High Performance Parallel Graph Coloring on GPGPUs

Pingfan Li, Xuhao Chen, Zhe Quan, Jianbin Fang, Huayou Su, Tao Tang and Canqun Yang
College of Computer, National University of Defense Technology, Changsha 410073, China
{lipingfan14,chenxuhao,zhequan,j.fang,shyou,taotang84,canqun}@nudt.edu.cn

***Abstract*—Graph coloring has been broadly used to discover concurrency in parallel computing, where vertices with the same color represent subtasks that can be processed simultaneously. To speedup graph coloring for large scale datasets, parallel algorithms have been proposed to leverage the massive hardware resources on modern multicore CPUs or GPGPUs. Existing GPU implementations either have limited performance or yield unsatisfactory coloring quality (too many colors assigned). We present a high performance parallel graph coloring implementation on GPGPUs with good coloring quality. Our approach employs the *speculative greedy* algorithm which usually yields better quality than the method based on *maximal independent set*. In order to achieve high performance on GPGPUs, we adapt the algorithm to improve work efficiency and reduce overhead, and incorporate several optimization techniques which reduce memory access latency and atomic operation overhead. Our method is evaluated with both synthetic and real-world graphs on the NVIDIA GPU. Experimental results show that our proposed implementations outperform the sequential implementation (3.0× speedup) and the existing GPU implementation from the NVIDIA CUSPARSE library (1.5× speedup), while yielding good coloring quality close to the sequential implementation.**

***Keywords*-Graph Coloring; GPGPU; Speculative Greedy**

I. Introduction

The problem of graph coloring is to assign colors to all the vertices of a graph such that no neighboring vertices have the same color. Graph coloring is a fundamental graph algorithm that has been utilized in many applications [1]–[5], and is also employed by scientific and engineering computing to discover concurrency, e.g. high performance conjugate gradient (HPCG) [6] and incomplete-LU factorization [7], where coloring is used to identify subtasks that can be carried out or data elements that can be updated simultaneously.

To deal with large scale graphs, parallel graph coloring algorithms [8], [9] have been proposed to leverage the massive hardware resources on modern multicore CPUs or GPUs. Existing parallel implementations of graph coloring can be classified into two categories: 1) speculative greedy (SGR) scheme based [10] and 2) maximal independent set (MIS) based [11]. General-purpose graphics processing units (GPGPUs) have been widely used for high performance computing (HPC) during the last decade, owing to their high throughput and energy-efficiency. In this paper, we explore the parallel graph coloring on GPGPUs using CUDA [12] programming model. There are existing GPU implementations of both catigories, but with different algorithms, they exhibit different characteristics of performance and coloring quality. MIS implementations [7] are usually fast since multiple threads can find MIS in parallel independently, but they yield too many colors. On the other hand, SGR implementations [13] generally use less colors than MIS based implementations, but without careful optimizations, they spend much more time to complete coloring.

To overcome the limitations of existing approaches, we propose a high performance GPU implementation of graph coloring which can produce high-quality coloring. Our method is built based on the speculative greedy scheme, and optimized specifically for the GPGPU architecture. We provide both topology-driven and data-driven implementations, and make tradeoffs on task mapping and data movement to take advantage of GPU's compute capability. Meanwhile, we leverage the cache hierarchy in GPUs to reduce the memory access latency. The main contributions of this paper are:

1) We present a simple yet efficient parallel graph coloring algorithm for GPGPUs based on the greedy scheme. The algorithm is carefully designed to better leverage the bulk-synchronous model of GPUs than existing approaches.

2) We employ optimization techniques specifically for the GPU architecture to take advantage of the massive computation resources and memory hierarchies of GPUs.

3) We implement the proposed algorithm and optimizations using CUDA, and evaluate it on the NVIDIA GPU with several synthetic as well as real-world graphs. Experimental results show that our implementation achieves high performance with good coloring quality.

The rest of the paper is organized as follows: the existing sequential and parallel algorithms as well as the state-of-the-art GPU implementations are introduced in Section II. Our proposed schemes are presented in Section III. Section IV presents the experimental results, and Section V concludes.

II. Background and Motivation

Graph coloring refers to the assignment of colors to elements (vertices or edges) of a graph subject to certain constraints. In this paper, we focus on *vertex coloring* which assigns colors to vertices so that no two neighboring vertices (vertices connected by an edge) are assigned the same color. There are several known applications of graph coloring, such as time-tabling and scheduling [1]–[3], register allocation [4], high-dimensional nearest-neighbor search [5],

Algorithm 1 Sequential Greedy Algorithm [10]

```
1: procedure GREEDY(G(V, E))
2:     for each vertex v ∈ V do
3:         for each vertex w ∈ adj(v) do
4:             colorMask[color[w]] ← v
5:         end for
6:         c ← min{i > 0 : colorMask[i] ≠ v}
7:         color[v] ← c
8:     end for
9: end procedure
```

Algorithm 2 Parallel GM Algorithm [10]

```
1: procedure GM(G(V, E))
2:     W ← V                                  ▷ Initialize the worklist
3:     while W ≠ ∅ do
4:         for each vertex v ∈ W in parallel do
5:             for each vertex w ∈ adj(v) do
6:                 colorMask[color[w]] ← v
7:             end for
8:             c ← min{i > 0 : colorMask[i] ≠ v}
9:             color[v] ← c
10:        end for
11:        R ← ∅                  ▷ Initialize the remaining worklist
12:        for each vertex v ∈ V in parallel do
13:            for each vertex w ∈ adj(v) do
14:                if color[v] = color[w] and v < w then
15:                    R ← R ∪ {v}
16:                end if
17:            end for
18:        end for
19:        W ← R                              ▷ Update the worklist
20:    end while
21: end procedure
```

sparse-matrix computation [6], [7] and assigning frequencies to wireless access points [14].

