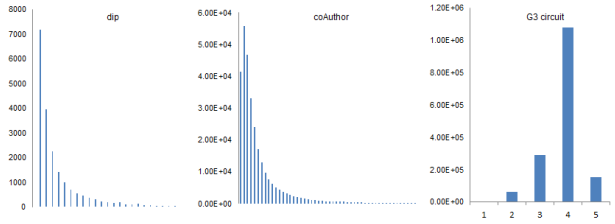

Fig. 5. Degree distribution of three graph inputs *dip*, *G3-Circuit* and *coAuthor*. The y-axis shows vertex counts while the x-axis shows degrees. The long tails of the *dip* and *coAuthor* dataset are only drawn up to 50 for demonstration purpose. The largest degrees are approximately 300 and 240 respectively.

structure to store irregular graphs [3] which requires data preprocessing. We leave a comparison with other approaches to future research.

B. Performance Benefits

We evaluate performance by comparing the hybrid algorithm and the baseline algorithm on an AMD Radeon HD 7950. Fig. 6 shows execution times and associated performance improvement. The execution time is measured on the main computation part, including the PCIe® transfer overhead but excluding I/O and graph parsing.

The results show that our hybrid algorithm achieves better performance compared to the baseline implementation, especially for unstructured graphs. For example, the hybrid implementation is 23% faster than the baseline for the dip20090126_MAX input, and 27% faster than the baseline for the coAuthorsDBLP input. As discussed, the benefit is because the improved design labels colors for high-degree vertices earlier in program execution, so the later iterations process vertices with relatively similar, small degrees resulting in less load imbalance. Compared to work stealing which is only capable of rebalancing loads in the workgroup granularity, the advantage of our approach is that it is effective to deal with bottleneck threads within a workgroup. The performance may be further improved by combining hybrid coloring and work-stealing, which we leave for future work. In Section VI-C, we will provide a clear illustration on how our approach provides benefit by showing changes of degree distribution of active vertices.

G3-Circuit does not get benefit because this input is a structured graph as shown in Fig. 5. If using degree-based coloring, it is highly likely that a vertex cannot be colored. The reason is the probability of a vertex having a neighbor with the same degree and also being the local maximum is high for a structured graph. This suggests that our approach is more useful to process a relatively unstructured graph.

Regarding the portions of GPU kernel vs. PCIe transfer times (excluding I/O and parsing), majority of time is spent on GPU kernel execution for coloring. For example, 5% for dip20090126_MAX, 3% for coAuthorsDBLP and 21% for G3-Circuit are spent on PCIe transfers. Also, G3-Circuit spent relatively more time on PCIe because for structured graphs, it take fewer iterations (i.e., colors) to label a graph.

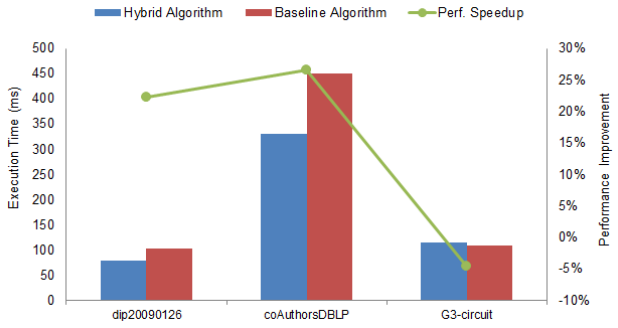

Fig. 6. Execution time comparison between the new hybrid and baseline algorithms. The y-axis shows the execution time and percentage of performance improvement with the hybrid algorithm.

C. Load Distribution over Time

Fig. 7 shows snapshots of workload distribution for active threads in a few representative iterations. We use *dip* as an example. The x-axis is organized by increasing vertex IDs (i.e., vertices are number from 0 to N-1). In the first iteration, all the vertices are involved in the color-labelling process. As shown, there are a number of vertices with extremely high degrees, some of which are as high as 300. Most vertices are with a degree less than 100. After iteration 1 (using degree-based labelling), some high-degree vertices are colored and thus removed from the set for future coloring. The number of the active vertices keeps decreasing in subsequent iterations. In iteration 7, most vertices have a degree less than 50.