Graph coloring is a well explored problem, and various approaches have been taken to solve it. This is a NP-complete problem to solve optimally, and is known to be NP-hard even sloved approximately [15]. In this paper, we focus on *approximate graph coloring* which yields near-optimal coloring quality. Many heuristics have been developed for approximate solutions, including First Fit (FF), Largest Degree First (LF), etc. These heuristics make trade-off between minimizing the number of colors and execution time, but generally the faster algorithms have poor coloring quality while the slower ones tend to yield fewer colors. In the following, we introduce some existing sequential algorithm as well as parallel ones.

A. Sequential Graph Coloring

A sequential algorithm [10], [16] that performs approximate graph coloring with the greedy scheme is shown in Algorithm 1. It is not optimal, but it is fast and easy to implement. In all the algorithms specified in this paper, we use similar data structures to those introduced in [10]. $adj(v)$ denotes the set of vertices adjacent to the vertex v, $color$ is a vertex-indexed array that stores the color of each vertex, and $colorMask$ is a color-indexed mask array used to mark the colors that are impermissible to a particular vertex v. At the beginning of the procedure, the array $color$ is initialized with each entry $color[w]$ set to zero to indicate that vertex w is not yet colored, and each entry of the array $colorMask$ is initialized with some value $a \notin V$.

When processing the vertex v, the algorithm scans all its neighbors (line 3), and their colors are forbidden to be assigned to the vertex v (line 4). By the end of the inner for-loop, all of the colors that are impermissible to the vertex v are recorded in the array $colorMask$. It is then scanned from left to right in search of the lowest positive index i at which a value different from the current vertex v is encountered; this index corresponds to the smallest permissible color c to the vertex v (line 6). The color c is then assigned to the vertex v (line 7). Note that since the colors impermissible to the vertex v are marked in the array $colorMask$ using v (instead of a boolean flag) as a label, the array $colorMask$ does not need to be re-initialized in every iteration of the loop over the vertex set V.

B. Parallel Graph Coloring

When applied to large scale problems, such as sparse-matrix computation [6], [7] and chromatic scheduling [3], parallel graph coloring is required to meet the performance requirement. Because of its sequential nature, the greedy scheme is challenging to parallelize. Basically, two classes of approaches have been investigated in the past to tackle this issue.

Gebremedhin and Manne (GM) [9] proposed a *speculation* scheme to deal with the inherent sequentiality of the greedy scheme. The main idea is to color as many vertices as possible in parallel, tentatively tolerating potential conflicts, and detect and resolve conflicts afterwards. Algorithm 2 shows the details of the GM algorithm. It can be divided into two parts: the first part (from line 4 to line 10) is the same as the sequential algorithm but done in parallel. The second part (from line 12 to line 18) does the conflict detection (line 14) and puts the conflicting vertices into the remaining worklist (line 15). Based on this *speculative greedy* (SGR) algorithm, Çatalyürek *et al.* developed OpenMP implementations for the multi-core and massively multithreaded architectures [10]. Rokos *et al.* improved Çatalyürek's algorithm and implemented it on the Intel® Xeon Phi coprocessor [17].

The other approach relies on iteratively finding a *maximal independent set* (MIS) of vertices in a progressively shrinking graph and coloring the vertices in the independent

Algorithm 3 Parallel JP Algorithm [7]

```
1: procedure JP(G(V, E))
2:     W ← V, c ← 1
3:     while W ≠ ∅ do
4:         S ← ∅                          ▷ Initialize the independent set
5:         for each vertex v ∈ W in parallel do
6:             r(v_i) ← random()
7:         end for
8:         for each vertex v ∈ W in parallel do
9:             flag ← true
10:            for each vertex w ∈ adj(v) do
11:                if r(v) <= r(w) then
12:                    flag ← false
13:                end if
14:            end for
15:            if flag = true then
16:                S ← S ∪ {v}
17:            end if
18:        end for
19:        for each vertex v ∈ S in parallel do
20:            color[v] ← c               ▷ Color an independent set
21:        end for
22:        W ← W − S, c ← c + 1
23:    end while
24: end procedure
```

set in parallel. In many of the methods in this class, the independent set is computed in parallel using some variant of Luby's algorithm [11]. An example is the work of Jones and Plassmann (JP) [18]. Algorithm 3 shows the details of the JP algorithm. Gjertsen *et al.* [19] introduced an advanced parallel heuristic, PLF, that consistently generates better colorings than the JP heuristic with slight overhead. Two new parallel color-balancing heuristics, PDR(k) and PLF(k) are also introduced. Hasenplaugh *et al.* [20] futher improve the ordering heuristics based on the JP algorithm.

C. GPU Implementations

Grosset *et al.* [13] implement the GM algorithm using CUDA on GPUs. They use a 3-step graph coloring framework: 1) *Graph partitioning* which partitions the graph into subgraphs and identifies boundary vertices, 2) *graph coloring & conflicts detection* which colors the graph using the specified heuristic, e.g. FF, and identifies color conflicts, and 3) *sequential conflicts resolution* which goes back to CPU and resolves the conflicts. Note that step 2 is performed multiple times on GPU to reduce the number of conflicts before going back to CPU. Although this `3-step GM` algorithm can get as few color as (if not fewer than) the best sequential graph coloring algorithm, its performance is much worse than the sequential graph coloring, meaning the GPU computation horsepower is not leveraged very well.

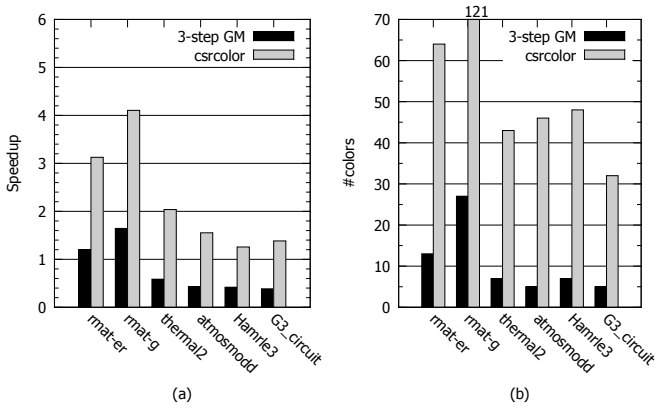

Figure 1. Comparison between two existing GPU graph coloring implementations: `3-step GM` and `csrcolor`. (a) runtime speedup normalized to the sequential implementation (the more the better); (b) the number of colors assigned (the less the better).

The CUSPARSE [21] library offered by NVIDIA includes `csrcolor` [7] routine which does graph coloring on a given graph in CSR format [22]. The algorithm of `csrcolor` is derived from the JP algorithm, but uses the *multi-hash* method to find maximal independent sets. Basically, several hash functions (instead of random number generators) are selected, and used to generate hash values for each vertex with the vertex number as the input of the hash functions. Given the generated hash values, local maximum and minimum values can be found, and distinct (maximal) independent sets are generated for each of the hash values. Assume N hash values are associated with each vertex, and used to create different pairs of (maximal) independent sets, this multi-hash method can generate $2N$ (maximal) independent sets at once. As reported [7], the `csrcolor` implementation runs pretty fast on modern NVIDIA GPUs. However, it usually produces several times more colors than the sequential algorithm, which is not satisfactory for many applications. For example, when applied to exploit concurrency in parallel computing, more colors means less parallelism, because tasks (vertices) with the same color can be processed concurrently.

We evaluate the two existing GPU implementations of graph coloring on the NVIDIA K20c GPU. Fig. 1 shows the performance and coloring quality of both implementations. As illustrated, `3-step GM` yields much better coloring quality than `csrcolor`, but its performance becomes even worse than the sequential implementation, meaning this GPU implementation dose not exploit GPU hardware very well. On the other hand, `csrcolor` runs much faster than `3-step GM`, and gains a certain degree of speedup over the sequential implementation. However, this good performance comes at the expense of much worse coloring quality: it yields several times more colors than the sequential implementation and `3-step GM`. The limitations of `csrcolor` and `3-step GM` motivate us to design a

Algorithm 4 Topology-driven Parallel Graph Coloring

```
1: procedure TOPO-GC(G(V, E))
2:     do
3:         changed ← false
4:         for each vertex v ∈ V in parallel do
5:             if colored[v] = false then
6:                 for each vertex w ∈ adj(v) do
7:                     colorMask[color[w]] ← v
8:                 end for
9:                 c ← min {i > 0 : colorMask[i] ≠ v}
10:                color[v] ← c
11:                colored[v] ← true
12:                changed ← true
13:            end if
14:        end for
15:        for each vertex v ∈ V in parallel do
16:            for each vertex w ∈ adj(v) do
17:                if color[v] = color[w] and v < w then
18:                    colored[v] ← flase
19:                end if
20:            end for
21:        end for
22:    while changed = true
23: end procedure
```

Algorithm 5 Data-driven Parallel Graph Coloring

```
1: procedure DATA-GC(G(V, E))
2:     W_in ← V                          ▷ Initialize the in worklist
3:     while W_in ≠ ∅ do
4:         for each vertex v ∈ W_in in parallel do
5:             for each vertex w ∈ adj(v) do
6:                 colorMask[color[w]] ← v
7:             end for
8:             c ← min {i > 0 : colorMask[i] ≠ v}
9:             color[v] ← c
10:        end for
11:        W_out ← ∅                     ▷ Initialize the out worklist
12:        for each vertex v ∈ V in parallel do
13:            for each vertex w ∈ adj(v) do
14:                if color[v] = color[w] and v < w then
15:                    W_out ← W_out ∪ {v}
16:                end if
17:            end for
18:        end for
19:        swap(W_in, W_out)             ▷ Swap the worklists
20:    end while
21: end procedure
```

better implemention of parallel graph coloring for GPGPUs to achieve both good performance and coloring quality.

III. Design

Graph algorithms are typical irregular algorithms [23] that are considered to be difficult to parallelize on GPUs. However, recent works [24]–[28] in this area, show that GPUs are capable to substantially accelerate graph algorithms if they are carefully designed and optimized for the GPU architecture. In this section, we propose our design based on existing knowledge and previously proposed optimization strategies for other graph algorithms, and adapt the techniques for solving the graph coloring problem.

A. Design Overview

Nasre *et al.* [29] introduced the concept of *topology-driven* and *data-driven* imlementations of irregular applications on GPUs. For graph algorithms, the topology-driven implementation simply maps each vertex to a thread, and in each iteration, the thread stays idle or is responsible to process the vertex depending on whether the corresponding vertex has been processed or not. The topology-driven implementation is straightforward, and since GPUs are suitable for accelerating data-parallel applications, it is easy to map onto the GPU hardware and possibly get speedup. By contrast, the data-driven implementation maintains a worklist which holds the remaining vertices to be processed. In each iteration, threads are created in proportion to the size of the worklist (i.e. the number of vertices in the worklist). Each thread is responsible for processing a certain amount of vertices in the worklist, and no thread is idle. Therefore, the data-driven implementation is generally more work-efficient than the topology-driven one, but it needs extra overhead to maintain the worklist. Note that the data-driven implementation still suffers from load imbalance problem, since vertices may have different amount of edges to be processed by the corresponding threads.