As coloring proceeds, active vertices will have more and more neighbors with the same degree and be the local maximum in the mean time. Therefore, degree-based labelling will gradually get diminishing returns in terms of the number of colorable vertices (see Fig. 9). The algorithm switches to randomization-based coloring at the 9th iteration. In this second phase, load imbalance across threads still exists and there is still room for further improvement. However, the vertices which cause long runtimes have already been removed by the first phase, which allows the program to achieve overall better performance.

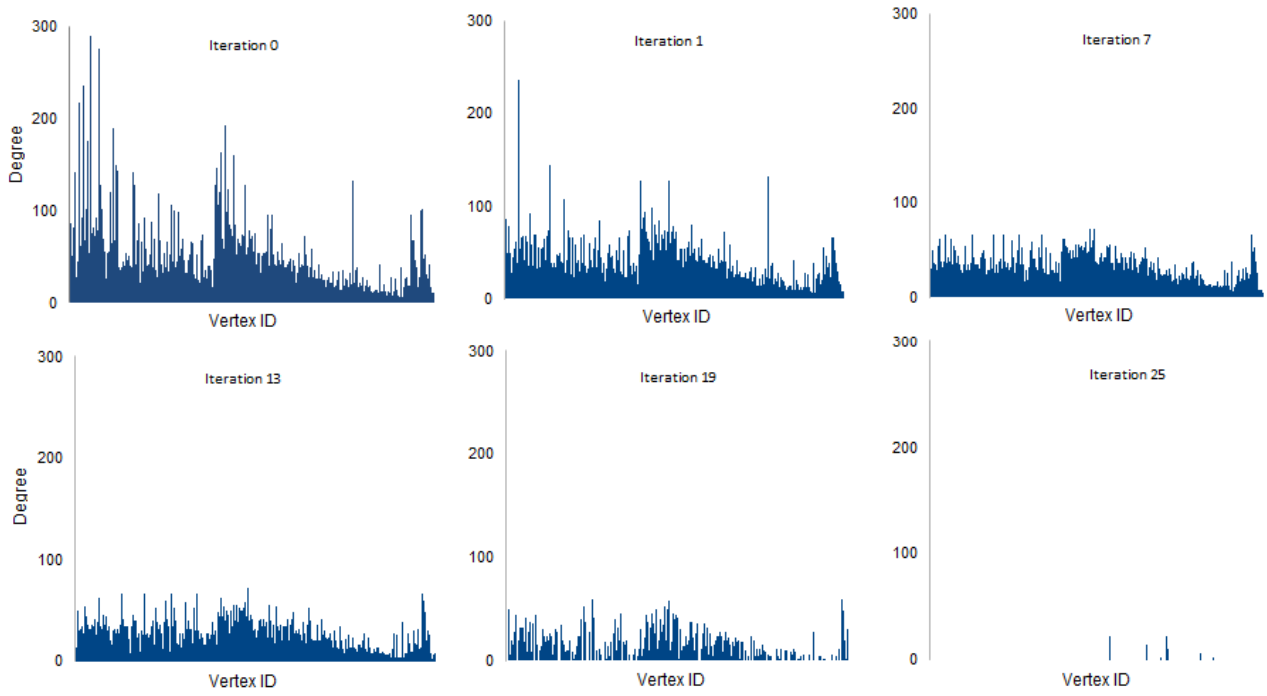

Fig. 7. Snapshot graphs showing the degrees of active vertices across different iterations. High-degree vertices are colored and removed during the first few iterations thanks to the degree-based coloring. The benefit is that later iterations present less load imbalance. X-axis represents vertex ID

D. The Impact of Phase Change

To determine the optimal point (i.e., in which iteration) to switch labelling strategies is an interesting future research problem. Currently, some simple strategies with programmers' hints are used. For example, one way is using a counter tracking the number of colored vertices for degree-based coloring. If there is no or a limited number of vertices which can be colored by degree-based labelling, or the number is not big enough to fill the GPU, it is efficient to switch. For the graph inputs we use, we find that most of the high-degree vertices can be colored with only a few iterations.