We implement our graph coloring in these two fashions, and then compare their performance and coloring quality. In the previous evaluation we find that speculative greedy (i.e. GM) algorithm inherently yields better coloring quality than the maximal independent set (i.e. JP) method. Thus we choose to use the speculative greedy scheme and design our algorithm skeleton on top of it. Compared to the `3-step GM` algorithm, our proposed GPU implementation maps the entire coloring work onto the GPU, consequently removing the data transfer between the CPU and the GPU. The rationale behind this change of data movement is, as throughput-oriented processors, GPUs are good at exploiting data level parallelism, so recomputing the conflicted work rather than serializing it onto the CPU would be more straightforward and efficient. To achieve performance close to the MIS method (e.g. `csrcolor`), we employ GPU specific optimizations to take advantage of the hardware.

B. Algorithmic Adaptation

Algorithm 4 shows the topology-driven graph coloring algorithm. In this topology-driven algorithm, a flag *changed*

is used to indicate whether all the vertices are colored or not. It is cleared at the beginning of each iteration, and set by one or more threads if any vertex is colored. Once all the vertices are colored, the flag remains $false$ and the algorithm finally terminates. The vertex coloring as well as the conflict detection and resolve are similar to the GM algorithm. Algorithm 5 shows the data-driven graph coloring algorithm. It is almost the same as the GM algorithm except that Algorithm 5 uses *double buffering* [29] to avoid copying the worklist. The two worklists W_{in} and W_{out} are referenced by pointers, and they are swapped at the end of each iteration. Since they are operated using pointers instead of data values, no copy operation is required between the two worklists.

C. Optimization Techniques

We use the well-known compressed sparse row (CSR) [22] sparse matrix format to store the graph in memory consisting of two arrays. Fig. 2 provides a simple example. The column-indices array C is formed from the set of the adjacency lists concatenated into a single array of m (m is the number of edges) integers. The row-offsets R array contains $n+1$ (n is the number of vertices) integers, and entry $R[i]$ is the index in C of the adjacency list of the vertex v_i. We store graphs in the order they are defined and do not perform any preprocessing in order to improve locality or load balance.

Memory irregularity may lead to poor GPU system performance [30]. As illustrated in Fig. 3, the CUDA kernels are highly memory latency bound. Since graph coloring uses CSR format to store the indices and data in memory, this indirect memory access pattern leads to statically unpredictable memory access behavior which is dependent on the input dataset. To mitigate the effect of irregular memory access, efforts should be made to reduce as many off-chip memory accesses as possible. Therefore, the optimization should focus on keeping the main data structures (i.e. the C array, the R array and the $color$ array) on chip.

The GPU memory hierarchy consists of register file, L1 memories (scratchpad, L1 cache, and read-only data cache), shared L2 cache, and off-chip GDDR DRAM [31]. The L1 memory is private per-SM and shared by sibling warps.

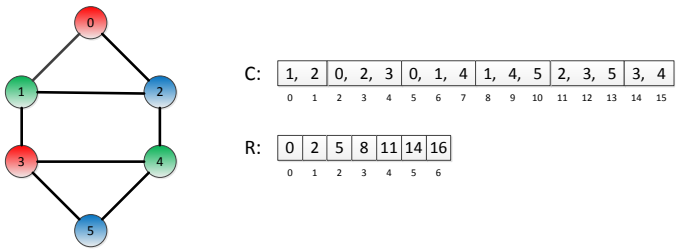

Figure 2. An example of the compressed sparse row (CSR) format. For this graph, at least three colors (red, green, blue) are needed.

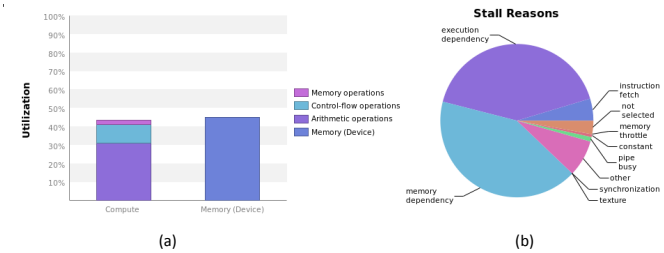

Figure 3. Graph coloring is highly memory latency bound. (a) Achieved compute throughput and memory bandwidth are both below 60% of peak, indicating the kernel is most likely limited by the latency of memory operations. (b) The break-down of instruction stalls reasons averaged over the entire execution of the kernel, among which `memory dependency` dominates. (Instruction stall reasons indicate the condition that prevents warps from executing on any given cycle.)

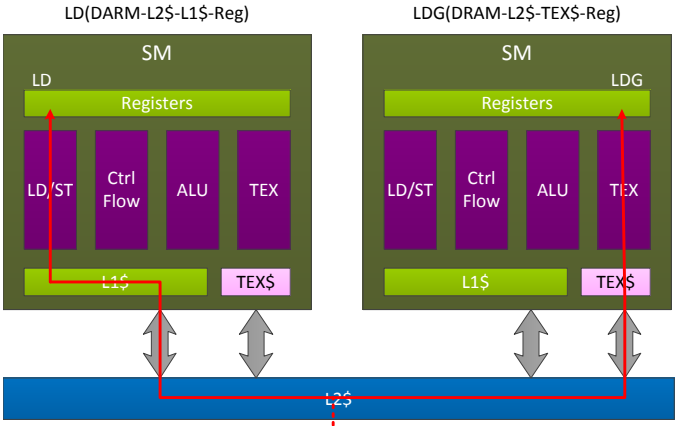

Figure 4. The difference between `ld` (left) and `ldg` (right).

Scratchpad memory (shared memory in CUDA terminology) is programmer visible and can be used for explicit intra thread block communication. The same on-chip memory (64KB in total in Kepler GPUs) is used for both L1 data cache and scratchpad. Note that L1 caching in Kepler GPUs is reserved only for local memory accesses, such as register spill and stack data, which is different from that in Fermi GPUs. Global loads are cached in L2 only (or in the read-only data cache). Each SM also has a read-only data cache of 48 KB to speed up reads from device memory. It accesses this cache either directly or via a texture unit. When accessed via the texture unit, the read-only data cache is also referred to as texture cache. The L2 cache works as the central point of coherency, and is shared across all threads of the entire kernel. It is partitioned into multiple banks that are connected to each memory channel. Atomic operations are performed at each memory partition by the Atomic Operation Unit (AOU).

Read-only Data Caching. In CUDA devices of compute capability 3.5 and higher, data that is read-only for the entire lifetime of the kernel can also be cached in the read-only data (unified L1/texture) cache by reading it using the

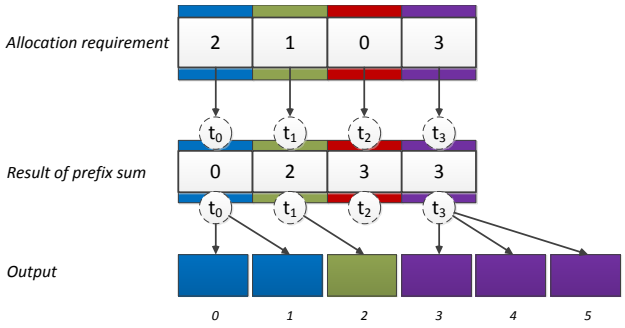

Figure 5. Example of prefix sum for computing scatter offsets for updating the remaining worklist. Input order is preserved. The figure is duplicated from [24].

intrinsic `__ldg()` [12]. Fig. 4 illustrates the difference between `__ld()` and `__ldg()`. We propose to use the read-only data cache to hold read-only data, i.e. the C array and the R array. `__ld()` is the normal load operation with which the data walks through DRAM, L2 cache, L1 cache to the register file (as mentioned data in global memory is actually not cached by the L1 cache). `__ldg()` is the read-only data cache load operation with which the data walks through DRAM, L2 cache, L1 read-only cache to the register file. In this case, more read-only data can be cached in the L1 read-only cache whose access latency is around 30 cycles which is much shorter than the DRAM access latency (about 300 cycles). Therefore, `__ldg()` can improve the performance because of reduced DRAM accesses.

Atomic Operation Reduction. For the data-driven implementation, another overhead comes from the atomic operations. In Algorithm 5, since the *out* worklist is a shared data structure, pushing elements into the worklist (line 15) requires atomic operations to ensure correctness. Although GPU architects have paid a lot of effort to optimize atomic operation, serialization from atomic synchronization is still expensive for GPUs [24], because generally mutual exclusion does not scale to thousands of threads, and the fine-grained dynamic serialization within the SIMD width is much more expensive than between overlapped SMT threads on CPUs. Thus Merrill *et al.* [24] proposed to use software *prefix sum* for updating the shared worklist in the data-driven implementation. Parallel prefix sum [32], [33] is a bulk-synchronous algorithmic primitive that can be used to compute scatter offsets for concurrent threads to place dynamic data within shared data structures such as global queues.

Fortunately, efficient GPU prefix sums [34] have been proposed, and the CUB [35] library has already provided standard routines for CUDA users to invoke. This allows us to easily reorganize sparse and uneven workloads into dense and uniform ones. Given a list of allocation requirements for each thread, prefix sum computes the offsets for where each thread should start writing its output elements. Fig. 5 from [24] illustrates updating the worklist using prefix sum. In this example, the thread t_0 wants to produce two items (e.g. two vertices), t_1 one item, t_2 zero items, and so on. The prefix sum computes the scatter offset needed by each thread to write its output element. Thread t_0 writes its items at offset zero, t_1 at offset two, t_3 at offset three, etc. In the context of parallel graph coloring, parallel threads use prefix sum when assembling confilicting vertices into the remaining worklist. In this case, local atomic operations are not necessary, and the globally shared worklist is only atomically updated once for each thread block in each iteration.

IV. Evaluation

We use the R-MAT [36] graph generator to create synthetic graphs. The R-MAT algorithm determines the distribution by using four non-negative parameters (a; b; c; d) whose sum equals one. We generated two graphs (RMAT-ER and RMAT-G) with 1M vertices size but varying structures by using the following set of parameters: (0:25; 0:25; 0:25; 0:25); (0:45; 0:15; 0:15; 0:25). We also pick some real-world graphs with more than 1M vertices from the University of Florida Sparse Matrix Collection [37]. Smaller datasets are excluded from the experiment since we focus on large-scale problems which are more common in real-world applications. The 2 symmetric positive definite (s.p.d.) and 4 nonsymmetric matrices with the respective number of vertices (i.e. rows) and edges (non-zero elements) are grouped and shown according to their increasing order in Table I. The graphs vary widely in degree distribution of the vertices and density of local subgraphs, and they are from different application domains.