Fig. 8 shows the execution time and total number of used colors by varying the iteration when we switch from degree-based to randomization-based labelling. The figure uses dip20090126_MAX as an example. Even for the same input, different numbers of iterations used to run the first phase can lead to different overall execution times and number of used colors (i.e. number of iterations). Note that these are all meaningful coloring results. Varying when to switch will affect the set of vertices involved in the local-maximum evaluation, and thus eventually affect the coloring result. In the case of this input, we observe 15% execution time differences, with the best case happening when we use the 4th iteration to change labelling strategies. We leave the study of the optimal switch point for future work. Fig. 9 shows the accumulative number of colored vertices in each iteration. Each bar also shows a breakdown of number of vertices labelled with different colors.

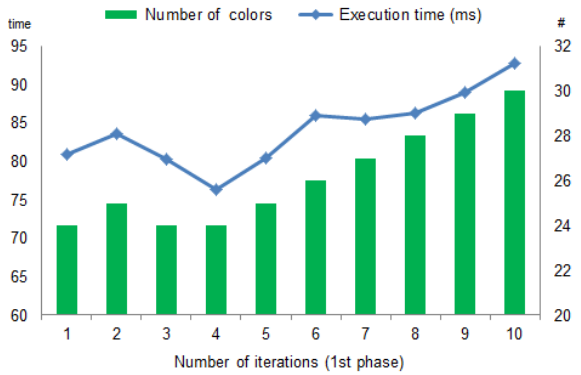

Fig. 8. The y-axis represents two sets of data: the total number of used colors and associated execution time. X-axis represents the number of iterations configured to run the degree-based algorithm. The remaining iterations use the base-line algorithm until the program converges.

VII. Related Work

There are several prior works applying the *work sharing* or *work stealing* techniques to the GPU computing [6], [16], [21], [22]. Our work uses an approach similar to Tsigas and Cedermann [21] to implement a version of graph coloring. We found that the work-stealing approach in the workgroup level is not capable of dealing with fine-grained load imbalance within a workgroup very effectively. It is an open research question whether it is cost effective to apply work stealing within a workgroup.

The baseline version of coloring is a parallelized version of

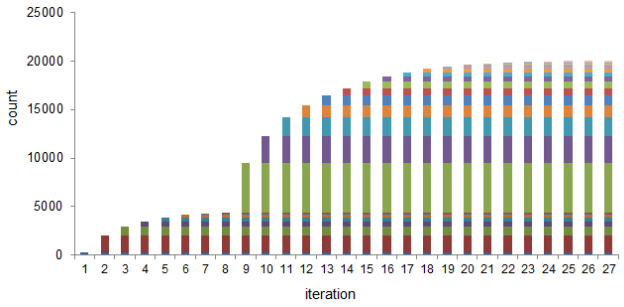

Fig. 9. Number of vertices over time with the dip20090126_MAX input. The y-axis represents the vertex counts while the x-axis represents different iterations. The vertex count bars also show a breakdown of vertices with different colors. At the iteration 9, the algorithm switches from degree-based to the randomization-based labelling.