We compare seven schemes including (1) the sequential implementation [13], (2) `3-step GM` [13], (3) the basic topology-driven implementation (T-base), (4) T-base with `ldg` (T-ldg), (5) the basic data-driven implementation with reduced atomic operations (D-base), (6) D-base with `ldg` (D-ldg) and (7) `csrcolor` [7]. Among them, (3)$\sim$(6) are our proposed implementations. We conduct the experiments on NVIDIA K20c GPU with CUDA Toolkit 7.0 release. The sequential implementation is executed on Intel Xeon E5 2670 2.60 GHz CPU. All the benchmarks are executed 10 times and we collect the average execution time to avoid system noise. The timing is only performed on the computation part of each program, and the I/O part is excluded from the evaluation for all the implementations.

Fig. 6 shows the number of colors needed by different implementations for each graph. It is not surprising that the first six schemes need similar amount of colors, since they are all based on the speculative greedy scheme. The slight difference among the four schemes may result from the different orderings that are caused by different thread mapping stratigies and so on. `csrcolor`, however, needs much (4.9$\times\sim$23$\times$) more colors than the sequential algorithm, making this MIS based scheme unattractive or even unapplicable in many scenarios. This substantial difference of

Graph	No. vertices	No. edges	Min. deg.	Max. deg.	Avg. deg.	Variance	s.p.d	Application
rmat-er	1048576	20971268	2	59	20.00	23.37	no	Synthetic
rmat-g	1048576	20964268	0	899	20.00	472.81	no	Synthetic
thermal2	1228045	8580313	1	11	6.99	0.66	yes	Thermal Simulation
atmosmodd	1270432	8814880	4	7	6.94	0.06	no	Atmospheric Model
Hamrle3	1447360	11028464	4	15	7.62	7.21	no	Circuit Simulation
G3_circuit	1585478	7660826	2	6	4.83	0.41	yes	Circuit Simulation

Table I
SUITE OF BENCHMARK GRAPHS

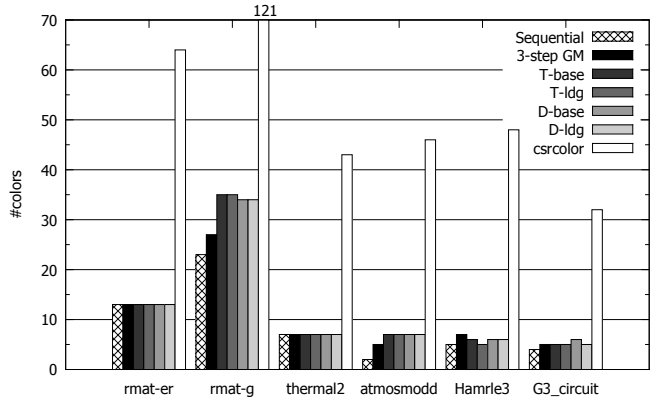

Figure 6. Comparing the number of colors among different schemes.

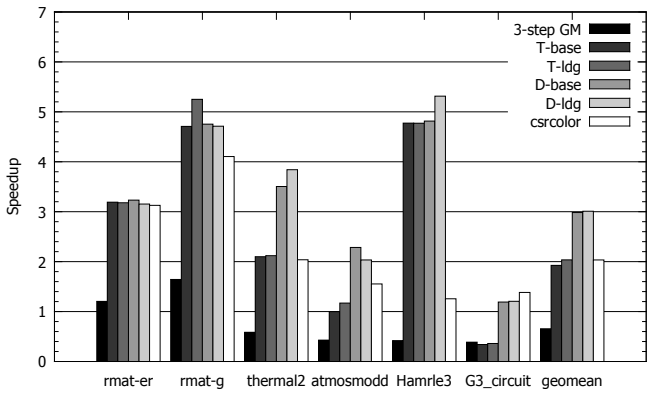

Figure 7. Runtime speedup normalized to the sequential algorithm.

coloring quality between csrcolor and the other schemes stems from the different processing methods of SGR scheme and MIS scheme. SGR scheme uses essentially the same method as the sequential greedy scheme but optimistically does coloring in parallel with later conflict detection and resolve, while MIS scheme tries to find independent sets iteratively, which does not cause any conflict, but for performance concern, the methods used to find independent sets should be simple enough and thus generate solutions that are far away from the optimal.

Fig. 7 illustrates the execution time speedup over the sequential implementation. As mentioned before, 3-step GM gets unacceptable performance: it is much slower than the sequential implemention (0.66× slowdown on average). Our proposed implementations, although based on the same speculative greedy scheme as 3-step GM, achieve significant speedup over the sequential implemention. Moreover, the topology-driven implementations get the performance close to csrcolor, while the data-driven ones are even 1.5× faster than csrcolor. For some benchmarks like Hamrle3, our implementions even significantly outperform csrcolor. This performance boost comes from the refinement of the algorithm structure as well as the optimizations specific to the GPU architecture. As illustrated, with the changes of task mapping and data movement in the algorithm, we boost the performance with an average speedup of 2× and 3× for the topology-driven and data-driven implementations respectively. Furthermore, __ldg() can bring a certain degree of speedup for some benchmarks such as thermal2 and Hamrle3, although on average its impact on performance is not very distinct. Another interesting observation is that the data-driven implementations perform much better than the topology-driven ones for thermal2, atmosmodd and G3_circuit, since the data-driven implementations are more work efficient and improve the utilization of the SIMT hadware.

We also notice that for G3_circuit which has the most vertices and is the sparsest (with average degree of 4.83) among the selected benchmarks, our proposed implementations do not perform very well, while csrcolor still get some speedup over the sequential implementation. This implies that our approach might be limited by the scale and/or sparsity of the given graph. This is because the sparser the graph is, the less temporal locality it has, and with very large scale the kernel becomes extremly memory latency bound, which could not be mitigated by the optimizations that we employ. On the contrary, csrcolor is only slightly affected by these features, possibly due to its better memory access behavior (the *color* array is not frequently accessed). In this extreme case, csrcolor is relatively more efficient, and this limitation of our approach is left to be solved for our future work.

To find out the influence of the thread block size on performance, we conduct the experiment with varying thread block sizes. Fig. 8 shows that the thread block size does have siginificant effect on performance. Generally, with small thread block sizes, e.g. 32-thread, there are few warps running simultaneously on each stream multiprocessor (SM), and thus the memory access latency can not be hidden very well by warp interleaving. For graph coloring which is

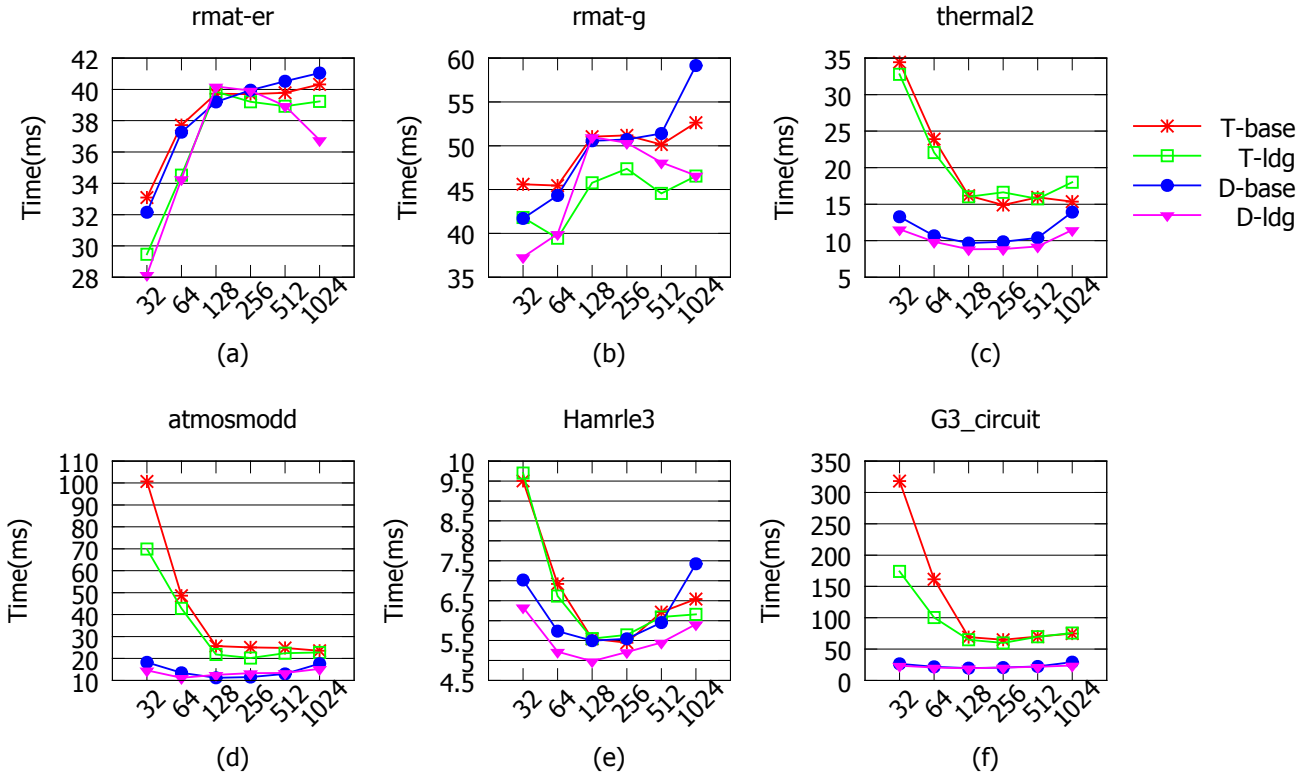

Figure 8. Performance with different thread block size.

highly latency bound, the performance would be extremely limited if the memory access latency is not carefully dealt with. However, larger thread block size doesn't always mean better performance. As shown in the figure, in most cases the performance peaks at 128-thread or 256-thread block size, but in some cases the thread block size 32 leads to best performance. This indicates that further optimization is required to improve parallelism. Note that thread block sizes larger than 256 usually can not get optimal performance due to resource oversaturation. Since 128-thread block size leads to best average performance, and does not cause significant performance loss in all cases, we choose to use this configuration as the default.

V. Conclusion

Graph coloring is an important graph algorithm that has been applied in many application areas. To process large scale graphs, parallel graph coloring has been intensively studied in the past. GPGPUs have been broadly utilized to speed up compute intensive kernels in high performnce computing (HPC) applications in the past decade. In this paper, we explore the parallel graph coloring implementation on GPGPUs. Existing implementations either achieve limited performance or yield unsatisfatory coloring quality. We present a high performance graph coloring implementation for GPGPUs with good coloring quality. Our method is derived from the speculative greedy algorithm, but improved with GPU specific optimizations. Experimental results on the NVIDIA K20c GPU show that our implementation outperforms existing GM or JP implementations on GPUs in terms of performance and coloring quality.