Jones-Plassmann [14]. Welsh-Powell [9] proposes a largest-degree-first algorithm. Our hybrid algorithm takes advantage of the strengths of both Jones-Plassmann and Welsh-Powell, and optimizes coloring for SIMD hardware. Other prior research [7], [12], [13] has studied parallelization of graph coloring on the GPU platform. However, none have addressed the load imbalance issue. There are some other GPGPU graph work. Our techniques may be applied to solve some of these problems too. Prior benchmarking efforts (e.g. Rodinia [8] and Parboil [19]) included and evaluated a few graph or tree-based algorithms. Burtscher et al. [4] performed a quantitative study of irregular programs on GPUs. Pannotia [7] is a recent work on developing and characterizing graph algorithms specifically on the GPU platform. Other works studied how to accelerate graph algorithms efficiently on GPUs [5], [15], [17], [23].

VIII. Conclusion and Future Work

This paper studies graph coloring and shows the cause of SIMD load imbalance when processing graphs, especially for unstructured graphs. The work stealing mechanism is implemented to balance workloads across GPU workgroups, but it can only achieve limited performance improvement due to significant imbalance within individual workgroups/wavefronts. We then propose a hybrid coloring algorithm to perform coloring with the combination of degree and randomization-based strategies. The algorithm is able to color vertices and mask off the threads assigned with large-degree vertices early in program execution, so that a balanced execution in later processing can be achieved with a better overall performance.

Future work directions include: expanding graph coloring to a multi-machine node configuration; applying the approach to other applications (e.g., independent set and other graph algorithms); evaluating performance of the proposed techniques using different data layouts and more graph inputs; and studying the issue of the optimal point to switch labelling strategies.

Acknowledgement

We would like to thank the reviewers for their constructive comments and suggestions. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. PCIe is a registered trademark of PCI-SIG Corporation. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

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

[2] AMD Accelerated Parallel Processing: OpenCL Programming Guide. Web resource. http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/.

[3] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec 2008.

[4] 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*, Nov 2012.

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

[6] S. Chatterjee, M. Grossman, A. S. Sbirlea, and V. Sarkar. Dynamic task parallelism with a GPU work-stealing runtime system. *Languages and Compilers for Parallel Computing*, pages 203–217, 2011.

[7] S. Che, B. Beckmann, S. Reinhardt, and K. Skadron. Pannotia: Understanding irregular GPGPU graph algorithms. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Sept 2013.

[8] 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 IEEE International Symposium on Workload Characterization*, Oct 2009.

[9] M. B. Powell D. J. A. Welsh. An upper bound for the chromatic number of a graph and its application to timetabling problems. *Computer Journal*, 10:85–86, 1967.

[10] Graph input for interacting proteins. Web resource. http://www.sommer.jp/graphs/.

[11] Graphics Core Next (GCN). Web resource. http://www.amd.com/us/products/technologies/gcn/Pages/gcn-architecture.aspx.

[12] A. V. P. Grosset, P. Zhu, S. Liu, S. Venkatasubramanian, and M. Hall. Evaluating graph coloring on GPUs. In *Poster of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb 2015.

[13] 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.

[14] M. T. Jones and P. E. Plassmann. A parallel graph coloring heuristic. *SIAM Journal of Scientific Computing*, 14:654–669, 1992.

[15] D. G. Merrill, M. Garland, and A. S. Grimshaw. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb 2012.

[16] R. Nasre, M. Burtscher, and K. Pingali. Data-driven versus topology-driven irregular computations on GPUs. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium*, May 2013.

[17] V. M. A. Oliveira and R. A. Lotufo. A study on connected components labeling algorithms using GPUs. In *Proceedings of the 23rd SIBGRAPI Conference on Graphics, Patterns and Images*, Aug 2010.

[18] OpenCL. Web resource. http://www.khronos.org/opencl/.

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

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

[21] P. Tsigas and D. Cedermann. *GPU Computing Gems Jade Edition*, chapter Dynamic Load Balancing Using Work-Stealing. Morgan Kaufmann, 2011.

[22] S. Tzeng, A. Patney, and J. D. Owens. Task management for irregular-parallel workloads on the GPU. In *Proceedings of High Performance Graphics*, June 2010.

[23] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the GPU. In *Proceedings of the Conference on High Performance Graphics*, July 2009.