Acknowledgment

We thank the anonymous reviewers for their insightful comments and suggestions, and A.V. Pascal Grosset from University of Utah for generously sharing his source code. This work is partly supported by the National Natural Science Foundation of China (NSFC) No.61502514, No.61402488, and No.61502509, Research Fund for the Doctoral Program of Higher Education of China (RFDP) No.20134307120035 and the National High Technology Research and Development Program of China (863 Program) No.2015AA01A301.

References

[1] D. J. A. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, pp. 85–86, 1967.

[2] V. Lotfi and S. Sarin, "A graph coloring algorithm for large scale scheduling problems," *Computers & Operations Research*, vol. 13, no. 1, pp. 27–32, Jan. 1986.

[3] T. Kaler, W. Hasenplaugh, T. B. Schardl, and C. E. Leiserson, "Executing dynamic data-graph computations deterministically using chromatic scheduling," in *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 154–165, 2014.

[4] G. J. Chaitin, "Register allocation & spilling via graph coloring," in *Proceeding of the 1982 SIGPLAN symposium on Compiler Construction*, pp. 98–101, June 1982.

[5] S. Berchtold, C. Böhm, B. Braunmüller, and D. A. Keim, "Fast parallel similarity search in multimedia databases," in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pp. 1–12, June 1997.

[6] E. Phillips and M. Fatica, "A cuda implementation of the high performance conjugate gradient benchmark," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, vol. 8966, pp. 68–84, 2015.

[7] P. C. M. Naumov and J. Cohen, "Parallel graph coloring with applications to the incomplete-lu factorization on the gpu," NVIDIA Research, Tech. Rep., 2015.

[8] J. R. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. L. Martin, "A comparison of parallel graph coloring algorithms," Northeast Parallel Architecture Center, Syracuse University, Tech. Rep., 1995.

[9] A. H. Gebremedhin and F. Manne, "Scalable parallel graph coloring algorithms," *Concurrency: Practice and Experience*, 2000.

[10] U. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen, "Graph coloring algorithms for multicore and massively multithreaded architectures," *Parallel Computing*, vol. 38, no. 10-11, pp. 576–594, Oct. 2012.

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

[12] *CUDA C Programming Guide v7.0*, NVIDIA, March 2015.

[13] 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*, pp. 297–298, 2011.

[14] J. Riihijarvi, M. Petrova, and P. Mahonen, "Frequency allocation for wlans using graph colouring techniques," in *Second Annual Conference on Wireless On-demand Network Systems and Services*, pp. 216–222, Jan 2005.

[15] D. Zuckerman, "Linear degree extractors and the inapproximability of max clique and chromatic number," in *Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing*, pp. 681–690, 2006.

[16] S. Dalton, N. Bell, L. Olson, and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," 2014, version 0.5.0. [Online]. Available: http://cusplibrary.github.io/

[17] G. Rokos, G. Gorman, and P. Kelly, "A fast and scalable graph coloring algorithm for multi-core and many-core architectures," in *Euro-Par 2015: Parallel Processing*, vol. 9233, pp. 414–425, 2015.

[18] M. T. Jones and P. E. Plassmann, "A parallel graph coloring heuristic," *SIAM J. Sci. Comput.*, vol. 14, no. 3, pp. 654–669, May 1993.

[19] R. K. Gjertsen, Jr., M. T. Jones, and P. E. Plassmann, "Parallel heuristics for improved, balanced graph colorings," *Journal of Parallel and Distributed Computing*, vol. 37, pp. 171–186, 1996.

[20] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson, "Ordering heuristics for parallel graph coloring," in *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 166–177, 2014.

[21] NVIDIA, "CUSPARSE Library," 2015. [Online]. Available: http://docs.nvidia.com/cuda/cusparse/

[22] J. Dongarra, "Compressed row storage." [Online]. Available: http://web.eecs.utk.edu/dongarra/etemplates/node373.html

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

[24] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 117–128, 2012.

[25] A. McLaughlin and D. A. Bader, "Scalable and high performance betweenness centrality on the gpu," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 572–583, 2014.

[26] J. Barnat, P. Bauch, L. Brim, and M. Ceska, "Computing strongly connected components in parallel on cuda," in *Proceedings of the IEEE 25th International Parallel Distributed Processing Symposium*, pp. 544–555, May 2011.

[27] R. Nasre, M. Burtscher, and K. Pingali, "Morph algorithms on gpus," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 147–156, 2013.

[28] S. Nobari, T.-T. Cao, P. Karras, and S. Bressan, "Scalable parallel minimum spanning forest computation," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 205–214, 2012.

[29] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus topology-driven irregular computations on gpus," in *Proceedings of the IEEE 27th International Parallel Distributed Processing Symposium*, pp. 463–474, May 2013.

[30] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive cache management for energy-efficient gpu computing," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 343–355, 2014.

[31] *NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110*, NVIDIA, 2012.

[32] G. Blelloch, "Scans as primitive parallel operations," *IEEE Transactions on Computers*, vol. 38, no. 11, pp. 1526–1538, Nov 1989.

[33] S. Sengupta, M. Harris, and M. Garland, "Efficient parallel scan algorithms for gpus," Tech. Rep. NVR-2008-003, December 2008.

[34] S. Yan, G. Long, and Y. Zhang, "Streamscan: fast scan algorithms for GPUs without global barrier synchronization," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pp. 229–238, 2013.

[35] D. Merrill, "CUB," NVIDIA Research, 2015. [Online]. Available: http://nvlabs.github.io/cub/

[36] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *SDM*, 2004.

[37] "The university of florida sparse matrix collection." [Online]. Available: http://www.cise.ufl.edu/research/sparse/matrices/